Yale University
Department of Computer Science

Auditing the Structural Reliability of the Clouds

Ennan Zhai David Isaac Wolinsky Hongda Xiao
Hongqiang Liu Xueyuan Su Bryan Ford

YALEU/DCS/TR-1479
July 2013

Auditing the Structural Reliability of the Clouds

Ennan Zhai David Isaac Wolinsky Hongda Xiao Hongqiang Liu Xueyuan Su
Bryan Ford

Abstract

Large scale systems, common in cloud computing, rely on redundancy for reliability and availability. Modern clouds have become ever-increasingly complex and diverse creating large messes that experience long outages when failures occur. While there exist significant effort in resolving faults after they occur, we propose a novel approach to untangling this mess before it occurs by auditing the underlying structure of a cloud, which we call the cloud **S**tructural **R**eliability **A**uditor (SRA). SRA achieves our goal by auditing a cloud with the following steps: 1) collecting comprehensive component and its dependency information, 2) using this data to construct a system-wide fault tree, 3) and leveraging fault tree analysis algorithms to determine and rank sets of components based on the likelihood of causing a cloud service outage. SRA enables a cloud administrator to be able to evaluate risks within the cloud beforehand and improve the reliability of her service deployments before the occurrences of critical failure events. We have built a prototype implementation that performs all three tasks. Using this prototype, our experimental evaluation shows that SRA is practical: auditing a cloud containing $13,824$ servers and $3,000$ switches spends about 6 hours.

1 Introduction

As companies increasingly move their computations and data onto third-party cloud resources, the debate continues as to whether these services are as reliable as providers claim [13]. As more individuals depend on cloud services for critical services, clouds have greater need to eliminate failures before they occur as to avoid downtime and lost business for both them and their consumers. To support this effort, cloud providers employ extensive redundancy for their services. Amazon S3, for example, stores each object at multiple servers in a given Amazon S3 region [3]. Apple's iCloud service, similarly, rents infrastructure from both Amazon's EC2 and Microsoft's Azure [4].

Nevertheless, as increasingly common reports of cloud outages attest [5, 6, 8, 23], reliability remains imperfect. The complex interdependent structure found within today's clouds coupled with the fact that these interdependencies often extend across multiple providers or administrative domains none of whom has a "complete picture" of the entire system creates pitfalls that may compromise reliability [28–30, 38]. These complex dependencies may unintentionally introduce unknown common dependencies leading to unexpected correlated failures, reducing the effectiveness of redundancy as a means to improve availability and reliability. Suppose for example a cloud provider, **A**, replicates critical state in two data centers **B** and **C** to ensure reliability. **A**, however, does not know both **B** and **C** depend on a common power supply **P**. If **P** fails, both **B** and **C** fail simultaneously, resulting in a unexpected correlated failure causing **A** to fail despite redundancy.

This preceding example has happened in reality. Despite Amazon EC2 having multiple data centers located in Northern Virginia for reliability, a recent lightning storm in this region taking out both Amazon's main power supply and its backup generator, disabling all the data centers and EC2 service in the area [8].

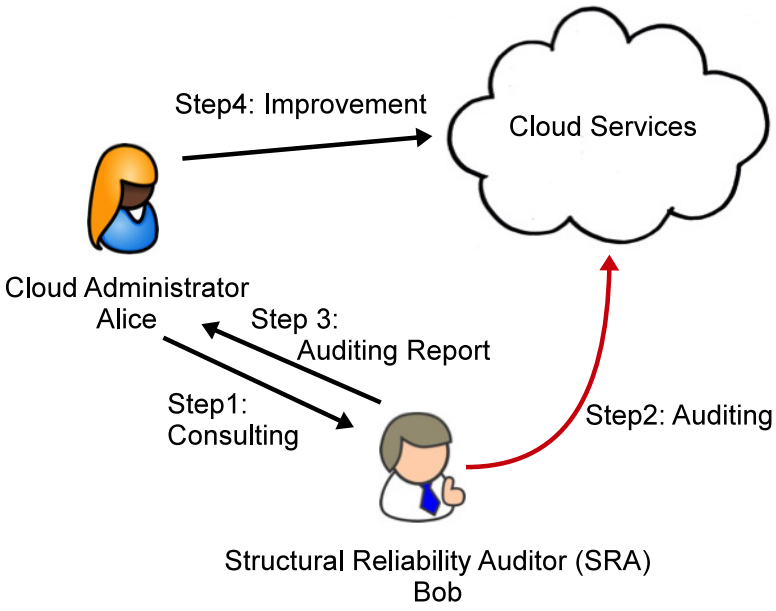

Figure 1: The relationships among cloud provider, its services and cloud reliability auditor.

Worse, the effect carried over to other services disabling Netflix, Instagram, Pinterest, Heroku and other services that relied heavily on EC2 there.

If cloud providers could easily determine and account for all critical dependencies that may result in failures ahead of time, then uptime would be nearly 100% except maybe in cases involving "acts of God." For instance, Google's investigation efforts indicate that although they understand close to 37% of failure correlations hidden in their global storage systems can truly cause failure burst disabling their service, they do not know how to directly detect or identify them [29]. Leaving the discovery of unknown correlated failures an important but unresolved challenge.

To address this problem, we propose a novel approach to auditing the structure of cloud services called the structural reliability auditor (SRA). The SRA, shown in Figure 1, is a set of tools that a cloud provider may employ in this process. To determine unknown dependencies resulting in correlated failures within a cloud service, SRA first collects cloud component and component dependency information (e.g., network, hardware and software dependencies). Then the SRA builds a fault tree [21, 45] using collected dependencies. Finally, the SRA uses fault tree analysis algorithms to generate reliability auditing reports, which can be used to determine high-risk events that will result in correlated failures causing service outages.

To the best of our knowledge, SRA is the first system for auditing common dependencies resulting in potential correlated failures within clouds. In order to provide effective and practical approach, SRA makes the following contributions:

- automated collection of dependencies including network, hardware and software dependencies from a cloud for structural auditing;
- adapt fault trees for use in reasoning about common dependencies within clouds;

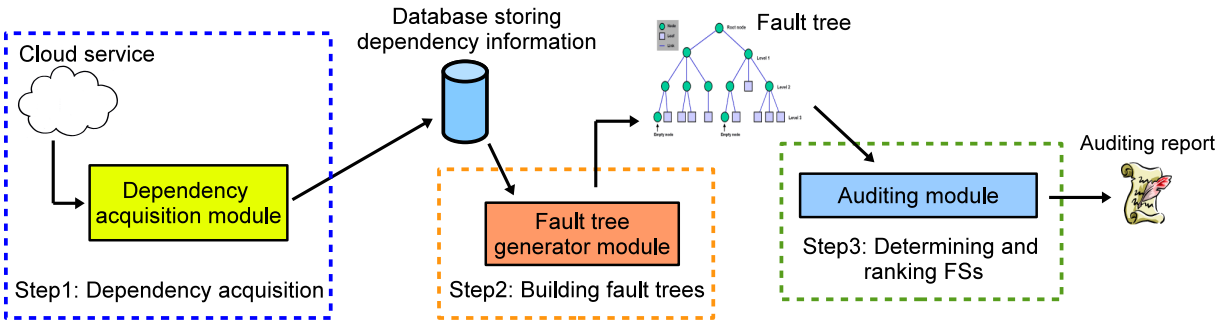

Figure 2: The overview of SRA architecture.

- determine common dependencies resulting in correlated failures within a cloud and rank the dependencies based on their likelihood of occurrence; and
- a privacy-preserving cloud auditing mechanism.

Road-map. The rest of this paper is organized as follows. The next section describes our target problem and the challenges on building SRA. Section 3 presents SRA's architecture and detailed design of our approach, and Section 4 describes our implementation and evaluations. Some discussions are given in Section 5. Section 6 presents related work and finally Section 7 concludes this paper.

2 Problem & challenges

2.1 Problem

Cloud providers often leverage redundancy techniques as a means to improve the structural reliability and availability of their services by reducing the likelihood of occurrence of a *failure set* (FS), a set of components whose simultaneous failure would cause a cloud service outage. The knee-jerk reaction of deploying redundancies as a means to address FSs may overlook important issues and backfire, as these efforts might inadvertently produce unexpected dependencies resulting in small FSs and increasing the likelihood of correlated failures causing service outages [28, 30].

Many existing service outage accidents result from unexpected FSs. Well-known individual cloud services, e.g., EC2 [2] and Azure [9], make use of redundancy to ensure the structural reliability of their services, e.g., introducing backup servers and switches into their data centers [17, 26, 31, 39, 55]. However, additional redundancies may not mitigate the likelihood of failure due to a failed FSs, derailing cloud providers' efforts for improving reliability. Amazon, for example, recently experienced a significant disruption in the Northern Virginia EC2 data center due to correlated failures resulting from incorrect configuration within a few aggregation switches [5]. While Amazon may look at efforts to prevent a repeat of this issue, the general problem, an unexpected compromised FS, likely will rear its ugly head again in the future.

As the cloud diversifies, application service providers no longer depend upon a single cloud service (e.g., IaaS) and have begun using multiple cloud service providers for enhanced reliability [22]. Netflix [10], for instance, utilizes three independent Amazon EC2 availability regions [1], while Zynga [16], developer of many Facebook games, uses both EC2 and an internal "cloud". Despite these efforts, application service providers may be unaware and unable to mitigate failures due to unknown FSs shared by distinct cloud

providers. For instance, a recent ferocious lightning storm in Northern Virginia [8, 11] took out both Amazon's primary and backup power supplies resulting in unavailability of the three EC2 redundancies supporting Netflix, thus causing a Netflix service outage in this area.

Our goal. We begin by exploring the challenges associated with these problems. We then propose a solution and share our initial prototype that can help cloud providers not only determine FSs within their clouds, but also rank them in order of importance assisting cloud administrators in their efforts to improve the structural reliability of their clouds as effectively as possible by fixing potential issues before they occur. Beyond that, these efforts provide depth and understanding about the complexities within a cloud and can even be used as a diagnosis tool to assist in isolating live errors within a cloud.

2.2 Challenges

Several technical challenges appear in detailed analysis of these problems. The following list focuses on those we view as critical toward solving our goal.

Challenge 1: Acquiring dependencies. In order to discover FSs for a given cloud service, the components information within the cloud and their associated dependencies must be acquired. Since infrastructures underlying cloud services tend to be complex, asking an administrator to populate this data set may be an infeasible task. Therefore, acquiring the dependencies efficiently becomes an important problem. Existing approaches towards this effort found in monitoring and diagnosis systems have been limited to networking ignoring hardware, software and other dependencies [19]. History has shown unsuitability of networking alone, as failures resulting from software and hardware become rather commonplace [14].

Challenge 2: Determining and ranking FSs. Even with a set of components and their dependencies, obtaining FSs and ranking them remains unresolved. Within this challenge, there exists the need to construct an useful dependency graph and instrumenting it with potential failures. Determining FSs collected from a service provides a difficult challenge due to potentially complex dependencies. Existing efforts [19, 37] have tried to solve similar problems with diagnosis analysis; however, these approaches may fail to provide accurate results when confronted with complex dependencies [50].

Challenge 3: Privacy Preservation. Auditing application cloud services that rent from different cloud services introduces another complex challenge: preserving the privacy of the cloud providers while performing auditing of the application service and its dependencies. The problem also relates to auditing a cloud service and its external dependencies such as Internet and electricity providers. To the best of our knowledge, there is no existing work done in this area. The obvious approach of using a third-party who signs a NDA (non-disclosure agreement) with each cloud provider may not work in practice due to lack of motivation by cloud providers [28, 41]. Ideally, cloud providers need trust no one with their internal secrets.

3 System Architecture

In this section, we describe a structural reliability auditor (SRA), a system capable of acquiring dependency information and then discovers and ranks FSs within a cloud service.

3.1 Overview

The SRA performs an audit of a cloud service using the following three steps, diagrammed in Figure 2:

Step 1: Dependency acquisition. As depicted in Figure 2, SRA's dependency acquisition module processes a cloud service collecting as much component and dependency information as possible storing the results into a database for post processing. Components may include any resource and service that runs within or outside of a cloud limited only by the ability to mine them for useful information. Section 3.2 presents the detailed design of this module.

Step 2: Building fault trees. Component and dependency information only presents a part of the story: without a clear question or goal, processing this amount of information can be cumbersome and unhelpful. The next component, the fault tree generator module, takes as input a target or goal, processes the data acquired in the previous step in order to build a fault tree, a type of dependency graph. While traditionally fault trees [45] have been used to evaluate the risks of automatic control systems, we adapt it for reasoning about the reliability of cloud services. We present detailed descriptions on building fault trees in Section 3.3.

Step 3: Determining and ranking FSs. Using the fault tree, SRA begins the analysis phase of the audit with the specific goal of finding high-risk FSs in the cloud service based on the likelihood of failure. As shown in Figure 2, the auditing module uses the fault tree generated in the previous step and outputs an auditing report that includes a ranked list for FSs. The process for analyzing the fault tree in order to determine and rank FSs is given in Section 3.4.

Auditing services that span multiple providers. There still exists situations in which the auditing requires sensitive information from an external party. In these scenarios, the external party or parties may be unwilling to share this information. For example, Amazon may not be willing to share detailed information about its data centers with Netflix, but nonetheless Netflix would like to audit the reliability of their cloud structure. Thus, the auditor may need to use a privacy preserving mechanism for determining FSs. We discuss one possible solution in Section 3.5.

3.2 Dependency acquisition

The dependency acquisition module, depicted in Figure 3, collects as much information as possible about a cloud's components and their dependencies, and then stores that data into a database for use later by SRA's other modules. We separate the dependency acquisition module into two parts: 1) various types of dependency collectors that acquire component information and then report information upstream to a 2) Dependency Acquisition Manager (DAM). Dependency collectors target different hardware types, network information, software, and other resources as well such as air conditioning, power, and geographical location. A collector can exist for anything measurable. The DAM acquires all this information from the various collectors and stores it within a database. While the DAM may discover many dependencies, explicit definition of dependencies occurs later based upon the information collected.

In the rest of this section, we focus discussion on three dependency collectors, each of which have been implemented: 1) network collector that discovers components and dependencies within data center network, e.g., servers, routers, switches and connections between them; 2) hardware collector that acquires various components and configurations from each server, e.g., CPU, disks, network cards of servers and relationships between them; and a 3) software collector that analyzes cloud software stacks to determine correlations between programs within the applications running on hosts, and the calls and libraries used by these programs.

In practice, these dependency collectors may be inadequate and it may be necessary for cloud providers to build their own. Customized collectors can take advantage of likely proprietary management infrastructure that might provide significantly more information than general purpose protocols and databases. Our discussion of these three collectors is intended to illustrate the architecture and general functionality, not to

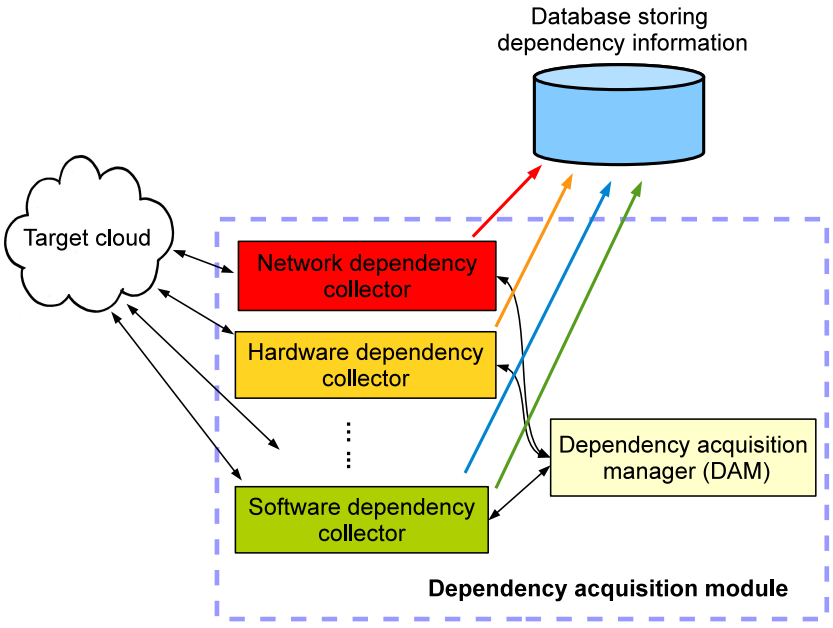

Figure 3: The architecture of dependency acquisition module.

propose that these specific modules would be directly suitable for any particular real cloud environment.

Network dependency acquisition. The network collector obtains information on the hardware used to support the network infrastructure within the cloud and connecting it to the Internet, but it also plays a key role in understanding the cloud topology, locations and relationships of the various switches, servers, and other networked resources. Using Figure 4 as a simple but illustrative example, the collector gathers network information about aggregation switch 1-4, server 1-4 and all the communication paths between the components. This data can be automatically acquired by leveraging existing network management protocols, e.g., SNMP, to collect network topology within data centers. In most of data centers, switches have LLDB (Link Layer Discovery Protocol) enabled, SNMP can be used to obtain the set of directly connected neighbors for each switch. Recursing through the switches using SNMP, the collector can determine the entire network topology in real-time. SNMP also contains the routing tables, which can then be used to analyze the path(s) between each pair of switches [51].

Hardware dependency acquisition. While the cloud provider may have detailed records of all purchases, that information may not be consistent with reality and cannot include dynamic information present in hardware. The hardware collector iterates across the entire set of servers inside a cloud obtaining detailed information about each server. This information includes the various component types, such as CPUs, PCIs, network cards, memory, disks and drivers; their product information such as vendors, machine life, model numbers, and uptime; and even software installed and its versions. In the Figure 4, the hardware dependency collector contacts each of the Servers (Server 1-4) and obtains this information using existing hardware information acquisition tools (e.g., lshw and hwinfo). The hardware collector can be extended to obtain additional information or additional collectors can be written that would collect unique information about a server that does not fit into the framework for the default collector.

Software dependency acquisition. Cloud computing software stacks form another topology and understanding its relationship becomes critical to understanding potential failures as network and hardware

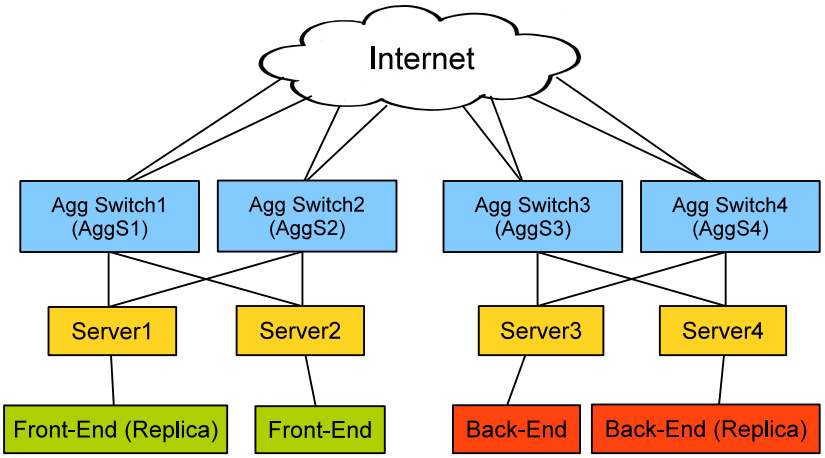

Figure 4: An example on topology underlying a typical service.

failures could result in a software failure. The software collector obtains the following information: 1) location of each application component, for example, in Figure 4, using Hadoop as an example, the master, slaves, file system services, etc; 2) dependencies for application components, or rather the relationship among the various moving pieces and how a failure in one may render the remaining into a failure state as well; and 3) library dependencies of each program. The collector obtains these dependencies by utilizing existing profiling tools (e.g., gprof and perf). We currently rely on the cloud operator to provide the remaining information: location of each component and their dependencies. Ideally, the dependencies could be obtained by analyzing the flow of RPC (remote procedure calls) through the system while introducing faults in order to discover dependencies that the provider may be unaware of.

Failure probability acquisition. In order to achieve the goal of determining the likelihood of FSs' failures, these and other dependency acquisition modules can be extended to obtain failure probability. Hard drives, 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, SRA could make use of an online crowd-sourced database in which individuals post failure (or non-failure) information regarding various hardware 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. On the other hand, the probability of stumbling across a software failures (bugs) or configuration problems can not easily be computed. So for events for which we are unable to obtain failure probabilities from reliable sources, we make use of logs to estimate historical reliability as a measure for future reliability. For each component, a cloud provider can extract how much time the component was offline, τ over a given time period T and obtain the failure probability by $F = \tau/T$. The existing study [30] successfully obtained the failure probability of each of component failure events in a Microsoft data center based on this approach.

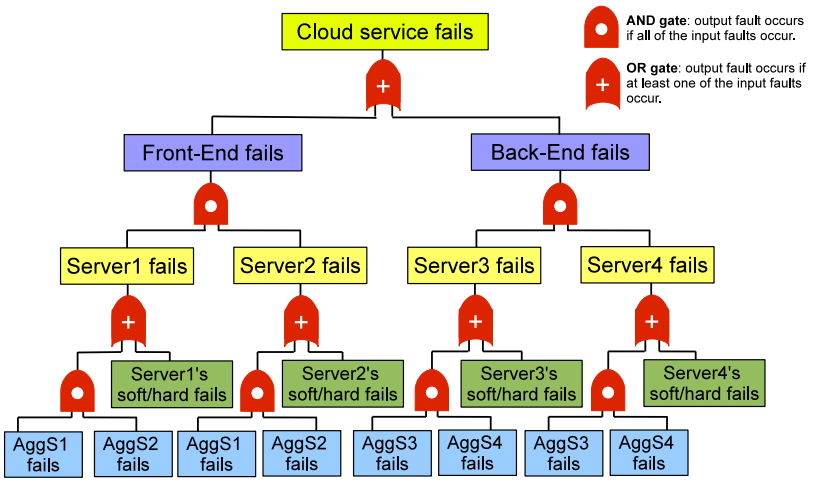

Figure 5: Fault tree which is constructed based on our example.

3.3 Building fault trees

Using the component and dependency information acquired earlier, a cloud provider can now use SRA to examine dependencies within their services. This occurs in two steps: 1) defining the goal of the reliability evaluation and 2) constructing a dependency graph using the acquired dependency data. Examples of evaluation goals include ensuring that at least one server is online, some threshold of servers are online, or a particular cloud application runs correctly, e.g., Hadoop. The goal assists in processing and filtering the component and dependency data in order to produce the dependency graph. Since our eventual goal is to determine reliability, we chose to model our dependency graphs as fault trees [45]

3.3.1 Building a dependency graph

To generate a dependency graph, the SRA must know the goal of the dependency graph or the components of interest. Without this information, a dependency graph can still be generated, except that it may contain potentially unimportant information and take an exceptionally long time to produce and process as the dependency database contains a significant amount of information. To determine the useful components, the SRA makes use of a *templating mechanism*, which predefines the relationships of various components and queries the dependency database in order to construct a dependency graph that satisfies the SRA's goal.

Figure 5 shows a completed dependency graph in the form of a *fault tree* based on the example shown in Figure 4. For now, we focus on the components and their relationships saving the discussion on fault trees to the proceding section, Section 3.3.2. The approach taken in constructing the dependency graph involves determining the set of aggregation switches (AggS), core routers (Core), servers and top switches of racks (ToR) that satisfy the goal in evaluation. Then the network topology information defines another dependency relationship where servers rely on the various networking components. As the final high level component, the SRA maps out the software components into the dependency graph. With the high level components laid out, the SRA again queries the dependency database, but this time filling out details for each of the high level components as the various templates define, these details include power, hardware components, and network cables.

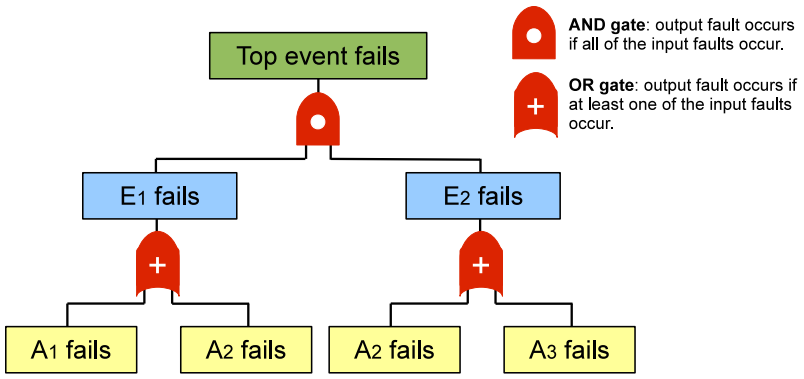

Figure 6: A typical example for fault tree.

3.3.2 Using Fault Trees (FTs)

A fault tree is a type of dependency graph that focuses on the logical relationships between events causing component failures within a given system. A typical fault tree has two types of building blocks: *failure events* and *logic gates*. A failure event represents one possibile way a component may fail, due to an internal failure or by loss of power, for example. Fault trees represent the occurrence of a failure event as a 1 and normal behavior as a 0. Logic gates connect the failure events together and propagate the failure upwards based upon the set of failure inputs. The selection of logic gate depends on the relationship shared by the connected events. For example, if a failure should propagate upward due to any one failure occurring an "OR" gate is used. While if all failure events must occur for the failure to propagate, an AND gate is used. Leaf nodes or a fault event for individual components use the name *basic failure events*, while the root node, or the service or goal of interest, is named *top event* or a system failure.

We illustrate the fault tree concept in Figure 6. The top event branches logically into equally contributing events. Tracing through the fault tree reveals the various events whose occurrence could lead to the failure of the top event. The figure shows that each failure event connects directly to an *input gate*, which connects an event to its children.

Returning to our earlier example in Figure 5, the fault tree represents the dependency graph in terms of failing components. This fault tree determines and analyzes the conditions under which the cloud service will behave normally. This implies that at least one server running the front-end software and one server running the back-end software remains online and properly working. Furthermore, the software itself must not fail either, for example, due to bugs.

3.4 Determining and ranking FSs

As the final step in the SRA's process, the auditing module, depicted in Figure 7, determines and ranks the FSs from a given fault tree using the FSs determination algorithms container (FDC) and the FSs ranking metrics container (FRC), respectively. The FDC currently supports two algorithms for obtaining the FSs trading off speed and accuracy. The FRC then processes the list of resulting FSs and ranks them based upon the chosen metric, thus producing the SRA's output.

The remainder of this section presents algorithms for determining FSs within a fault tree: *minimal cut set algorithm* and *failure sampling algorithm*, for determining FSs; and we conclude the section with an overview of our ranking metrics.

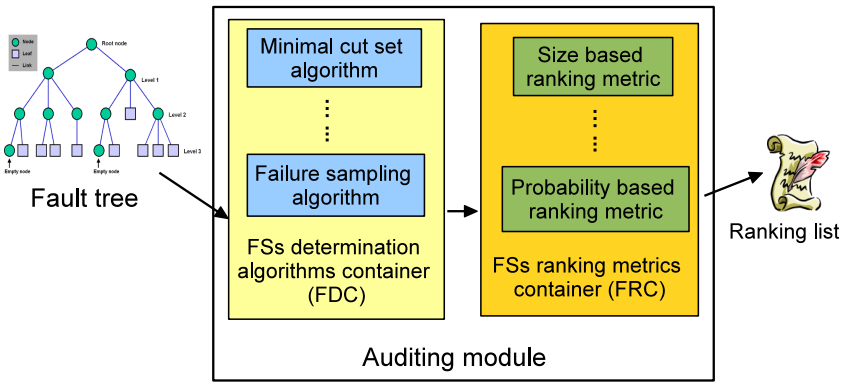

Figure 7: The architecture of the auditing module.

3.4.1 Minimal cut set algorithm

The minimal cut set algorithm is a deterministic approach for obtaining all the FSs within a given fault tree. We first introduce two terms: *cut set* and *minimal cut set*.

Cut sets. A cut set consists of a collection of basic events with the property that if all these events occur simultaneously, then top event occurs as well. For example, for the fault tree in Figure 6, if events A_1 and A_3 occur simultaneously, the top event occurs. Thus, $\{A_1, A_3\}$ is a cut set. Similarly, $\{A_1, A_2\}$, $\{A_1, A_2, A_3\}$, $\{A_2\}$ and $\{A_2, A_3\}$ are cut sets. In this sense, a cut set applied to a cloud service corresponds to a FS.

Minimal cut sets. A minimal cut set is the smallest combination of basic events with the property that if all of these events occur simultaneously, then the top event of fault tree to occurs as well. In other words, if any element were removed from a minimal cut set, it would no longer be a cut set. Using our example, consider the following two cut sets: $\{A_1, A_2\}$ and $\{A_2, A_3\}$. These are not minimal cut sets, because $\{A_2\}$ alone is sufficient to cause top event to occur. The fault tree in Figure 6 has two minimal cut sets: $\{A_2\}$ and $\{A_1, A_3\}$.

Minimal cut set algorithm. Our first approach to determining the minimal cut sets uses a reversed breadth-first algorithm to traverse each of the events in the fault tree with pseudocode presented in Algorithm 1. The algorithm produces cut sets for each of the visited events. Basic events generate cut sets containing only themself, while non-basic events produce cut sets based on the cut sets of their children events and their gate type. OR gates produces an aggregate of the children events' cut sets to produce the visited event's cut set, while AND gate produces cut sets by using the Cartesian Product among the cut sets of its children events. The last step reduces the top event's cut sets to minimal cut sets.

3.4.2 Failure sampling algorithm

In practice, computing all the minimal cut sets tends to be very expensive as counting minimal cut sets is NP-hard [44]. To address this issue, we also consider a randomized sampling algorithm that makes a tradeoff between accuracy and run-time with pseudocode presented in in Algorithm 2.

The failure sampling algorithm uses a breadth-first traversal of the fault tree. At each leaf node, or basic event, the algorithm flips a random coin assigning with equal probability either a 1 or a 0. Upon visiting non-basic events, the algorithm evaluates the children inputs. In the same fashion, the root value is assigned a 1 or 0 based upon all its children. When the top event's value is 1, the algorithm defines a cut set consisting

Algorithm 1: Minimal cut set algorithm

Input: Fault tree T
Output: MinimalCutSet

foreach $event_i \in T$ *by reversed breadth-first traversal* **do**
  if $event_i$ *is* basic event **then**
    $event_i.CutSet.append(event_i)$;
  else
    if $event_i.InputGate$ *is* OR **then**
      foreach $c_i \in event_i.ChildList$ **do**
        foreach $cs_i \in c_i.CutSet$ **do**
          $event_i.CutSet.append(cs_i)$
    else /* Namely $event_i.InputGate$ 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 $c_i.CutSet$ and $c_j.CutSet$ */
          $list \leftarrow c_i.CutSet \times c_j.CutSet$;
          $event_i.CutSet.append(list)$;

/* reduce redundant items in $root.CutSet$ and assign the result to $minimalCutSet$, and then simplify minimalCutSet. */
$minimalCutSet \leftarrow reduce\_redundancy(root.CutSet)$;
$minimalCutSet \leftarrow simplify(minimalCutSet)$;
return minimalCutSet;

Algorithm 2: Failure sampling algorithm

```
Input: Fault tree T and the number of samples N
Output: CutSet
for i ← 1 to N do
    foreach event_i ∈ T by reversed breadth-first traversal do
        if event_i is basic event then
            event_i.occurrence ← 0 or 1 based on random flipping a fair coin
        else
            event_i.occurrence ← 0;
            if event_i.InputGate is OR then
                foreach c_i ∈ event_i.ChildList do
                    if c_i.occurrence is 1 then
                        event_i.occurrence ← 1;
                        break;
            else                                   /* Namely, event_i.InputGate is AND */
                foreach c_i ∈ event_i.ChildList do
                    if c_i.occurrence is 0 then
                        break;
                event_i.occurrence ← 1;
    if root.occurrence is 1 then
        TmpSet ← ∅;
        foreach event_i ∈ T do
            if event_i.occurrence is 1 then
                TmpSet.append(event_i);
        CutSet.append(TmpSet);
return CutSet;
```

of basic events assigned 1. We define this entire process as a *sampling round*. Our algorithm executes a large number of sampling rounds and aggregates the resulting cut sets (i.e., our FSs) in those rounds. While the cut sets computed by this method may not be minimal, we have done proof (see Appendix) to provide guarantees that we can find out most of the critical cut sets.

3.4.3 Ranking metrics for FSs

The cut sets give only minimal information about the likelihood of a failure, in this section, we introduce ranking metrics that help cloud administrators focus their attentions on the important FSs.

Size based ranking approach. The first method assumes that FSs with fewer components are more likely to be reliability bottlenecks, thus sorting occurs by using the number of components found in each of the FSs.

Probability based ranking approach. The other approach we consider uses the probability of failure for a given component. Precisely how to obtain the failure for components remains an open question, though we discuss some options earlier at the end of Section 3.2. In this scenario, sorting uses the *relative importance* of each FS in comparison to the top event in the fault tree:

$$I_c = \frac{\Pr(C)}{\Pr(T)} \tag{1}$$

Where I_c is the relative importance of cut set C and T is the top event. The cut set C's probability, $\Pr(C)$, is the probability that all the events in C happen simultaneously. To compute the probability of the top event T, i.e., $\Pr(T)$, we use the inclusion-exclusion rule:

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

Where MC_i means minimal cut set i. The module ranks the FSs by I_c, their relative importance. For the example shown in Figure 6, if we assume the probabilities of events A_1, A_2 and A_3 are 0.2, i.e., $\Pr(A_1) = 0.2$, $\Pr(A_2) = 0.2$ and $\Pr(A_3) = 0.2$, we have the probability of the top event (i.e., the failure probability of the target service) is: $0.2 \cdot 0.2 + 0.2 - 0.2 \cdot 0.2 \cdot 0.2 = 0.232$. The relative importance of minimal cut sets $\{A_2\}$ and $\{A_1, A_3\}$ is: $\frac{0.2}{0.232} = 0.862$ and $\frac{0.04}{0.232} = 0.1724$ respectively. Thus, the cut set $\{A_2\}$ should be located at the top of ranking list.

3.5 Privacy-preserving SRA

In reality, not all clouds own all aspects of their environment. Most providers likely do not have their own Internet backend or power generators, while application providers may own no hardware but only provide a service using existing cloud infrastructures (e.g., EC2). We already have seen evidence that much like cloud consumers taking advantage of various cloud services to produce content for their consumers, so too have cloud services begun to build on other cloud services. While using different clouds gives the consumer an opportunity to improve reliability by subscribing to multiple cloud providers, doing so blindly may cost significantly more and not necessarily result in better reliability. While the SRA could help evaluate these hierarchical clouds, cloud service providers may be hesitant to share their private information with third-parties. All is not lost, a derivate SRA supporting privacy preservation, called Privacy-preserving SRA (P-SRA), can avoid the leakage of cloud provider secrets.

The P-SRA provides the following privacy guarantees: 1) each of cloud providers cannot obtain the dependency information from any of other providers, 2) only consumer will know the FSs of underlying the cloud providers, and 3) the P-SRA provider is not able to obtain any information during the whole process of auditing.

In order to achieve our goal, we use secure multi-party computation (MPC) [53] for privacy protection. Using MPC maintains privacy for all cloud providers involved so long as a majority of them honestly execute their portion of the MPC. Specifically, all parties involved including the consumer (or application service provider), infrastructure service providers, and the P-SRA provider perform multi-round communications and local computations interactively. Intermediary data remains encrypted with only the output being decrypted by the consumer.

Because P-SRA reveals to the consumer the FSs shared by the different clouds, cloud service providers must carefully select how to represent their components. If done injudiciously, common failure events may be overlooked or worse it may reveal potentially private information to the P-SRA consumer. A consumer

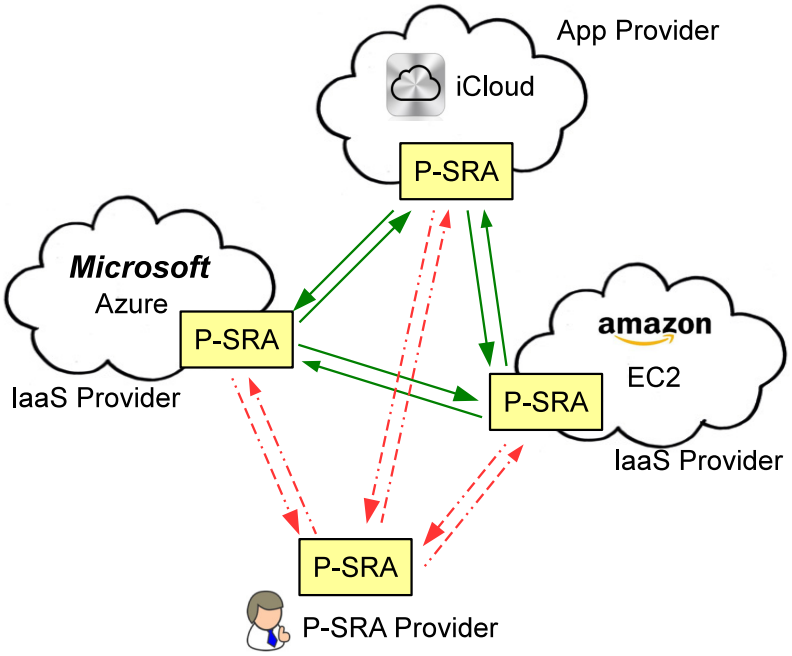

Figure 8: Auditing process of the P-SRA based on multi-party computations.

should be able to determine if two cloud providers share common internal or external resources but at the granularity as to not reveal what those resources are. For example, iCloud, an application service provider, uses Microsoft's Azure and Amazon's EC2 as backup for the reliability; however, the power for them is supplied by the same provider [7]. If a lightning storm or some other event were to cause a power outage, iCloud would be disabled due to lack of redundancies in power.

Figure 8 illustrates privacy preserving auditing process for iCloud's deployment across Microsoft's Azure and Amazon's EC2 with the P-SRA provider. As what we mentioned before, the privacy preserving mechanism the P-SRA uses is MPC. In this environment, each party hosts a P-SRA locally with each of the cloud providers *first* establishing a Shamir secret share [43] for the purpose of securely encrypting and sharing dependency information about the infrastructures provided by third parties. *Then*, the cloud providers deliver encrypted the dependencies to other cloud providers and notifies the P-SRA provider upon completing this process. Once all clouds have completed this process, the P-SRA provider broadcasts a *secure multi-party calculation script* to each of cloud providers. *Finally*, the P-SRA running on each cloud provider executes the script, generates a result, and sends that to the application service provider, in this case iCloud revealing the auditing results, i.e., a ranking list on FSs across multiple cloud providers. The secure multi-party calculation script of the P-SRA performs the following operations: 1) securely combining the private dependencies from various providers, 2) from the combined results generating the shares of the fault tree 3) performs either the minimum cut set or the failure sampling algorithm on the results to obtain shares of FSs, and 4) ranks and opens the results of shares.

3.6 Limitations

SRA depends on both the level of detail and accuracy of the input data in order to successfully find important FS. A majority of failures occur due correlated failures involving software [14] and therefore an area of particular interest is in more thorough collection of dependency information from distributed systems.

Important data might lie outside the provider's control, such as, fibre-optic and power cables as well network and power loads. Capturing this data is critical to discovering unknown FS, for example, the Baltimore tunnel fire in 2001 [40] had a significant effect on Internet traffic due to wide-spread use of the passage way for laying fibre-optic cables.

4 Implementation and evaluation

This section introduces our SRA and P-SRA prototypes and then uses the SRA prototype to evaluate the following two challenges: can the SRA discover potential reliability issues before they occur and 2) the SRA's performance.

4.1 SRA prototype

Following from the discussion in Section 3, we constructed a proof-of-concept SRA implementation in order to demonstrate the feasibility of our approach. We have implemented all three basic modules: dependency acquisition, fault tree generator, and auditing/ranking. The source code can be found at: `https://github.com/ennanzhai/CRR`.

The dependency acquisition modules were written in Python. The network dependency collector makes use of the SNMPv2 library support from NetSNMP in order to collect network information. This method has one caveat: all switches must have LLDP enabled. Our hardware collector uses lshw, a light weight tool that extracts detailed hardware configuration from the local machine. The software collector made use of ps to determine the location of deployment for the different services and gprof to acquire call-dependency information.

The fault tree generator module was also written in Python using the NetworkX library [12] to operate on tree/graph structures. We implemented the templates as XML files, which the graph generator uses to query the dependency database to produce a fault tree.

Our auditing module implements both minimal cut set and failure sampling algorithms as well as the two ranking metrics described in Section 3 into the prototype.

4.2 P-SRA prototype

Our initial version of the P-SRA uses SecreC [15], a C-like secure multi-party computation language. The share of each cloud provider was split and stored in a secret XML file belonging to the P-SRA hosted by that cloud provider. Each provider's P-SRA only can perform secure, local computations and open its own results. SecreC's specifications guarantee security.

4.3 Using SRA

In this case study, Alice, an cloud service administrator has deployed MapReduce within her cloud as depicted as a topology map in Figure 9. For robustness, Alice deploys her MapReduce Master on S5 with

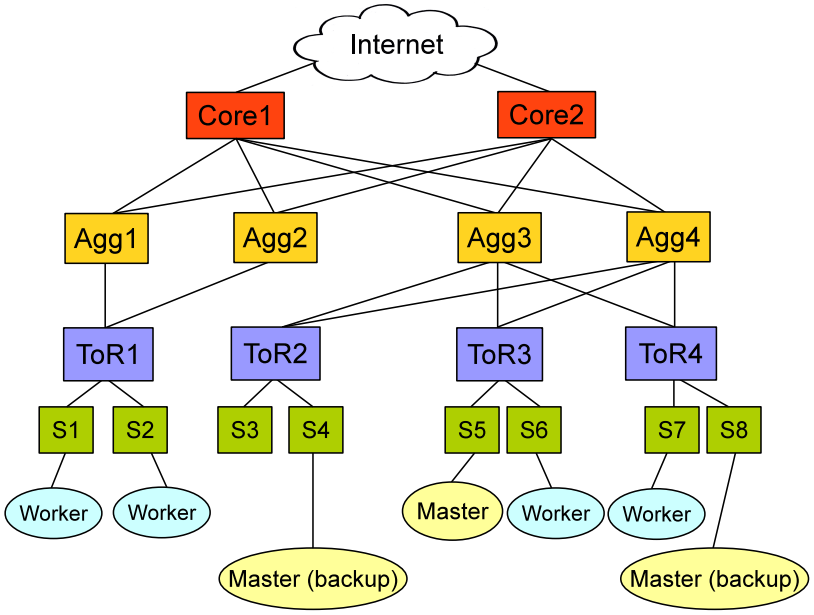

Figure 9: A sample topology of a cloud for the case study illustrating the effectiveness of the SRA. Core, Agg and ToR means core router, aggregation switch and top of rack switch respectively.

Table 1: Ranking list on FSs by size based ranking metric.

NO.	Hidden FSs
1	Agg3, Agg4
2	Core1, Core2
3	ToR2, ToR3, ToR4
4	S4, S5, S8

backups on two independent servers, S4 and S8. She runs Workers on the remaining servers. During her deployment, Alice only considers the need for independent servers for reliability, unfortunately her choices of servers for the Master result in a weakness in her deployment as the entire service will become unavailable if Agg3 and Agg4 fail concurrently. Alice may be completely unaware of this issue, because she thought it was reasonable enough to assign her Masters to different racks. If she had realized the relationship between the aggregation switches and the racks and that earlier measurement results [5, 30] revealed aggregation switches' failure probabilities tend to be higher than other network components, her deployment decisions may have been improved.

SRA can rescue Alice by informing her of this situation before an issue ever occurred. Alice begins by executing SRA's dependency acquisition module accumulating the data present in the dependency graph depicted in Figure 9. The SRA uses this information to produce a fault tree for Alice's environment with the goal to ensure that the MapReduce framework remains online. Finally, the SRA processes the fault tree using the minimal cut set algorithm and sorts the results by using our size based ranking metric. Table 1 presents the top 4 FSs. Agg3 and Agg4 produce a potentially high risk FS and Alice can view the dependency graph in Figure 9 to determine that her software configuration could be improved by moving one of the Masters to either S1 or S2 or alternatively she could improve network robustness by connecting ToR2 with Agg1

Table 2: Configurations of our simulated data sets.

	Case 1	Case 2	Case 3	Case 4	Case 5
# of switch ports	4	8	16	24	48
# of core routers	4	16	64	144	576
# of agg switches	8	32	128	288	1152
# of ToR switches	8	32	128	288	1152
# of servers	16	128	1024	3456	13824
Total # of components	40	216	1360	4200	16752

and Agg2. Future versions of the SRA could potentially incorporate a mechanism that would assist Alice in determining reasonable software configurations based upon reliability versus performance trade-offs.

4.4 Large-scale simulations

In this evaluation, we explore the performance/accuracy trade-offs between the SRA's two methods, minimal cut set algorithm and the failure sampling algorithm, for determining FSs given a fault tree. The minimal cut set algorithm produces a complete set of all FSs but does so within NP-hard complexity, while the failure sampling algorithm generates various inputs for basic events to produce cut sets with cost in polynomial time. We compare the run-time of the minimal cut set algorithm to the failure sampling algorithm and the coverage of the failure sampling algorithm using different sampling rounds. The analysis consists of various sized topologies and the number of samples performed (10^3 through 10^7) for the failure sampling algorithm.

Because we are unable to obtain real cloud data sets due to privacy concerns, we produce simulated data sets at various scales up to as large as what we expect to find in real clouds using the widely accepted three-stage fat tree model [39]. Our simulated data sets include servers, top of rack (ToR) switches, aggregation switches, and core routers, and the specifics are listed in Table 2. All the simulations are run on a server which is equipped with two 2.8GHz 4-Core Intel Xeon CPUs and 16GB of memory.

Our evaluation for total run-time of performing audits can be found in Figure 10. The second half of our evaluation focuses on accuracy, Figure 11. Using the minimal cut set algorithm as a baseline, we were able to obtain all the minimal cut sets for a given fault tree. We compare the effectiveness of the failure sampling algorithm with different sampling rounds by computing the coverage of the failure sampling algorithm against that of the result baseline provided by the minimal cut set algorithm.

During analysis of Case 4 and 5, the minimal cut set algorithm took a significantly long period of time to complete, therefore, we exclude these results from our evaluation. While the minimal cut set algorithm produces guaranteed results, we can see that given the right sampling rate, the failure sampling algorithm produces nearly as useful results in one-fourth, Case 3 10^6 samples to minimal cut set algorithm.

In the current form, one may argue that the failure sampling approach may not be entirely useful if we are going for complete coverage. One instance in which it would be more useful is if the cloud were to be running the SRA frequently, for example, if clouds offered a service for customers that would help the cloud determine the best layout for a user's deployment in the cloud.

4.5 Initial evaluation on the P-SRA

In order to evaluate the initial version of P-SRA, we generated a data set simulating the iCloud scenario mentioned in Section 3.5 where the P-SRA audits an application cloud service depending on multiple cloud

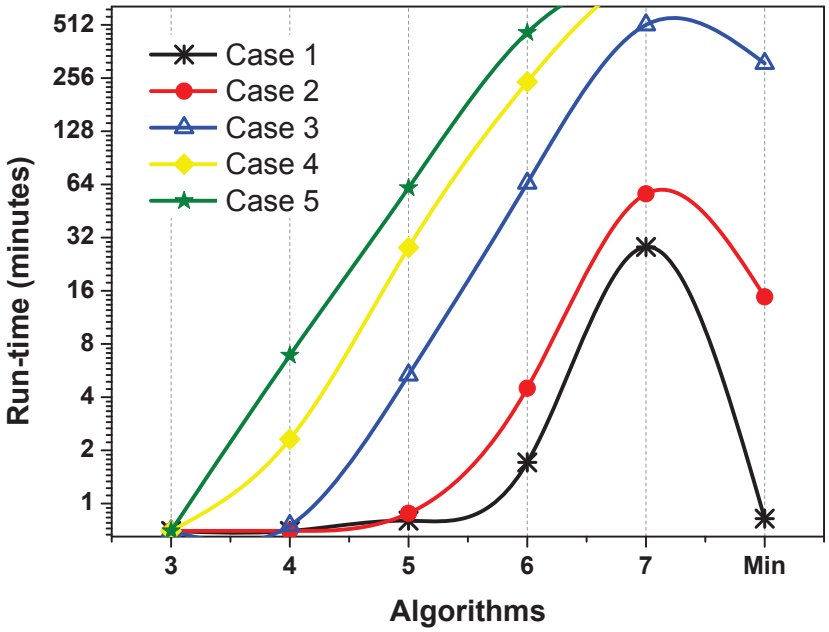

Figure 10: Running time comparing different fault tree analysis algorithms. $3-7$ located at X-axis are used to denote failure sampling algorithm at various powers of 10. Min represents the minimal cut set algorithm.

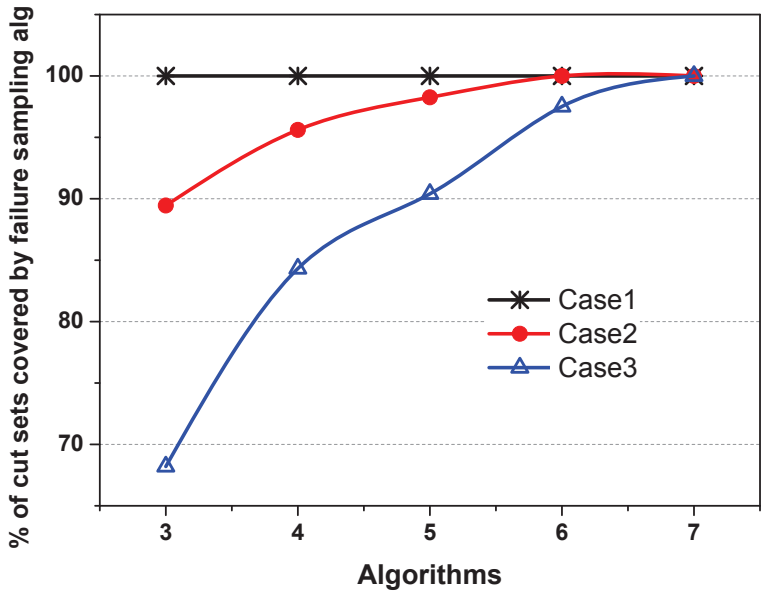

Figure 11: Comparison of the failure sampling algorithm with different sampling rounds. $3-7$ located at X-axis are used to denote failure sampling algorithm at various powers of 10. Min represents the minimal cut set algorithm.

providers. In this data set, the application service uses two different IaaS cloud providers with each IaaS having 5 data centers and a total of 4 power supplies and 6 Internet routers. The IaaS share one power supply and two Internet routers. Using P-SRA the minimal cut set algorithm completed in 14.6 minutes. Given that this data set consists of only 20 nodes, we can see that the performance of the P-SRA prototype is far too slow to be practical most likely due to the use of MPC.

5 More applications of SRA

In this section we discuss some potential applications of the SRA that fit outside the scope of what we have discussed thus far.

Complementing diagnosis systems. Diagnosis systems attempt to determine the root-cause of a fault upon detecting an error. Most of the current diagnosis systems infer and localize the root-cause of a fault through traversing the entire dependency graph resulting in long delays in fault resolution. A cloud operator can use the SRA's auditing report ranked list as a road-map in isolating faults more quickly.

Recommending cloud services. If there exists a trusted third-party, the SRA can be used to provide cloud service recommendations. In this model, a cloud consumer could post a job requirement and the trusted third-party would analyze sample deployments in various clouds recommending to the consumer the most reasonable deployment. Alternatively, the P-SRA should be able to perform the same operation but at significant cost and perhaps less clarity about the final result.

Making the cloud insurable. While not necessarily an application of the SRA, the SRA makes understanding the risks associated with using a cloud more pronounced. An insurance provider could use the SRA as a means to determine more effective rates for cloud insurance and potentially ignore clouds that have too much associated risk.

6 Related Work

Providing audits for clouds has been suggested as a means to achieve reliability [41]; however, general and efficient cloud auditing remains an open question.

As the earliest advocators, Shah et al. [41] present the definition and classification of cloud reliability auditing and propose a general auditing scheme to ensure the reliability of cloud storage services. They classify cloud auditing techniques into two groups: *internal auditing* and *external auditing*. The former checks the internal structure and processes of a cloud provider to assess the likelihood the provider will fail to meet its SLAs, while the latter makes use of samples from third-parties without internal knowledge. While there exist much work on external cloud reliability efforts [42, 46–49, 52], to the best of our knowledge, the SRA is the first effort at providing internal reliability audits for the cloud.

Diagnosis, unlike auditing, attempts to discover problems after they occur; however, often times the techniques used in one can be applied to the other. Our work follows more similarly with inference-based diagnosis [18–20,24,25,27,35,36,36,37] which obtain dependency graphs of a cloud when a problem occurs. Different from existing diagnostic systems, NetPilot [50] aims to mitigate the failures rather localizing their sources.

Accountability techniques differ from auditing approaches as accountability attempts to place blame after a failure, while auditing can be used to prevent failures in the first place. Haeberlen et al. [32] propose using third-party verifiable evidence to determine whether the cloud consumer or provider should be responsible if a problem occurs. Similarly the accountable virtual machines [33] (AVM) enables users to

monitor software executing on a remote virtual machine. Upon detecting a fault, AVM produces produces third-party verifiable evidence to determine whether or not the virtual machine is correct.

FTCloud [54] and FTM [34] focus on fault tolerance in clouds. The former proposes a ranking metric for the components with a given cloud application, thus enhancing the fault tolerance capability of the cloud. The later one introduces fault tree model for providing fault tolerance to the clouds. Compared with these two efforts, SRA employs fault tree analysis in combination with a ranking metric and SRA has been shown to be feasible through prototyping and evaluation.

7 Conclusion

In this paper, we have presented a cloud structural reliability auditing system named SRA that uniquely targets the important problem of resolving failures before they occur by first discovering a clouds' components and their dependencies and then using fault tree analysis to determine and evaluate the impact of FSs. Beyond this use, SRA's techniques may be useful for improving diagnosis and recommendation systems as well as making clouds more easily insurable. However, what we present is only the beginning of the process in this area, there remains significant work to be done that would make system's like this even more useful, namely, improved methods for obtaining component information and ensuring that the detail covered is sufficient as well as developing more efficient privacy-preserving mechanisms.

Acknowledgements

We thank Jeff Mogul, Ruichuan Chen, James Aspnes and Gustavo Alonso for their helpful comments. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1149936.

References

[1] Amazon EBS. `http://aws.amazon.com/ebs/`.

[2] Amazon EC2 web services. `http://aws.amazon.com/ec2/`.

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

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

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

[6] EC2 Outages on Christmas Day of 2012. `http://my.chicagotribune.com/#section/-1/article/p2p-73813597/`.

[7] iCloud service. `http://www.theregister.co.uk/2011/09/02/icloud_runs_on_microsoft_azure_and_amazon/`.

[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] Microsoft Azure. http://www.windowsazure.com/.

[10] Netflix. https://signup.netflix.com/.

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

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

[13] Survey: Cloud computing 'No Hype', but fear of security and control slowing adoption. http://www.circleid.com/posts/20090226_cloud_computing_hype_security.

[14] The 10 Biggest Cloud Outages of 2011. http://www.crn.com/slide-shows/cloud/231000954/the-10-biggest-cloud-outages-of-2011-so-far.html/.

[15] The SecreC Language. http://sharemind.cyber.ee/the-secrec-language.

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

[17] Hussam Abu-Libdeh, Paolo Costa, Antony I. T. Rowstron, Greg O'Shea, and Austin Donnelly. Symbiotic Routing in Future Data Centers. In *ACM SIGCOMM*, pages 51–62, 2010.

[18] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Reveiw*, pages 74–89, 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*, pages 13–24, 2007.

[20] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 259–272, 2004.

[21] Tim Bedford and Roger Cooke. *Probabilistic Risk Analysis: Foundations and Methods*. Cambridge University Press, 2001.

[22] 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)*, pages 31–46, 2011.

[23] Jon Brodkin. Hurricane Sandy takes data centers offline with flooding, power outages, 10 2012. http://arstechnica.com/information-technology/2012/10/hurricane_sandy_takes_data_centers_offline_with_flooding_power_outages/.

[24] Mike Y. Chen, Anthony Accardi, Emre Kiciman, David A. Patterson, Armando Fox, and Eric A. Brewer. Path-based failure and evolution management. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, pages 309–322, 2004.

[25] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 117–130, 2008.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *ACM Symposium on Operating System Principles (SOSP)*, pages 205–220, 2007.

[27] John Dunagan, Nicholas J. A. Harvey, Michael B. Jones, Dejan Kostic, Marvin Theimer, and Alec Wolman. Fuse: Lightweight guaranteed distributed failure notification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 151–166, 2004.

[28] Bryan Ford. Icebergs in the Clouds: the Other Risks of Cloud Computing. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[29] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 61–74, 2010.

[30] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *ACM SIGCOMM*, pages 350–361, 2011.

[31] 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*, pages 75–86, 2008.

[32] Andreas Haeberlen. A Case for the Accountable Cloud. In *Proceedings of the 3rd ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware (LADIS'09)*, oct 2009.

[33] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 119–134, 2010.

[34] Ravi Jhawar, Vincenzo Piuri, and Marco D. Santambrogio. Fault tolerance management in cloud computing: A system-level perspective. *IEEE Systems Journal*, 7(2):288–297, 2013.

[35] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. Shrink: A Tool for Failure Diagnosis in IP Networks. In *ACM SIGCOMM Workshop on Mining Network Data (MineNet)*, pages 173–178, 2005.

[36] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed Diagnosis in Enterprise Networks. In *ACM SIGCOMM*, pages 243–254, 2009.

[37] Ramana Rao Kompella, Jennifer Yates, Albert G. Greenberg, and Alex C. Snoeren. IP Fault Localization Via Risk Modeling. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[38] Jeffrey C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 293–304, 2006.

[39] 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*, pages 39–50, 2009.

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

[41] Mehul A. Shah, Mary Baker, Jeffrey C. Mogul, and Ram Swaminathan. Auditing to Keep Online Storage Services Honest. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

[42] Mehul A. Shah, Ram Swaminathan, and Mary Baker. Privacy-Preserving Audit and Extraction of Digital Contents. *IACR Cryptology ePrint Archive*, 2008:186, 2008.

[43] Adi Shamir. How to Share a Secret. *Communcations of ACM*, 22(11):612–613, 1979.

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

[45] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, January 1981.

[46] 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.

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

[48] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing. In *IEEE INFOCOM*, pages 525–533, 2010.

[49] 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 Distributed Systems*, 22(5):847–859, 2011.

[50] 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*, pages 419–430, 2012.

[51] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks. In *IEEE INFOCOM*, pages 2170–2183, 2005.

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

[53] Andrew Chi-Chih Yao. Protocols for Secure Computations (Extended Abstract). In *IEEE FOCS*, pages 160–164, 1982.

[54] Zibin Zheng, Tom Chao Zhou, Michael R. Lyu, and Irwin King. FTCloud: A Component Ranking Framework for Fault-Tolerant Cloud Applications. In *IEEE International Symposium on Software Reliability Engieering (ISSRE)*, 2010.

[55] Ming Zhong, Kai Shen, and Joel I. Seiferas. Replication Degree Customization for High Availability. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 55–68, 2008.

A Failure sampling algorithm analysis

In this appendix, we present the analysis of our proposed failure sampling algorithm.

A.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.

A.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 $\mathscr{P} = \mathscr{NP}$. 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 $1 \le i \le n$ and $\bigcup_{1\le i\le n} s_i = U$. 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 $\mathscr{NP}$-hard.

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

Proof. 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 $\mathscr{NP}$-hard, the MSA problem is also $\mathscr{NP}$-hard. □

A.3 Target Cover

A.3.1 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. 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. By Markov's inequality, the probability that more than m uniform samples are need to cover the target is

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

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

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

□

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.

A.3.2 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. 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 $1 \leq i \leq t$. Let $p = \sum_{i=1}^{t} p_i$. It follows the definition that for any $1 \leq i \leq t$,

$$q_i \leq \hat{q}_i \leq q'_i \tag{3}$$

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

$$\begin{aligned} \mathbf{E}[X] &= \mathbf{E}\left[\sum_{i=1}^{t} X_i\right] = \sum_{i=1}^{t} \mathbf{E}[X_i] \\ &= \sum_{i=1}^{t} \frac{1}{p - \hat{q}_{i-1}} \end{aligned} \tag{4}$$

Define $q_0 = q'_0 = 0$ for convention. Combining (3) and (4) gives

$$\sum_{i=1}^{t} \frac{1}{p - q_{i-1}} \leq \mathbf{E}[X] \leq \sum_{i=1}^{t} \frac{1}{p - q'_{i-1}} \tag{5}$$

[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.

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

$$p - q_{i-1} = q'_{t-i+1} \tag{6}$$

Finally, combining (5) and (6) gives

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

□

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 $24.49 \le \mathbf{E}[X] \le 44.35$.

A.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. 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 $1 \le i \le d$,

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

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 $2^5 \cdot (H_{10} - H_7) \approx 10.76$.