Towards Proactive Auditing of Cascading Vulnerabilities across Inter-Cloud Replications

Ennan Zhai, *Yale University*
Zhenhua Li, *Tsinghua University*
Jiang Ming, *UT at Arlington*
William Dower, *IBM*
Cameron Yick, *Enigma Technologies*

Abstract

Today's cloud customers intend to replicate important states across multiple cloud providers to ensure the availability of their data or applications. These seemingly independent cloud providers, nevertheless, may actually rely on common third-party components, *e.g.*, DNS services, open-source software and libraries. We call the vulnerabilities hidden in such shared dependencies as *cascading vulnerabilities*, which have recently been exploited by attackers to launch DDoS and worm attacks resulting in cascading outages across the entire inter-cloud replications. Even worse, cloud providers tend to keep their internal component dependencies a secret due to business concerns, thus making it hard for customers to diagnose the cascading vulnerabilities. This paper presents iAudit, a novel auditing system capable of *proactively* evaluating the independence of inter-cloud replications through a *fine-grained* and *efficient* approach, while preserving the business privacy of audited clouds. With the help of iAudit, a cloud customer can select the most independent replication deployment *before* her data or application adoption, preventing potential cascading outage risks at the best efforts. We have implemented an iAudit prototype that exhibits accurate and efficient auditing capability under a real-world case study and large-scale datasets.

1 Introduction

Cloud outages severely influence their customers. Under the assumption that clouds fail independently, increasingly more customers replicate important states of their data or applications across multiple cloud providers to ensure high availability [1, 8, 64]. iCloud, for example, rents Microsoft Azure and Amazon S3 for redundancy [25], while a recent survey [35] reports 73% of current applications have employed inter-cloud replications.

These seemingly independent cloud providers, nevertheless, may share third-party dependencies, such as DNS services, switches, open-source software and libraries [41, 72]. Attackers are able to *remotely* exploit the vulnerabilities in these common dependencies, and then launch DDoS and worm attacks [38, 42] to cause cascading outages across the entire inter-cloud replications, undermining customers' availability enhancement efforts [16, 42, 71, 72]. Security specialists, on Lawfare [34], have pointed:

> *"An insidious source of common-mode failures is a vulnerability that causes redundant software processes to fail under identical conditions" [20]*

In the widely-reported DDoS attack on Oct. 21, 2016 [52], for example, many cloud providers became unavailable during the same period of time, because they use the same DNS service, Dyn, whose vulnerability was compromised by DDoS attackers. Many customers were severely affected in this event, because their data replicas on multi-cloud providers became unaccessible simultaneously [31]. Besides DNS services, vulnerabilities in switches provided by the same vendors (*e.g.*, Cisco) may also be exploited by remote attackers to cause cascading outages across multiple cloud providers [15, 19].

Similar to network dependencies, vulnerable third-party software components and libraries shared by different cloud providers also become the sources of cascading outages [42]. For example, vulnerability CVE-2016-0737 [14] can lead to critical availability issues in OpenStack Swift, and many cloud providers, *e.g.*, Oracle and IBM, were affected, because their services heavily rely on OpenStack Swift. Despite publicly known risks, these vulnerable third-party software components and libraries, unfortunately, widely exist in many systems, because patching or upgrading these components would lead to many system compatibility and stability issues [2, 33, 43]. For example, Kula *et al.* observed that 81.5% of prevalent third-party systems still keep their outdated vulnerable dependencies [33]. In addition, Pham *et al.* revealed that software reuse is another important reason making providers share a large number of vulnerabilities [49].

Learning from the above evidence, vulnerabilities shared by cloud providers have become the perfect points of invalidating inter-cloud replication efforts [18, 31]. We define *cascading vulnerabilities* as the vulnerabilities in the infrastructure (both network and software) components shared by different cloud providers.

To localize vulnerabilities, many post-failure forensics, *e.g.*, diagnosis systems [7, 11, 30, 36] and accountability systems [22–24], have been proposed. Even so, once outages occur, cloud customers would be put at a loss until one of the cloud providers becomes available, which requires prolonged recovery time and significant human intervention [66]. *Worse*, these post-failure forensics are hardly used by cloud customers or providers to localize

cascading vulnerabilities across cloud providers [16, 72], since few providers are willing to share their network and software dependency information, for the business concerns. The cloud community, therefore, have been in a dilemma when dealing with cascading vulnerabilities.

To address the above dilemma, we propose a *proactive* auditing system, iAudit, that *privately* evaluates the independence of each alternative inter-cloud replication for customers and returns the customers a list ranking those replications based on the independence. Thus, the customers can select the most independent inter-cloud replications before their data or application adoptions, preventing "severe damages" resulting from cascading vulnerabilities at the early stage.

"While preventing disease is the best, it requires considering much larger problem spaces than reactively diagnosing and curing disease [65]". Similarly, building a practical proactive auditing system requires our solution to be both *fine-grained* and *efficient*, because we try to search and assess cascading vulnerabilities in the entire underlying structures of audited clouds before outages occur. This problem is more challenging than post-failure forensics, because the latter case only needs to localize the root causes specific to observed outage events.

To this end, we enable iAudit to simultaneously achieve fine-grained and efficient auditing. The insight of iAudit design is to reduce the privacy-preserving fine-grained independence auditing to a *private attack tree similarity* problem—less similarity indicating better independence. In particular, for an alternative inter-cloud replication R, iAudit splits a private attack tree similarity computation into the following two phases:

- **Local risk group dataset generation.** Each involved cloud provider j locally generates an attack tree [57], which is a model representing system components and their dependencies as a Directed Acyclic Graph (DAG) with logical gates, and then analyzes the generated attack tree to produce a local dataset D_j recording the top-k critical *risk groups* and their severities. A risk group is a set of vulnerabilities that disable a cloud if they are triggered simultaneously. This step is challenging, since existing attack tree analysis algorithms [28,51,68] are poorly scalable to cloud-scale systems containing thousands of components [58]. Inspired by scalable program verification techniques [26, 27], we design a scalable attack tree analysis algorithm that transforms an attack tree into a weighted Boolean formula and then solves this formula by applying a high-performance weighted partial MaxSAT (WP-MaxSAT) solver [5].
- **Private set-similarity computation.** All the involved providers take their own D_j as *sensitive* inputs to collaboratively perform a private set-similarity protocol to output a value quantifying R's independence; however, making such a computation practical is not straightforward. Because vulnerabilities have different impacts and exploitability, risk groups (consisting of different vulnerabilities) in D_j have different severity values. This phase, therefore, requires our protocol computing set similarity by taking into account the weights of different elements. Nevertheless, existing private set-similarity protocols *either* focus on efficiency while failing to distinguish elements by their weights [9, 72], *or* take elements' weights into account in computationally expensive ways [67, 70, 71]. To this end, we construct the first efficient Private Weighted Jaccard Similarity protocol (named PW-JS) by first constructing a weighted MinHash primitive inspired by the weighted sampling [12], and then combining this primitive with a private set intersection cardinality protocol [32, 56].

We have implemented and evaluated iAudit. Our case study shows iAudit successfully assists the customer avoid cascading vulnerability risks (§6.2). Furthermore, our results show that iAudit is applicable to larger-scale industrial scenarios due to its sound performance (§6.3). For example, iAudit only needs 9 minutes to complete a fine-grained auditing to a four-way inter-cloud replication, where each cloud contains 70,656 components.

Besides benefiting customers, the results of iAudit (*e.g.*, local risk group dataset) can guide cloud providers to better understand local vulnerabilities. Furthermore, participating in the iAudit community enables cloud providers to have "certified labels", attracting more customers. We, therefore, believe iAudit represents one practical step towards deploying *truly robust* inter-cloud replications.

In summary, this paper makes four contributions:

- iAudit is the first proactive system capable of offering *both* efficient and fine-grained independence auditing to inter-cloud replications, while preserving the privacy of cloud providers.
- Our WP-MaxSAT solver-based approach offers an efficient solution that extracts the top-k risk groups from attack trees modeling cloud-scale systems (§4.2).
- We construct the first efficient private weighted Jaccard similarity protocol to facilitate the computation of independence of inter-cloud replications (§4.3).
- We implement an iAudit prototype and evaluate it with large-scale datasets and a real-world case study (§6).

2 Overview

This section describes iAudit's motivation (§2.1), scenario (§2.2), threat model (§2.3), and goals (§2.4).

2.1 Motivating Example

As shown in Figure 1, a cloud customer wants to select two out of three potential cloud providers, A, B and C, to deploy a replication for her application. Thus, *al-*

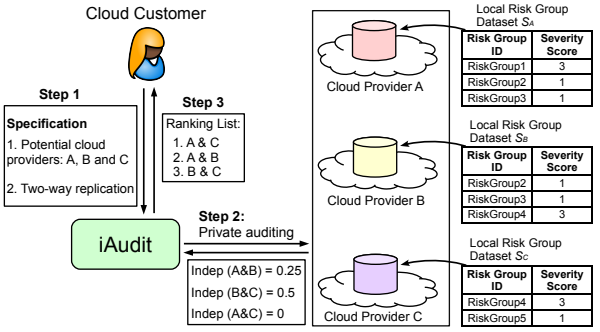


Figure 1: Motivating example. There are three parties in this scenario: cloud customer, iAudit, and potential cloud providers. Each cloud provider locally maintains a sensitive dataset, named *risk group dataset*. No cloud provider is wiling to share such a dataset.

ternative replication deployments, in this example, are: $\{A\&B\}$, $\{B\&C\}$, and $\{A\&C\}$. Due to non-transparency, this customer does not know which two providers she should choose to avoid potential cascading vulnerability risks. *The goal of our system is to assist this customer by privately auditing the independence of alternative inter-cloud replications deployments before adoption.*

Step 1: This customer sends iAudit her specification, including potential cloud providers, and the number of cloud providers she wants to use for the replications. In Figure 1 example, the former one is Cloud A, B and C, while the latter one is two.

Step 2: With such a specification, iAudit audits the independence of all the alternative replication deployments, *i.e.*, $\{A\&B\}$, $\{B\&C\}$, and $\{A\&C\}$, while preserving their privacy. Preserving the privacy means iAudit, cloud customer and providers should not learn any information of other cloud providers, including software and network components, and their dependencies. As shown in Figure 1, by the end of this step, iAudit only learns a score quantifying the independence of each alternative inter-cloud replication: `indep(A & B)=0.25`, `indep(B & C)=0.5`, and `indep(A & C)=0`—lower score indicating better independence. See §4 for more details in Step 2.

Step 3: iAudit generates a list ordering all the alternative inter-cloud replications based on their independence scores (received in Step 2), and then sends the customer this ranking list (without independence scores). With such a list, the customer can learn which two providers she needs to rent to adopt her replication. In Figure 1 example, the customer should rent cloud providers *A* and *C*, because this alternative replication deployment is at the top of the ranking list, although she does not know concrete independence scores.

2.2 Target Scenario and Entities

As shown in Figure 1, our scenario consists of three types of entities: cloud customers, cloud providers, and iAudit.

A cloud customer deploys her data or application across multiple cloud providers to enhance availability. She aims to select at best-effort independent inter-cloud replication, avoiding cascading vulnerabilities. A cloud customer, in practice, might be a large organization (*e.g.*, iCloud, Netflix, and Zynga), or an individual.

Cloud providers, *e.g.*, Amazon S3 and Microsoft Azure, host physical infrastructure and software services. In practice, these providers rely on third-party components, *e.g.*, DNS services, switches, software components and libraries, which may contain cascading vulnerabilities. We wish to emphasize that vulnerable third-party components widely exist in prevalent cloud systems, because 1) upgrading these components may introduce more critical system compatibility and stability problems [43], 2) fixing these vulnerabilities typically needs significant efforts [63], and 3) the providers of these vulnerable components may never tried to patch potential vulnerabilities.

iAudit fills the gap between cloud customers and providers. A cloud customer expresses her specifications to the iAudit who in turn sends the customers a list ranking alternative inter-cloud replications based on their independence. In practice, iAudit should be maintained by a third-party auditing company, separately from cloud customers and providers [53].

2.3 Threat Model and Trust Assumptions

Cloud customers are potentially malicious, *i.e.*, dishonestly following our protocol, and wish to learn as much as possible about the providers' secret. Cloud customers may collude, but they cannot collude with iAudit.

Cloud providers are honest-but-curious: they try to learn whatever sensitive information they can, but they will not deviate from our protocol. This assumption is reasonable in practice, because accepting and collaborating with external auditing have been the compliance of today's cloud providers [53]. We also offer solutions to detecting malicious cloud providers (in §5). Cloud providers will not collude with iAudit.

iAudit is also honest-but-curious, which means it faithfully follows our designed protocol. Nevertheless, iAudit tries to learn as much as possible about cloud customers' and cloud providers' sensitive information, *e.g.*, components and dependencies.

We wish to emphasize that our honest-but-curious assumption on both cloud providers and the iAudit is appropriate in the realistic scenario, because 1) truly malicious party in our context should be network attackers who want to invalidate customers' replication efforts, and customers, cloud providers and iAudit should work together (due to commercial benefits) to defend against these attackers, and 2) third-party auditing has been a widely-accepted business and no cloud provider or auditor violates it due to the business contract [3].

Finally, we assume that all the cryptographic operations are operated correctly and computationally secure.

2.4 Goals and Non-Goals

System goal. The most important goal of iAudit is to enable cloud customers to select the most independent inter-cloud replications from alternative deployments, thus minimizing potential cascading vulnerability risks for their data or application adoptions.

Privacy goals. For cloud customers, they should only learn the ranking of alternative inter-cloud replication deployments (as shown in Figure 1). They should not learn any privacy information of cloud providers.

iAudit is allowed to learn independence score of each of alternative inter-cloud replication deployments (as shown in Figure 1), but, the same as customers, iAudit should not learn any privacy information of audited cloud providers.

For each of cloud providers, it knows its own sensitive information, and the independence scores of replications it involves. A cloud provider should not learn or infer any business privacy information of other cloud providers.

Non-goals. First, iAudit relies on existing daily-updated vulnerability databases (detailed in §4.2) and thus iAudit is not responsible for discovering or detecting any new vulnerabilities. Second, iAudit is not a tool used to repair or patch vulnerabilities. Finally, iAudit cannot be used as a tool to defend against DDoS or worm attacks. In fact, many defenses (*e.g.*, server provisioning, selective traffic blocking and proof-of-work challenges) have been developed to deal with the DDoS and work attacks.

3 Technical Building Blocks

This section describes techniques that we will utilize throughout the iAudit design.

3.1 Attack Trees and Risk Groups

An attack tree [57] is an abstraction of the details of potential availability risks in any given system. The attack tree focuses on vulnerabilities causing availability impact on a specific service. Attack tree is an appropriate technique in our context because: 1) the architectures of many cloud services could be represented as layered Directed Acyclic Graphs (DAGs) [16], and 2) the potential dependencies of components within cloud services are naturally modeled by the logic gates (defined later) of attack trees [67].

Figure 2(a) shows a simple data access service example, and Figure 2(b) shows an attack tree modeling the potential risks against the availability of this service. Note that what the attack tree in Figure 2(b) focuses on are the software and application services running on the network devices and servers shown in Figure 2(a). For example, Agg1 switch system and certificate system shown in Figure 2(b) represent the embedded switch software running

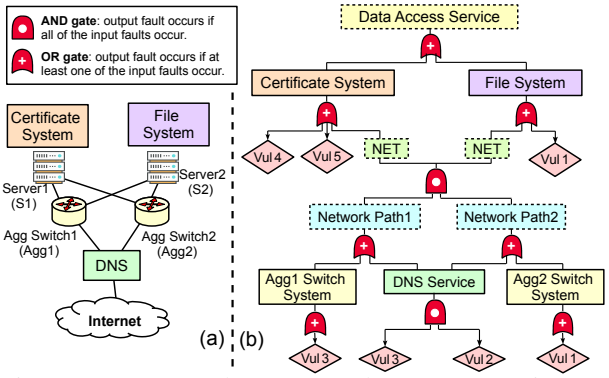


Figure 2: An example customer data access service (a) and its corresponding attack tree (b). In (b), diamond-shaped nodes, rectangle nodes and gate nodes denote vulnerabilities, components and logic gates, respectively. Dashed nodes, *e.g.*, NET, denote logical components which do not exist in the physical topology.

on physical Agg1 switch and certificate software component running on server 1 in Figure 2(a), respectively. In addition, dashed nodes in Figure 2(b) (*e.g.*, Network and Data Access Service) denote logical components, which do not exist in the physical topology. In particular, an attack tree has three types of nodes: *vulnerability nodes*, *component nodes* and *logic gates*.

All the component nodes in an attack tree are rectangle. A component node may denote a software component (*e.g.*, file system), network component (*e.g.*, switch) or a logical component (*e.g.*, network path).

A *vulnerability node* (*i.e.*, diamond-shaped node in Figure 2(b)) represents one potential risk leading to the availability impact on its higher-layer node. In the attack tree analysis, if a vulnerability node is compromised (or not), it outputs a 1 (or 0) to its higher-layer nodes. For example, in Figure 2(b), Vul2 and Vul4 mean vulnerabilities in DNS service and certificate system, respectively. All the vulnerability nodes must be leaf nodes, but in some case component nodes may also be leaf nodes, if the components do not have any vulnerability.

The attack tree has two types of logic gates, *AND* and *OR*, used to depict different logic relationships among components. For an OR gate, if at least one of its child nodes is assigned 1, its parent node is assigned 1. For an AND gate, only if all of its child nodes are affected (*i.e.*, 1), the gate propagates an impact upwards.

Risk groups. An *risk group* within an attack tree is a set of vulnerabilities with the property that if all of them are triggered simultaneously, then the availability of root node (*e.g.*, the service node in Figure 2(b)) is affected. For example, in Figure 2(b), if Vul2 and Vul3 are triggered simultaneously, the top data access service will be unavailable. In addition, {Vul1}, {Vul4}, {Vul5} and {Vul3, Vul4} are also risk groups. In principle, each risk group is a wanted set containing potential cascading vul-

nerabilities. Extracting risk groups from an attack tree is an NP-hard problem [73].

The severity of a risk group. For a given risk group P containing n vulnerabilities V_i, $P = \{V_i\}_{i=1}^{n}$, and its severity is computed by:

$$Severity(P) = \lceil \sum_{i=1}^{n} (Exploit(V_i) \cdot Impact(V_i)) \rceil, \quad (1)$$

where $Severity(P)$ is the severity value (or called weight) of the risk group P. $Exploit(V_i)$ and $Impact(V_i)$ denote the exploitability (*i.e.*, the likelihood of being triggered) and the availability impact of vulnerability V_i, respectively. Both values of $Exploit(V_i)$ and $Impact(V_i)$ could be obtained from existing vulnerability scoring systems (*e.g.*, CVSS [44]). Thus, we define the top-k critical risk groups as k risk groups with the highest severity scores (*i.e.*, $Severity(P)$ in equation (1)) in the given attack tree.

3.2 Boolean Formula & WP-MaxSAT

Boolean formula. A Boolean formula consists of many Boolean variables, in the value of either 1 (TRUE) or 0 (FALSE), and Boolean operators (conjunctions and disjunctions). For a given Boolean formula, *e.g.*, $\phi = (A \wedge B) \vee (A \wedge C)$, we assign 1 or 0 to each variable, learning the formula's value (1 or 0). If an assignment makes the value of the given formula ϕ evaluate to 1, we say this assignment is a satisfying assignment for ϕ. For example, an assignment $\{A = 1, B = 1, C = 0\}$ is a satisfying assignment for the formula $\phi = (A \wedge B) \vee (A \wedge C)$.

Weighted partial MaxSAT solver. WP-MaxSAT solver is a high-performance solver variant [5], and its purpose is different from conventional SAT solvers. Given a Boolean formula ϕ with n variables a_1, ..., a_n, and a weight vector, $\{w_i | w_i \geq 0, 1 \leq i \leq n\}$, a WP-MaxSAT solver can *efficiently* output a satisfying assignment to the set of ϕ's variables $a_i \in \{0,1\}^n$ that maximizes the function: $C = \sum_{i=1}^{n} w_i \cdot a_i$.

3.3 MinHash-Based Jaccard Similarity

We now describe a decentralized MinHash-based Jaccard similarity protocol without preserving privacy of any party. Jaccard similarity [29] is a metric widely used to measure the similarity between multiple datasets. Each dataset is maintained by an individual party. Conventional Jaccard similarity is defined as $J(S_0, ..., S_{k-1}) = |S_0 \cap ... \cap S_{k-1}| / |S_0 \cup ... \cup S_{k-1}|$, where S_i denotes the dataset maintained by party i. In Figure 1, the Jaccard similarity between A and B is 0.5 without considering weights, because $|A \cap B| / |A \cup B| = 2/4 = 0.5$. A Jaccard similarity J close to 1 indicates high similarity. Datasets with similarity $J \leq 0.15$ are considered uncorrelated [29].

While there are many other similarity metrics, *e.g.*, the Sørensen-Dice index [55], we choose Jaccard similarity because it is efficient and easy to understand.

Computing the Jaccard similarity incurs a complexity linear with the dataset sizes. Thus, an approximation of the Jaccard similarity based on MinHash is proposed [10]. The MinHash [10] extracts a vector $\{h_{min}^{(i)}(S)\}_{i=1}^{m}$ of a dataset S, where $h^{(1)}(\cdot), ..., h^{(m)}(\cdot)$ denote m different random sampling functions, and $h_{min}^{(i)}(S)$ is the item $e \in S$ with the minimum value $h^{(i)}(e)$. Let δ denote the number of datasets satisfying $h_{min}^{(i)}(S_0) = ... = h_{min}^{(i)}(S_{k-1})$. Then, the MinHash-based Jaccard similarity $J(S_0, ..., S_{k-1})$ can be approximated as δ/m. The parameter m correlates to the expected error to the precise Jaccard similarity – a larger m (*i.e.*, more sampling functions) yields a smaller approximation error. Broder [10] proved the expected error should be $O(1/\sqrt{m})$.

4 Design of iAudit

This section first describes iAudit's workflow (§4.1), then details the designs of two important components of iAudit (§4.2 and §4.3), and finally discusses practical issues and the potential solutions (§4.4). We use Figure 1 and Figure 3 as the examples to clarify iAudit's design.

4.1 iAudit's Workflow

We now describe the iAudit's workflow using the scenario in Figure 1. In the following descriptions, we use Step 2.x to denote sub-step within Step 2 in Figure 1.

Step 1: Specification submission. A cloud customer, Alice, plans to deploy her data or application across multiple cloud providers. She contacts iAudit and conveys her specifications: 1) which potential cloud providers she may use, and 2) the number of cloud providers involved in an alternative replication deployment.

Step 2.1: Specification forwarding. Upon receiving Alice's specification, iAudit contacts the cloud providers specified by Alice, and informs them of other potential cloud providers and the number of providers involved in each alternative replication deployment.

Step 2.2: Local risk group dataset generation. With the information received from iAudit, each of Alice's specified cloud providers (say, j) executes a local tool, named RiskGroupGen, to automatically generate a dataset, called *risk group dataset*, which contains the top-k critical risk groups within j's service, and these risk groups' severities. RiskGroupGen is a tool developed and offered by iAudit. Each of cloud providers participating in the iAudit auditing community should install a RiskGroupGen locally (as shown in Figure 3). This step is one of the most important designs of iAudit, since this step makes iAudit capable of offering the fine-grained auditing at the graph level of detail. §4.2 details this step. Although we assume providers are honest-but-curious, we discuss how can iAudit check if cloud providers dishonestly run RiskGroupGen in §4.4.

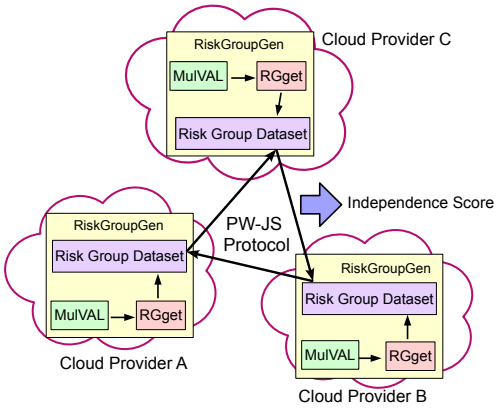


Figure 3: An example for local risk group dataset generation (Step 2.2) and private weighted Jaccard similarity computation (Step 2.3). This example represents an alternative three-cloud replication.

Step 2.3: Private weighted Jaccard similarity computation. After each of Alice's specified cloud providers produces the local risk group dataset, the providers involved in each alternative replication deployment cooperatively perform *private weighted Jaccard similarity* (or PW-JS) protocol to generate an independence score for each alternative replication deployment (detailed in §4.3). To quantify the independence of a replication deployment R, the PW-JS protocol takes as input the risk group dataset of each cloud provider involved in R, and outputs a score quantifying R's independence (as shown in Figure 3).

For example in Figure 1, as one of alternative replications, Cloud Provider A and B run the PW-JS protocol by taking each risk group dataset as the private input. The output is 0.25 (in this example), which is the independence score of replication {A & B}. During the protocol execution process, A and B learn no information on the risk groups or internal network topology of each other.

Step 2.4: Replying results. iAudit collects all the computed independence scores. For example in Figure 1, iAudit receives: `indep(A & B)=0.25`, `indep(B & C)=0.5`, and `indep(A & C)=0`.

Step 3: Auditing results feedback. iAudit uses the independence scores to sort these alternative replication deployments, producing a ranking list, which Alice can use to choose a highly independent set of cloud providers for her replication. During the above workflow, Alice and iAudit learn nothing about the internal dependencies of potential cloud providers.

4.2 Local Risk Group Dataset Generation

We now detail the design of Step 2.2 in the iAudit workflow. In principle, this step enables iAudit to analyze and capture common dependencies (*i.e.*, potential shared risk groups) at the graph level of detail.

As shown in Figure 3, iAudit offers each cloud provider a risk group dataset generation tool, RiskGroupGen. The RiskGroupGen on each provider can automatically finish the following three operations (*i.e.*, the three boxes in Figure 3): 1) generating an attack tree modeling the underlying dependencies (§4.2.1); 2) extracting the top-k critical risk groups from the generated attack tree (§4.2.2); and 3) computing the severity of each extracted risk group, and putting all of them in a local dataset, *i.e.*, risk group dataset, for the post process (§4.2.3).

4.2.1 Attack Tree Generation

RiskGroupGen applies MulVAL [46, 47], an efficient tool for automatically generating an attack tree to representing the underlying structures of any given service. Namely, the task of MulVAL is to transform the service and its infrastructure components shown in Figure 1(a) to the attack tree shown in Figure 1(b). Thus, each cloud provider first uses RiskGroupGen's MulVAL to generate a local attack tree (see Figure 3).

Given the fact that each cloud provider knows its local infrastructure dependency information (including network topology and application configurations), each provider only needs to translate this information into Datalog representations, and then puts them into MulVAL, thus obtaining a local attack tree. MulVAL knows the vulnerability information of components within target services, because it integrates Common Vulnerability Scoring System (CVSS) [44]. CVSS is a widely used vulnerability scoring database, and records almost all the exploited and potential vulnerabilities, and their corresponding exploitability and impact, which are given by security experts.

Note that we make no claim that attack tree generation is novel in itself, as many attack tree generation systems have been widely deployed [51, 72, 74]. We merely utilize MulVAL as a "black box" towards our goal of automatically building an attack tree for the post process.

4.2.2 Extracting the Top-k Critical Risk Groups

With the generated attack tree G, RiskGroupGen invokes its another module, *RGget*, to extract the top-k critical risk groups from G. Nevertheless, it is difficult to offer an accurate and efficient analysis approach to determine the top-k critical risk groups in a large attack tree, since analyzing an attack tree is NP-hard [73].

Insight 1 *Our target,* i.e.*, extracting the top-k critical risk groups from G, can be reduced to WP-MaxSAT problem (defined in §3.2). An attack tree G can be represented by a Boolean formula ϕ, and G's vulnerabilities correspond to the Boolean variables in ϕ. If we can obtain the severity of each vulnerability in G, we can use these severity values as the weights for Boolean variables in ϕ, thus getting the weight vector of ϕ (defined in §3.2). This allows us to put the ϕ and its weight vector into*

a WP-MaxSAT solver, obtaining an assignment with the highest weight, which exactly corresponds to the most critical risk group in G. We can iteratively run the above process to obtain the top-k critical risk groups. Because WP-MaxSAT solvers can efficiently solve the problem set with millions of Boolean variables, such a construction makes our solution scalable to large-scale systems.

Based on the above insight, we design RGget's workflow as the following three steps.

First, RGget transforms an input attack tree G to a Boolean formula ϕ in Conjunctive Normal Form (CNF). For example, the attack tree shown in Figure 2(b) can be transformed to the following Boolean formula:

$$\begin{aligned}\phi &= (Service = (CertificateSys) \vee (FileSys)) \\ &\Leftrightarrow ((Vul3 \vee (Vul2 \wedge Vul3)) \wedge (Vul1 \vee (Vul2 \\ &\quad \wedge Vul3))) \vee Vul4 \vee Vul1 \vee Vul5 \\ &\Leftrightarrow (Vul1 \vee Vul2 \vee Vul4 \vee Vul5) \wedge (Vul1 \vee Vul3 \vee Vul4 \vee Vul5)\end{aligned}$$

Second, for each vulnerability V_i in G, RGget extracts $Impact(V_i)$ and $Exploit(V_i)$ from CVSS, and then computes $Severity(V_i) = Exploit(V_i) \cdot Impact(V_i)$ to get the weight of V_i. All the vulnerabilities' weights are put in a vector, forming a weight vector mentioned in §3.2.

Finally, RGget applies a WP-MaxSAT solver by taking the above generated CNF formula ϕ and its weight vector as input, to extract k assignments with the highest weights through k rounds. In each round, RGget performs three steps: 1) we input the current formula ϕ and its weight vector into WP-MaxSAT solver to generate the maximal weight satisfying assignment for the current ϕ, 2) we get risk group by extracting all the "TRUE" variables from the resulting assignment, and 3) we use a conjunction to connect the current ϕ and the negation of the resulting assignment, thus generating a new ϕ for the next round.

The extracted k assignments exactly correspond to the top-k critical risk groups. Furthermore, because modern WP-MaxSAT solver is efficient, it is very fast to obtain the desired results (*i.e.*, the top-k critical risk groups).

4.2.3 Risk Group Dataset Generation

After extracting the top-k critical risk groups, RiskGroupGen organizes these k risk groups in a dataset, called risk group dataset. The output of RiskGroupGen (*i.e.*, the Step 2.2 in iAudit workflow) is a k-size risk group dataset. Each item in this dataset, *i.e.*, a risk group, is a tuple:

$$\langle RiskGroup\ i,\ \ Severity(RiskGroup\ i)\rangle,$$

where RiskGroup i is a combination of CVSS IDs of all the vulnerabilities involved in risk group i, and Severity(RiskGroup i) is the computed severity score of the RiskGroup i (see Equation 1). This format is the same as the dataset example in Figure 1. This risk group dataset will be used as the input of Step 2.3 in iAudit workflow.

Insight 2 *An important advantage of using CVSS is that each vulnerability in CVSS has a unique identifier (*e.g.*, CVE-xx-yy), so that a shared vulnerability in different cloud providers has the same identifier and weight, thus enabling us to assess independence by privately computing set intersection cardinality.*

4.3 Private Weighted Jaccard Similarity

So far, each cloud provider has locally generated a risk group dataset by analyzing its own attack tree. For any alternative inter-cloud replication R, we now need to perform a private set-similarity protocol among the cloud providers of R to output a value quantifying R's independence (as shown in Figure 3). However, it is challenging to make such a computation practical, since risk groups in the risk group dataset of each cloud have different severity values (computed by Equation 1). This requires the employed set-similarity protocol capable of taking into account the weights of different elements (*i.e.*, the severity values of different risk groups). Nevertheless, the state-of-the-art private set-similarity protocols *either* focus on efficiency while failing to distinguish elements by their weights [9, 72], *or* take elements' weights into account in computationally expensive ways [67, 70, 71].

We construct the first *efficient* private *weighted* set-similarity protocol, which utilizes Jaccard similarity to quantify the similarity between the datasets of different cloud providers. We have described the working principle of MinHash-based Jaccard similarity computation in §3.3. However, conventional MinHash-based Jaccard similarity computation cannot be directly applied to achieve our goal, due to the following two reasons.

- Because datasets in our case contain elements' weights (*i.e.*, risk groups' severities) rather than unweighted case in §3.3, we need a weighted MinHash approach to "capture" elements' weights (§4.3.1).
- Because no party in our target scenario has the global view, we need an approach that can privately compute $h^{(i)}_{min}(S_0) = ... = h^{(i)}_{min}(S_{k-1})$, *i.e.*, computing a privacy-preserving set intersection cardinality (§4.3.2).

If the above two primitives can be obtained, then we are able to construct a Private Weighted Jaccard Similarity computation protocol, named PW-JS (see §4.3.3).

4.3.1 Weighted MinHash

We construct a new weighted MinHash approach, called *weighted MinHash*, based on the weighted sampling theory [12]. Our weighted MinHash approach (with m MinHash functions) extracts a vector $\{h'^{(j)}_{min}(S)\}_{j=1}^{m}$ from a dataset S, where $h'^{(i)}(x)$ is computed with Equation 2:

$$h'^{(j)}(x) = \frac{-\log h^{(j)}(x)}{d_x}, \tag{2}$$

Algorithm 1: Weighted MinHash primitive.

Input: Cloud provider i's local risk group dataset: S_i
Input: The number of MinHash functions: m
Input: Empty set: S'_i
Output: S'_i
for $j \leftarrow 1$ *to* m **do**
  $h^{(j)}_{min}(S_i)$ = MAX;
  foreach $x \in S_i$ **do**
    $h'^{(j)}(x) = \frac{-\log h^{(j)}(x)}{d_x}$;
    if $h'^{(j)}(x) < h^{(j)}_{min}(S_i)$ **then**
      $h^{(j)}_{min}(S_i) = h'^{(j)}(x)$;
  Append $h^{(j)}_{min}(S_i)$ to S'_i;
return S'_i;

where x denotes each element in S, $h^{(j)}(.)$ is the jth MinHash function, and d_x is the element x's weight.

Chum *et al.* [12] have proved that the weighted sampling shown in Equation 2 can make an element x with a weight of d_x will be exactly d_x times as likely to produce the minimum value as an element with a weight of 1. In other words, an element e with higher weight would generate smaller $h'^{(j)}(e)$, thus making itself have higher probability to be S's minimum value, *i.e.*, $h'^{(j)}_{min}(S)$. For example in Figure 1, RiskGroup 1 with weight 3 would generate a value $h'^{(j)}$(RiskGroup 1) which has $3\times$ probability than another element (with weight 1), *e.g.*, RiskGroup 2, to become the minimum value across datasets. Algorithm 1 details the weighted MinHash primitive.

4.3.2 Private Set Intersection Cardinality

A typical private set intersection cardinality protocol [32, 56] allows a group of k parties, each with a set S_i to privately compute $|\cap_i S_i|$ (*i.e.*, set intersection cardinality), without learning the specific elements in $\cap_i S_i$. To enable private set intersection cardinality operation, we apply an existing private set intersection cardinality protocol proposed by Vaidya and Clifton (VC) [56]. Note that our framework can also apply other private set intersection cardinality protocol, *e.g.*, Kissner and Song [32]. This section briefly presents VC protocol, and the next section (§4.3.3) details how we construct our PW-JS protocol.

Vaidya and Clifton (VC) protocol. Initially, all parties form a logical ring, and agree upon the same cryptographic hash function (*e.g.*, SHA-256 or MD5). Each party has its own permutation used to shuffle the items in its local dataset, and its own public key used for commutative encryption. 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 the protocol, 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 $\{e\|1, ..., e\|t\}$, with '$\|$' being a concatenation operator. Each party then hashes and encrypts every individual element in its dataset, and randomly permutes all the encrypted elements. Then, each party sends the encrypted and permuted dataset to its successor in the ring. 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.*, $|\cap_i S_i|$, due to the commutative property.

4.3.3 PW-JS Protocol Construction

With the above two primitives—*i.e.*, weighted MinHash (§4.3.1) and private set intersection cardinality (§4.3.2)—in hand, we now construct our PW-JS protocol. Initially, iAudit system setting uniformly generates m different sampling functions and sends them to every cloud provider in the target replication deployment. First, each provider p_i ($i = 0, ..., k-1$) generates $S'_i = \{h^{(j)}_{min}(S_i)\}^m_{j=1}$ through performing m weighted MinHash computations (Algorithm 1). This step maps the input risk group dataset S_i to a much smaller dataset S'_i. Second, each cloud provider takes these MinHash-generated datasets, *i.e.*, S'_i, as input to run the private set intersection cardinality protocol (*i.e.*, VC protocol) to get the number of common components across them, *i.e.*, $|\cap_i S'_i|$. Finally, the Jaccard similarity can be approximated as $|\cap_i S'_i|/m$. As discussed in §3.3, the MinHash-based approach leads to much higher efficiency but lower accuracy. To increase the accuracy, we can use more sampling functions in MinHash. How to make the trade-off between efficiency and accuracy depends on the application domain.

4.4 Practical Issues and Solutions

We now discuss how to deal with dishonest cloud providers and what incentives the cloud providers need to join an "iAudit auditing community".

4.4.1 Dealing with Dishonest Providers' Behaviors

In practice, cloud providers may dishonestly execute iAudit, especially local risk group dataset generation. For example, dishonest providers may declare a subset of their actual risk group datasets, thus benefiting themselves. This is because, by doing so, each dishonest cloud provider would have a smaller set similarity result and hence greater independence. Thus, dishonest providers might rank higher in the resulting lists.

Even worse, a malicious cloud provider may only take some specific risk groups as the input of the PW-JS protocol, in order to detect (or called prob) whether other

cloud providers have the same risk groups. We call this behavior as *malicious probing behavior*.

To address the malicious cloud provider issues, we give the following three solutions.

- We can equip RiskGroupGen with trusted computing techniques, *e.g.*, Trusted Platform Module (TPM), to remotely attest whether cloud providers are performing iAudit as required. Since existing efforts such as Ecalibur [50] have deployed TPM into some cloud platforms successfully, the TPM approach in principle can attest to the behaviors of a cloud provider, verifying whether the provider honestly performs iAudit, without leaking business privacy of the cloud provider.
- Alternatively, an impressive technique, named privacy-preserving accountability technique was recently proposed by Antonis *et al.* [48]. This approach could also be used to verify whether the potential cloud providers honestly perform RiskGroupGen, while preserving their business privacy. Compared with the TPM-based scheme, this solution does not need to install any additional hardware components on the cloud providers.
- For the malicious probing behavior case, iAudit can deploy a simple mechanism explicitly specifying that the number of elements (*i.e.*, risk groups) in each risk group dataset D_j must be higher than a pre-defined threshold (*e.g.*, > 500); otherwise, a cloud provider with a small D_j (*e.g.*, < 500) is not allowed to perform PW-JS. While such a mechanism is simple, we believe it works reasonably in our context. Because each risk group dataset D_j records risk groups in an enterprise-scale cloud provider, the number of elements in D_j should be very high in reality. Thus, a cloud provider holding a small risk group dataset is highly suspicious, and we need to disallow it to involve in PW-JS.

We have developed a tool detecting malicious cloud providers' behaviors (detailed in §5).

4.4.2 Motivating Cloud Providers to Join

Cloud providers not participating in iAudit's auditing community will not appear in potential cloud provider list that iAudit offers to cloud customers. Thus, non-participating cloud providers will lose customers due to not being on the iAudit "certified provider list". Furthermore, cloud providers participating in iAudit community need to locally install RiskGroupGen provided by iAudit, which offers them the opportunity to learn risk groups hidden in their systems.

5 Detecting Dishonest Cloud Providers

To detect the misbehaviors of dishonest cloud providers, iAudit offers an dishonest-provider-detection mechanism that checks the correctness of each cloud provider's behaviors, while preserving the business privacy of audited cloud providers. Figure 4 shows how does iAudit check dishonest cloud providers for each step.

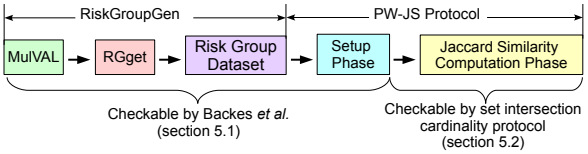


Figure 4: Checking dishonest behaviors.

5.1 Checking RiskGroupGen and Setup Phase

We check RiskGroupGen and the setup phase of PW-JS protocol by applying an existing privacy-preserving accountability protocol [6].

Accountability systems and privacy-preserving accountability. The basic idea of accountability systems (without privacy-preserving) [22–24] is that every party in a decentralized network can generate a tamper-evident log which contains a complete trace of the performed computations (*e.g.*, $y = F(x)$). Later, an auditor (any other node in the network) can check the correctness of this party's operations by inspecting the logs, replaying the execution of a reference implementation, and finally comparing the results.

However, the above accountability systems cannot preserve the privacy of audited parties, because they need to check the local log of audited parties. Backes *et al.* [6] construct a privacy-preserving accountability protocol, where each party can generate a "blinded" log used to convince an auditor of the correctness of a computation $y = F(x)$, while preserving the secrecy of specific inputs and outputs (*i.e.*, x and y) of the computation. This effort requires the auditor knowing $F()$. The key design of Backes *et al.* is a non-interactive zero-knowledge proof system (constructed based on Groth and Sahai proof system [21]) that supports statements in the pairing-product equations and a pseudo-random function which works in bilinear groups. Note that Backes *et al.* protocol not only supports general computational functions, but also supports non-deterministic computations. The protocol's efficiency and security have been proved [6].

Using Backes *et al.* to detect dishonest local behaviors by cloud providers. We have deployed Backes *et al.* protocol in both RiskGroupGen and the setup phase of PW-JS protocol to assist iAudit to detect dishonest cloud providers. In particular, the input and output of Backes *et al.* protocol in our design are cloud configuration (*i.e.*, the input of RiskGroupGen—*config*) and S'_j (defined in §4.3.3), respectively. The operation $F()$ is RiskGroupGen plus the setup phase of PW-JS (see Figure 4). After each cloud provider j runs RiskGroupGen and the setup phase

of PW-JS, our prototype can generate a zero-knowledge proof p_j (based on Backes *et al.* protocol) used to check whether the provider j correctly performs the above operations locally. Then, the cloud provider j sends p_j to iAudit. Because iAudit knows the implementation of RiskGroupGen and PW-JS (the requirement using Backes *et al.* protocol), iAudit is capable of checking whether the correctness of cloud provider j's behaviors based on the received proof p_j without learning any sensitive information of provider j.

Note that even if the cloud provider j compromises the implementation of RiskGroupGen to generate a fake proof p', such misbehaviors cannot bypass the checking of iAudit, because iAudit checks p' based on a reference (*i.e.*, correct) implementation of RiskGroupGen which is maintained by iAudit.

5.2 Checking Dishonest Similarity Computation Phase

While most of dishonest behaviors of cloud providers should occur during the execution of RiskGroupGen or the setup phase of PW-JS, cloud providers may also dishonestly perform set intersection cardinality protocol in the Jaccard similarity computation phase of PW-JS. In practice, detecting these misbehaviors is straightforward, and has been sufficiently studied by almost all the set intersection cardinality protocols [13]. For example, state-of-the-art private set intersection cardinality protocols (*e.g.*, VC [56] and Kissner and Song [32]) have been able to generate publicly checkable zero-knowledge proofs for all the operations conducted by cloud providers, thus making cloud providers' behaviors checkable.

Note that generating these zero-knowledge proofs will not leak the sensitive information of each cloud provider, since all the items have been hashed, encrypted and permuted in the setup phase (*i.e.*, before the Jaccard similarity computation phase).

6 Implementation and Evaluations

This section first describes the iAudit implementation (§6.1), and then presents a real-world case study to evaluate the practicality of iAudit (§6.2). Finally, we evaluate the overhead and performance of iAudit in §6.3.

6.1 Prototype Implementation

We fully implemented an iAudit prototype in Java. In the implementation of RiskGroupGen, we used MulVAL library [45] to build the attack tree generation component, and used Toysolver [39], a fast WP-MaxSAT solver, to develop the RGget. In the implementation of PW-JS, we used SHA-256 as the cryptographic hash function to hash the elements in each dataset. For MinHash, we used an open-source MinHash library [37]. We used RSA as commutative encryption (with a 1024-bit key).

Table 1: Ranking list based on the Jaccard similarities of two and three-way inter-cloud replication deployments. Cloud1, 2, 3 and 4 are equipped with Cassandra, Dynamo, HBase and Riak, respectively.

Rankings	2-Way Inter-Cloud Replications	Jaccard
NO.1	Cloud3 & Cloud4	0.1221
NO.2	Cloud2 & Cloud4	0.1514
NO.3	Cloud1 & Cloud4	0.1838
NO.4	Cloud1 & Cloud2	0.2244
NO.5	Cloud2 & Cloud3	0.2466
NO.6	Cloud1 & Cloud3	0.3016
Rankings	**3-Way Inter-Cloud Replications**	**Jaccard**
NO.1	Cloud2 & Cloud3 & Cloud4	0.1167
NO.2	Cloud1 & Cloud3 & Cloud4	0.1184
NO.3	Cloud1 & Cloud2 & Cloud4	0.1386
NO.4	Cloud1 & Cloud2 & Cloud3	0.1647

6.2 Case Study

We now present a case study—similar to our motivating example (§2.1)—to evaluate iAudit's practicality. In this case study, a customer wants an availability enhancement storage solution, like iCloud redundantly renting S3 and Azure storage, which needs more than one individual cloud storage providers. The customer has found four potential cloud providers: Cloud 1-4, each of which offers a key-value store. Then, she consults iAudit for the most independent inter-cloud replication deployment to avoid outages from cascading vulnerabilities [20].

In our case study, these four cloud providers deploy popular key-value storage systems (Apache Cassandra, Dynamo, HBase, and Riak) as back-ends, respectively. Specifically, we assigned one to each cloud provider as follows, Cloud1: Cassandra, Cloud2: Dynamo, Cloud3: HBase, and Cloud4: Riak. The four providers adopt the same software components for their front-ends, DNS services and switches, because we mainly focus on potential correlations between the above four storage systems.

After we perform the entire iAudit workflow, the customer gets the ranking lists for two-way and three-way inter-cloud replications, as shown in Table 1. Table 1 presents the following suggestions.

- Using two cloud storages cannot give us truly robust guarantees in the most cases. Specifically, datasets with Jaccard similarity $J \leq 0.15$ are considered uncorrelated [29], so that we observe that only the replication using Cloud3 (HBase) and Cloud4 (Riak) can give us the similarity less than 0.15.
- Three-way inter-cloud replication can typically give us a truly availability enhancement.
- Storage systems implemented by the same organizations are more likely to introduce cascading vulnerability risks, since these systems typically use common libraries and packages that may have the same buggy code. For example, HBase and Cassandra are devel-

Table 2: Configurations of the evaluated topologies.

	Topology A	Topology B	Topology C
# Switch ports	24	48	64
# Core routers	144	576	1,024
# Agg switches	288	1,152	2,048
# ToR switches	288	1,152	2,048
# Servers	3,456	27,648	65,536
Total # devices	4,176	30,528	70,656

Table 3: Comparing the run-time (seconds) of extracting the top-100 critical risk groups between RGget, Huang *et al.*, and INDaaS.

	RGget	Huang *et al.*	INDaaS's SIA
Topology A	4.108 +2.602 =6.71	16.74	53
Topology B	28.776 +51.396 =80.172	384.2	1454
Topology C	68.149 +203.366 =271.515	1862	4066

oped under the support and regulation of Apache, so that they share many libraries and packages.

6.3 Performance Evaluation

Configuration. The performance of iAudit was evaluated based on three real-world topologies extracted from the Topology Zoo [75] dataset, which consists of 261 actual cloud deployments. The extracted topologies follow the fat tree model [40] and represent small-scale, medium-scale and large-scale cloud deployments, respectively. Table 2 details these three topologies. The performance evaluation was conducted on a cluster of 40 workstations equipped with Intel Xeon Quad Core HT 3.7 GHz CPU and 16 GB RAM. By default, we use 1000 MinHash functions for our evaluations, since using more functions cannot effectively increase the accuracy but introduces much more computational overhead [10].

6.3.1 Microbenchmark

In our prototype, the operations of iAudit include: 1) RGget, *i.e.*, the top-k critical risk groups extraction via WP-MaxSAT solver (§4.2.2), 2) weighted MinHash sampling (§4.3.1), and 3) private set intersection cardinality (§4.3.2). Thus, this section evaluates their overhead.

RGget. The purpose of RGget is to extract the top-k risk groups from a given attack tree representing some cloud topology. Thus, we measured the run-time of RGget from reading a given attack tree to extracting the top-100 risk groups. We also evaluated two state-of-the-art attack tree analysis algorithms, Huang *et al.* [28] and INDaaS's SIA algorithm [72], for comparison. Table 3 presents the performance (seconds) comparison between the three approaches on extracting the top-100 critical risk groups under different topologies (detailed in Table 2). The performance of RGget shown in Table 2 includes two parts: 1) the run-time from reading an attack tree to generating the corresponding formula coupled with its weight vector, and 2) the run-time of using WP-MaxSAT solver to output the desired results. With the help of WP-MaxSAT solver, RGget is 20× more efficient than INDaaS's SIA algorithm in Topology B and Topology C.

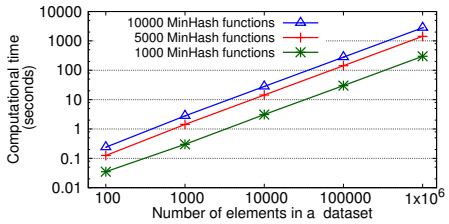


Figure 5: Weighted MinHash evaluation.

Weighted MinHash sampling. While Broder has proved expected error of MinHash-based Jaccard similarity as $O(1/\sqrt{m})$ [10], we still want to understand the run-time of running m different MinHash functions on a dataset with n elements. We set m to 1000, 5000 and 10000, and vary n between 100 and one million to cover a wide range of real-world settings. As shown in Figure 5, the computational overhead increases linearly with the number of elements in a dataset.

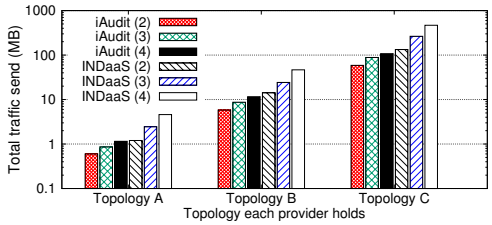
Private set intersection cardinality. To evaluate private set intersection cardinality protocol (*i.e.*, VC [56]) used in PW-JS, we focused on the computational and bandwidth overhead for different sizes of input datasets. The cryptographic operations of VC include hashing, commutative encryption, and permutation. As shown in Table 4, the bandwidth overhead of VC increases slowly, while VC's computational overhead increases almost linearly with the size of the datasets.

6.3.2 System Overhead

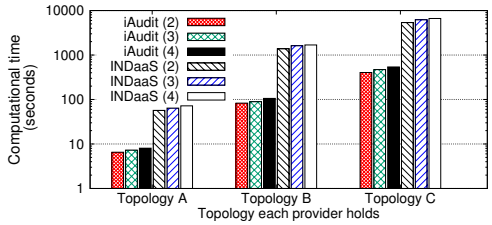
To evaluate iAudit's system overhead, we compared iAudit with a representative private independence auditing system, INDaaS [72]. INDaaS can offer private independence auditing on inter-cloud replications at component-set levels of detail. At the heart of INDaaS lies a simple weighted set similarity protocol, named PIA. In a PIA workflow, each cloud provider first duplicates each element i in its local dataset w_i times, where w_i is the element i's weight. Then, involved providers privately and coordinately compute the number of overlapping elements (with the VC protocol) across their "inflated" datasets, computing weighted set similarity. Thus, the main operations of PIA include element duplication, hashing, commutative encryption and permutation. In iAudit, the cryptographic

Table 4: The overhead of VC with the 1024-bit key.

# of elements in each provider's dataset	10^3	5000	10^4	50000	10^5
Traffic overhead (MB)	0.28	1.5	2.92	14.7	29.4
Comp. overhead (Sec.)	1.76	3.74	6.48	32.79	67.51



(a) Bandwidth overhead.



(b) Computational overhead.

Figure 6: Comparing the system overhead of iAudit and INDaaS in different topologies. iAudit (k) and INDaaS(k) mean that there are k cloud providers participating in the iAudit and INDaaS systems, respectively.

primitives of PW-JS are weighted hashing, commutative encryption, and permutation.

In our evaluation, there are k cloud providers, and each locally maintains the same topology model. We set k to 2, 3 and 4, and vary the topology model from Topology A to Topology C to cover a wide range of real-world settings. We measured and compared iAudit with INDaaS in terms of their bandwidth and computational overheads at each such cloud provider. Figure 6a and 6b show the bandwidth and computational overheads, respectively.

With a small number of cloud providers (*e.g.*, $k = 2$), the bandwidth overhead of INDaaS is comparable to that of iAudit. However, with an increasing number of cloud providers, the bandwidth overhead of INDaaS increases much faster than iAudit's. For example, when auditing a four-way inter-cloud replication where each cloud contains 70,656 components, iAudit is 4× more efficient in bandwidth than INDaaS. With respect to the computational overhead, iAudit outperforms INDaaS by a few orders of magnitude although both systems' computational overheads increase almost linearly with the number of elements in each cloud provider's dataset.

7 Related Work

Following the cloud auditing concept [53], many privacy-preserving auditing protocols have been proposed to audit the integrity of cloud storages [54, 59–62, 69]. Different from iAudit, these protocols do not focus on auditing the independence of inter-cloud replications.

Private set operations. There have been many private set operational protocols proposed in the past years. Secure multi-party computation (SMPC) [70] is the first effort, but it is not scalable in our scenario [67]. Arawal *et al.* [4] proposed the first two-party private set operation protocol based on commutative encryption. Vaidya and Clifton [56] extended it to more than two parties, and also proposed an optimisation technique to increase the efficiency. Freedman *et al.* [17] proposed the first private set intersection cardinality protocol based on homomorphic encryption. The protocol uses the polynomial generation approach to privately compute the number of the overlapping elements between two datasets. Kissner and Song [32] construct a collection of privacy-preserving set operation protocols to handle multi-party cases.

Independence auditing systems. INDaaS [72] is the first systematic effort to privately evaluate cascading failure risks across multi-cloud. Compared with iAudit, INDaaS has the following disadvantages. First, INDaaS only supports private auditing at *component-set level of detail*. Component-set level of detail simply enumerates components involved by a service as a set, *e.g.*{Certificate System, File System, httpd}; on the contrary, graph level of detail provided by iAudit focuses on structural dependencies between components, thus enabling fine-grained auditing capability. Second, INDaaS is not equipped with scalable attack tree analysis algorithms. Finally, INDaaS only supports unweighted private Jaccard similarity protocol.

8 Conclusion

This paper has presented iAudit, a third-party auditing system that assists cloud customers to quantify the independence of alternative inter-cloud replication deployments before their data or application adoptions, while preserving the privacy of the underlying cloud providers. Our design, implementation and experiments show that cloud customers can proactively prevent potential risk resulting from cascading vulnerabilities across their inter-cloud replications.

References

[1] ABU-LIBDEH, H., PRINCEHOUSE, L., AND WEATHERSPOON, H. RACS: A case for cloud storage diversity. In *1st ACM Symposium on Cloud Computing (SoCC)* (June 2010).

[2] ADMIN, J. Jingdong user data leakage back to say from 3 years ago, the problem of vulnerability. https://goo.gl/hQkD9N, 2016.

[3] ADSHEAD, A. Why you need a cloud storage compliance audit. https://goo.gl/9TXy6i, 2012.

[4] AGRAWAL, R., EVFIMIEVSKI, A. V., AND SRIKANT, R. Information sharing across private databases. In *ACM International Conference on Management of Data (SIGMOD)* (June 2003).

[5] ALVIANO, M., DODARO, C., AND RICCA, F. A MaxSAT algorithm using cardinality constraints of bounded size. In *24th International Joint Conference on Artificial Intelligence (UJCAI)* (July 2015).

[6] BACKES, M., FIORE, D., AND MOHAMMADI, E. Privacy-preserving accountable computation. In *18th European Symposium on Research in Computer Security (ESORICS)* (Sept. 2013).

[7] BAHL, P., CHANDRA, R., GREENBERG, A. G., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2007).

[8] BESSANI, A. N., CORREIA, M. P., QUARESMA, B., ANDRÉ, F., AND SOUSA, P. DepSky: Dependable and secure storage in a cloud-of-clouds. In *6th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)* (Apr. 2011).

[9] BLUNDO, C., DE CRISTOFARO, E., AND GASTI, P. EsPRESSo: Efficient privacy-preserving evaluation of sample set similarity. In *DPM/SETOP* (Sept. 2012).

[10] BRODER, A. Z. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES)* (June 1997).

[11] CHEN, X., ZHANG, M., MAO, Z. M., AND BAHL, P. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[12] CHUM, O., PHILBIN, J., AND ZISSERMAN, A. Near duplicate image detection: Min-Hash and tf-idf weighting. In *the British Machine Vision Conference (BMVC)* (Sept. 2008).

[13] CRISTOFARO, E. D., KIM, J., AND TSUDIK, G. Linear-complexity private set intersection protocols secure in malicious model. In *ASIACRYPT* (2010).

[14] CVE. CVE-2016-0738. https://vulners.com/cve/CVE-2016-0738, 2016.

[15] DAVIDSON, D., MOENCH, B., JHA, S., AND RISTENPART, T. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium* (Aug. 2013).

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

[17] FREEDMAN, M. J., NISSIM, K., AND PINKAS, B. Efficient private matching and set intersection. In *Conference on Theory and applications of cryptographic techniques (EUROCRYPT)* (May 2004).

[18] GIEICHER, N. The big lesson we must learn from the Dyn DDoS attack. https://goo.gl/nQ3qzk, 2016.

[19] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2011).

[20] GREER, D. Heartbleed as metaphor. *Lawfare* (Apr. 2014). http://www.lawfareblog.com/2014/04/heartbleed-as-metaphor/.

[21] GROTH, J., AND SAHAI, A. Efficient non-interactive proof systems for bilinear groups. In *27th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)* (Apr. 2008).

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

[23] HAEBERLEN, A., ADITYA, P., RODRIGUES, R., AND DRUSCHELND, P. Accountable virtual machines. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2010).

[24] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. PeerReview: Practical accountability for distributed systems. In *21st ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2007).

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

[26] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S. T. V., AND ZILL, B. IronFleet: Proving practical distributed systems. In *25th ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2015).

[27] HAWBLITZEL, C., HOWELL, J., LORCH, J. R., NARAYAN, A., PARNO, B., ZHANG, D., AND ZILL, B. Ironclad Apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2014).

[28] HUANG, H., ZHANG, S., OU, X., PRAKASH, A., AND SAKALLAH, K. A. Distilling critical attack graph surface iteratively through minimum-cost SAT solving. In *27th Annual Computer Security Applications Conference (ACSAC)* (Dec. 2011).

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

[30] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2009).

[31] KARTCH, R. DDoS attacks: Four best practices for prevention. https://goo.gl/fZD3c8, 2016.

[32] KISSNER, L., AND SONG, D. X. Privacy-preserving set operations. In *25th Annual International Cryptology Conference (CRYPTO)* (Aug. 2005), Springer.

[33] KULA, R. G., GERMAN, D. M., OUNI, A., ISHIO, T., AND INOUE, K. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. In *11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (Sept. 2017).

[34] LAWFARE. Lawfare Blog. https://www.lawfareblog.com/, 2010.

[35] LENTEJAS, R. Use multiple cloud providers for redundancy. https://goo.gl/neKZKZ, 2012.

[36] LI, F., DURUMERIC, Z., CZYZ, J., KARAMI, M., BAILEY, M., MCCOY, D., SAVAGE, S., AND PAXSON, V. You've got vulnerability: Exploring effective vulnerability notifications. In *25th USENIX Security Symposium*.

[37] LIBS, C. MinHash library. https://github.com/codelibs/minhash, 2014.

[38] MOURA, G. C. M., DE O. SCHMIDT, R., HEIDEMANN, J., DE VRIES, W. B., MÜLLER, M., WEI, L., AND HESSELMAN, C. Anycast vs. DDoS: Evaluating the November 2015 root DNS event. In *Internet Measurement Conference (IMC)* (Nov. 2016).

[39] MSAKAI. Toysolver. https://github.com/msakai/toysolver, 2015.

[40] MYSORE, R. N., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2009).

[41] NAPPA, A., JOHNSON, R., BILGE, L., CABALLERO, J., AND DUMITRAS, T. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *36th IEEE Symposium on Security and Privacy (S&P)* (May 2015).

[42] NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Subtleties in tolerating correlated failures in wide-area storage systems. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (May 2006).

[43] NG, B. H., HU, X., AND PRAKASH, A. A study on latent vulnerabilities. In *IEEE Symposium on Reliable Distributed Systems (SRDS)* (Oct. 2010).

[44] OF STANDARDS, N. I., AND TECHNOLOGY. Common Vulnerability Scoring System. http://nvd.nist.gov/, 2016. Online; accessed May 16 2017.

[45] OU, X. MulVAL 1.1. http://people.cis.ksu.edu/~xou/argus/software/mulval/, 2016.

[46] OU, X., BOYER, W. F., AND MCQUEEN, M. A. A scalable approach to attack graph generation. In *13th ACM Conference on Computer and Communications Security (CCS)* (Nov. 2006).

[47] OU, X., GOVINDAVAJHALA, S., AND APPEL, A. W. MulVAL: A logic-based network security analyzer. In *14th USENIX Security Symposium (USENIX Security)* (July 2005).

[48] PAPADIMITRIOU, A., ZHAO, M., AND HAEBERLEN, A. Towards privacy-preserving fault detection. In *9th Workshop on Hot Topics in Dependable Systems (HotDep)* (Nov. 2013).

[49] PHAM, N. H., NGUYEN, T. T., NGUYEN, H. A., AND NGUYEN, T. N. Detection of recurring software vulnerabilities. In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Sept. 2010).

[50] SANTOS, N., RODRIGUES, R., GUMMADI, K. P., AND SAROIU, S. Policy-sealed data: A new abstraction for building trusted cloud services. In *21st USECIX Security Symposium (USENIX Security)* (Aug. 2012).

[51] SAWILLA, R. E., AND OU, X. Identifying critical attack assets in dependency attack graphs. In *13th European Symposium on Research in Computer Security (ESORICS)* (Oct. 2008).

[52] SCROXTON, A. Dyn DDoS attack highlights vulnerability of global Internet infrastructure. http://www.computerweekly.com/news/450401576/Dyn-DDoS-attack-highlights-vulnerability-of-global-internet-infrastructure, 2016.

[53] SHAH, M. A., BAKER, M., MOGUL, J. C., AND SWAMINATHAN, R. Auditing to keep online storage services honest. In *11th Workshop on Hot Topics in Operating Systems (HotOS)* (May 2007).

[54] SHAH, M. A., SWAMINATHAN, R., AND BAKER, M. Privacy-preserving audit and extraction of digital contents. *IACR Cryptology ePrint Archive 2008* (2008), 186.

[55] SØRENSEN, T. *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.

[56] VAIDYA, J., AND CLIFTON, C. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security 13*, 4 (2005), 593–622.

[57] VESELY, W. E., GOLDBERG, F. F., ROBERTS, N. H., AND HAASL, D. F. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, Jan. 1981.

[58] VIGO, R., NIELSON, F., AND NIELSON, H. R. Automated generation of attack trees. In *27th IEEE Computer Security Foundations Symposium (CSF)* (July 2014).

[59] WANG, C., CHOW, S. S. M., WANG, Q., REN, K., AND LOU, W. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers 62*, 2 (2013), 362–375.

[60] WANG, C., REN, K., LOU, W., AND LI, J. Toward publicly auditable secure cloud data storage services. *IEEE Network 24*, 4 (2010), 19–24.

[61] WANG, C., WANG, Q., REN, K., AND LOU, W. Privacy-preserving public auditing for data storage security in cloud computing. In *29th IEEE INFOCOM (INFOCOM)* (Mar. 2010).

[62] WANG, Q., WANG, C., REN, K., LOU, W., AND LI, J. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Transactions on Parallel and Distributed Systems 22*, 5 (2011), 847–859.

[63] WELSH, M. What I wish systems researchers would work on. https://goo.gl/MTVQUu, 2013.

[64] WILLIAMS, D., JAMJOOM, H., AND WEATHERSPOON, H. The Xen-Blanket: Virtualize once, run everywhere. In *European Conference on Computer Systems (EuroSys)* (Apr. 2012).

[65] WORLD HEALTH ORGANIZATION. Prevention is better than cure. http://www.who.int/bulletin/volumes/89/4/11-030411/en/, 2011.

[66] WU, X., TURNER, D., CHEN, C.-C., MALTZ, D. A., YANG, X., YUAN, L., AND ZHANG, M. NetPilot: Automating datacenter network failure mitigation. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2012).

[67] XIAO, H., FORD, B., AND FEIGENBAUM, J. Structural cloud audits that protect private information. In *ACM Cloud Computing Security Workshop (CCSW)* (Nov. 2013).

[68] XIE, P., LI, J. H., OU, X., LIU, P., AND LEVY, R. Using Bayesian networks for cyber security analysis. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (June 2010).

[69] YANG, K., AND JIA, X. Data storage auditing service in cloud computing: Challenges, methods and opportunities. *World Wide Web 15*, 4 (2012), 409–428.

[70] YAO, A. C.-C. Protocols for secure computations (Extended abstract). In *23rd Annual Symposium on Foundations of Computer Science (FOCS)* (Nov. 1982).

[71] ZHAI, E., CHEN, R., WOLINSKY, D. I., AND FORD, B. An untold story of redundant clouds: Making your service deployment truly reliable. In *9th Workshop on Hot Topics in Dependable Systems (HotDep)* (Nov. 2013).

[72] ZHAI, E., CHEN, R., WOLINSKY, D. I., AND FORD, B. Heading off correlated failures through Independence-as-a-service. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2014).

[73] ZHAI, E., WOLINSKY, D. I., XIAO, H., LIU, H., SU, X., AND FORD, B. Auditing the Structural Reliability of the Clouds. Tech. Rep. YALEU/DCS/TR-1479, Department of Computer Science, Yale University, 2013. Available at http://cpsc.yale.edu/sites/default/files/files/tr1479.pdf.

[74] ZHOU, W., FEI, Q., SUN, S., TAO, T., HAEBERLEN, A., IVES, Z. G., LOO, B. T., AND SHERR, M. NetTrails: a declarative platform for maintaining and querying provenance in distributed systems. In *ACM International Conference on Management of Data (SIGMOD)* (June 2011).

[75] ZOO, T. The Internet topology zoo. http://www.topology-zoo.org/, 2013.