Structural Cloud Audits that Protect Private Information

Hongda Xiao
Yale Univ., EE Dept.
hongda.xiao@yale.edu

Bryan Ford
Yale Univ., CS Dept.
bryan.ford@yale.edu

Joan Feigenbaum
Yale Univ., CS Dept.
joan.feigenbaum@yale.edu

ABSTRACT

As organizations and individuals have begun to rely more and more heavily on cloud-service providers for critical tasks, cloud-service reliability has become a top priority. It is natural for cloud-service providers to use redundancy to achieve reliability. For example, a provider may replicate critical state in two data centers. If the two data centers use the same power supply, however, then a power outage will cause them to fail simultaneously; replication *per se* does not, therefore, enable the cloud-service provider to make strong reliability guarantees to its users. Zhai *et al.* [28] present a system, which they refer to as a *structural-reliability auditor* (SRA), that uncovers common dependencies in seemingly disjoint cloud-infrastructural components (such as the power supply in the example above) and quantifies the risks that they pose. In this paper, we focus on the need for structural-reliability auditing to be done in a *privacy-preserving* manner. We present a privacy-preserving structural-reliability auditor (P-SRA), discuss its privacy properties, and evaluate a prototype implementation built on the Sharemind SecreC platform [6]. P-SRA is an interesting application of *secure multi-party computation* (SMPC), which has not often been used for graph problems. It can achieve acceptable running times even on large cloud structures by using a novel data-partitioning technique that may be useful in other applications of SMPC.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

General Terms

Reliability, Security, Measurement

Keywords

Cloud computing, reliability, secure multi-party computation

1. INTRODUCTION

Cloud computing and cloud storage now play a central role in the daily lives of individuals and businesses. For example, more than a billion people use Gmail and Facebook to create, share, and store personal data, 20% of all organizations use the commercially available cloud-storage services provided both by established vendors and by cloud-storage start-ups [8, 9], and programs run on Amazon EC2 and Microsoft Azure perform essential functions.

As people and organizations perform more and more critical tasks "in the cloud," reliability of cloud-service providers grows in importance. It is natural for cloud-service providers to use redundancy to achieve reliability. For example, a provider may replicate critical state in two data centers. If the two data centers use the same power supply, however, then a power outage will cause them to fail simultaneously; replication *per se* does not, therefore, enable the cloud-service provider to make strong reliability guarantees to its users. This is not merely a hypothetical problem: Although Amazon EC2 uses redundant data storage in order to boost reliability, a lightning storm in northern Virginia took out both the main power supply and the backup generator that powered all of Amazon's data centers in the region [16]. The lack of power not only disabled EC2 service in the area but also disabled Netflix, Instagram, Pinterest, Heroku, and other services that relied heavily on EC2. This type of dependence on common components is a pervasive source of vulnerability in cloud services that believe (erroneously) that they have significantly reduced vulnerability by employing simple redundancy.

Zhai *et al.* [28] propose *structural-reliability auditing* as a systematic way to discover and quantify vulnerabilities that result from common infrastructural dependencies. To use their SRA system, a cloud-service provider proceeds in three stages: (1) It collects from all of its infrastructure providers (*e.g.*, ISPs, power companies, and lower-level cloud providers) a comprehensive inventory of infrastructure components and their dependencies; (2) it constructs a service-wide fault tree; and (3) using fault-tree analysis, it estimates the likelihood that critical sets of components will cause an outage of the service. Prototype implementation and testing presented in [28] indicates that the SRA approach to evaluation of cloud-service reliability can be practical.

A potential barrier to adoption of SRA is the sensitive nature of both its input and its output. Infrastructure providers justifiably regard the structure of their systems, including the components and the dependencies among them, as proprietary information. They may be unwilling to disclose this information to a customer so that the latter can improve its reliability guarantees to *its* customers. Fault trees and failure-probability estimates computed by the SRA are also proprietary and potentially damaging (to the cloud-service provider as well as the infrastructure providers). All of the parties to SRA computation thus have an incentive not to participate. On the other hand, they have a countervailing incentive *to* participate: Each party stands to lose reputation (and customers) if it promises more reliability than it can actually deliver because it is unaware of common dependencies in its supposedly redundant infrastructure.

In this paper, we investigate the use of *secure multi-party computation* (SMPC) to perform SRA computations in a privacy-preser-

CCSW'13, November 8, 2013, Berlin, Germany.
ACM 978-1-4503-2490-8/13/11.
http://dx.doi.org/10.1145/2517488.2517493.

ving manner. SRA computation is a novel and challenging application of SMPC, which has not often been used for graph computations[1] (or, more generally, for computations on complex, linked data structures). We introduce a novel data-partitioning technique in order to achieve acceptable running times for SMPC even on large inputs; this approach to SMPC efficiency may be applicable in other contexts. Our preliminary experiments indicate that our P-SRA (for "private structural-reliability auditing") approach can be practical.

2. RELATED WORK

2.1 Secure Multi-Party Computation

The study of secure multi-party computation (SMPC) began with the seminal papers of Yao [25,26] and has been pursued vigorously by the cryptographic-theory community for more than 30 years. SMPC allows n parties $P_1,\ldots,P_n$ that hold private inputs $x_1,\ldots,x_n$ to compute $y = f(x_1,\ldots,x_n)$ in such a way that they all learn y but no P_i learns anything about x_j, $i \neq j$, except what is logically implied by the result y and the particular input x_i that he already knew. Typically, the *input providers* $P_1,\ldots,P_n$ wish not only to compute y in a privacy-preserving manner but also to do so using a protocol in which they all play equivalent roles; in particular, they don't want simply to send the x_i's to one trusted party that can compute y and send it to all of them. Natural applications include voting, survey computation, and set operations. One of the crowning achievements of cryptographic theory is that such privacy-preserving protocols can be obtained for any function f, provided one is willing to make some reasonable assumptions, *e.g.*, that certain cryptographic primitives are secure or that some fraction of the P_i's do not cheat (*i.e.*, that they follow the protocol scrupulously).

Many SMPC protocols have the following structure: In the first round, each P_i splits its input x_i into *shares*, using a *secret-sharing scheme*, and sends one share to each P_j; the privacy-preserving properties of secret sharing guarantee that the shares do not reveal x_i to the other parties (or even to coalitions of other parties, provided that the coalitions are not too large). The parties then execute a multi-round protocol to compute shares of y; the protocol ensures that the shares of intermediate results computed in each round also do not reveal x_i. In the last round, the parties broadcast their shares of y so that all of them can reconstruct the result. Alternatively, they may send the shares of y to an outside entity (resp., to a subset of the P_j's) if none (resp., only a subset) of the P_j's is supposed to learn the result. The maximum size of a coalition of cheating parties that the protocol must be able to thwart and the "adversarial model," *i.e.*, the capabilities and resources available to the cheaters, determine which secret-sharing scheme the P_i's should use. Because secret-sharing-based SMPC is common (and for ease of exposition), we will refer to parties' "sharing" or "splitting" their inputs. Note, however, that some SMPC protocols use other techniques to encode inputs and preserve privacy in multi-round computation.

The past decade has seen great progress on general-purpose platforms for SMPC, including Fairplay [14], FairplayMP [4], SEPIA [7], VIFF [10], and Tasty [13]. For our prototype implementation of P-SRA, we use the Sharemind SecreC platform [6]. Thorough comparison of SMPC platforms is beyond the scope of this paper, but we note briefly the properties of SecreC that make it a good choice in this context. Because it has a C-like programming language and optimizing compiler, assembler, and virtual machine, programmers can more easily write efficient programs with SecreC than with most of the other SMPC tools. Scalability to large numbers of input providers and reliable predictions of running times of programs are better in SecreC than in other SMPC environments. SecreC makes it easy for programs to use both private data (known to only one input provider) and public data (known to all parties to the computation) in the same program – something that is useful in our reliability-auditing context but is not provided by all SMPC platforms. On the downside, SecreC is not especially flexible or easily configurable.

[1] A notable exception is the work of Gupta *et al.* [12] on SMPC for interdomain routing.

2.2 Cloud Reliability

The case for "audits" as a method of achieving reliability in cloud services was originally put forth by Shah *et al.* [17], who advocated both *internal* and *external* auditing. Internal audits use information about the structure and operational procedures of a cloud-service provider to estimate the likelihood that the provider can live up to its service-level agreements. To the best of our knowledge, the first substantial effort to design and implement a general-purpose internal-auditing system that receives the structural and operational information directly from the cloud-service providers is the recent work of Zhai *et al.* [28]; the privacy issue and the possibility of addressing it with SMPC were raised in [28] but were not developed in detail.[2] External audits use samples of the cloud-service output provided by third parties through externally available interfaces to evaluate the quality of service; they have been investigated extensively, *e.g.*, in [18, 20–24]. Bleikertz *et al.* [5] present a cloud-auditing system that Shah *et al.* [17] would probably classify as "internal," because it uses structural and operational information about the cloud services to estimate reliability rather than using sampled output, but it obtains that structural and operational information through external interfaces rather than receiving it directly from the cloud-service providers.

In addition to auditing, technical approaches that researchers have taken to cloud reliability include *diagnosis*, the purpose of which is to discover the causes of failures after they occur and, in some cases, to mitigate their effects, *accountability*, the purpose of which is to place blame for a failure after it occurs, and *fault tolerance*. Further discussion of these approaches and pointers to key references can be found in Section 6 of [28].

2.3 Fault Trees

Fault-tree analysis [19] is a deductive-reasoning technique in which an undesirable event in a system is represented as a boolean combination of simpler or "lower-level" events. Each node in a fault tree[3] represents either an event or a logic gate. Event nodes depict failure events in the system, and logic gates depict the logical relationships among these events. The links of a fault tree illustrate the dependencies among failure events. The root node represents a "top event" that is the specific undesirable state that this tree is designed to analyze. The leaf nodes are "basic events," *i.e.*, fail-

[2] Concurrently with this work, Zhai, Chen, Wolinsky, and Ford [27] also explored the problem of privacy in cloud-reliability analysis, with a different goal of recommending good cloud configurations in a privacy-preserving manner. Their work was done independently of ours and differs from ours both in its target problem and in its technical approach. Briefly, Zhai *et al.* [27] simply compute the set of components that are common to two cloud-service systems, while our P-SRA system involves more participants and a richer set of outputs. The main technical tool in [27] is privacy-preserving set intersection, whereas P-SRA does a variety of privacy-preserving distributed computations using a general-purpose SMPC platform.

[3] What are called "fault trees" in the literature are not, in general, trees but rather DAGs. Because it is standard and widely used, we adopt the term "fault tree" in this paper.

ures that may trigger the top event; in order to use a fault tree, one must be able to assess (at least approximately accurately) the probabilities of these basic failures. Figure 4 is an example of a fault tree.

Fault-tree analysis has been applied very widely, *e.g.*, in aerospace, nuclear power, and even social services [11], but, to the best of our knowledge, was first used for cloud-service-reliability auditing by Zhai *et al.* [28]. It is an appropriate technique in this context for at least two reasons. First, the architectures of many cloud platforms can be accurately represented as leveled DAGs; therefore, potential cloud-service failures are naturally modeled by fault trees. Second, to construct a fault tree, one must uncover and represent the dependency relationships among components in a cloud system, and this inventory of dependencies is itself helpful in identifying potential failures (especially correlated failures).

3. PROBLEM FORMULATION

In order to specify in full detail the goals of our P-SRA system and how it achieves them, we start with a brief explanation of the SRA system of Zhai *et al.* [28]. Here and throughout the rest of this paper, a *failure set* (FS) is a set of components whose simultaneous failure results in cloud-service outage.[4] For example, the main and backup power supplies in the Amazon EC2 example described in Section 1 are an FS. A *minimal FS* is an FS that contains no proper subset that is also an FS.

The first necessary and nontrivial task of SRA is *data acquisition*. SRA's *data-acquisition unit* (DAU) collects from a target cloud-service provider S and all of the service providers that S depends on the details of network dependencies, hardware dependencies, and software dependencies, as well as the failure probability of each component. Using this inventory of components and the dependencies among them, SRA builds a model of S and the services on which it depends in the form of a *dependency graph*. Zhai *et al.* [28] assume that the dependency graph of a cloud service is a leveled DAG, and we also make this assumption; we are aware that it is a simplification (see Section 6), but it is an important first step. There are many potential technical and administrative challenges involved in modeling cloud components, discovering their dependencies, and assigning realistic failure probabilities; in particular, all cloud-service providers that participate in an SRA computation must agree on a taxonomy of components and types of dependencies. We defer to Subsection 3.2 of Zhai *et al.* [28] for discussion of these challenges and for details about data acquisition and dependency-graph construction in SRA. Here, we merely assume that cloud providers have *some* usable modeling and dependency-gathering infrastructure.

The next step in SRA is fault-tree analysis for the target cloud service S. Ideally, the output of this step is a complete set of minimal FSes for S. Note that an outage may occur because multiple entities that S relied on for redundancy had a common dependency on a set of components that failed; so accurate reporting of all minimal FSes requires information about all of the other service providers (*e.g.*, ISPs, power suppliers, and lower-level cloud services) that S uses. For some of these other services, the information might be publicly available (or, in any case, available to S) and thus not require the other service to participate in the computation; for example, SRA makes the simplifying assumption that it can obtain the information it needs about the ISPs and power suppliers that S uses without their participation, and we continue with that assumption in P-SRA. In other cases, participation in the SRA computation by other service providers is required; this is true, for example, of lower-level and peer cloud services on which S depends. If the ideal of reporting all minimal FSes is unattainable because it is too time-consuming, then SRA may produce a collection of (not necessarily minimal) FSes using a *failure-sampling algorithm*; this algorithm uses both the dependency graph and the individual components' failure probabilities.

Figure 3 depicts a simple dependency graph. Figure 4 depicts a corresponding fault tree. The semantics of an OR gate in the fault tree are that, if any input fails, the output of the gate is "fail." For an AND gate, only if all of the inputs fail does the gate output "fail." So Data Center #1 fails if Power #1 fails or if both Router #1 and Router #2 fail. Note that the logic-gate nodes in Figure 4 cannot be inferred from Figure 3; SRA collects additional information during its data-acquisition phase that is needed for fault-tree construction. The minimal FSes for Cloud Service #1 are {Data Center #1, Data Center #2}, {Router #1, Router #2}, {Power #1, Power #2}, {Power #1, Data Center #2}, and {Data Center #1, Power #2}.

The goal of P-SRA is to perform structural-reliability auditing in a privacy-preserving manner; to do this, we must modify all phases of SRA – data acquisition, fault-tree construction and analysis, and delivery of output. Our basic approach to the first two is to use SMPC. Instead of sending their data to one machine that integrates them and performs fault-tree analysis, P-SRA participants split their data into shares and perform fault-tree construction and analysis in a distributed, privacy-preserving fashion. However, the output of this computation cannot simply be a comprehensive list of S's minimal FSes, as it was in SRA, because these sets may contain infrastructural components that are used only by other service providers (*i.e.*, not by S). So the first technical challenge in the design of P-SRA is to specify SMPC outputs that reveal to S the components of its own infrastructure that could cause an outage while not revealing private information about other service providers' infrastructure. The second technical challenge is to reduce the size of the data sets that are input to the SMPC; the complete dependency graph of a cloud-service provider could have millions of nodes, which is more than current SMPC technology can handle, even in an off-line procedure like reliability auditing. P-SRA deals with this challenge by requiring each service provider that participates in the SMPC to partition its components into those that are known to be "private" and those that might be shared with other participants. For example, if the storage devices in a data center owned and operated by S are not accessible by anyone outside of S, then their failure cannot cause any service other than S to fail – they can be marked "private" by S and not entered individually into the SMPC. Rather, S can collapse certain "private" subgraphs of its dependency graph into single nodes, treat each such node as a "component" when entering its input to the SMPC, and perform SRA-style fault-tree analysis on the private subgraph locally. We refer to this data-partitioning technique as *subgraph abstraction*. Finally, P-SRA must provide useful, privacy-preserving output to cloud-service users as well as cloud-service providers. These three technical challenges are addressed in detail in Section 4.

In Section 5, we present a P-SRA prototype implemented on the Sharemind SecreC platform. The properties of this platform guarantee security in the semi-honest (or honest-but-curious) adversarial model. See the beginning of Section 5 for a more detailed explanation of Sharemind's computational model and adversarial model.

4. SYSTEM DESIGN

4.1 System Overview

There are three types of participants in the P-SRA system: the P-SRA host, cloud-service providers, and cloud-service users; see

[4]These are called *cut sets* in the fault-tree literature.

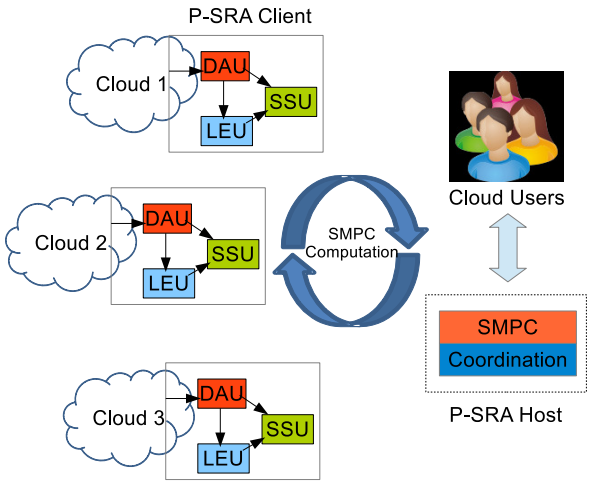

Figure 1: System Overview

Figure 1. The input supplied by each cloud-service provider is its topology information; this is private information and cannot be revealed to any other participants. The input supplied by the P-SRA host is the SMPC protocol. The inputs supplied by the cloud-service users are the set of cloud-service providers that they use or plan to use. The inputs of the P-SRA host and the cloud-service users are not private.

The P-SRA host consists of two modules. One is the SMPC execution unit (SMPC), which is responsible for execution of the SMPC protocol. The other is the coordination unit, which is responsible for establishing the SMPC protocol and coordinating the communication among the P-SRA host and the other participants.

Each cloud-service provider installs and controls a **P-SRA client** that processes local data and communicates with the P-SRA host. The P-SRA client consists of three modules: the Data-Acquisition Unit (DAU), the Secret-Sharing Unit (SSU), and the Local-Execution Unit (LEU). The DAU collects component and dependency information from the cloud-service provider and stores it in a local database. The SSU (1) abstracts the dependency information of "private" components in order to reduce the size of the input to the SMPC, (2) splits the dependency information into secret shares, and (3) connects to the P-SRA host and the SSUs of other cloud-service providers to execute the SMPC. The LEU performs local structural-reliability analysis within each "abstracted" macro-component.

Step 1: Privacy-preserving dependency acquisition: The DAU of the P-SRA client in each cloud-service provider S collects as much dependency information as possible from S, including network dependencies, hardware dependencies, software dependencies, and component-failure probabilities. The DAU stores this information in a local database. Because the P-SRA client is fully controlled by S, and the DAU does not communicate with any other cloud-service providers or the P-SRA host, there is no risk that private information will leak through the DAU.

Step 2: Subgraph abstraction. After data acquisition by the DAU, the SSU processes the dependency information and creates the macro-components to generate the SMPC input according to some abstraction policy. For instance, if cloud-service provider S_1 uses cloud-service provider S_2 as a lower-level infrastructure provider, S_1 can abstract S_2 as a macro-component that its services depend on. The SSU treats macro-components, the number of which is much smaller than total number of components in a cloud-service provider, as individual inputs to the SMPC. We leave the choices of abstraction policies to the cloud-service providers, which can tailor the policies based on the features of their architectures. However, we provide a standard example of subgraph abstraction in Subsection 4.3.

Step 3: SMPC protocol execution and local computation. After the subgraph-abstraction step, the SSUs of the cloud-service providers and the P-SRA host execute the SMPC protocol. The SSU of each cloud-service provider first adds some randomness to conceal statistical information about the input (without changing the output) and splits the randomized input into secret shares. It then establishes connections with the SSUs of other providers and the P-SRA host to execute the SMPC protocol, which identifies common dependency, performs fault-tree analysis, and computes reliability measures in a privacy-preserving manner. Meanwhile, the SSU passes the dependency graphs of the macro-components to the LEU, which performs local computation. The LEU mainly performs fault-tree analysis to obtain minimal FSes of the macro-components. After both the SSU and the LEU finish their execution, the SSU combines the results of the SMPC protocol and local computation to generate the comprehensive outputs for the cloud-service providers and users.

Step 4: Privacy-preserving output delivery. The output of the P-SRA system should satisfy two requirements: preserving privacy of the cloud-service providers and illustrating reliability risk caused by correlated failure. The SRA system of Zhai *et al.* [28] fully reveals all minimal FSes; P-SRA cannot do this, because the full specification of all minimal FSes may compromise the privacy of cloud-service providers. Although P-SRA is flexible in that cloud-service providers can specify the output sets that are most appropriate for them, we recommend some sets of benchmark outputs for cloud-service providers and users. For cloud-service providers, we recommend *common dependency* and *partial failure sets*. For cloud-service users, we recommend *common dependency ratio*, *failure probabilities of relevant cloud services*, and a small set of *top-ranked FSes*. All the outputs are delivered by an SMPC protocol in a privacy-preserving manner. We discuss these recommended outputs in Subsection 4.5.

4.2 Privacy-preserving Data Acquisition

The DAU of each cloud-service provider collects as much information as possible about the components and dependencies of this provider and then stores the information in a local database for later use by the P-SRA's other modules. The DAU can collect network dependencies, hardware dependencies, software dependencies, and failure probabilities of each component. For network dependencies, it collects information about a variety of components in the cloud structure including servers, racks, switches, aggregation switches, routers, and power stations, as well as the connections between these components within the cloud infrastructure and from the cloud infrastructure to the Internet. For hardware dependencies, the DAU inventories the CPUs, network cards, memory, disks, and drivers, and collects product information about each piece of hardware, including vendor, machine life, model number, and uptime. For software dependencies, the DAU analyzes the cloud-service provider's software stacks to determine the correlations between programs within the applications running on servers and the calls and libraries used by these programs. Failure probabilities can be obtained via a variety of methods, including examining the warranty documents of a vendor or searching online based on hardware type and serial number.

The dependency information can be encoded in XML files to store in the local databases of the cloud providers. We use the *topology-path form* to store graph information. The definition of the topology-path form and our reasons for choosing it are given in Subsection 4.4.

4.3 Subgraph Abstraction

Recall that a macro-component is an abstracted (or virtual) node in the dependency graph of a cloud-service provider that can be considered an atomic unit for the purpose of SMPC protocol execution. Creating macro-components allows us to reduce the input-set size to something that is feasible for SMPC execution. A subgraph H of the full dependency graph of a cloud-service provider S should have two properties in order to be eligible for abstraction as a macro-component. First, all components in H must be used only by S; intuitively, this is a "private" part of S's infrastructure. Second, for any two components v and w in H, the dependency information of v with respect to components outside of H is identical to that of w; that is, if v has a dependency relationship (as computed by the DAU) with a component y outside of H, then w has exactly the same dependency relationship with y. (Note that y may be inside or outside of S.) Abstraction of a subgraph that does not satisfy these properties would destroy dependency information that is needed for structural-reliability auditing.

Recall that different cloud-service providers may wish to use different abstraction policies. That is, we do not *require* that all subgraphs that satisfy the two properties given above be abstracted – some providers may wish to use a more stringent definition of a macro-component.

Suppose that G is the full dependency graph of cloud-service provider S and that G contains macro-component H. To transform G into a smaller graph G' via subgraph abstraction of H, S "collapses" H to a single node in G'; that is, S replaces H with a single node, say h, and, for every node y in G but not in H, replaces all dependency relationships in G of the form (w,y,ℓ), where w was a node in H and ℓ is a label that describes the nature of the dependency relationship between w and y, with a single dependency relationship (h,y,ℓ) in G'. (Note that there will, in general, be many nodes y that have no dependency relationships with nodes in H.) Of course, there may be more than one subgraph H that is abstracted before the reduced dependency graph is entered into the SMPC. After S receives the results of the SMPC, it combines them with the results of local fault-tree analysis of the macro-components H. For example, if F is an FS of G', $h \in F$, and f is an FS of H, then $(F-\{h\}) \cup f$ is an FS of G.

As promised, we now provide a standard example in order to illustrate the abstraction process. In this example, the SSU creates a macro-component to represent all of the components in a data center. In most cloud structures, the data centers are eligible for subgraph abstract. First, all the nodes in the data centers are owned and used by exactly one cloud-service provider. Second, all nodes in a data center communicate with the rest of the world only through the data-center gateways; they therefore have identical dependency relationships with components outside of the data center.

Figures 2 and 3 illustrate this process. Suppose that Figure 2 is the full dependency graph of cloud-service provider C_1, which contains a storage-cloud service. C_1's users' files are stored in server S_2, with two backup copies stored in server S_5 and S_7. The components inside the red box belong to a data center DC_1, which has the two properties required for asbtraction. After abstracting both DC_1 and another data-center subgraph, the SSU obtains Figure 3 as the input to the SMPC. After the abstraction process, the SSU executes the SMPC protocol with the abstracted inputs and passes

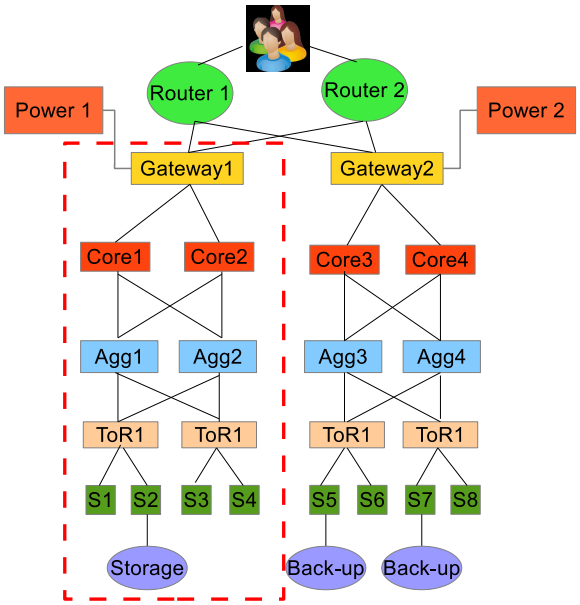

Figure 2: Full Dependency Graph of C_1

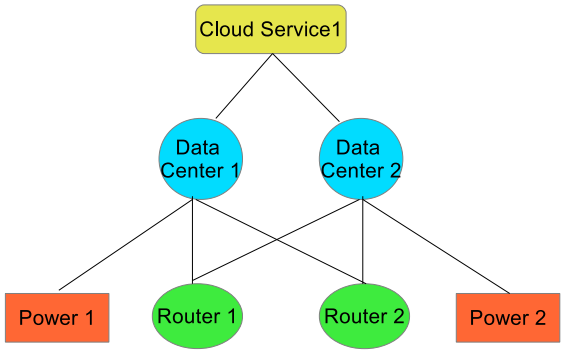

Figure 3: Abstracted Dependency Graph, suitable for SMPC

the dependency information within the macro-components (such as the red box in Figure 2 for DC_1) to the LEU for local computation.

Because the number of components in a data center is often huge, this kind of abstraction can be a crucial step toward the feasibility of SMPC.

4.4 SMPC and Local Computation

4.4.1 SSU Protocol:

Fault-tree construction: Recall that a fault tree contains two kinds of information: dependency information about components (events and links) and logical relationships among components (represented as logic gates). As we said in Subsection 4.2, we use the *topology-path form* to store dependency information. That is, we represent a leveled DAG as a set of (directed) paths in which the first node of each path is the root node of the leveled DAG, the last node is one of the leaf nodes of the leveled DAG, and the other nodes form a path from the root to the leaf. Figure 5 depicts the topology-path form of the dependency graph in Figure 3. The topology-path form of a DAG can, in the worst case, be ex-

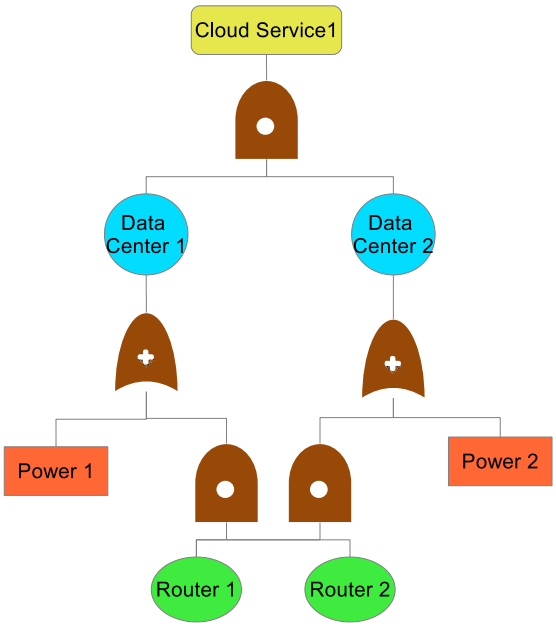

Figure 4: Fault Tree Based on Dependency Graph in Figure 3

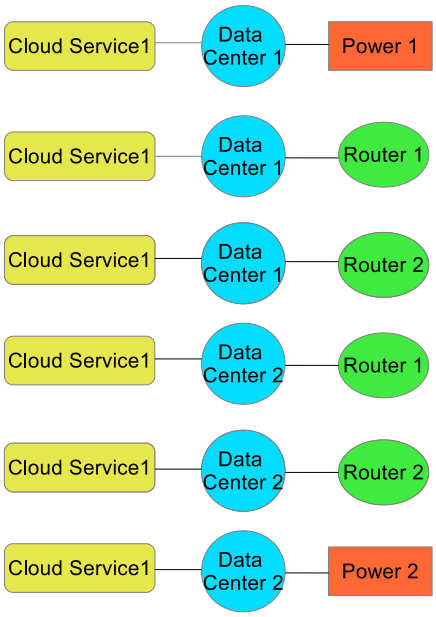

Figure 5: Topology-path Form of Dependency Graph in Figure 3.

ponentially larger than the DAG itself; thus, subgraph abstraction is crucially important, because we need to start with modest-sized DAGs. On the positive side, the topology-path form enables us to avoid using conditional statements in our SecreC code – something we must do to avoid leaking private information.

In order to capture the logical relationships among components of a cloud-service provider, we extend this representation to what we call the *topology-path form with types*. The SSU builds a "disjunction of conjunctions of disjunctions" data structure by assigning different "types" to the topology paths. Failure of the top event in the fault tree is the OR of a set of "type failures"; if any "type" that is an input to this OR fails, then the top event fails. Each "type failure" is the AND of failures of individual topology paths in the type; the "type failure" occurs only if all of the topology paths in that type fail. Failure of a topology path is the OR of failures of individual nodes on the path.

The SSU assigns a "type ID" to each topology path; the type ID is a function of the component IDs in the nodes on the path. Type IDs and the mapping from sets of component IDs to type IDs can be agreed upon by all of the relevant cloud-service providers and stored in a table before the P-SRA execution starts; so the SSU simply needs to look up type IDs during the protocol execution. To construct the fault tree from the topology-path form with types, the SSU traverses each path and constructs an OR gate for each path, the inputs to which are the nodes on the path. It then constructs an AND gate for each type of path, the inputs to which are the outputs of the OR gates of the paths in the type. Finally, the SSU constructs an OR gate whose inputs are the outputs of all the AND gates in the previous step.

For example, starting with the fault tree of Figure 4, the SSU can classify the topology paths of Figure 5 into two types. Type 1 includes the two topology paths ($Cloud\ Service_1$, DC_1, $Power_1$) and ($Cloud\ Service_1$, DC_2, $Power_2$). Type 2 includes the other four topology paths. It can be verified that the minimal FSes of the fault tree generated by the topology path form with types are the same as the minimal FSes of the fault tree in Figure 4; we defer a formal statement and proof of this fact to a future, longer version of this paper that includes all of the necessary details of fault-tree analysis given in [28].

Generate input for the SMPC: After constructing the topology paths with types, the SSU "pads" the paths so that they all have the same length L, where L is an agreed-upon global parameter distributed by the P-SRA host. Padding is accomplished by adding the required number of "dummy" nodes in which the component ID is 0. (Here, "0" is any fixed value that is *not* a valid, real component ID.) Similarly, the SSU adds a random number of "0 paths," which are topology paths with types in which all of the nodes have component ID 0. The types of these 0 paths can be assigned randomly, because they do not affect the result – the 0 paths never fail. The purpose of this padding step is to prevent leakage of structural information about the cloud-service providers' architectures, including the number of topology paths or the size of each path. Finally, the SSU splits the padded paths into secret shares that are input to the SMPC protocol.

Identify common dependencies: A component is in the *common dependency* of cloud-service provider S_i if it is in the fault tree of S_i *and* in the fault tree of at least one other cloud-service provider S_j, $j \neq i$. Conceptually, the common dependency is very easy to compute by doing multiple (privacy-preserving) set intersections, followed by one (privacy-preserving) union. However, we need to do this computation without conditional statements; see Algorithm 1 for a method of doing so.

Calculate failure sets: Finally, the SMPC protocol integrates the fault trees of all participating cloud-service providers into a unified, global fault tree and performs fault-tree analysis. It can execute either algorithm 2, which computes minimal FSes, or algorithm 3, a heuristic "failure-sampling" algorithm that is faster than algorithm 2 and computes FSes but does not guarantee that the FSes returned are minimal.

Algorithm 2 works as follows. Let T denote the unified, global fault tree; because we represent fault trees as padded, topology

Algorithm 1: Common-Dependency Finder

Input: Fault tree T_i, $i = 1$ *to* N, where N is the number of participating cloud-service providers
Output: Common Dependency

foreach T_I *and* $T_J, I \neq J$ **do**
  private mask.clear();
  foreach $node_i \in T_I$ *and* $node_j \in T_J$ **do**
    $private\ mask[i][j] = (node_i.ID == node_j.ID)$;
  private CommonDep.clear();
  foreach $node_i \in T_i$ *and* $node_j \in T_j$ **do**
    $private\ CommonDep[i] = mask[i][j] \times node_j.ID + CommonDep[i]$;
  private CommonDependent.append(CommonDep);
return private CommonDependent;

Algorithm 2: Minimal-FS algorithm

Input: Global Fault tree T
Output: MinimalFS

foreach $private\ path_i \in T$ **do**
  foreach $private\ node_j \in private\ path_i$ **do**
    $private\ path_i.FS.apped(node_j)$;
    /* each path corresponds to an OR gate with input as the nodes along the path */
foreach $AndGate_i \in T$ **do**
  $AndGate_i.FS.clear()$;
  foreach $path_j \in AndGate_i$ **do**
    $AndGate_i.FS \leftarrow AndGate_i.FS \times path_j.FS$;
    /* process the AndGate for each type of topology paths */
    /* FS of $AndGate_i$ is the Cartesian Product of $AndGate_i.FS$ and $path_j.FS$. */
private minimalFS.clear();
foreach $AndGate_i \in T$ **do**
  $minimalFS.append(AndGate_i.FS)$;
  /* process the OR gate connecting to the And Gates */
/* reduce redundant items in $minimumFS$ and assign the result to $minimalFS$, and then simplify minimalFS. */
$minimalFS \leftarrow reduce\_redundancy(minimalFS)$;
$minimalFS \leftarrow simplify(minimalFS)$;
return minimalFS;

Algorithm 3: Failure-Sampling Algorithm

Input: Global Fault tree T and the number of samples N
Output: FSes

private FSes.clear();
for $i \leftarrow 1$ *to* N **do**
  foreach $private\ path_j \in T$ **do**
    private tmp = 0;
    foreach $private\ node_s \in path_j$ **do**
      foreach $private\ node_k \in T$ **do**
        private random = 0 or 1 based on randomly flipping a fair coin;
        $tmp+ = random \times (node_s.ID == node_k.ID)$;
    /* calculate whether $path_j$ fails */
    $path_j.failure = (tmp > 0)$;
  foreach $AndGate_i \in T$ **do**
    $AndGate_i.failure = true$;
    foreach $path_j \in AndGate_i$ **do**
      $AndGate_i.failure = AndGate_i.failure$ && $path_j.failure$;
  private serviceFailure = false;
  foreach $AndGate_i \in T$ **do**
    $serviceFailure = AndGate_i.failure \parallel serviceFailure$;
  open(serviceFailure);
  if *serviceFailure* **then**
    FS.clear();
    foreach $path_i \in T$ **do**
      FS.append($path_i.failure$);
    FSes.append(FS);
return FS;

paths with types, T is simply the union of the fault trees of the individual cloud-service providers. The algorithm traverses T, producing FSes for each of the visited events. Basic events generate FSes containing only themselves, while non-basic events produce FSes based on the FSes of their child events and their gate types. For an OR gate, any FS of one of the input nodes is an FS of the OR. For an AND gate, we first take the cartesian product of the sets of FSes of the input nodes and then combine each element of the cartesian product into a single FS by taking a union. The last step of algorithm 2 reduces the top event's FSes to minimal FSes.

Algorithm 3 works as follows. For each sampling round, the algorithm randomly assigns 1 or 0 to the basic events (leaves) of the fault tree T, where 1 represents failure and 0 represents non-failure. Starting from such an assignment, the algorithm can assign 1s and 0s to all non-basic events in T, using the logic gates. At the end of each sampling round, the algorithm checks whether the top event fails. If the top event fails, then the failure nodes in this sampling round are an FS. The algorithm runs for a large number of sampling rounds to find FSes. In [28], it is proven that most of the critical FSes can be found in this fashion but that the FSes are not necessarily minimal.

4.4.2 *LEU Protocol:*

The LEU in the P-SRA client of cloud-service provider S performs fault-tree analysis on S's macro-components. The LEU can use algorithm 2 or algorithm 3. Note that these computations are done locally and do not involve SMPC; so, large macro-components are not necessarily bottlenecks in P-SRA computation. It is very advantageous when a cloud-service provider can partition its infrastructure in a way that produces a modest number of large macro-components, each one of which is a "virtual node" in the SMPC.

4.5 Privacy-preserving Output Delivery

Recall that P-SRA performs an SMPC on dependency information that is potentially shared by multiple cloud-service providers and performs local computation on dependency information that is definitely relevant to only one provider. The intermediate results include common dependency and minimal FSes (or FSes if algorithm 3 was used). We now turn our attention to the outputs that P-SRA delivers to cloud-service providers and to cloud-service users. P-SRA gives cloud services the flexibility to choose exactly what should be output. However, we argue that the outputs should not compromise the privacy of cloud-service providers and must be illustrative of correlated-failure risk and reliability. We propose some specific outputs that satisfying these two requirements.

4.5.1 *Output for Cloud-Service Providers*

Common dependency: The common dependency set, as defined in Subsection 4.4, includes components shared by more than one cloud-service provider. It is useful for cloud-service providers, in that it can make them aware of unexpected correlation with other

providers. They can then deploy independent components as backups to mitigate the impact of the common dependency or switch to independent components to improve the reliability of their service and decrease the correlation with other cloud-service providers.

Partial failure sets: If F is a (minimal) FS for cloud-service provider S, then the corresponding partial (minimal) FS is simply all of the components in F that are used by S. Such a partial FS gives S information about components whose failure may lead to an outage because equipment that is controlled by some other service provider fails. If S can build enough redundancy into its internal infrastructure to avoid failure of all of the components in this partial FS, then it will not suffer an outage because of F, regardless of what happens outside.

Sometimes the number of FSes is huge. If this is the case, we need to rank the FSes first and only output the partial failure sets of the top-ranked FSes. Ranking of comprehensive failure sets can be either probability-based or size-based [28].

4.5.2 Output for Cloud-Service Users

Common-dependency ratio: Cloud-service users can obtain a *common-dependency ratio* for each cloud-service provider. We define the common-dependency ratio of cloud-service provider S as the fraction of components in S that are shared with at least one other cloud-service provider. Intuitively, the larger the common-dependency ratio, the higher the risk of correlated failure. In the extreme case, if a cloud service is deployed entirely on an external cloud infrastructure (as is the case with some Software-as-a-Service providers), then its common-dependency ratio is 1. If a cloud-service provider shares no components with other providers, then its common-dependency ratio is 0. Cloud-service users can evaluate risk and choose cloud providers in part based on this ratio. This common-dependency ratio does not reveal any information about internal architecture of the cloud providers.

Overall failure probabilities of cloud services: Cloud-service users can compare these failure probabilities with the reliability measures promised by the providers in their service-level agreements and evaluate whether they are subject the risk of unexpected, correlated failure. Failure probabilities, like common-dependency ratios, do not reveal the architectures of the service providers.

Top-ranked failure sets: Recall from Subsection 2.1 that, in its SMPC, P-SRA computes the *secret shares* of the (minimal) FSes of the cloud-service providers. As we have seen, an SMPC program can compute from those shares the partial (minimal) FSes that are delivered to the providers. However, an alternative SMPC program could use those shares to rank the (minimal) FSes based on failure probability or size. Then a small set of *top-ranked* (minimal) FSes can be delivered to cloud-service users. Just a few top-ranked sets can give users useful information about how to avoid correlated failures; they reveal some information about the cloud-service architectures, but this may be tolerable in some markets.

5. IMPLEMENTATION

5.1 P-SRA Prototype

The Sharemind SecreC platform includes a set of **miners** to execute the SMPC protocols and a controller to coordinate the miners. The SMPC protocols run by the miners are coded in **SecreC**, a C-like programming language for SMPC programs. Variables in SecreC may be declared **public** or **private**. The language supports basic arithmetic, and some matrix and vector operations. SecreC uses a client/server model, with multiple clients providing (secret-shared) input to the miners, which execute the SMPC protocol.

Our implementation of P-SRA is illustrated in Figure 6. The miners are installed in the SMPC module of the P-SRA host. The P-SRA clients and P-SRA host upload their SecreC scripts to the miners. The SecreC scripts are executed by the P-SRA clients remotely through the C++ interface of the controller or by the P-SRA host locally. The P-SRA clients execute the SecreC scripts to split their inputs into secret shares and to read and write shares of inputs or intermediate results from the miners' secure databases. The P-SRA host executes the SecreC scripts to perform the SMPC protocol that identifies common dependencies and performs fault-tree analysis. SecreC uses SSL for secure communication between miners and clients.

From Figure 6, it is not immediately obvious what one gains from using the Sharemind platform and SMPC instead of a trusted-party SRA as in [28]: All of the miners, *i.e.*, the nodes that execute the SMPC protocol, run inside the P-SRA host; if they share information, then together they constitute a trusted party. However, this system configuration is merely the default of the currently available Sharemind "demo," and we have used it only in order to be able to build this proof-of-concept prototype as quickly as possible. In a real, deployed P-SRA (or any real SMPC-based application coded in SecreC), the miners would run on separate, independently administered machines and communicate over a network; no substantive changes to the SecreC compiler are needed to create executables that run on separate networked nodes, and we expect future Sharemind releases to create them. Thus, moving to P-SRA from the SRA of Zhai *et al.* [28], in which one trusted auditor handles all of the sensitive information supplied by the cloud-service providers, is tantamount to "distributing trust" over a number of independently administered auditors no one of which is trusted with any sensitive information, in the sense that each receives only a secret share of every input; if the independent owners of the networked nodes that run the auditors do not collude, then the clients' inputs will remain private. This SMPC architecture, in which clients (or "input providers"), rather than executing an SMPC protocol themselves, instead send their input shares to independently administered computational agents that then execute the SMPC protocol, is known as *secure outsourcing* in the SMPC literature; see, *e.g.*, Gupta *et al.* [12] for more information about secure outsourcing's history, its practical advantages, and its use in a routing application.

The SecreC compiler relieves programmers of the need to code standard cryptographic functionality. In particular, it generates secret-sharing code automatically. Currently, it uses additive secret sharing and thus guarantees privacy only against honest-but-curious adversaries. We expect future releases to incorporate more elaborate secret-sharing techniques and hence to protect input providers against stronger classes of adversaries.

The DAU and LEU in the P-SRA client are written in Python. The DAU uses the SNMPv2 library support from NetSNMP to collect network dependencies; it uses lshw, a lightweight tool that extracts detailed hardware configuration from the local machines, to collect hardware dependencies; it uses ps and gprof to collect software dependencies. The LEU uses the Network-X library [2] to process the dependency-graph data structures.

5.2 Case Study

This section outlines a case study to illustrate the prototype's operation. Let CS_1 denote a cloud service provided by cloud provider C_1. To improve the reliability of CS_1, C_1 decides to use providers C_2 and C_3 for redundant storage. Only C_1 serves users directly, while C_2 and C_3 provide lower-level services to C_1. This architec-

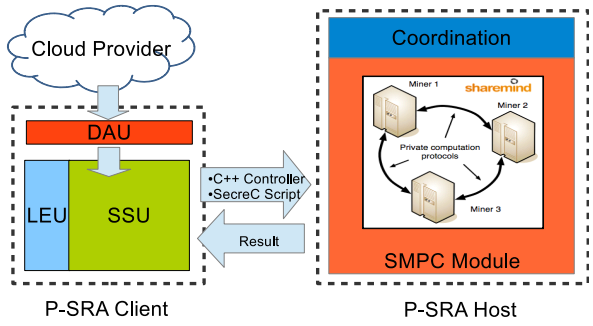

Figure 6: Implementation in Sharemind SecreC

ture is analogous to iCloud, Apple's storage service, which uses Amazon EC2 and Microsoft Azure for redundant backup storage.

Suppose Alice, a user of CS_1, wants to deploy a MapReduce function using CS_1. Alice deploys the MapReduce Master on a data center DC_1 of C_1, and C_1 uses a data center DC_2 of C_2 and a data center D_3 of C_3 as backup for the MapReduce Master. However, as in Figure 7, C_1, C_2, and C_3 depend on the same power station P_1. Alice and all three cloud providers are unaware of this situation. Therefore, they may overestimate the reliability of the MapReduce Master and underestimate the risk of correlated failure. If P_1 goes down, Alice's MapReduce may not work, because all the backup data centers may fail simultaneously.

The P-SRA system can help to identify P_1 as the common dependency in the cloud structure supporting CS_1 and provide multiple measures of reliability and correlated failure risk (the failure probability for Alice and partial FSes for C_1), without revealing significant private information about C_1, C_2, and C_3. Alice need not learn private topological information about the three cloud providers (or even learn of the existence of C_2 and C_3) but can accurately assess the failure risk via the P-SRA system. Meanwhile, C_1 can improve the reliability of CS_1 by connecting to alternative power stations or seeking redundancy from cloud providers other than C_2 and C_3, without learning private topological information about C_2 and C_3.

To further illustrate P-SRA, we display the details within a data center. There are a large number of components in data centers including servers, racks, switches, aggregate switches and routers. For simplicity, we generate the same topology for all the data centers and show only the components in DC_1 – see Figure 8. The MapReduce Master is installed on server 5 of DC_1. The DAUs of C_1, C_2, and C_3 collect the dependency information of each cloud provider. Then the SSUs abstract macro-components for each cloud provider using standard data-center abstraction. The SSUs pass the information within the data centers to the LEUs and establish connections with each other and the P-SRA host to execute the SMPC protocol. The LEUs perform fault-tree analysis on the dependency information within the data centers locally. The results of the SMPC and the local computation are then combined as explained in Subsection 4.3.

The P-SRA system is practical in this case study. Even using a laptop with little computational power, equipped only with a 2.5GHz 2-core Intel i5 CPU and 2.00GB of memory, the running time used by the SSUs and P-SRA host to find the common dependency was approximately 20 seconds; the time to perform the fault-tree analysis was approximately 13 minutes using the minimal-FS algorithm and 55 seconds using the failure-sampling algorithm with 100 rounds. The running time for the LEUs deployed on

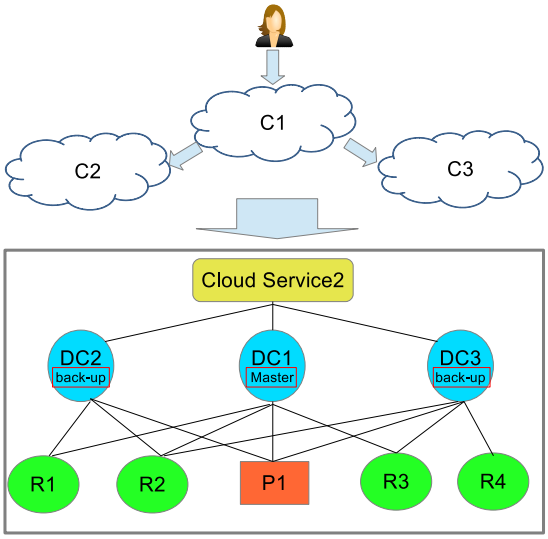

Figure 7: Multi-level Structure of Cloud Service

servers equipped with two 2.8GHz 4-core Intel Xeon CPUs and 16GB of memory was less than 30 seconds for both the minimal-FS algorithm and the failure-sampling algorithm.

5.3 Large-Scale Simulation

This section evaluates the P-SRA prototype using larger-scale simulations. Our data set is synthesized based on the widely accepted three-stage fat-tree cloud model [15] and scaled up to what we expect to find in real cloud structures. For the SMPC protocol run by the P-SRA host and the SSUs of P-SRA clients, we test the running time of the common-dependency-finder Algorithm 1, the minimal-FS Algorithm 2, and the failure-sampling Algorithm 3. Our output for both cloud-service providers and users can be computed efficiently from the common dependency and the (minimal) FSes.

We test the five cases summarized in Table 1. For simplicity, we generate only homogeneous cloud providers. In Table 1, the numbers of data centers, Internet routers, and power stations are numbers per cloud provider. The common-dependency ratio is as defined in Subsection 4.5.2. The padding ratio is the number of zeros with which the topology paths were padded divided by the total number of nodes on the topology paths after padding.

The five cases are intended to be illustrative of configurations broadly comparable to realistic multi-cloud services. To the best of our knowledge, it is uncommon for any cloud services to be deployed on more than three cloud providers or distributed over more than 10 data centers, because the total number of data centers worldwide is limited, and cloud-service management costs increase quickly as data centers are added. Amazon, one of the giant cloud providers, owns only 15 data centers globally [1]; Microsoft Azure has fewer than 10 data centers [3].

Figure 9 summarizes measured P-SRA computation performance. The P-SRA host and SSUs of the P-SRA clients were run on laptops with 2.5GHz 2-core Intel i5 CPU and 2.00GB of memory. We used these machines because the SecreC platform supported only Microsoft Windows when we started this work. We expect that performance would improve using higher-powered machines.

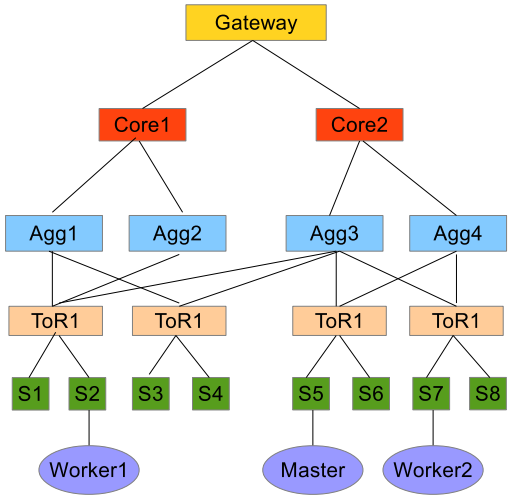

Figure 8: Components in Data Center DC_1: Core, Agg, and ToR represent core router, aggregation switch, and top-of-rack switch.

	Case 1	Case 2	Case 3	Case 4	Case 5
# of cloud providers	2	2	3	3	2
# of data center	1	3	8	10	3
# of internet router	3	5	10	15	5
# of power stations	1	2	3	5	2
ratio of common dep.	0.8	0.2	0.2	0.2	0.2
ratio of padding	0.0	0.0	0.0	0.0	0.5

Table 1: Configuration of Test Data Sets

The common-dependency finder exhibits reasonable efficiency in all five cases, the runtimes of which are all less than 3 minutes. The minimal-FS algorithm yields exact minimal FSes (but takes exponential time in the worst case, because the problem in NP-hard), while the failure-sampling algorithm produces FSes approximating the minimal FSes and runs in polynomial time. In Cases 4 and 5, the minimal-FS algorithm was aborted before it finished, and thus no results are shown for them in Figure 9. The runtimes of other simulations of the minimal-FS algorithm and the failure-sampling algorithm range from 1 to 50 hours depending on the configuration. As the number of nodes increases, the efficiency of fault-tree analysis drops quickly. Case 5 shows that the cost of padding to conceal the statistical information of each topology path is high. Therefore, subgraph abstraction to reduce the size of the dependency graphs is important for the efficiency of fault-tree analysis in P-SRA.

For the LEUs in the P-SRA clients performing local computations, we also test the running times of both the minimal-FS algorithm and the failure-sampling algorithm. For the LEUs running on servers with two 2.8GHz 4-core Intel Xeon CPUs and 16GB of memory, the failure-sampling algorithm with 10^6 rounds on a data center with 13,824 servers and 3000 switches takes around 6 hours. For details, see Table 2. "FS round 10^n" denotes the runtime (in minutes) of the failure-sampling algorithm running 10^n rounds. "Minimal FS" denotes the runtime of the minimal-FS algorithm.

6. CONCLUSIONS AND FUTURE WORK

We have designed P-SRA, a private, structural-reliability auditor for cloud services based on secure, multi-party computation, and

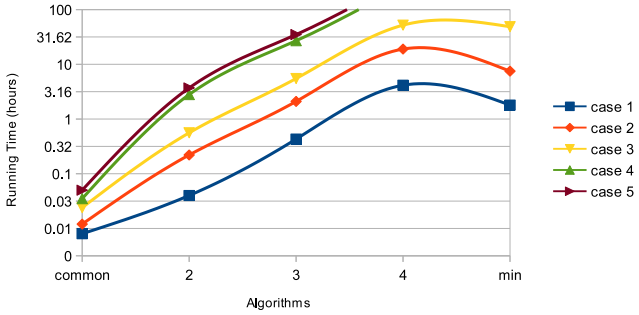

Figure 9: Performance of algorithms. On the X axis, "Common" represents the common-dependency finder, 2 through 4 represent the failure-sampling algorithm with sampling rounds at various powers of 10, and "Min" represents the minimal-FS algorithm.

Table 2: Performance of the LEU of a P-SRA client

Configuration	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 # components	40	216	1360	4200	16752
Runtime (minutes)					
FS round 10^3	< 0.7	< 0.7	< 0.7	< 0.7	< 0.7
FS round 10^4	0.7	0.7	1.7	2.3	6.9
FS round 10^5	0.8	0.9	5.3	28.1	6.9
FS round 10^6	1.7	4.5	65.0	243.5	462.9
FS round 10^7	28.3	56.6	512.1	NA	NA
Minimal FS	0.8	14.8	309.7	NA	NA

prototyped it using the Sharemind SecreC platform. In addition, we have explored the use of data partitioning and subgraph abstraction in secure, multi-party computations on large graphs, with promising results. Our preliminary experiments and simulations indicate that P-SRA could be a practical, off-line service, at least for small-scale cloud services or for ones that permit significant subgraph abstraction. There are many interesting directions for future work, including: (1) Although our preliminary experiments indicate that the cost of privacy in structural reliability auditing (*i.e.*, the additional cost of using P-SRA instead of SRA) is not prohibitive, it would be useful to measure this cost more precisely with more exhaustive experiments. (2) It will be interesting to seek more efficient algorithms for fault-tree analysis and/or a more efficient P-SRA implementation; both would enable us to test P-SRA on larger cloud architectures. (3) Note that we assumed, following Zhai *et al.* [28], that dependency graphs of cloud services are acyclic, but they need not be. Tunneling-within-tunneling of the type already in use in MPLS and corporate VPNs could (perhaps unintentionally) create cyclic dependencies if used in clouds. Thus, it will be worthwhile to develop structural-reliability auditing techniques that apply to cyclic dependency graphs. (4) P-SRA partitions components based on the fact that some physical equipment is used by exactly one service provider and hence cannot cause the failure of another provider's service, but this type of partitioning has limitations. If, for example, two cloud-service providers purchase large numbers of hard drives of the same make and model from the same batch, and that batch is discovered to be faulty, then the two services have

a common dependency on this faulty batch of drives. P-SRA's data partitioning could hide this common dependency, because the hard drives could be considered "private" equipment by both services. It will be worthwhile to extend P-SRA so that it can discover this type of common dependency while retaining the efficiency provided by data partitioning and subgraph abstraction.

7. ACKNOWLEDGEMENTS

We thank Ennan Zhai, Aaron Segal, Debayan Gupta, and the anonymous reviewers for their helpful comments. This material is based on research sponsored by NSF grants CNS-1016875 and CNS-1149936, ONR grant N00014-12-1-0478, DARPA contract FA8750-13-2-0058, and a gift from Google Research.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, ONR, DARPA, or the U.S. Government.

8. REFERENCES

[1] Amazon web services global infrastructure. `http://aws.amazon.com/en/about-aws/globalinfrastructure/`.

[2] NetworkX. `http://networkx.github.com/`.

[3] Windows azure. `http://en.wikipedia.org/wiki/Windows_Azure`.

[4] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM Symposium on Computer and Communication Security*, pages 257–266, 2008.

[5] S. Bleikertz, M. Schunter, C. W. Probst, D. Pendarakis, and K. Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *ACM Cloud Computing Security Workshop*, pages 93–102, 2010.

[6] D. Bogdanov and A. Kalu. Pushing back the rain – how to create trustworthy services in the cloud. *ISACA Journal*, 3:49–51, 2013. Available at `http://www.isaca.org/Journal/Past-Issues/2013/Volume-3/Pages/default.aspx`.

[7] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, 2010.

[8] B. Butler. Cloud storage viable option, but proceed carefully, 2013. Available at `http://www.networkworld.com/news/2013/010313-gartner-storage-265460.html`.

[9] B. Butler. Top 10 cloud storage providers, 2013. Available at `http://www.networkworld.com/news/2013/010313-gartner-cloud-storage-265459.html`.

[10] I. Damgård, M. Geisler, M. Krøigård, and J. Nielsen. Asynchronous multiparty computation: Theory and implementation. In S. Jarecki and G. Tsudik, editors, *Public Key Cryptography – PKC 2009*, pages 160–179. Springer Verlag, LNCS 5443, 2009.

[11] C. A. Ericson II. *Hazard analysis techniques for system safety*. John Wiley and Sons, 2000.

[12] D. Gupta, A. Segal, A. Panda, G. Segev, M. Schapira, J. Feigenbaum, J. Rexford, and S. Shenker. A New Approach to Interdomain Routing Based on Secure Multi-Party Computation. In *ACM SIGCOMM Workshop on Hot Topics in Networks*, 2012.

[13] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: tool for automating secure two-party computations. In *ACM Conference on Computer and Communications Security*, pages 451–462, 2010.

[14] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – a secure two-party computation system. In *USENIX Security Symposium*, pages 298–302, 2004.

[15] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *ACM SIGCOMM*, pages 39–50, 2009.

[16] W. Oremus. Internet outages highlight problem for cloud computing: Actual clouds, 2012. Available at `http://www.slate.com/blogs/future_tense/2012/07/02/amazon_ec2_outage_netflix_pinterest_instagram_down_after_aws_cloud_loses_power.html`.

[17] M. A. Shah, M. Baker, J. C. Mogul, and R. Swaminathan. Auditing to keep online storage services honest. In *USENIX Workshop on Hot Topics in Operating Systems*, 2007.

[18] M. A. Shah, R. Swaminathan, and M. Baker. Privacy-preserving audit and extraction of digital contents. Cryptology ePrint Archive, Report 2008/186, 2008. Available at `http://eprint.iacr.org/2008/186/`.

[19] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. US Nuclear Regulatory Commission, 1981.

[20] C. Wang, S. S. M. Chow, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers*, 62(2):362–375, 2013.

[21] C. Wang, K. Ren, W. Lou, and J. Li. Toward publicly auditable secure cloud data storage services. *IEEE Network*, 24(4):19–24, 2010.

[22] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *IEEE INFOCOM*, pages 525–533, 2010.

[23] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):847–859, 2011.

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

[25] A. C. Yao. Protocols for secure computation. In *IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[26] A. C. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[27] E. Zhai, R. Chen, D. I. Wolinsky, and B. Ford. An untold story of redundant clouds: Making your service deployment truly reliable. In *ACM Workshop on Hot Topics in Dependable Systems*, 2013.

[28] E. Zhai, D. I. Wolinsky, H. Xiao, H. Liu, X. Su, and B. Ford. Auditing the structural reliability of the clouds. Technical Report YALEU/DCS/TR-1479, July 2013. Available at `http://www.cs.yale.edu/publications/techreports/tr1479.pdf`.