An Auditing Language for Preventing Correlated Failures in the Cloud[*]

ENNAN ZHAI, Yale University, USA
RUZICA PISKAC, Yale University, USA
RONGHUI GU, Columbia University, USA
XUN LAO, Yale University, USA
XI WANG, Yale University, USA

Today's cloud services extensively rely on replication techniques to ensure availability and reliability. In complex datacenter network architectures, however, seemingly independent replica servers may inadvertently share deep dependencies (*e.g.*, aggregation switches). Such unexpected common dependencies may potentially result in correlated failures across the entire replication deployments, invalidating the efforts. Although existing cloud management and diagnosis tools have been able to offer post-failure forensics, they, nevertheless, typically lead to quite prolonged failure recovery time in the cloud-scale systems. In this paper, we propose a novel language framework, named RepAudit, that manages to prevent correlated failure risks *before* service outages occur, by allowing cloud administrators to proactively audit the replication deployments of interest. In particular, RepAudit consists of three new components: 1) a declarative domain-specific language, RAL, for cloud administrators to write auditing programs expressing diverse auditing tasks; 2) a high-performance RAL auditing engine that generates the auditing results by accurately and efficiently analyzing the underlying structures of the target replication deployments; and 3) an RAL-code generator that can automatically produce complex RAL programs based on easily written specifications. Our evaluation result shows that RepAudit uses 80× less lines of code than state-of-the-art efforts in expressing the auditing task of determining the top-20 critical correlated-failure root causes. To the best of our knowledge, RepAudit is the first effort capable of simultaneously offering expressive, accurate and efficient correlated failure auditing to the cloud-scale replication systems.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; *Fault tree analysis*; • **Computer systems organization** → Reliability;

Additional Key Words and Phrases: Cloud reliability, Correlated failures, Cloud auditing, Fault tree analysis

ACM Reference Format:
Ennan Zhai, Ruzica Piskac, Ronghui Gu, Xun Lao, and Xi Wang. 2017. An Auditing Language for Preventing Correlated Failures in the Cloud. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 97 (October 2017), 27 pages. https://doi.org/10.1145/3133921

[*]This research was sponsored by the NSF under grants CCF-1302327 and CCF-1715387.

Authors' addresses: Ennan Zhai, Computer Science, Yale University, 51 Prospect St. New Haven, CT, 06511, USA; Ruzica Piskac, Computer Science, Yale University, 51 Prospect St. New Haven, CT, 06511, USA; Ronghui Gu, Computer Science, Columbia University, New York, NY, 10027, USA; Xun Lao, Computer Science, Yale University, 51 Prospect St. New Haven, CT, 06511, USA; Xi Wang, Computer Science, Yale University, 51 Prospect St. New Haven, CT, 06511, USA.

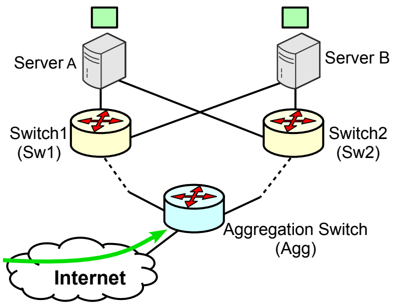


Fig. 1. An example for our target problem. Once the shared aggregation switch fails, the replicated states (green boxes) would become unavailable simultaneously.

1 INTRODUCTION

Today's cloud computing systems typically ensure availability and reliability through replication techniques, *i.e.*, replicating the important data and states across multiple nodes [Bessani et al. 2011; Bonvin et al. 2010]. However, complex multi-layered datacenter structures may unwittingly introduce dependencies shared by seemingly independent infrastructure components, thus resulting in unexpected correlated failures across the entire replication systems. Figure 1 illustrates a typical scenario where correlated failures occur: suppose an administrator deploys a replication system on two servers *A* and *B*. Unbeknownst to her, if servers *A* and *B* share a deep aggregation switch, *Agg*, a glitch of this common switch will result in correlated failures making *A* and *B* become unavailable *simultaneously*, causing the entire system to fail.

This example, while simplistic, nevertheless illustrates pervasive documented failures. In a recent Rackspace outage report [Steven 2014], a glitch occurred on a core switch in the Rackspace cloud caused multiple servers to be unaccessible, thus making customers fail to access their data and backups in Rackspace. In another example, Amazon AWS reported that a connection glitch occurred in Amazon Elastic Block Store (EBS) service for one of availability zones [The AWS Team 2012]. This failure resulted in Relational Database Storage (RDS) service and its backups failing as well, due to their dependencies on EBS. A recent Microsoft service measurement study [Wu et al. 2012] has revealed that service outages resulting from unexpected network-level common dependencies account for the largest proportion (38%) among other types in Microsoft datacenters. Furthermore, Gunawi *et al.* analyzed 242 public cloud outage events in news reports, and revealed that 54 publicly-known outage events were caused by network component failures [Gunawi et al. 2016]. The problem of correlated failures was even discussed in a popular news magazine [Bradbury 2016] recently, where the reporters concluded, after the discussions with the most prominent researchers in the area, that not enough work was done in this field.

For cloud administrators, discovering or avoiding such unexpected common dependencies has been known as an important but challenging problem in current enterprise-scale datacenters [Ford et al. 2010]. Many diagnostic and troubleshooting tools are proposed to localize such failures *after* service outages occur [Bahl et al. 2007; Chen et al. 2017, 2016, 2008; Cohen et al. 2004; Kandula et al. 2009; Kompella et al. 2005; Leners et al. 2011; Reynolds et al. 2006; Wu et al. 2014; Zhou et al. 2011a,b]. However, these post-failure forensics require significant human interventions, which typically lead to quite prolonged failure recovery time [Wu et al. 2012]. Google has estimated that the majority of failures in Google cloud are truly correlated, but their engineers usually spent many hours on identifying the root causes due to the complexity of datacenter [Ford et al. 2010].

We propose a fundamentally different approach to this dilemma. Rather than localizing or diagnosing failures after they occurred—which is too late, we aim to enable cloud administrators to *proactively* understand and prevent correlated failure risks *before* the service outages occur.

In particular, this paper presents a novel framework, named RepAudit. RepAudit enables cloud administrators to easily express heterogeneous auditing tasks, such as ranking all the components underlying the given replication deployment based on their relative importance, or calculating the failure probability. Additionally, RepAudit can also suggest to the administrator which servers should be selected so that the replication deployment has the lowest correlated failure risks. Using RepAudit, the cloud administrator can better understand the correlated failure risk situations, before service outages occur.

Building a practical and usable RepAudit, nevertheless, requires addressing several key challenges. First, it is non-trivial for administrators to express auditing tasks for diverse purposes. Currently administrators either write analysis scripts manually [Huang et al. 2015], which is error-prone and tedious, or they adapt some of existing tools to meet specific purposes, *e.g.*, INDaaS [Zhai et al. 2014]. These efforts, unfortunately, are not only *ad hoc* and difficult to support heterogeneous auditing tasks, but also need administrators to understand complex underlying structures of replication systems. To address this issue, we propose a declarative domain-specific auditing language, RAL, which is intuitive for administrators to express their auditing tasks by writing, but it is also abstract enough to hide the details of system underlying structures. Our evaluation result shows that RepAudit uses 80× less lines of code than state-of-the-art efforts [Zhai et al. 2014] in expressing the auditing task of determining the top-20 critical correlated-failure root causes.

Inside the auditing engine, the underlying structural dependency information is modeled as fault graph [Zhai et al. 2014]. A fault graph is a data structure representing a system as a Directed Acyclic Graph (DAG) with logical gates. For the modern datacenter networks with tens or hundreds of thousands of communication components, it is challenging to develop efficient analysis algorithms. To solve this issue, we propose a collection of novel and scalable fault graph analysis algorithms, which are heavily relying on modern SAT solvers. We use a reduction of fault graph analysis to the weighted partial MaxSAT (WP-MaxSAT) problem [Alviano et al. 2015]. Specifically, we model a fault graph into a Boolean formula and assign failure probability of each component in the fault graph as the weight of the corresponding variable in the Boolean formula. Because a WP-MaxSAT solver can efficiently compute the top-k satisfiable assignments with the maximum weights (*i.e.*, highest failure probabilities in our case), such a reduction significantly speeds up the fault graph analysis with guaranteed accuracy. We employ these analysis algorithms as primitives of the RAL auditing engine, thus enabling RepAudit to offer cloud administrators efficient and accurate auditing capabilities even in a cloud-scale replication system. For example, our evaluation results show that RepAudit can determine the top-20 critical correlated failure root causes in a replication system containing 30,528 devices within 20 minutes.

In summary, this paper makes the following contributions:

- RepAudit is the first practical a framework that allows cloud administrators to conveniently express their auditing tasks.
- RepAudit offers a new declarative domain-specific language, RAL (Sec. 4).
- We construct the primitives of RAL by leveraging weighted partial MaxSAT solver, which significantly benefits both accuracy and efficiency of RepAudit's auditing engine (Sec. 5).
- RepAudit offers an automatic RAL-code generator that can output complex auditing programs satisfying easily-written specifications (Sec. 6).

- We implement a RepAudit prototype (Sec. 7), and evaluate its expressiveness, accuracy and efficiency based on large-scale datasets (Sec. 8).

2 MOTIVATING EXAMPLES

In this section, we present four examples to illustrate how can RepAudit be applied in practice. All the examples are coming from real-world cloud failure and management events [Gill et al. 2011; Gunawi et al. 2016]. In order to highlight the use of RAL and clarify how RAL can help the administrators, we simplify these scenarios and their corresponding datacenter network topology. This simplified replication deployment with only two replica servers is given in Figure 2a. More realistic examples are given in Sec. 8.

Example 1: Finding and ranking of risk component groups. A *risk component group* [Kaminow and Koch 1997; Zhai et al. 2014], or RCG, is a set of components whose simultaneous failures can cause the failure of the entire replication deployment. For example in Figure 1, {Agg} and {Sw1, Sw2} are two RCGs. One of the main concerns for the cloud administrator is to find such potential RCGs, and understand which ones are the most critical [Ford et al. 2010; Zhai et al. 2014]. Once understanding potential RCGs, the cloud administrator can easier decide if the current deployment is acceptable or should it be changed.

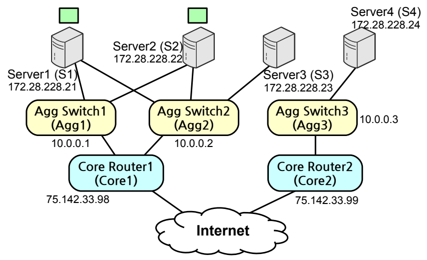
Given the replication deployment shown in Figure 2a, to get the information on critical RCGs, the cloud administrator writes the following auditing program in RAL language:

```
let s1 = Server("172.28.228.21");
let s2 = Server("172.28.228.22");
let rep = [s1, s2];
let ft = FaultGraph(rep);
let list = RankRCG(ft, 2, SIZE);
print(list);
```

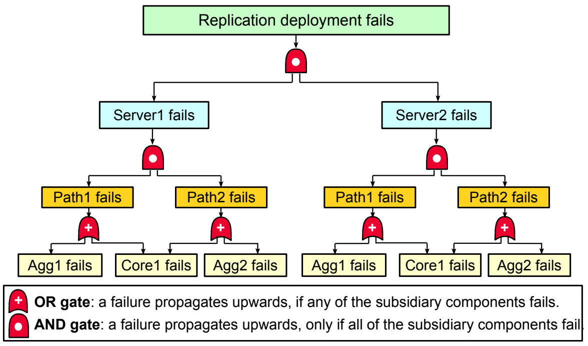
The first two lines specify which replica servers the administrator is interested in. In RAL, we can specify replica servers using their IP addresses, but we also support other ways, *e.g.*, using server identifiers. The variable `rep` is a list of servers representing the target replication deployment. If this list is large, RAL also supports the use of the ellipsis operator, as in example `[s1,...,s9]`. `FaultGraph(rep)` is a command that builds a data structure, called fault graph [Kaminow and Koch 1997; Zhai et al. 2014]. This fault graph contains all the components and their dependencies underlying servers $S1$ and $S2$, as depicted in Figure 2b. We detail more about fault graph in Sec. 3. The administrator cannot observe a fault graph directly due to its size. Instead, the administrator calls `RankRCG(ft, 2, SIZE)` that takes the generated fault graph as input and ranks RCGs. In this call, the administrator asks for two highest ranked critical RCGs and the results should be ranked based on their sizes (specified by `SIZE`). The above program outputs the following ranking list.

```
1. {Core-Router-1["75.142.33.98"]}
2. {Agg-Switch-1["10.0.0.1"], Agg-Switch-2["10.0.0.2"]}
```

This ranking shows to the cloud administrator that the most critical RCG is `{Core-Router-1}`. Identifying RCGs with fewer components—especially any of size 1, indicating no redundancy—can point to areas of the system that require a closer manual inspection. While size-based ranking does not distinguish which potential component failures are more likely to happen, there is also ranking based on failure probabilities. This option is working under the assumption that failure probabilities of components could be obtained [Gill et al. 2011]. To do that the administrator is using parameter `PROB` instead of `SIZE`.



(a) An example datacenter network topology. Green boxes are used to denote replica data.


Replication deployment fails
Server1 fails
Server2 fails
Path1 fails
Path2 fails
Path1 fails
Path2 fails
Agg1 fails
Core1 fails
Agg2 fails
Agg1 fails
Core1 fails
Agg2 fails
OR gate: a failure propagates upwards, if any of the subsidiary components fails.
AND gate: a failure propagates upwards, only if all of the subsidiary components fail.

(b) An example fault graph. This fault graph models our example replication deployment.

Fig. 2. Our motivating example. This example represents a two-way replication deployment, which means cloud administrator replicates each data across two servers (called replica servers).

For the Rackspace outage example [Steven 2014] described in Sec. 1, a glitch occurred on a core switch in Rackspace, disabling many replica servers. If the Rackspace administrators had applied the above method, they would have noticed that this switch is a critical RCG.

Example 2: Computing the failure probabilities of the target replication deployments. The administrators should be able to estimate the failure probabilities of their replication deployments [Ford et al. 2010; Nath et al. 2006]. To do that they should know the failure probabilities of individual infrastructure components in datacenters, which can be successfully extracted via daily log information [Gill et al. 2011]. RepAudit introduces a primitive `FailProb()` to achieve this goal. Note that even though all the components' failure probabilities are known, efficiently computing the entire deployment's failure probability is challenging, because different servers depend on multiple overlapping network components that may have deep and complex dependencies layer by layer. Details on how to efficiently compute the failure probability are given in Sec. 5.3. The following program computes the failure probability for replication deployment given in Figure 2a.

```
let s1 = Server("172.28.228.21");
let s2 = Server("172.28.228.22");
let rep = [s1, s2];
let ft = FaultGraph(rep);
let prob = FailProb(ft);
print(prob);
```

If we assume the failure probability of every device in the Figure 2a network topology is 0.2, then the above program returns that the failure probability of the target replication deployment is 0.232. Clearly, such a high failure probability is not acceptable in a practical scenarios. Thus, the administrator can submit a RAL-program to rank individual devices by quantifying their relative importance. The following program: `let list = RankNode(ft); print(list);` returns the list:

```
1. Core-Router-1 ["75.142.33.98"]
2. Agg-Switch-1 ["10.0.0.1"]
2. Agg-Switch-2 ["10.0.0.2"]
```

The returned result indicates that the most critical device in the given scenario is `Core-Router-1`. Note that `RankNode()` is different from `RankRCG()`. Each item in the ranking list represents a component in the target replication deployment, and its failure may not cause the outage of the entire replication deployment.

Example 3: Exploring better replica server options by manipulating a fault graph. Previous examples showed that the current servers $S1$ and $S2$ do not offer a desired reliability of a replication deployment. To enhance the reliability, the administrator wants to replicate the data on one more server. She can easily find the best candidate by writing the following RAL program:

```
let s1 = Server("172.28.228.21");
let s2 = Server("172.28.228.22");
let s3 = Server("172.28.228.23");
let s4 = Server("172.28.228.24");
let s5 = Server("172.28.228.25");
let candList = [s3, ... , s5];
let i = 0;
while (i != len(candList)){
  let newRep = [s1, s2, candList[i]];
  let ft = FaultGraph(newRep);
  let prob = FailProb(ft);
  print("Using" + candList[i] + prob);
  ++i;
}
```

Example 4: Recommending candidate servers for the most independent replication. Our last scenario shows: given a set of alternative servers, how can the administrator ask RepAudit to "recommend" a replication deployment whose servers have the lowest underlying correlations in the given set. We call such a replication deployment as the most independent replication deployment for the given set of servers. More formally, given a set of n servers and m ($m < n$), our goal is to find the most independent m-way replication. In RAL, we offer a primitive `RecRep()`, to compute the most independent replication deployment by taking as input a list of (candidate) servers and the number of replicas. In our motivating example, the administrator writes the following RAL program (for $n = 4$ and $m = 2$):

```
let s1 = Server("172.28.228.21");
let s2 = Server("172.28.228.22");
let s3 = Server("172.28.228.23");
let s4 = Server("172.28.228.24");
let serverList = [s1, s2, s3, s4];
let num_replica = 2;
let list = RecRep(serverList, num_replica);
print(list);
```

The RepAudit system outputs the following list of servers. Note that the server groups with the same ranking have the same correlated failure risks.

```
1. {Server-1["172.28.228.21"],Server-4["172.28.228.24"]}
1. {Server-2["172.28.228.22"],Server-4["172.28.228.24"]}
```

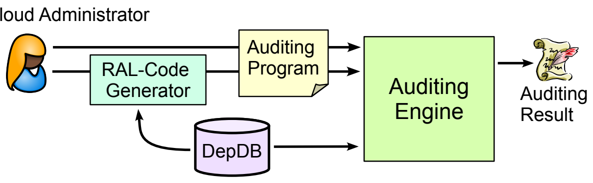


Fig. 3. An overview of the RepAudit framework.

```
1. {Server-3["172.28.228.23"],Server-4["172.28.228.24"]}
4. {Server-1["172.28.228.21"],Server-2["172.28.228.22"]}
5. {Server-1["172.28.228.21"],Server-3["172.28.228.23"]}
5. {Server-2["172.28.228.22"],Server-3["172.28.228.23"]}
```

The final remark is that the primitives of RepAudit do not necessarily rely on whether failure probabilities can be obtained, because all the primitives support the RCG size-based metric as their default measure.

3 FRAMEWORK OVERVIEW

Figure 3 depicts the RepAudit framework overview and a typical workflow of RepAudit. RepAudit has three main components: an auditing language, an auditing engine and an automatic RAL-code generator. In a typical RepAudit workflow, the cloud administrator uses our proposed language, RAL (described in Sec. 4), to write a program expressing an auditing task. Alternatively, the administrator may leverage the RAL-code generator to automatically generate an auditing program (in Sec. 6).

The auditing engine (described in Sec. 5) parses the RAL program and executes the RAL primitives on the underlying structures of the cloud system. We use the well-known techniques from the systems area research to construct formal description of that system: fault graph and dependency information database (DepDB).

A fault graph. We use a fault graph to model and reason about the independence of replication deployments. The name "a fault graph", is used in the system literature [Kaminow and Koch 1997; Zhai et al. 2014], while in formal reasoning research a fault graph corresponds to a monotone circuit [Alon and Boppana 1987]. An example of a fault graph is given in Figure 2b. The fault graph has two types of nodes: *failure events* and *logic gates*. Failure event nodes in any given fault graph can be grouped into three categories: a *top event*, many *basic events*, and multiple *intermediate events*. In particular, for event nodes which do not have child nodes, we call them as basic events, *e.g.*, "Agg1 fails" and "Core1 fails" in Figure 2b. The root node in a fault graph is called top event. Any fault graph's root node is the target replication deployment, and the root node is connected by an AND gate to multiple nodes representing servers. The rest of nodes are intermediate event nodes.

If a failure event occurs (or not), it outputs a 1 (or 0) to its higher-layer logic gate. The fault graph has two types of logic gates, *AND* and *OR*, which are used to depict different logic relationships among components' failures. For an OR gate, if any of its subsidiary components fails, a failure propagates upwards; for an AND gate, only if all of its subsidiary components fail, the gate propagates a failure upwards. Because the root node in a fault graph represents the target replication deployment failure event, the fault graph's root node should be connected by an AND gate to multiple nodes representing servers holding replicas.

In addition, every node in a fault graph is allowed to be assigned a weight, which is expressing the failure probability of the associated event.

```
Network dependencies of S1, S2 and S3:
<src="S1" dst="Internet" route="Agg1,Core1"/>
<src="S1" dst="Internet" route="Agg2,Core1"/>
<src="S2" dst="Internet" route="Agg1,Core1"/>
<src="S2" dst="Internet" route="Agg2,Core1"/>
<src="S3" dst="Internet" route="Agg3,Core2"/>
<src="S1" dst="S2" route="Agg1"/>
<src="S1" dst="S2" route="Agg2"/>
<src="S1" dst="S2" route="Agg1,Core1,Agg2"/>
```

Fig. 4. An example for the dependency information on the topology in Figure 2a.

The dependency information database (DepDB). RepAudit aims to help administrators to analyze, detect and reduce correlated failure risks, and thus RepAudit heavily relies on dependency data describing underlying structures of real cloud systems. We build upon the fact that many automatic dependency acquisition tools have been developed and deployed in today's cloud providers [Aguilera et al. 2003; Bahl et al. 2007; Barham et al. 2004; Chen et al. 2004, 2008; Dunagan et al. 2004; Kandula et al. 2005, 2009; Kompella et al. 2005; Peddycord III et al. 2012; Zhai et al. 2014].

In our prototype, we employ NSDMiner [Natarajan et al. 2012; Peddycord III et al. 2012] to automatically collect network dependencies. NSDMiner is a traffic-based network data collector, which discovers network dependencies by analyzing network traffic flows collected from network devices or individual packets. We choose NSDMiner, because it does not need to install any extra agents or software on hosts and can get more accurate results than other representative network dependency collectors, *e.g.*, Sherlock [Bahl et al. 2007] and Orion [Chen et al. 2008].

We store all the acquired dependency information needed for RepAudit. Next, we transform the collected information to a formal description, as illustrated in Figure 4. Every network path dependency information in DepDB has a uniform representation: `<src="S" dst="D" route="x,y,z"/>`, which describes a route from the source `S` to the destination `D` through network components `x`, `y`, and `z`, such as routers and/or switches.

This way RepAudit does not need to collect any dependency information from the underlying target services—it extracts all needed information from the DepDB.

4 AUDITING LANGUAGE: RAL

We propose and develop RAL, a domain-specific language that allows cloud administrators to express their auditing tasks. In principle, RAL is a strong and dynamic typed query language, which is interpreted by RepAudit's auditing engine (cf. Sec. 5), without any compilation involved.

RAL syntax. Figure 5 shows the RAL's expressions, which include global variable g, constant value c, list l and query primitive q. The expressions do not have side effect. Constant values in RAL are either real numbers or strings. Lists we sometimes also denote as [] and $[e_1, \ldots, e_n]$. Query primitives in RAL enable various auditing purposes. These primitives are typically designed for analyzing the underlying structures of target replication deployments. `Server(e)` and `Switch(e)` are used to return a server node instance and a switch node instance according to the IP address e, respectively. `FaultGraph(e)` generates a fault graph that represents the replication deployment consisting of the input server list e. `RankRCG(`e_1`, `e_2`, metric)` and `RankNode(`e`, metric)` rank RCGs and individual device nodes within the given fault graph (*i.e.*, e_1 and e) according to the ranking metric parameters, respectively. `FailProb(e)` computes the failure probability of a replication

e	::=	$g \mid c \mid l\langle e\rangle \mid q \mid e_1\ op\ e_2$	Expression
c	::=	$i \mid str$	Real number or string
$l\langle e\rangle$	::=	`nil` $\mid e :: l$	List
q	::=	`Server`(e)	Initializing server node
	|	`Switch`(e)	Initializing switch node
	|	`FaultGraph`(e)	Generating fault graph
	|	`RankRCG`$(e_1, e_2, metric)$	Ranking RCGs
	|	`RankNode`$(e, metric)$	Ranking devices
	|	`FailProb`(e)	Failure probability
	|	`RecRep`(e_1, e_2)	Recommending replication
	|	`AddPath`(e_1, e_2)	Adding path
	|	...	
$metric$	::=	`SIZE` | `PROB`	Ranking metric

Fig. 5. Expression of RAL

deployment represented by fault graph e. `RecRep(`e_1`, `e_2`)` takes an candidate server list e_1, and returns a list containing e_2 server nodes, which means given e_1 alternative servers, using these e_2 servers to hold replicas would offer a replication deployment with the lowest correlated failure risk. `AddPath(`e_1`,`e_2`)` can be used to automatically add a network path e_2 to fault graph e_1. Note that making the query primitives accurate and scalable is very challenging in practice, because they typically need to analyze a large-scale and complex fault graph. Sec. 5 details the primitive design.

Statements in RAL include assignments to global variables, output instructions, and structured control commands (*e.g.*, branching and loops). While function calls are not supported in our current design, they can easily to be added in the future work.

S	::=	`let` $g = e$	Assignment
	|	`print`(e)	Output
	|	$S_1; S_2$ | `if`$(e)\{S_1\}$ `else`$\{S_2\}$ | `while`$(e)\{S\}$	

Query evaluation. In order to represent the auditing result in RAL, *i.e.*, the return values of query primitives, we introduce the *abstract query value*.

v	$\in$	`aval`	Abstract query value
	::=	c | `node` | $l\langle v\rangle$	Constant value or node or list
`node`	:=	`leaf` i | `inner` r $l\langle$`node`$\rangle$	Abstract node
r	::=	`AND` | `OR`	Logical relation

The query results consist of constant value, abstract network node (or abstract node), and list. The abstract node can be used to represent the underlying topology structures of replication deployments of interest. An abstract node n is either a *leaf* node (`leaf` i), where i is the failure probability of the node, or an inner node (`inner` r $l\langle$`node`$\rangle$), where $l\langle$`node`$\rangle$ represents the list of lower-level child nodes of the node n and r is the logical relation among these child nodes. The `AND` relation means that only if all of lower-level child nodes fail, the node n fails, while `OR` indicates that the failure of any lower-level child nodes would make the node n fail. In principle, a leaf node represents basic event in fault graph, and an inner node corresponds to either an intermediate event or a root event in fault graph. The replication deployment topology shown in Figure 2b can be represented as the following abstract nodes (we omitted the tail `nil` in the list construction).

$$\frac{\tau(g) = v}{\Gamma, \tau \vdash g \downarrow v} \qquad \frac{e = c \mid \texttt{nil}}{\Gamma, \tau \vdash e \downarrow e} \qquad \frac{\Gamma, \tau \vdash e \downarrow v_1 \qquad \Gamma, \tau \vdash l \downarrow v_2}{\Gamma, \tau \vdash e :: l \downarrow v_1 :: v_2} \qquad \frac{\Gamma, \tau \vdash e \downarrow str \qquad \Gamma(str) = \texttt{node}}{\Gamma, \tau \vdash \texttt{Server}(e) \downarrow \texttt{node}}$$

$$\frac{\Gamma, \tau \vdash e \downarrow str \qquad \Gamma(str) = \texttt{node}}{\Gamma, \tau \vdash \texttt{Switch}(e) \downarrow \texttt{node}} \qquad \frac{\Gamma, \tau \vdash e_1 \downarrow i_1 \qquad \Gamma, \tau \vdash e_2 \downarrow i_2}{\Gamma, \tau \vdash e_1\ op\ e_2 \downarrow i_1\ op\ i_2}$$

Fig. 6. Selected rules of expression evaluation.

$$\begin{aligned} S_1 &= \texttt{inner AND } (\text{Path}_1 :: \text{Path}_2) \\ S_2 &= \texttt{inner AND } (\text{Path}_1 :: \text{Path}_2) \\ \text{Path}_1 &= \texttt{inner OR } (\text{Agg}_1 :: \text{Core}_1) \\ \text{Path}_2 &= \texttt{inner OR } (\text{Core}_1 :: \text{Agg}_2) \\ \text{Agg}_1 &= \texttt{leaf } 0.5 \\ \text{Agg}_2 &= \texttt{leaf } 0.5 \\ &\ldots \end{aligned}$$

Thus, all the abstract nodes can be evaluated as a *mono-circuit* [Alon and Boppana 1987] containing one or more `leaf` nodes. We write `monoCircuit` to denote the function that takes an abstract node and returns a mono-circuit formula containing all the leaf nodes underlying this node.

$$\frac{n = \texttt{leaf \_}}{\texttt{monoCircuit}(n) = n} \qquad \frac{n = \texttt{inner AND } l \qquad f = \bigwedge_{\forall n' \in l} \texttt{monoCircuit}(n')}{\texttt{monoCircuit}(n) = f}$$

$$\frac{n = \texttt{inner OR } l \qquad f = \bigvee_{\forall n' \in l} \texttt{monoCircuit}(n')}{\texttt{monoCircuit}(n) = f}$$

For example, transforming the abstract node S_1 to the mono-circuit formula gives us:

$$\texttt{monoCircuit}(S_1) = (\text{Agg}_1 \vee \text{Core}_1) \wedge (\text{Core}_1 \vee \text{Agg}_2)$$

The semantics of the query primitives are defined in terms of this `monoCircuit` function. Because expressions have no side effects, we fix a finite map Γ from IP address to abstract node and a finite map τ from global variable to abstract query value. We use $\Gamma, \tau \vdash e \downarrow v$ to denote the expression evaluation, which takes the IP table Γ, global environment τ and returns an abstract query value v : `aval`. Figure 6 shows selected expression evaluation rules.

We now detail the definition of several non-trivial queries' evaluations as examples. The evaluation of `FaultGraph`(e) adds an `AND` dependency over the nodes list e and introduces a new abstract node, which does not exist in the original topological structure. This newly introduced abstract node represents the replication deployment upon e. The replication deployment shown in Figure 2b is represented as $(\texttt{inner AND } (S_1 :: S_2))$, where the `AND` gate indicates that the replicated deployment will fail only if all the replicated server nodes fail.

$$\frac{\Gamma, \tau \vdash e \downarrow l\langle\texttt{node}\rangle}{\Gamma, \tau \vdash \texttt{FaultGraph}(e) \downarrow (\texttt{inner AND } \langle\texttt{node}\rangle)}$$

Since we have introduced this abstract node and its corresponding node graph representing the target replication deployment, the evaluation of the query expressions can be defined as the operations over this node graph. Take the query RankRCG as an example. We write $\texttt{RCG}(\texttt{monoCircuit}(n), l\langle\texttt{leaf}\rangle) = \texttt{True}$ to denote that the leaf node list l is a RCG of the abstract node n, which means the failure of all the nodes in l will lead to the failure of the node n.

$$\frac{c \in l\langle\texttt{leaf}\rangle}{\texttt{RCG}(c, l\langle\texttt{leaf}\rangle) = \texttt{True}} \qquad \frac{v = \texttt{RCG}(c_1, l) \wedge \texttt{RCG}(c_2, l)}{\texttt{RCG}(c_1 \wedge c_2, l\langle\texttt{leaf}\rangle) = v} \qquad \frac{v = \texttt{RCG}(c_1, l) \vee \texttt{RCG}(c_2, l)}{\texttt{RCG}(c_1 \vee c_2, l\langle\texttt{leaf}\rangle) = v}$$

For the fault graph shown in Figure 2b, we have:

$$\texttt{RCG}\ (\texttt{monoCircuit}(S_1), \texttt{Agg}_1) = \texttt{False}$$
$$\texttt{RCG}\ (\texttt{monoCircuit}(S_1), \texttt{Agg}_1 :: \texttt{Core}_1) = \texttt{True}$$

Therefore, with the ranking metric SIZE, RankRCG will return a list of minimal RCG. The minimal RCG is defined as a list of leaf node and removing any of them will make the list fail to form the RCG (more definitions in Sec. 5.2). For instance, ($\texttt{Core}_1$ :: nil) is a minimal RCG of the node S_1.

$$\frac{\begin{array}{ccccc} & \Gamma, \tau \vdash e_1 \Downarrow n & \Gamma, \tau \vdash e_2 \Downarrow i & c = \texttt{monoCircuit}(n) & \\ v = l\langle l\langle\texttt{leaf}\rangle\rangle & size(v) = i & \forall l \in v, \texttt{RCG}(c, l) = \texttt{True} & & \forall l \in v, \forall x \in l, \texttt{RCG}(c, l/x) = \texttt{False}\end{array}}{\Gamma, \tau \vdash \texttt{RankRCG}(e_1, e_2, \texttt{SIZE}) \Downarrow v}$$

Among these query primitives, the $\texttt{AddPath}(e_1, e_2)$ is special in the sense that it will change the abstract node e_1 in the IP table Γ by introducing a new abstract *path* node as a child of e_1. The abstract path node does not exists in the original topology network and is an inner node having a list of leaf child nodes e_2 associated with OR.

$$\frac{\begin{array}{cccc} & \Gamma, \tau \vdash e_1 \Downarrow n & \Gamma, \tau \vdash e_2 \Downarrow l\langle\texttt{leaf}\rangle & \\ n = \texttt{inner AND } l' & p = \texttt{inner OR } l & n' = \texttt{inner AND } (l' :: p) & \Gamma' = \Gamma[n \leftarrow n']\end{array}}{\Gamma', \tau \vdash \texttt{AddPath}(e_1, e_2) \Downarrow n'}$$

Note that the AddPath query requires that the target node e_1 is an inner node with AND gate, since the practical meaning of this query is to add a redundant network path to the original replication deployment topology. For example, after the query $\texttt{AddPath}(S_1, \text{Core}_2 :: \text{Agg}_2)$, the mono-circuit formula representing S_1 becomes:

$$\texttt{monoCircuit}(S_1) = \quad (\text{Agg}_1 \vee \text{Core}_1) \wedge (\text{Core}_1 \vee \text{Agg}_2) \wedge \ (\text{Core}_2 \vee \text{Agg}_2)$$

In this case, ($\texttt{Core}_1$ :: nil) is no longer a RCG of S_1, thus making the replication deployment more reliable. We have more examples for AddPath() in Sec. 6.

Semantics. Figure 7 defines the semantics of RAL under the form of a big-step semantics. We write $\Gamma \vdash S : \tau \Downarrow (out; \tau')$ for the semantics of statements: from the global environment τ, execution of S terminates and yields output *out* and global environment τ'.

5 AUDITING ENGINE

RepAudit's auditing engine parses and executes RAL programs. The main challenge for the engine is to design and implement RAL primitives capable of scaling to large-scale, complex datacenter network topologies containing tens or hundreds of thousands of communication devices. In this section, we describe the designs of five important primitives in the RAL language, FaultGraph(),

$$\frac{\Gamma, \tau \vdash e \Downarrow v \qquad \tau' = \tau[g \leftarrow v]}{\Gamma \vdash \texttt{let}\ g = e : \tau \Downarrow (\epsilon; \tau')} \qquad \frac{\Gamma, \tau \vdash e \Downarrow v}{\Gamma \vdash \texttt{print}(e) : \tau \Downarrow (v; \tau)}$$

$$\frac{\Gamma \vdash S_1 : \tau \Downarrow (o_1; \tau') \qquad \Gamma \vdash S_2 : \tau' \Downarrow (o_2; \tau'')}{\Gamma \vdash S_1; S_2 : \tau \Downarrow (o_1 :: o_2; \tau'')} \qquad \frac{\Gamma, \tau \vdash e \Downarrow i \qquad i \neq 0 \qquad \Gamma \vdash S_1 : \tau \Downarrow (o; \tau'))}{\Gamma \vdash \texttt{if}(e)\{S_1\}\ \texttt{else}\{S_2\} : \tau \Downarrow (o; \tau')}$$

$$\frac{\Gamma, \tau \vdash e \Downarrow 0 \qquad \Gamma \vdash S_2 : \tau \Downarrow (o; \tau'))}{\Gamma \vdash \texttt{if}(e)\{S_1\}\ \texttt{else}\{S_2\} : \tau \Downarrow (o; \tau')} \qquad \frac{\Gamma, \tau \vdash e \Downarrow 0}{\Gamma \vdash \texttt{while}(e)\{S\} : \tau \Downarrow (\epsilon; \tau)}$$

$$\frac{\Gamma, \tau \vdash e \Downarrow i \qquad i \neq 0 \qquad \Gamma \vdash S : \tau \Downarrow (o; \tau') \qquad \Gamma \vdash \texttt{while}(e)\{S\} : \tau' \Downarrow (o'; \tau'')}{\Gamma \vdash \texttt{while}(e)\{S\} : \tau \Downarrow (o :: o'; \tau'')}$$

Fig. 7. Operational semantics of RAL.

`RankRCG()`, `FailProb()`, and `RecRep()`, that enables administrators to perform diverse auditing tasks in both efficient and accurate way even in cloud-scale networks.

We omit other primitives whose designs are straightforward. In addition, RepAudit allows administrators to define and customize new primitives to support their specific goals.

5.1 The Fault Graph Generation: `FaultGraph()`

An RAL language primitive `FaultGraph()` takes as input a list of servers and returns a fault graph, represented as a mono-circuit formula.

The fault graph is constructed automatically, using only the information from the dependency information database (DepDB). First, RepAudit constructs the root node which denotes the failure of the target replication deployment. Then, all the servers from the input list become children nodes of the top event, connected with an AND gate. Using the AND gate indicates that the replication deployment will fail iff all the replica servers fail. For each of servers, we query the DepDB to obtain all the network paths relevant to the server. A path would fail iff any of its devices (*e.g.*, some switch) fails. Thus, network devices on each path are connected with an OR gate.

Finally, we translate the constructed fault graph into a Boolean formula, to every variable we also assign the failure probability, as described in Sec. 4. For every leaf n node we define a new Boolean variable B_n. The translation algorithm $\mathcal{T}$, which takes as input a root node in fault graph and returns a Boolean formula, is straightforward:

$$\mathcal{T}(n) = \begin{cases} \mathcal{T}(n_1) \wedge \mathcal{T}(n_2) & n = \texttt{inner AND}\ (n_1 :: n_2) \\ \mathcal{T}(n_1) \vee \mathcal{T}(n_2) & n = \texttt{inner OR}\ (n_1 :: n_2) \\ B_n & n = \texttt{leaf}\ \_ \end{cases}$$

A fault graph describing the replication deployment topology from Fig. 2b is a formula

$$(\text{Agg}_1 \vee \text{Core}_1) \wedge (\text{Core}_1 \vee \text{Agg}_2) \wedge (\text{Agg}_1 \vee \text{Core}_1) \wedge (\text{Core}_1 \vee \text{Agg}_2)$$

When this formula evaluates to true, that means that the depicted replication deployment failed. Thus, any satisfying assignment of F denotes a risk component group.

```
Algorithm 1: RCGs auditing primitive
Input: $F \leftarrow$ the input fault graph
Input: $k \leftarrow$ the number of critical RCGs desired
Input: $c \leftarrow$ critical metric for ranking
Output: $R \leftarrow$ ranked RCGs list
costVector $\leftarrow \emptyset$;        // cost vector
$x \leftarrow \emptyset$;        // minimal cost assignment list
$rcg \leftarrow \emptyset$;
switch $c$ do
    case SIZE        // Extracting the top-k RCGs with the smallest sizes
        foreach variable $i$ in $F$ do
            costVector[i] $\leftarrow 1$;
    case PROB        // Extracting the top-k RCGs with the highest failure probabilities
        foreach variable $i$ in $F$ do
            costVector[i] $\leftarrow (-100)\times$ the logarithm of failure probability of variable $i$;
$\phi \leftarrow F$;
for $i \leftarrow 1$ to $k$ do
    $rcg \leftarrow \emptyset$;
    $x \leftarrow$ WP-MaxSATSolver($\phi$, costVector);
    if $x$ is a model then
        $k \leftarrow k - 1$;
        foreach literal $i$ in the assignment list $x$ do
            if $x[i]$ = TRUE then
                $rcg$.append($x[i]$);
        $R$.append($rcg$);
        $\phi \leftarrow \phi \wedge \neg x$;
    else
        break;        // No feasible result
return $R$;
```

5.2 RCG Auditing Primitive

A risk component group (or RCG) [Kaminow and Koch 1997; Zhai et al. 2014] is a group of basic failure events such that if all of them occur simultaneously, then the top event occurs as well. We define an RCG as a *minimal RCG* if the removal of any of its constituent failure events makes it no longer an RCG. Consider the following two RCGs in Figure 2b: {Agg1 fails, Core1 fails} and {Agg2 fails, Core1 fails}. None of them is a minimal RCG because {Core1 fails} alone is sufficient to cause the top event to occur.

Analyzing minimal RCGs in a large-scale fault graph structure is an NP-hard problem [Zhai et al. 2013], and this is a potential obstacle for auditing current datacenter networks. The previous attempts, using Monte Carlo simulations did not scale well [Zhai et al. 2013]. On the other hand, existing SAT solvers can successfully check satisfiability of Boolean formulas arising from practical and industry-scale problems. There are reports [Balyo et al. 2016] showing that formulas with millions of variables and clauses can be solved in seconds. We thus decided to use advances in modern SAT solvers and de the task of finding minimal RCGs in a fault graph to the *weighted partial MaxSAT (WP-MaxSAT) problem.*

Definition 5.1. [Ansótegui et al. 2009, 2010] For a given Boolean formula ϕ with n variables $x_1, x_2, \ldots, x_n$, and a corresponding cost vector, $\{c_i | c_i \geq 0, 1 \leq i \leq n\}$, the goal is to find a satisfying assignment for ϕ that minimizes the formula:

$$C = \sum_{i=1}^{n} c_i x_i \tag{1}$$

An assignment sets every variable x_i to 1 or 0.

Algorithm 1 provides an implementation of the `RankRCG()` language primitive. The algorithm works in two phases. In the first phase we initialize the cost vector according to the criterion for ranking (lines 4-10 in Algorithm 1). For the `SIZE` criterion (*i.e.*, extracting the top-k RCGs with the smallest sizes), each leaf node's cost is 1. For `PROB` option (*i.e.*, extracting the top-k RCGs with the *highest* failure probabilities), each leaf's value $c_i = (-100) \log p_i$, where p_i is its failure probability.

To explain the value of c_i, note that computing a failure probability of a single RCG is a *product* of the failure probabilities of all of its components. However, the WP-MaxSAT problem is about finding the satisfiability assignment where the *sum* of the costs of variables, that are set to 1, is minimal. That is why we use the log function. Having an RCG with $\{y_1, \ldots, y_m\}$ components, its cost will be computed as $\Sigma_{i=1}^{m} c_i = \Sigma_{i=1}^{m} (-100) \log p_i = (-100) \log \Pi_{i=1}^{m} p_i$. If we just used $c_i = \log p_i$, the resulting costs would be negative numbers, which cannot be the applied in the WP-MaxSAT problem. Therefore we multiply each value with a negative number. This way the finding the minimal cost assignment will result in an RCG with the highest failing probability.

Theorem 5.2. *Given a fault graph with $\{x_1, \ldots, x_n\}$ variables, where every x_i is associated cost c_i. An RCG $\{y_1, \ldots, y_k\} \subseteq \{x_1, \ldots, x_n\}$ is minimal iff it is a solution of the corresponding weighted partial MaxSAT problem.*

In the second phase of the algorithm, we invoke a WP-MaxSAT solver on the formula ϕ and find the top k- critical RCGs through k loop iterations. We used an open source weighted partial MaxSAT solver Maxino [Alviano 2015]. Since the numbers $-\log p_i$ were too small and too close in the value for the solver, we use $c_i = (-100) \log p_i$.

5.3 Computing the Failure Probability

The RAL primitive `FailProb()` computes the failure probability of the fault graph's top event. Because the intermediate nodes in fault graph are not independent, computing the failure probability of the top event can be done by computing a conditional probability for a Markov chain. However, previous attempts did not scale for a large fault graph [Zhai et al. 2014]. We again reduce this computation task to the weighted partial MaxSAT problem for guaranteed accuracy and efficiency. This way we delegate the burden of heavy computations to a WP-MaxSAT solver problem.

To compute the failure probability of the top event T, *i.e.*, $\Pr(T)$, we first call `RankRCG()` to output the top-k RCGs of the T, and then compute $\Pr(T)$ based on the inclusion-exclusion rule:

$$\begin{aligned} \Pr(T) = & \sum_{i=1}^{k} \Pr(RCG_i) - \sum_{1 \leq i < j \leq k} \Pr(RCG_i) \cdot \Pr(RCG_j) \\ & + \sum_{1 \leq i < j < m \leq k} \Pr(RCG_i) \cdot \Pr(RCG_j) \cdot \Pr(RCG_m) \\ & + \cdots + (-1)^{k-1} \Pr(RCG_1) \cdot \Pr(RCG_2) \cdots \Pr(RCG_k) \end{aligned} \tag{2}$$

where RCG_i means the ith RCG output by `RankRCG()`. In Figure 2b example, for $k = 2$, there are two RCGs, *i.e.*, {Core1 fails} and {Agg1 fails, Agg2 fails}. Let us assume the failure probability of each component is 0.2 (clearly, too high for practical purpose, we use it only for a demonstration). We compute the probability of the top event (*i.e.*, the failure probability of the target replication deployment) by: $\Pr(T) = 0.2 \cdot 0.2 + 0.2 - 0.2 \cdot 0.2 \cdot 0.2 = 0.232$.

Note that `FailPro()` is not guaranteed to always return 100% accurate failure probability, since we consider only the top-k RCGs during the process of computation. Nevertheless, because the administrators use the probabilities only to get an estimate on how reliable the replication systems are, we believe that using an efficient algorithm that efficiently obtains (almost) accurate results is more important than having a completely accurate but computationally expensive algorithm.

5.4 Recommending Replication Deployments

The RAL primitive `RecRep()` recommends the most independent m-way replication deployment according to a given list specifying n candidate servers. With such a candidate server list in hand, the most straightforward way designing `RecRep()` is to traverse all the possible m-way replication deployment cases and use `FailProb()` or `RankRCG()` to compute a metric for picking out the most suitable one. However, such an approach is impractical in a datacenter topology with tens of thousands of nodes, since it needs to try C_n^m cases.

We design a scalable `RecRep()` inspired from an existing theory named *network transformation* [Plotkin et al. 2016]. In principle, network transformation theory can transform a given structurally symmetric network N (*e.g.*, datacenter network) into a "simplified" network N', with equivalent connectivity and reachability but with much less nodes, by exploiting datacenter network symmetry and network surgery (in which irrelevant sets of nodes and paths are sliced away). We follow the network transformation effort proposed by Plotkin *et al.* [Plotkin et al. 2016] to develop a network transformation solver that can transform a given "symmetric" CNF into a simpler CNF with the same reachability and validity but with much less variables.

The algorithm of `RecRep()` includes three steps. First, by taking candidate server list as input, we generate a Boolean formula representing dependencies underlying all the servers in the given candidate list. Second, we put the generated formula into the network transformation tool [Plotkin et al. 2016], obtaining a tailored formula representing the simplified topology. Finally, we traverse all the possible redundant deployment cases on the transformed small topology and extract the most independent deployment. The independence metric could be computed by `FailProb()` or `RankRCG()`.

We make no claim that network transformation is novel in itself, as it is an existing theory and has been applied in other domains to scale verifications; we merely utilize it in a new domain (*i.e.*, independent redundant deployment extraction) to enable our efficient auditing.

6 RAL CODE GENERATION

We believe that expressing basic auditing tasks in RAL (*e.g.*, Example 1, 2 and 4 in Sec. 2) is easy. However, once the cloud administrators understand correlated failure risks in their replication deployments, they often need to fix risks in their deployments. This is typically done by: 1) additionally deploying replicas on new servers (cf. Example 3 in Sec. 2), 2) adding new paths to connect existing servers and switches, 3) moving replicas from some of current servers to another servers, or 4) using new devices with low failure probabilities to replace vulnerable ones.

RAL offers language primitives (*e.g.*, `AddNode()`, `AddPath()`, `MvRep()`, and `ChNode()`) to allow administrators to "simulate" a desired replication deployment at the logical level. The administrators, therefore, can estimate whether the new replication deployments satisfy their goals. For example, in Example 3 (Sec. 2) an administrator tried to find a new replication deployment using RAL program.

Nevertheless, because these advanced RAL programs are done by administrators manually, it is an error-prone process and there is no guarantee that the resulting program finds the optimal solution such as adding the minimal number of paths. In order to avoid this tedious process, we extend the RAL language with a possibility to automatically generate the RAL code that helps administrators to obtain better deployments.

Note that the RAL-code generator only "changes" replication deployments at the logical level, rather than physically updating deployments. In addition, the generator aims to produce RAL code for common but difficult-to-write auditing tasks, rather than generating simple RAL code like the Example 1, 2 and 4 in Sec. 2, because RAL itself has been expressive enough for those simple cases.

6.1 Grammar and Usage

An administrator specifies what code should be generated using the following template call:

```
goal (spec | scheme | CONSTRAINTS*)
```

The specification, scheme and constraints are formed according to the following grammar:

```
       spec  :=   failProb(ft) op N | sizeMinRCG(ft) op N
         op  :=     < | ≤ | = | > | ≥
     scheme  ::=  ChNode | MvRep | AddPath | AddNode | ···
CONSTRAINTS  :=     l⟨PosConstr⟩, l⟨NegConstr⟩
  PosConstr  :=     !DeviceID |!Serv
  NegConstr  :=     ~ DeviceID |~ Serv
```

Here, N is a number, given by the administrator, to compare the failure probability or the size of the minimal RCGs to this threshold N. Current *scheme* contains four types of reliability enhancement schemes: adding new paths (*i.e.*, AddPath), moving replicas from some of current servers to another servers (*i.e.*, MvRep), additionally deploying replicas on new servers (*i.e.*, AddNode), and using more reliable devices to replace the old ones (*i.e.*, ChNode). Our RAL-code generator is extendible to more schemes as long as administrators define more primitives. CONSTRAINTS is an optional parameter and it stands for a list of constraints. There are two type of constraints: positive and negative. In negative constraints, we indicate that certain devices cannot be used, or certain servers should not be used to hold replica data. In positive constraints (or a better name: enforcing constraints), we specify to the generator that some devices or servers must be used.

Usage example. The example below illustrates how to use the feature of RAL-code generator in our small replication system example shown in Figure 2a:

```
let s1 = Server("172.28.228.21");
let s2 = Server("172.28.228.22");
let rep = [s1, s2];
let ft = FaultGraph(rep);
goal (failProb(ft) < 0.08 | AddPath | ~Agg3);
```

In this program, the target replication deployment consists of two servers s1 and s2. The goal for an enhancement is to make the replication deployment's failure probability lower than 0.08. Additionally, we require adding new paths to the underlying topology (*i.e.*, AddPath scheme), and the new paths cannot contain the switch Agg3. Taking this specification as input, the generator outputs the following RAL program.

```
let ls1 = [Switch("Agg1"), Switch("Core2")];
let path1 = (s1, ls1);
```

Algorithm 2: RAL-code generation algorithm for `AddPath` scheme.

```
1  Algorithm generation()
       Input: Spec ← the given specification;
       Input: map() ← mapping the solution to Prog
       Output: Prog ← RAL program for Spec
2      F ← Spec.faultGraph;
3      g ← Spec.goal;
4      c ← Spec.constraints;
5      F′ ← findPaths(F, g, c);
6      if F′ = noSolution then
7          return noSolution;
8      else
9          Prog.append(map(F'));

1  Procedure findPaths()
       Input: F ← the input fault graph;
       Input: g ← the specified goal;
       Input: c ← the constraints;
       Output: F′ ← new fault graph satisfying g;
2      f ← F;
3      v ← classify all the candidate paths into groups;
4      for each group v[i] in v do
5          j ← len(v[i]) − 1;
6          // v[i][0 : j] means the former j paths in the group v[i]
7          if check(f ∧ v[i][0 : j], g, c) = ok then
8              F′ ← binarySearch(f, v[i][0 : j], g, c);
9              if F′! = nil then
10                 return solution F′;
11             else
12                 return EXCEPTION;
13         else
14             f ← f ∧ v[i][0 : j];
15     return noSolution;
```

```
ft = AddPath(ft, path1);
let ls2 = [Switch("Agg2"), Switch("Core2")];
let path2 = (s2, ls2);
ft = AddPath(ft, path2);
```

The generated RAL program adds the lowest number of paths to the existing fault graph and it is guaranteed that the new fault graph has the failure probability lower than 0.08. The generated code is presented to the administrator, and she then may decide if the physical network topologies should be updated accordingly.

6.2 Code Generation Algorithm

The algorithms employed in the RAL-code generator for different schemes (*e.g.*, `AddPath`, `MvRep`, `AddNode`, and `ChNode`) are all based on efficient searching through the space of all possible solutions

Algorithm 3: RAL-code generation algorithm for `MvRep` scheme.

```
1  Procedure findServers()
       Input: $F \leftarrow$ the input fault graph;
       Input: $g \leftarrow$ the specified goal;
       Input: $c \leftarrow$ the constraints;
       Output: $F' \leftarrow$ new fault graph satisfying $g$;
2      $f \leftarrow$ network_transform(CNF_transform($F$));
3      $v \leftarrow$ classify all the candidate servers into groups;
4      $u \leftarrow$ classify all the servers of $f$ into groups;
5      for each group $v[i]$ in $v$ do
6          $j \leftarrow len(v[i]) - 1$;
7          // $v[i][0:j]$ means the former $j$ servers in the group $v[i]$
8          if $check(f - u[i][0:j] \wedge v[i][0:j], g, c) = ok$ then
9              $F' \leftarrow$ binarySearch($f, u[i][0:j], g, c$);
10             if $F' != nil$ then
11                 return solution $F'$;
12             else
13                 return EXCEPTION;
14         else
15             $f \leftarrow f - u[0:j] \wedge v[i][0:j]$;
16     return noSolution;
```

for the specified goal. During the search process, the algorithms cut off unnecessary solution branches based on specified constraints. We omit the generation algorithm for `ChNode` and `AddNode` schemes, since they are straightforward to design. We detail the design of generation algorithms for `AddPath` and `MvRep` schemes.

RAL-code generation for `AddPath` scheme. The algorithm works in two phases. In the preprocessing phase, all network paths are classified into groups according to the number of overlapping devices, between that group and the topology of original replication deployment. Intuitively, all the paths in the same group have the *equal effects* in terms of improving the original replication deployment's independence. For example in Figure 2a, a candidate path `<src="S4" dst="Internet" route="Agg3,Core2"/>` has no overlapping device with the current replication deployment. In the second phase, the algorithm finds new paths to be added to the existing fault graph. Adding each of these paths in fault graph would influence the independence of the entire replication deployment. Using the binary search, we find the minimal number of paths. The constraints are enforced by discarding the paths that do not satisfy them.

Algorithm 2 describes the process. First, the algorithm gathers all the candidate paths into the corresponding groups according to the number of overlapping devices between these paths and the target replication deployment. With $v[i]$, we denote a group of all candidate paths, which have i overlapping devices with the replication deployment. Our algorithm searches a solution from $v[0]$ to $v[len(v) - 1]$ (in that order), because a path in $v[0]$ can make the "biggest" independence improvement to the original replication deployment. During searching through the $v[i]$ group, the algorithm uses binary search way to check whether the current solution is the optimal one satisfying the specified goal and constraints (line 8 in `findPaths()`).

RAL-code generation for `MvRep` scheme. Similar to `AddPath` scheme, the code generation algorithm for `MvRep` scheme also works in two phases. In the preprocessing phase, all available servers (servers not used by the current deployment) are classified into groups according to the number of overlapping devices, between that group and the topology of original redundant deployment. Intuitively, all the servers in the same group have the equal effects in terms of improving the original replication deployment's independence. For example, in Figure 2a, candidate server $S4$ has no overlapping device with the current replication deployment. In the second phase, the algorithm finds new servers to replace some of replica servers in the current system. Such a replacement would influence the independence of the entire replication deployment. To continue with the previous example, if we use $S4$ to replace $S2$, the independence of the new deployment is much better than the original one. We find the minimal number of servers that need to be replaced by using the binary search. We enforce the specification's constraints by discarding the paths that do not satisfy them.

Algorithm 3 describes the process of `MvRep` scheme. First, the algorithm gathers all the candidate servers into the corresponding groups v according to the number of overlapping devices between these servers and the target replication deployment. We use $v[i]$ to denote a group of all candidate servers, which have i overlapping devices with the target replication deployment. Meanwhile, the algorithm calls the network transformation toolbox [Plotkin et al. 2016] to generate a simplified fault graph (*i.e.*, f in Algorithm 3) and then puts all the replica servers in f into groups u. After the above preprocessing, our algorithm searches for a solution from $v[0]$ to $v[len(v) - 1]$ (in that order), because a server in $v[0]$ can make the "biggest" independence improvement to the original replication deployment. During searching through the $v[i]$ group, the algorithm uses binary search to check whether the current solution is the optimal one satisfying the specified goal and constraints.

Summary. The RAL-code generation algorithms (including `AddPath`, `MvRep`, `AddNode` and `ChNode` schemes) search at the best-effort independence improvement on the original deployment, and then narrow the solution space down with the binary search algorithm, finding this way a program that satisfies the specified goal and the constraints.

6.3 Extending RAL-Code Generator for More Advanced Goals

In practice, administrators may want their replication systems to achieve more sophisticated reliability goals, such as: 1) enabling their replication systems to balance the trade-off between the synchronization latency and independence of replicated data, and 2) obtaining a replication deployment with the lowest failure probability in a group of deployments where all the RCGs' sizes are larger than a certain number. These reliability goals are realistic requirements of current cloud administrators; however, achieving these targets in reality are non-trivial and error-prone [Bodik et al. 2012].

With RepAudit in hand, an administrator can easily obtain a replication deployment plan that meets complex reliability goals as shown above, by extending the template call of RAL-code generator. We now present an example for extending the RAL-code generator to obtain a replication deployment that balances the trade-off between the synchronization latency and independence of replication.

The only new language operator we need to add in the RAL-code generator is an `AND` operator, which will be used in `spec`. In particular, `AND` allows administrators to express a specification where multiple conditions should hold simultaneously. For instance, if we want to take into account both the synchronization latency (*e.g.*, sync < 0.01ms) and independence (*e.g.*, failure probability < 0.08), then an administrator can specify a goal like: `goal(latency < 0.01 AND failProb(ft) <`

`0.08 | MvPath)`. The RAL-code generator can directly read the above goal and then returns the results without changing any code in the generator. This is because we in fact only add more checking conditions in `g` (line 3 in `generation()` of Algorithm 2), and the `g` is checked by different procedures that correspond to specified schemes. In principle, any extension on specification means more conditions (or constraints) need to be checked to cut off branches that violate any of these conditions (*e.g.*, latency and failure probability in our example), and thus we can obtain desired replication deployments that should be the solutions in the whole space-search process in the invoked procedure.

For example in Figure 2a, we still assume the failure probability of each device is 0.2 and assume the latency among servers is lower than 0.1ms. The administrator conveys RepAudit a specification like `goal(latency < 0.01 AND failProb(ft) < 0.08 | MvNode)`, which means we want to generate such a deployment by moving replica data from some of current servers to another servers (defined earlier). Then, she gets the following results: Plan 1: Move replica data from S1 => S4; Plan 2: Move replica data from S2 => S4.

The above generated plans mean if the administrator moves one of her replica data from $S1$ or $S2$ to $S4$, then the failure probability of her replication system would be less than 0.08 and the synchronization latency between replica servers is less than 0.01ms. Note that the only additional information we require is the data for latency among replica servers. The RAL-code generator can get this information by querying DepDB (defined in Sec. 3), which is responsible for providing the profiling information of the underlying topology.

7 IMPLEMENTATION

This section first describes the implementation of our RepAudit prototype in Sec. 7.1. In order to deeply evaluate RepAudit, we also implemented two auditing tools—based on two state-of-the-art fault graph analysis approaches, minimal cut-set algorithm and failure sampling algorithm—to compare with RepAudit. We describe the implementation of these two systems in Sec. 7.2.

7.1 RepAudit Prototype Implementation

We developed an RepAudit prototype that fully implemented our design in a mix of C++, shell scripts and several open-source software components. The prototype consists of RAL, the auditing engine and the RAL-code generator. We used C++ to develop RAL as a domain-specific language following the grammar defined in Sec. 4. In the auditing engine, there are two key implementations that significantly optimize the performance: 1) we adapted a high-performance weighted partial MaxSAT solver, called Maxino [Alviano 2015], to enable the WP-MaxSAT capability, and 2) we developed a fault graph parser which can transform any given fault graph into a CNF Boolean formula. The implementation of RAL-code generator introduced a set of corner-case optimizations to speed up the solutions search.

The source code for our implementation is available at (https://github.com/ennanzhai/repaudit).

7.2 The Implementation of Comparable Systems

We now present how we implement two state-of-the-art fault graph analysis tools. For simplicity, we call these two tools as MCS (Minimal Cut-Set) and FSA (Failure Sampling), respectively. This is because they rely on minimal cut-set and failure sampling algorithms, respectively. Both MCS and FSA are also implemented in a mix of C++, shell scripts and several open-source libraries, which makes them fair to compare with RepAudit. In the rest of this section, we detail the working principle of the core algorithms of these two tools.

Table 1. Expressiveness (in LOC) comparison between RAL, MCS and FSA on various auditing tasks.

Auditing tasks	RAL	MCS	FSA
Modeling underlying topologies	4	213	224
Extracting and ranking RCGs	5	244	433
Computing failure probability	9	287	562
Ranking components	10	289	No Support
Recommending the most independent deployments	16	562	1395

Minimal cut-set approach. The first tool, MCS, is implemented based on minimal cut-set algorithm [Kaminow and Koch 1997], which produces the precise minimal RCGs but is not scalable as it takes exponential time in the input size. In particular, the algorithm traverses a fault graph T in a reverse breadth-first order (from basic events to the top event). Basic events first generate RCGs containing only themselves, while non-basic events produce RCGs based on their child events' RCGs and their input gates. For a non-basic event, if its input gate is an *OR* gate, the RCGs of this event include all its child events' RCGs; otherwise, if its input gate is an *AND* gate, each RCG of this event becomes an element of the Cartesian product among the RCGs of its child events. Traversing the fault graph T generates all the RCGs, and in turn all the minimal RCGs through simplification procedures.

Failure sampling approach. Another auditing tool, FSA, implements the failure sampling algorithm [Zhai et al. 2014]. It is designed based on random sampling, and aims to randomly determine RCGs efficiently but not accurately. In a typical process, this algorithm uses multiple sampling rounds, each of which performs a breadth-first traversal of the fault graph T. Within each sampling round, the algorithm assigns either a 1 or a 0 to each basic event of T based on random coin flipping, where 1 represents failure and 0 represents non-failure. Starting from such an assignment, the algorithm assigns 1's and 0's to all non-basic events from bottom to top based on their logic gates and the values of their child events. After each sampling round, the algorithm checks whether the top event fails. If it fails (*i.e.*, its value is 1), then the algorithm generates an RCG consisting of all the basic events being assigned a 1 in this sampling round. The algorithm executes a large number of sampling rounds and aggregates the resulting RCGs in all rounds. The failure sampling algorithm offers the linear time complexity, but is non-deterministic and cannot guarantee that the resulting RCGs are minimal.

8 EVALUATIONS

RepAudit consists of three key components (RAL, auditing engine and RAL-code generator); thus, our evaluation aims to answer the following three questions:

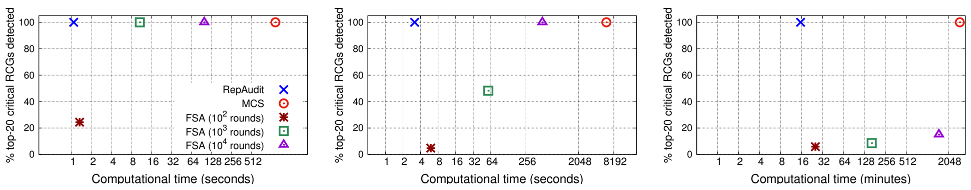
- Whether RAL is expressive (Sec. 8.1)?
- Whether the auditing engine is scalable to large-scale fault graphs (Sec. 8.2)?
- How expensive is our RAL-code generator (Sec. 8.3)?

8.1 Evaluating the Expressiveness of RAL

Our first goal is to evaluate the expressiveness of the RAL language, and thus we compare auditing programs written in RAL with MCS and FSA (described in Sec. 7) in terms of lines of the code (LOC). In particular, we use the three tools to express different auditing tasks, including: 1) modeling underlying topologies, 2) extracting and ranking RCGs in a given replication system, 3) computing

Table 2. Configurations of the evaluated topologies.

	Topology A	Topology B	Topology C	Topology D
# Switch ports	16	24	48	64
# Core routers	64	144	576	1,024
# Agg switches	128	288	1,152	2,048
# ToR switches	128	288	1,152	2,048
# Servers	1,024	3,456	27,648	65,536
Total # devices	1,344	4,176	30,528	70,656



(a) Topology A: 1, 344 devices. (b) Topology B: 4, 176 devices. (c) Topology C: 30, 528 devices.

Fig. 8. Performance evaluation of RepAudit, MCS and FSA.

failure probability, 4) ranking components based on their relative importance, and 5) recommending the most independent deployments.

Table 1 shows the expressiveness comparison results. We can observe that the auditing programs written in RAL are significantly more concise than the other two systems. For example, RAL program uses ~80× less lines of code than FSA in expressing an RCG-ranking task. This is because RAL is a domain-specific language for expressing the correlated failure risk auditing tasks. In addition, it is difficult to write task ranking components by using FSA, because FSA in principle cannot output accurate minimal RCGs and ranking components heavily rely on the results of minimal RCGs.

8.2 Evaluating Auditing Engine

We now compare the efficiency/accuracy trade-off of different auditing approaches, including RepAudit, MCS, and FSA, respectively.

We use FNSS [Saino et al. 2013], a standard and widely-accepted datacenter topology generator, to generate four topologies from a small-scale cloud deployment (with 1,344 devices) to a large-scale deployment (with 70,656 devices). All the topologies follow well-known fat tree datacenter model. Note that the topology configurations generated by FNSS are equivalent to realistic fat tree topologies [Mysore et al. 2009] in terms of our evaluation context. These topologies include the typical devices within a commercial datacenter: servers, Top-of-Rack (ToR) switches, aggregation switches, and core routers. Table 2 details the configurations of our evaluated topologies. Note that the topology configurations generated by FNSS are equivalent to real fat tree topologies in terms of our evaluation purposes. All the experiments in this section are conducted on a Dell Precision T3600 workstation equipped with Intel Xeon E5-1620 v2 Quad Core HT 3.7 GHz CPU and 16 GB memory.

Figure 8 compares the capabilities of evaluated tools by using them to rank the top-20 critical RCGs with size metric in Topology A, B and C. We only evaluate the performance of RepAudit on

Table 3. RAL-code generator processing time.

	Topology A	Topology B	Topology C	Topology D
Load time (ms.)	0.22	1.2762	12.65	33.004
Generation time (sec.)	0.2955	1.0245	7.353	23.279
Total time (sec.)	0.2957	1.0258	7.3657	23.312

Topology D, because MCS and FSA are too slow for Topology D. As shown in Figure 8, RepAudit not only runs much faster than the compared auditing tools MCS and FSA, but also obtains 100% accurate the top-20 critical RCGs. For MCS, which is implemented based on minimal cut-set algorithm, although it can also obtain the 100% accurate results, the running time is not acceptable in practice. For example, for the Topology C, MCS needs more than 2048 minutes to get the result. On the other hand, because FSA is implemented based on randomized sampling algorithm, it can run faster than MCS. However, as the sizes of target datasets increase, the accuracy of the FSA is significantly affected. For example in Topology C, FSA with 10^4 sampling rounds can only detect less than 20% critical RCGs. On the contrary, RepAudit can get 100% accurate results within 20 minutes. The reason RepAudit can work much better than MCS and FSA in both accuracy and efficiency aspects is RepAudit is built upon modern WP-MaxSAT solver, which is equipped with many smart heuristic algorithms.

Can RepAudit try Topology D? Furthermore, we also run a more challenging fat tree topology consisting of 5120 64-port switches and 65,536 servers. In such a huge dataset, RepAudit outputs the top-5 critical RCGs within 15 minutes; on the contrary, other two tools, MCS and FSA, are time-out (*i.e.*, > 48 hours).

8.3 Evaluating RAL-Code Generator

To evaluate the performance of our RAL-code generation, we measure the code generation time under different scales of datacenter topologies (shown in Table 2). In our experiments, we randomly select four replica servers, and generated RAL program reducing the failure probability of the selected replication deployment lower than 0.009. The scheme we choose is the `AddPath` scheme.

Table 3 shows the evaluation results. In Table 3, load time denotes the preprocessing time, and generation time denotes the time spent on generating RAL programs. We can observe two useful phenomena: 1) our code generation approach is scalable to large-scale datacenter network (even for Topology D); and 2) network path loading time (*i.e.*, preprocessing time for the RAL-code generator) is much faster than the running generation algorithm.

9 LIMITATIONS AND DISCUSSIONS

This section discusses the limitations of current RepAudit and potential solutions.

Correlated failures caused by dependencies other than network sources, *e.g.*, deep bugs. Our current RepAudit mainly focuses on correlated failures resulting from network-level components and their dependencies. However, in practice, many cloud service failures were caused by bugs within common software dependencies, such as packages and libraries [Gunawi et al. 2014, 2016]. In fact, collecting software dependencies has been out of our scope, because RepAudit is responsible for analyzing the structural data rather than acquiring this data. Given the fact that several existing systems [Zhao et al. 2016, 2014] offer automatic software dependency collection capability, RepAudit can be extended to handle the correlated failures caused by common bugs, as long as we can connect these tools to our framework.

Auditing tasks requiring solvers different than WP-MaxSAT. Our current RAL primitives heavily rely on WP-MaxSAT solver to ensure their efficient and accurate executions. However, for some practical auditing tasks, WP-MaxSAT solvers may not be the most suitable ones. As an illustration, suppose an administrator would want to introduce a new primitive which will return all the affected components if a certain set of components fails. In this case, the administrator should use an efficient reachability solver [Lal et al. 2012] for that task. Nevertheless, it is important to stress that our RepAudit prototype is modularly designed so that the administrator can easily add a new solver if she wants to extend the language with a new auditing primitive. She can also easily replace the currently used WP-MaxSAT solver with any other WP-MaxSAT solver.

Detecting RCGs across inter-cloud replications. The current RepAudit design cannot identify RCGs within inter-cloud replication systems, where each individual cloud provider is responsible for one replica. This is because in practice no provider is willing to share its own dependency information with other cloud providers. For example, as an application-level cloud provider, iCloud rents Amazon S3 and Microsoft Azure storage for reliability enhancement [Hardaware 2011]. If iCloud wants to use RepAudit to analyze its replication deployment, RepAudit needs to collect infrastructure information for both S3 and Azure storage to understand its own infrastructure dependencies. Unfortunately, the access may be difficult if not impossible as this information tends to be secret, proprietary information. Thus, the RepAudit cannot provide solution to the replication deployments across multiple individual infrastructure providers. Some alternative solutions include: 1) introducing a trusted third-party to collect all the infrastructure information from cloud providers (*e.g.*, EC2 and Azure for iCloud) and perform the RepAudit for target replication deployments, and 2) using secure multi-party computation (SMPC) [Yao 1982] to privately provide analysis. Xiao *et al.* [Xiao et al. 2013] and Zhai *et al.* [Zhai et al. 2013, 2014] proposed privacy-preserving approaches to auditing application services across multiple clouds.

10 RELATED WORK

Offering structural auditing to the cloud services has been advocated as an effective means to avoid correlated failures and ensure the reliability of replication in the cloud [Shah et al. 2007]. However, practical (*i.e.*, expressive, effective and efficient) auditing method still remains an open problem. To the best of our knowledge, RepAudit is the first effort offering expressive, effective and efficient auditing to cloud-scale replication deployments.

As the first effort for structural cloud auditing, INDaaS can quantify the independence of replication deployment of interest [Zhai et al. 2014]. INDaaS employs failure sampling algorithms to analyze fault graphs, which is the same as FSA. Compared with RepAudit, both effectiveness and efficiency of INDaaS are much worse than RepAudit (see the evaluation results performed by FSA in Sec. 8). More important, due to the lack of language-level support, INDaaS is too complex to be configured by administrators for expressing their tasks.

In the post-failure forensics (or called troubleshooting) field, diagnosis systems [Bahl et al. 2007; Barham et al. 2004; Chen et al. 2004, 2008; Dunagan et al. 2004; Kandula et al. 2005, 2009; Kompella et al. 2005; Peddycord III et al. 2012] and accountability systems [Haeberlen 2009; Haeberlen et al. 2010] identify failure root-causes after outages. Compared with proactive failure prevention techniques, *e.g.*, RepAudit and INDaaS, these troubleshooting approaches mainly have two disadvantages. First, all the troubleshooting efforts cannot avoid system downtime, because they aim to identify root causes of failures after outages occur. Second, existing investigations have shown troubleshooting systems eventually leads to prolonged failure recovery time in the face of complex cloud-scale systems [Wu et al. 2012].

Recent years, new language approaches have been developed to analyze dependency graphs representing systems or programs of interest, in order to offer safety guarantees [Huang et al. 2011; Johnson et al. 2015; von Hanxleden et al. 2014]. Existing dependency graph-based efforts mainly focus on checking and controlling data flow and information flow of programs or systems, rather than estimating correlated failure risks within interdependent structures underlying cloud-scale replication deployments. Because different system structures need very different dependency graph modeling, *e.g.*, PDG [Johnson et al. 2015], fault graph [Zhai et al. 2014, 2013] and attack trees [Huang et al. 2011; Zhai et al. 2015], existing dependency graph languages are not suitable to the correlated failure auditing problem.

To facilitate the management of datacenter networks, language-based approaches have been proposed to verify datacenter network properties or synthesize datacenter network rules and policies. In particular, NoD [Lopes et al. 2015] is used to check important properties of datacenter networks, *e.g.*, the reachability of some packets, through writing the corresponding specifications. McClurg *et al.* [McClurg et al. 2015] propose a synthesis approach that automatically produces datacenter network rule-update scripts that should be guaranteed to preserve specified properties. Different from RepAudit, these efforts aim to assist cloud administrators to understand and manage network traffic within their datacenter networks.

11 CONCLUSION

We have presented a novel language-based framework RepAudit for administrators to express diverse auditing tasks to cloud-scale replication systems. By comparing with state-of-the-art efforts, we demonstrate RepAudit is much easier to use, in the sense that we can express complex auditing tasks significantly more succinctly. Furthermore, RepAudit generates the auditing results much more efficiently and more accurately. To the best of our knowledge, RepAudit is the first effort capable of simultaneously achieving expressive, efficient and accurate auditing.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. We also thank Nikolaj Bjørner and Bryan Ford for their valuable feedback on the earlier version of this work.

REFERENCES

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

N. Alon and R. B. Boppana. 1987. The monotone circuit complexity of Boolean functions. *Combinatorica* 7, 1 (1987), 1–22.

Mario Alviano. 2015. Maxino: A fast MaxSAT solver. http://alviano.net/software/maxino/. (2015). Online; accessed Feb 24 2017.

Mario Alviano, Carmine Dodaro, and Francesco Ricca. 2015. A MaxSAT algorithm using cardinality constraints of bounded size. In *24th International Joint Conference on Artificial Intelligence (IJCAI)*.

Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. 2009. Solving Weighted partial MaxSAT through satisfiability testing. In *12th Theory and Applications of Satisfiability Testing (SAT)*.

Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. 2010. A new algorithm for weighted partial MaxSAT. In *24th Conference on Artificial Intelligence (AAAI)*.

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

Tomas Balyo, Marijn J. H. Heule, and Matti Jarvisalo. 2016. SAT Competition 2016 : Solver and Benchmark Descriptions. In *SAT*.

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

Alysson Neves Bessani, Miguel P. Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2011. DepSky: Dependable and Secure Storage in a Cloud-of-clouds. In *6th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*.

Peter Bodik, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. 2012. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM (SIGCOMM)*.

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

Danny Bradbury. 2016. The bigger they get, the harder we fall: Thinking our way out of cloud crash. http://www.theregister.co.uk/2016/07/29/bryan_ford_bigger_icebergs/. (2016).

Ang Chen, Yang Wu, Andreas Haeberlen, Boon Thau Loo, and Wenchao Zhou. 2017. Data provenance at Internet scale: Architecture, experiences, and the road ahead. In *8th Biennial Conference on Innovative Data Systems Research (CIDR)*.

Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2016. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *ACM SIGCOMM (SIGCOMM)*.

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

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

Ira Cohen, Jeffrey S. Chase, Moisés Goldszmidt, Terence Kelly, and Julie Symons. 2004. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

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

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

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

Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *5th ACM Symposium on Cloud Computing (SoCC)*.

Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why does the cloud stop computing? Lessons from hundreds of service outages. In *7th ACM Symposium on Cloud Computing (SoCC)*.

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

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

Devindra Hardaware. 2011. Apple's iCloud runs on Microsoft's Azure and Amazon's cloud. http://venturebeat.com/2011/09/03/icloud-azure-amazon/. (2011).

Heqing Huang, Su Zhang, Xinming Ou, Atul Prakash, and Karem A. Sakallah. 2011. Distilling critical attack graph surface iteratively through minimum-cost SAT solving. In *27th Annual Computer Security Applications Conference (ACSAC)*.

Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. ConfValley: A systematic configuration validation framework for cloud services. In *10th European Conference on Computer Systems (EuroSys)*.

Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and enforcing security guarantees via program dependence graphs. In *36th ACM Conference on Programming Language Design and Implementation (PLDI)*.

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

Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. 2005. Shrink: A Tool for Failure Diagnosis in IP Networks. In *MineNet*.

Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. 2009. Detailed Diagnosis in Enterprise Networks. In *ACM SIGCOMM (SIGCOMM)*.

Ramana Rao Kompella, Jennifer Yates, Albert G. Greenberg, and Alex C. Snoeren. 2005. IP Fault Localization Via Risk Modeling. In *2nd USENIX Symposium on Networked System Design and Implementation (NSDI)*.

Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. 2012. A solver for reachability modulo theories. In *24th International Conference on Computer Aided Verification (CAV)*.

Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos Kawazoe Aguilera, and Michael Walfish. 2011. Detecting failures in distributed systems with the Falcon spy network. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*.

Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked System Design and Implementation (NSDI)*.

Jedidiah McClurg, Hossein Hojjat, Pavol Cerný, and Nate Foster. 2015. Efficient synthesis of network updates. In *36th ACM Conference on Programming Language Design and Implementation (PLDI)*.

Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *ACM SIGCOMM (SIGCOMM)*.

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

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

Barry Peddycord III, Peng Ning, and Sushil Jajodia. 2012. On the Accurate Identification of Network Service Dependencies in Distributed Systems. In *26th Large Installation System Administration Conference (LISA)*.

Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling network verification using symmetry and surgery. In *43rd ACM Symposium on Principles of Programming Languages (POPL)*.

Patrick Reynolds, Charles Edwin Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. 2006. Pip: Detecting the unexpected in distributed systems. In *3rd Symposium on Networked Systems Design and Implementation (NSDI)*.

Lorenzo Saino, Cosmin Cocora, and George Pavlou. 2013. Fast Network Simulation Setup. https://github.com/fnss/fnss. (2013).

Mehul A. Shah, Mary Baker, Jeffrey C. Mogul, and Ram Swaminathan. 2007. Auditing to Keep Online Storage Services Honest. In *11th Workshop on Hot Topics in Operating Systems (HotOS)*.

Rew Steven. 2014. Rackspace Outage Nov 12th. http://www.realestatewebmasters.com/blogs/rew-steven/rackspace-outage-nov-12th/show/. (2014). Online; accessed Feb 24 2017.

The AWS Team. 2012. Summary of the October 22, 2012 AWS Service Event in the US-East Region. https://aws.amazon.com/message/680342/. (2012). Online; accessed Feb 24 2017.

Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquin Aguado, Stephen Mercer, and Owen O'Brien. 2014. SCCharts: sequentially constructive statecharts for safety-critical applications: HW/SW-synthesis for a conservative extension of synchronous statecharts. In *35th ACM Conference on Programming Language Design and Implementation (PLDI)*.

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

Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2014. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM (SIGCOMM)*.

Hongda Xiao, Bryan Ford, and Joan Feigenbaum. 2013. Structural Cloud Audits that Protect Private Information. In *ACM Cloud Computing Security Workshop (CCSW)*.

Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science (FOCS)*.

Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. 2013. An Untold Story of Redundant Clouds: Making Your Service Deployment Truly Reliable. In *9th Workshop on Hot Topics in Dependable Systems (HotDep)*.

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

Ennan Zhai, Liang Gu, and Yumei Hai. 2015. A risk-evaluation assisted system for service selection. In *International Conference on Web Services (ICWS)*.

Ennan Zhai, David Isaac Wolinsky, Hongda Xiao, Hongqiang Liu, Xueyuan Su, and Bryan Ford. 2013. *Auditing the Structural Reliability of the Clouds.* Technical Report YALEU/DCS/TR-1479. Department of Computer Science, Yale University. Available at http://cpsc.yale.edu/sites/default/files/files/tr1479.pdf.

Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. 2011a. Secure network provenance. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*.

Wenchao Zhou, Qiong Fei, Shengzhi Sun, Tao Tao, Andreas Haeberlen, Zachary G. Ives, Boon Thau Loo, and Micah Sherr. 2011b. NetTrails: a declarative platform for maintaining and querying provenance in distributed systems. In *ACM International Conference on Management of Data (SIGMOD)*.