Efficient Privacy-preserving Calculation of Cloud Independence

Cameron Yick

Advisor: Avi Silbershatz

Collaborators: William Dower, Ennan Zhai

Motivation

Today's cloud customers intend to replicate important data and states of their applications across multiple cloud providers to ensure the availability and reliability of their applications. These seemingly reasonable reliability enhancement efforts, however, might be undermined by correlated failures resulting from infrastructure components and software vulnerabilities shared by these redundant cloud providers. For example, a recent investigation study reported that many cloud storage providers employ a buggy Apache Thrift library. Suppose a cloud customer rents two individual cloud storage providers (sharing the Thrift library) for redundancy. If an attacker compromises and triggers the vulnerability within Thrift library, the two cloud providers would become unavailability simultaneously. Even worse, cloud providers tend to keep their internal component dependencies secret due to business concerns, thus making it hard for cloud customers to diagnose the root causes of such correlated failures.

System Design

In this project, we design and develop a novel auditing system, iAudit, that can efficiently quantify the independence of inter-cloud replications through a fine-grained scheme, while preserving the business privacy of audited cloud providers. With the help of iAudit, a cloud customer can select the most independent replication deployment for her data or application adoption, thus preventing correlated failure risks rather than diagnosing or troubleshooting root causes after failures occur - which is too late. The key insight of iAudit is to reduce the privacy-preserving independence auditing to a private fault graph similarity computation problem. We address the latter by implementing two novel components: a scalable fault graph analysis engine based on Weighted MaxSAT solver and a weighted private Jaccard similarity protocol inspired by weighted sampling theory. To evaluate the prototype, we will use several zoo machines to emulate different cloud providers and measure the performance of our prototype to demonstrate the efficiency.

My collaborator William Dower implemented the first component, he documents his results in a separate project. In this report, I document the design and

results with implementing the second component, the weighted private Jaccard similiarity protocol.

Implementation Details

Computing Intersection Privately: The Vaidya-Clifton Protocol

Each cloud provider has one node which runs the `worker.py` script, which runs the a client server for taking part in the ring pass protocol. The script is configured using a json file which contains shared public keys necessary for taking part in the ring-pass encryption, as well as a list of numbers to use for the minhash functionality. This script utilizes the `numpy` library to enable fast vectorized operations, and the Python Flask framework to run the server.

The iAudit recommender system runs the `master.py`, `setup.py`, and `trigger.py` scripts on an independent computer.

- `setup.py` is responsible for distributing shared modulus keys, configuration files, and minhash functions with each member of the ring, and is configured by a master public configuration file.
- `master.py` is a simple server script designed to receive data from the `worker.py` scripts.
- `trigger.py` asyncronously dispatches commands to each worker, instructing each worker to encrypt its local data and pass that data onto its neighbor. After the last pass is complete, this script computes the intersection scores for all pairs of workers, and outputs the independence rankings in descending order.

All systems make use of the `iaudit` module, which contains utility functions for generating acceptable large primes, modulus keys, and encrypting the cutsets using the `murmurhash` library. It also contains a `Cutset` class to allow the user to easily between encryption methods when obfuscating the cutsets.

Computing Intersection in a Fine-Grained Manner

To ensure the independence of recommendations, the system must consider the fact that different vulnerabilities have different degrees of significance if they are shared. Counting 2 minor shared vulnerabilities as a bigger problem than 1 critical shared vulnerability would lead to inaccurate rankings.

In order to extend Jaccard similarity to support weighted inputs, `iaudit`'s automatically handles weighted inputs by creating creates copies of each cutset, appends the index of the duplicate to the cutset name, and hashes the results. In order to keep the Python memory requirements down, this functionality is achieved through the use of Python generators, which generates each item on-the-fly instead of keeping everything in memory.

Computing Intersection with Sampling

A simple minhash function scheme is used to hash the outputs of the cutsets, which have already been processed by murmurhash.

$ h = (a * x + b) % c $

where `x` is the input message, `a` and `b` are random integers less than the maximum value of `x`. `c` is a prime number slightly larger than the miaxmum value of `x`. Each unique "minhash" function is a unique pair of `a`, `b`. Since all of our murmurhash functions result in 128 bit numbers, `x`'s maximum value is' 2^128 - 1. In order to preserve the commutative encryption property, all `worker.py` functions must use the same pairs of minhash parameters.

With some simple tests involving 3 clouds, ~ 10 cutsets per provider, and 30 minhashes, we found that sampling returned the same rankings as the non-sampling method. Further benchmarks using real clouds with larger cutsets are necessary to identify what the ideal number of minhash functions is needed for each problem size.

Performance Results

The original plan of testing the scheme on the multiple machines on the Zoo cluster could not be realized due to a firewall restriction on inter-node communications. However, equivalent benchmark tests were conducted on a single zoo computer to create the following graphs. Because of this difference, overhead from network communication is negligible, as all server programs were running on the same computer. The configuration files to run these tests are stored in the `test_configs` folder.

Effect of Larger Input Size

I tested the iAudit system using minhash functionality, using input sizes on each node of 10, 100, and 1000 cutsets per node. The sample input cutsets were provided from an output of the first portion of the iAudit protocol. As the cutset size on each node increased, the non-sampled version of the program increased in runtime linearly. However as expected, the runtime for the sampling based method ran in near constant time.

Effect of Extra Minhashes

As anticipated, the compute time scales linearly with an increase of the size of the inputs cutset on each worker node. This highlights the importance of picking a minhash count that is large enough to get correct ranking outputs, but not so large that the practical runtime becomes unreasonably long.

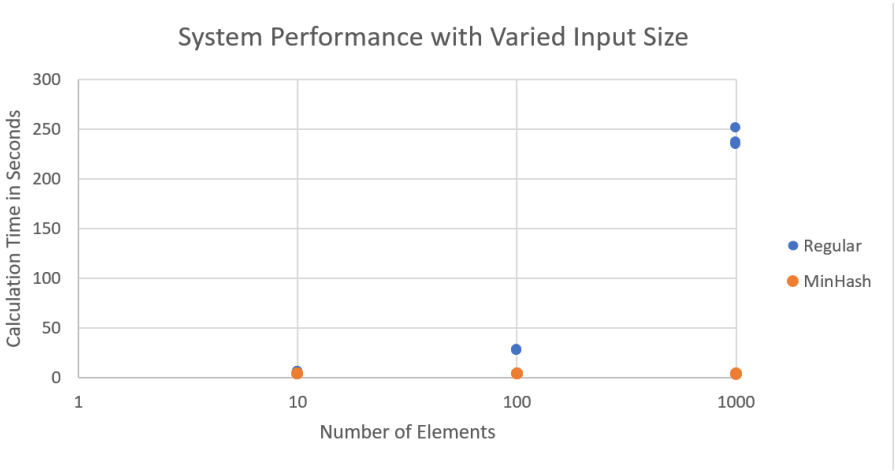

Figure 1: inputs_graph

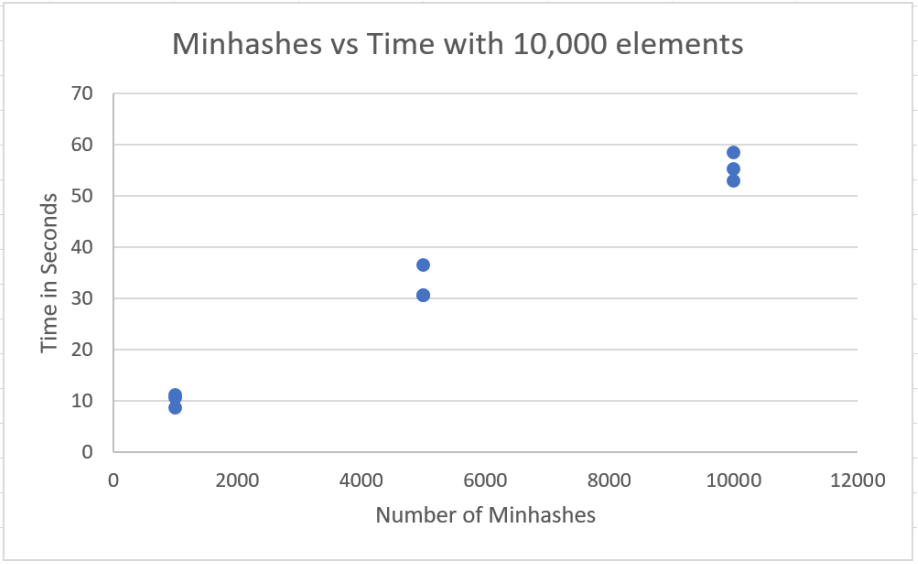

Figure 2: minhashes_graph

Avenues of Future Work

To improve performance, there are many opportunities to improve the system's throughput. First, each cloud provider could use a parallel computing framework such as OpenMP, CUDA, or OpenMPI to perform the hashing operations in parallel. Secondly, the computation of the intersection on the iAudit worker could be sped up through the usage of a Bloom filter. Thirdly, further investigation is necessary to perform the commutative or murmurhashes using smaller numbers of bits. However, these options can be configured using the `config.json` file that gets passed to `setup.py` at the start of each system run to enable future tuning.

To ensure the program returns the same results when sampling and minhash techniques are combined, more work is necessary to identify the threshhold at which there aren't enough minhash functions to compute the correct answer. Here, the classic engineering tradeoff between speed and accuracy rears its familiar head.

Lastly, in order to get the system to be adopted by commercial cloud providers, we need benchmarks for the system with the presence of real network traffic and latency. Further tests on the zoo nodes will become possible if the aforementioned firewall restrictions are lifted. The system should be combined with the graph generator portion of iAudit that William developed.

Summary

In this project, I implemented the second part of the iAudit Independence as as Service (INDaaS) architecture. I wrote programs that are configurable using `json` files, and created scripts that can run on each cloud provider as well as a central Auditor node. These scripts overcome the primary technical challenges preventing application developers from being able to know the independence of the clouds they deploy to, namely private calculation of independence, calculating indepdence in a fine-grained way, and computing independence efficiently. I also provide benchmarks indicating that the system performs as we theoretically anticipated, and outline some steps for future work.

References

Zander, Sebastian, Lachlan LH Andrew, and Grenville Armitage. "Scalable private set intersection cardinality for capture-recapture with multiple private datasets." Centre for Advanced Internet Architectures, Technical Report 130930A (2013).

Zhai, E., Chen, R., Wolinsky, D. I., & Ford, B. (2014, October). Heading Off Correlated Failures through Independence-as-a-Service. In OSDI (pp. 317-334).