Volume 3, No. 11, November 2012

Journal of Global Research in Computer Science

ISSN-2229-371X

REVIEW ARTICLE

Available Online at www.jgrcs.info

TRUSTED CLOUD – A SOLUTION FOR CLOUD CARTOGRAPHY

E. Kanimozhi
Assistant Professor,
Dept of Computer Science and Engineering,
Vivekanandha Engineering college for women,Sankari, India
kanivijay@gmail.com

Abstract: Cloud computing provides people a way to share large mount of distributed resources belonging to different organizations. That is a good way to share many kinds of distributed resources, but it also makes security problems more complicate and more important for users than before. Cloud cartography could be used by an attacker who wanted to place his own VM next to a target's VM and exploit vulnerabilities. To create the map, the attacker would deploy a large number of VMs in the service provider's cloud. He could then use the information he gets back from the service provider about his deployments to get a sense of how the provider assigns IP addresses for different instance types and accounts.

Key words: cloud computing, trusted computing, cartography, network probing, brute – force attack, side- channel attack.

INTRODUCTION

Seven of the specific security issues Gartner says customers should raise with vendors before selecting a cloud vendor.

Privileged user access: Sensitive data processed outside the enterprise brings with it an inherent level of risk, because outsourced services bypass the "physical, logical and personnel controls" IT shops exert over in-house programs. Get as much information as you can about the people who manage your data. "Ask providers to supply specific information on the hiring and oversight of privileged administrators, and the controls over their access," Gartner says.

Regulatory compliance: Customers are ultimately responsible for the security and integrity of their own data, even when it is held by a service provider. Traditional service providers are subjected to external audits and security certifications. Cloud computing providers who refuse to undergo this scrutiny are "signaling that customers can only use them for the most trivial functions," according to Gartner.

Data location: When you use the cloud, you probably won't know exactly where your data is hosted. In fact, you might not even know what country it will be stored in. Ask providers if they will commit to storing and processing data in specific jurisdictions, and whether they will make a contractual commitment to obey local privacy requirements on behalf of their customers, Gartner advises.

Data segregation: Data in the cloud is typically in a shared environment alongside data from other customers. Encryption is effective but isn't a cure-all. "Find out what is done to segregate data at rest," Gartner advises. The cloud provider should provide evidence that encryption schemes were designed and tested by experienced specialists. "Encryption accidents can make data totally unusable, and even normal encryption can complicate availability," Gartner says.

Recovery: Even if you don't know where your data is, a cloud provider should tell you what will happen to your data and service in case of a disaster. "Any offering that does not replicate the data and application infrastructure across multiple sites is vulnerable to a total failure," Gartner says. Ask your provider if it has "the ability to do a complete restoration, and how long it will take."

Investigative support: Investigating inappropriate or illegal activity may be impossible in cloud computing, Gartner warns. "Cloud services are especially difficult to investigate, because logging and data for multiple customers may be co-located and may also be spread across an ever-changing set of hosts and data centers. If you cannot get a contractual commitment to support specific forms of investigation, along with evidence that the vendor has already successfully supported such activities, then your only safe assumption is that investigation and discovery requests will be impossible."

Long-term viability: Ideally, your cloud computing provider will never go broke or get acquired and swallowed up by a larger company. But you must be sure your data will remain available even after such an event. "Ask potential providers how you would get your data back and if it would be in a format that you could import into a replacement application," Gartner says.

THE EC2 SERVICE

By far the best known example of a third-party compute cloud is Amazon’s Elastic Compute Cloud (EC2) service, which enables users to flexibly rent computational resources for use by their applications [5]. EC2 provides the ability to run Linux, FreeBSD, Open Solaris and Windows as guest operating systems within a virtual machine (VM) provided by a version of the Xen hypervisor [9].1 The hypervisor plays the role of a virtual machine monitor and provides isolation between VMs, intermediating access to physical memory and devices. A privileged virtual machine, called Domain0 (Dom0) in the Xen vernacular, is used to manage guest images, their physical resource provisioning, and any access control rights. In EC2 the Dom0 VM is configured to route packets for its guest images and reports itself as a hop in trace routes.

When first registering with EC2, each user creates an account—uniquely specified by its contact e-mail address and provides credit card information for billing compute and I/O charges. With a valid account, a user creates one or more VM images, based on a supplied Xen-compatible kernel, but with an otherwise arbitrary configuration. He can run one or more copies of these images on Amazon’s network of machines. One such running image is called an instance, and when the instance is launched, it is assigned to a single physical machine within the EC2 network for its lifetime; EC2 does not appear to currently support live migration of instances, although this should be technically feasible. By default, each user account is limited to 20 concurrently running instances.

In addition, there are three degrees of freedom in specifying the physical infrastructure upon which instances should run. At the time of this writing, Amazon provides two “regions”, one located in the United States and the more recently established one in Europe. Each region contains three “availability zones” which are meant to specify infrastructures with distinct and independent failure modes.

(e.g., with separate power and network connectivity). When requesting launch of an instance, a user specifies the region and may choose a specific availability zone (otherwise one is assigned on the user’s behalf). As well, the user can specify an “instance type”, indicating a particular combination of computational power, memory and persistent storage space available to the virtual machine. There are five Linux instance types documented at present, referred to as ‘m1.small’, ‘c1.medium’, ‘m1.large’, ‘m1.xlarge’, and ‘c1.xlarge’. The first two are 32-bit architectures, the latter three are 64-bit. To give some sense of relative scale, the “small compute slot” (m1.small) is described as a single virtual core providing one ECU (EC2 Compute Unit, claimed to be equivalent to a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor) combined with 1.7 GB of memory and 160 GB of local storage, while the “large compute slot” (m1.large) provides 2 virtual cores each with 2 ECUs, 7.5GB of memory and 850GB of local storage. As expected, instances with more resources incur greater hourly charges (e.g., ‘m1.small’ in the United States region is currently $0.10 per hour, while ‘m1.large’ is currently $0.40 per hour). When launching an instance, the user specifies the instance type along with a compatible virtual machine image.

Given these constraints, virtual machines are placed on available physical servers shared among multiple instances. Each instance is given Internet connectivity via both an external IPv4 address and domain name and an internal RFC 1918 private address and domain name. For example, an instance might be assigned external IP 75.101.210.100, external name ec2-75-101-210 100. compute1. amazonaws.com, internal IP 10.252.146.52, and internal name domU-12-31-38-00-8D-C6.compute-internal. Within the cloud, both domain names resolve to the internal IP address; outside the cloud the external name is mapped to the external IP address.

NETWORK PROBING

Network probing is used, both to identify public services hosted on EC2 and to provide evidence of co-residence (that two instances share the same physical server). In particular, we utilize nmap, hoping, and get to perform network probes to determine liveness of EC2 instances. We use nmap to perform TCP connect probes, which attempt to complete a 3-way hand-shake between a source and target. We use hoping to perform TCPSYN trace routes, which iteratively sends TCP SYN packets with increasing time-to-lives (TTLs) until no ACK is received. Both TCP connect probes and SYN trace routes require a target port; we only targeted ports 80 or 443. We used wget to retrieve web pages, but capped so that at most 1024 bytes are retrieved from any individual web server.

We distinguish between two types of probes: external probes and internal probes. A probe is external when it originates from a system outside EC2 and has destination an EC2 instance. A probe is internal if it originates from an EC2 instance (under our control) and has destination another EC2 instance. This dichotomy is of relevance particularly because internal probing is subject to Amazon’s acceptable use policy, whereas external probing is not.

We use DNS resolution queries to determine the external name of an instance and also to determine the internal IP address of an instance associated with some public IP address. The latter queries are always performed from an EC2 instance.

CLOUD CARTOGRAPHY

In this section we ‘map’ the EC2 service to understand where potential targets are located in the cloud and the instance creation parameters needed to attempt establishing co-residence of an adversarial instance. This will speed up significantly adversarial strategies for placing a malicious VM on the same machine as a target

To map EC2, we begin with the hypothesis that different availability zones are likely to correspond to different internal IP address ranges and the same may be true for instance types as well. Thus, mapping the use of the EC2 internal address space allows an adversary to determine which IP addresses correspond to which creation parameters. Moreover, since EC2’s DNS service provides a means to map public IP address to private IP address, an adversary might use such a map to infer the instance type and availability zone of a target service—thereby dramatically reducing the number of instances needed before a co-resident placement is achieved.

We evaluate this theory using two data sets: one created by enumerating public EC2-based web servers using external probes and translating responsive public IPs to internal IPs (via DNS queries within the cloud), and another created by launching a number of EC2 instances of varying types and surveying the resulting IP address assigned. To fully leverage the latter data, we present a heuristic algorithm that helps label /24 prefixes with an estimate of the availability zone and instance type of the included Internal IPs. These heuristics utilize several beneficial features of EC2's addressing regime. The output of this process is a map of the internal EC2 address space which allows one to estimate the availability zone and instance type of any target public EC2 server. Next, we enumerate a set of public EC2-based Web servers

a. All IPs from a /16 are from the same availability zone.
b. A /24 inherits any included sampled instance type.
c. A /24 containing a Dom0 IP address only contains Dom0 IP addresses. We associate to this /24 the type of the Dom0's associated instance.
d. All /24's between two consecutive Dom0 /24's inherit the former's associated type.

The last heuristic, which enables us to label /24's that have no included instance, is derived from the observation that Dom0 IPs are consistently assigned a prefix that immediately precedes the instance IPs they are associated with. There were 869 /24's in the data, and applying the heuristics resulted in assigning a unique zone and unique type to 723 of these; a unique zone and two types to 23 of these; and left 123 unlabeled. These last were due to areas (such as the lower portion of 10.253.0.0/16) for which we had no sampling data at all.

While the map might contain errors (for example, in areas of low instance sample numbers), we have yet to encounter an instance that contradicts the /24 labeling and we used the map for many of the future experiments. For instance, we applied it to a subset of the public servers derived from our survey, those that responded to wget requests with an HTTP 200 or 206. The resulting 6 057 servers were used as stand-ins for targets in some of the experiments in Section 7.

Preventing cloud cartography: Providers likely have incentive to prevent cloud cartography for several reasons, beyond the use we outline here (that of exploiting placement vulnerabilities). Namely, they might wish to hide their infrastructure and the amount of use it is enjoying by customers. Several features of EC2 made cartography significantly easier. Paramount is that local IP addresses are statically (at least over the observed period of time) associated to availability zone and instance type. Changing this would likely make administration tasks more challenging (and costly) for providers. Also, using the map requires translating a victim instance's external IP to an internal IP, and the provider might inhibit this by isolating each account's view of the internal IP address space (e.g. via VLANs and bridging). Even so, this would only appear to slow down our particular technique for locating an instance in the LAN—one might instead use ping timing measurements or trace routes (both discuss more in the next section) to help "triangulate" on a victim.

DETERMINING CO-RESIDENCE

Given a set of targets, the EC2 map from the previous section educates choice of instance launch parameters for attempting to achieve placement on the same physical machine. Recall that we refer to instances that are running on the same physical machine as being co-resident. In this section we describe several easy-to-implement co-residence checks. Looking ahead, our eventual check of choice will be to compare instances' Dom0 IP addresses. We confirm the accuracy of this (and other) co-residence checks by exploiting a hard-disk-based covert channel between EC2 instances.

Network-based co-residence checks: Using our experience running instances while mapping EC2 and inspecting data collected about them, we identify several potential methods for checking if two instances are co-resident.
Namely, instances are likely co-resident if they have

a. matching Dom0 IP address,
b. small packet round-trip times, or
c. numerically close internal IP addresses (e.g. within 7).

As mentioned, an instance's network traffic's first hop is the Dom0 privileged VM. An instance owner can determine its Dom0 IP from the first hop on any route out from the instance. One can determine an uncontrolled instance's Dom0 IP by performing a TCP SYN trace route to it (on some open port) from another instance and inspecting the last hop. For the second test, we noticed that round-trip times (RTTs) required a "warm-up": the first reported RTT in any sequence of probes was almost always an order of magnitude slower than subsequent probes. Thus for this method we perform 10 probes and just discard the first. The third check makes use of the manner in which internal IP addresses appear to be assigned by EC2. The same Dom0 IP will be shared by instances with a contiguous sequence of internal IP addresses.

Veracity of the co-residence checks. We verify the correctness of our network-based co-residence checks using as ground truth the ability to send messages over a cross-VM covert channel. That is, if two instances (under our control) can successfully transmit via the covert channel then they are co-resident, otherwise not. If the checks above (which do not require both instances to be under our control) have sufficiently low false positive rates relative to this check, then we can use them for inferring co-residence against arbitrary victims. We utilized for this experiment a hard-disk-based covert channel. At a very high level, the channel works as follows.

To send a one bit, the sender instance reads from random locations on a shared disk volume. To send a zero bit, the sender does nothing. The receiver times reading from a fixed location on the disk volume. Longer read times mean a 1 is being set, shorter read times give a 0.

We performed the following experiment.
Three EC2 accounts were utilized: a control, a victim, and a probe. (The "victim" and "probe" are arbitrary labels, since they were both under our control.) All instances launched were of type m1.small.

Two instances were launched by the control account in each of the three availability zones. Then 20 instances on the victim account and 20 instances on the probe account were launched, all in Zone 3.

We determined the Dom0 IPs of each instance. For each (ordered) pair (A,B) of these 40 instances, if the Dom0 IPs passed (check 1) then we had A probe B and each control to determine packet RTTs and we also sent a 5-bit message from A to B over the hard-drive covert channel.

EXPLOITING PLACEMENT IN EC2

Consider an adversary wishing to attack one or more EC2 instances. Can the attacker arrange for an instance to be placed on the same physical machine as (one of) these victims? In this section we assess the feasibility of achieving co-residence with such target victims, saying the attacker is successful if he or she achieves good coverage (co-residence with a notable fraction of the target set).

The brute-force strategy has an attacker simply launch many instances over a relatively long period of time. Such a naive strategy already achieves reasonable success rates (though for relatively large target sets). A more refined strategy has the attacker target recently-launched instances. This takes advantage of the tendency for EC2 to assign fresh instances to the same small set of machines.

Brute-forcing placement:

We start by assessing an obvious attack strategy: run numerous instances over a (relatively) long period of time and see how many targets one can achieve co-residence with. While such a brute-force strategy does nothing clever (once the results of the previous sections are in place), our hypothesis is that for large target sets this strategy will already allow reasonable success rates.

The strategy works as follows. The attacker enumerates a set of potential target victims. The adversary then infers which of these targets belong to a particular availability zone and are of a particular instance type using the map Then, over some (relatively long) period of time the adversary repeatedly runs probe instances in the target zone and of the target type. Each probe instance checks if it is co-resident with any of the targets. If not the instance is quickly terminated.

We experimentally gauged this strategy's potential efficacy. We utilized as "victims" the subset of public EC2- based web servers surveyed in Section 5 that responded with HTTP 200 or 206 to a wget request on port 80.

The gap in time between our survey of the public EC2 servers and the launching of probes means that new web servers or ones that changed IPs were not detected, even when we in fact achieved co-residence with them.

Our results suggest that even a very naive attack strategy can successfully achieve co-residence against a not-so-small fraction of targets. Of course, we considered here a large target set, and so we did not provide evidence of efficacy against an individual instance or a small sets of targets. We observed very strong sequential locality in the data, which hinders the effectiveness of the attack. In particular, the growth in target set coverage as a function of number of launched probes levels off quickly. This suggests that fuller coverage of the target set could require many more probes.

Abusing Placement Locality:

We would like to find attack strategies that do better than brute-force for individual targets or small target sets. Here we discuss an alternate adversarial strategy. We assume that an attacker can launch instances relatively soon after the launch of a target victim. The attacker then engages in instance flooding: running as many instances in parallel as possible (or as many as he or she is willing to pay for) in the appropriate availability zone and of the appropriate type. While an individual account is limited to 20 instances, it is trivial to gain access to more accounts. As we show, running probe instances temporally near the launch of a victim allows the attacker to effectively take advantage of the parallel placement locality exhibited by the EC2 placement algorithms.

But why would we expect that an attacker can launch instances soon after a particular target victim is launched? Here the dynamic nature of cloud computing plays well into the hands of creative adversaries. Recall that one of the main features of cloud computing is to only run servers when needed. This suggests that servers are often run on instances, terminated when not needed, and later run again.

So for example, an attacker can monitor a server's state (e.g., via network probing), wait until the instance disappears, and then if it reappears as a new instance, engage in instance flooding. Even more interestingly, an attacker might be able to actively trigger new victim instances due to the use of auto scaling systems. These automatically grow the number of instances used by a service to meet increases in demand.

Patching placement vulnerabilities:

The EC2 placement algorithms allow attackers to use relatively simple strategies to achieve co-residence with victims (that are not on fully-allocated machines). As discussed earlier, inhibiting cartography or co-residence checking (which would make exploiting placement more difficult) would seem insufficient to stop a dedicated attacker. On the other hand, there is a straightforward way to "patch" all placement vulnerabilities: offload choice to users. Namely, let users request placement of their VMs on machines that can only be populated by VMs from their (or other trusted) accounts.

In exchange, the users can pay the opportunity cost of leaving some of these machines under-utilized. In an optimal assignment policy (for any particular instance type), this additional overhead should never need to exceed the cost of a single physical machine.

CROSS-VM INFORMATION LEAKAGE

The previous sections have established that an attacker can often place his or her instance on the same physical machine as a target instance. In this section, we show the ability of a malicious instance to utilize side channels to learn information about co-resident instances. Namely we show that (time-shared) caches allow an attacker to measure when other instances are experiencing computational load.

Leaking such information might seem innocuous, but in fact it can already be quite useful to clever attackers. We introduce several novel applications of this side channel: robust co-residence detection (agnostic to network configuration), surreptitious detection of the rate of web traffic a co-resident site receives, and even timing keystrokes by an honest user (via SSH) of a co-resident instance. For the keystroke timing attack, we performed experiments on an EC2-like virtualized environment.

On stealing cryptographic keys. There has been a long line of work on extracting cryptographic secrets via cache-based side channels. Such attacks, in the context of third-party compute clouds, would be incredibly damaging—and since the same hardware channels exist, are fundamentally just as feasible. In practice, cryptographic cross-VM attacks turn out to be somewhat more difficult to realize due to factors such as core migration, coarser scheduling algorithms, double indirection of memory addresses, and unknown load from other instances and a fortuitous choice of CPU configuration

The side channel attacks we report on in the rest of this section are more coarse-grained than those required to extract cryptographic keys. While this means the attacks extract less bits of information, it also means they are more robust and potentially simpler to implement in noisy environments such as EC2. Other channels; denial of service. Not just the data cache but any physical machine resources multiplexed between the attacker and target forms a potentially useful channel: network access, CPU branch predictors and instruction cache

The Hadoop Approach:

Hadoop is designed to efficiently process large volumes of information by connecting many commodity computers together to work in parallel. The theoretical 1000-CPU machine described earlier would cost a very large amount of money, far more than 1,000 single-CPU or 250 quad-core machines. Hadoop will tie these smaller and more reasonably priced machines together into a single cost-effective compute cluster.

Comparison to Existing Techniques:

Performing computation on large volumes of data has been done before, usually in a distributed setting. What makes Hadoop unique is its **simplified programming model** which allows the user to quickly write and test distributed systems, and its **efficient, automatic distribution of data and work across machines** and in turn utilizing the underlying parallelism of the CPU cores.

Grid scheduling of computers can be done with existing systems such as Condor. But Condor does not automatically distribute data: a separate SAN must be managed in addition to the compute cluster. Furthermore, collaboration between multiple compute nodes must be managed with a communication system such as MPI. This programming model is challenging to work with and can lead to the introduction of subtle errors.

Data Distribution:

In a Hadoop cluster, data is distributed to all the nodes of the cluster as it is being loaded in. The Hadoop Distributed File System (HDFS) will split large data files into chunks which are managed by different nodes in the cluster. In addition to this each chunk is replicated across several machines, so that a single machine failure does not result in any data being unavailable. An active monitoring system then re-replicates the data in response to system failures which can result in partial storage. Even though the file chunks are replicated and distributed across several machines, they form a single namespace, so their contents are universally accessible.

Data is conceptually **record-oriented** in the Hadoop programming framework. Individual input files are broken into lines or into other formats specific to the application logic. Each process running on a node in the cluster then processes a subset of these records. The Hadoop framework then schedules these processes in proximity to the location of data/records using knowledge from the distributed file system. Since files are spread across the distributed file system as chunks, each compute process running on a node operates on a subset of the data. Which data operated on by a node is chosen based on its locality to the node: most data is read from the local disk straight into the CPU, alleviating strain on network bandwidth and preventing unnecessary network transfers. This strategy of **moving computation to the data**, instead of moving the data to the computation allows Hadoop to achieve high data locality which in turn results in high performance.

Map Reduce: Isolated Processes:

Hadoop limits the amount of communication which can be performed by the processes, as each individual record is processed by a task in isolation from one another. While this sounds like a major limitation at first, it makes the whole framework much more reliable. Hadoop will not run just any program and distribute it across a cluster. Programs must be written to conform to a particular programming model, named "MapReduce."

In Map Reduce, records are processed in isolation by tasks called *Mappers*. The output from the Mappers is then brought together into a second set of tasks called *Reducers*, where results from different mappers can be merged together.

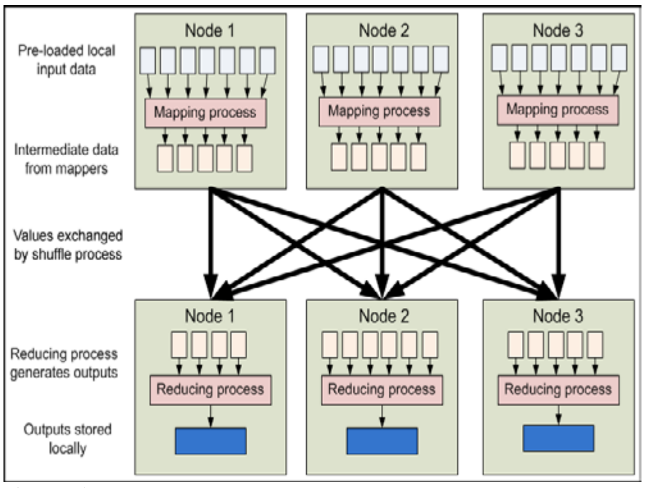

Figure 1

Distributed File System Basics:

A distributed file system is designed to hold a large amount of data and provide access to this data to many clients distributed across a network. There are a number of distributed file systems that solve this problem in different ways.

NFS, the Network File System, is the most ubiquitous distributed file system. It is one of the oldest still in use. While its design is straightforward, it is also very constrained. NFS provides remote access to a single logical volume stored on a single machine. An NFS server makes a portion of its local file system visible to external clients. The clients can then mount this remote file system directly into their own Linux file system, and interact with it as though it were part of the local drive.

One of the primary advantages of this model is its transparency. Clients do not need to be particularly aware that they are working on files stored remotely. The existing standard library methods like open(), close(), fread(), etc. will work on files hosted over NFS.

But as a distributed file system, it is limited in its power. The files in an NFS volume all reside on a single machine. This means that it will only store as much information as can be stored in one machine, and does not provide any reliability guarantees if that machine goes down (e.g., by replicating the files to other servers). Finally, as all the data is stored on a single machine, all the clients must go to this machine to retrieve their data. This can overload the server if a large number of clients must be handled. Clients must also always copy the data to their local machines before they can operate on it.

HDFS is designed to be robust to a number of the problems that other DFS's such as NFS are vulnerable to. In particular:

a. HDFS is designed to store a very large amount of information (terabytes or petabytes). This requires spreading the data across a large number of machines. It also supports much larger file sizes than NFS.
b. HDFS should store data reliably. If individual machines in the cluster malfunction, data should still be available.
c. HDFS should provide fast, scalable access to this information. It should be possible to serve a larger number of clients by simply adding more machines to the cluster.
d. HDFS should integrate well with Hadoop Map Reduce, allowing data to be read and computed upon locally when possible.

But while HDFS is very scalable, its high performance design also restricts it to a particular class of applications; it is not as general-purpose as NFS. There are a large number of additional decisions and trade-offs that were made with HDFS. In particular:

a. Applications that use HDFS are assumed to perform long sequential streaming reads from files. HDFS is optimized to provide streaming read performance; this comes at the expense of random seek times to arbitrary positions in files.
b. Data will be written to the HDFS once and then read several times; updates to files after they have already been closed are not supported. (An extension to Hadoop will provide support for appending new data to the ends of files; it is scheduled to be included in Hadoop 0.19 but is not available yet.)
c. Due to the large size of files, and the sequential nature of reads, the system does not provide a mechanism for local caching of data. The overhead of caching is great enough that data should simply be re-read from HDFS source.
d. Individual machines are assumed to fail on a frequent basis, both permanently and intermittently. The cluster must be able to withstand the complete failure of several machines, possibly many happening at the same time (e.g., if a rack fails all together). While performance may degrade proportional to the number of machines lost, the system as a whole should not become overly slow, nor should information be lost. Data replication strategies combat this problem.

The design of HDFS is based on the design of **GFS**, the Google File System. Its design was described in a paper published by Google.

HDFS is a block-structured file system: individual files are broken into blocks of a fixed size. These blocks are stored across a cluster of one or more machines with data storage capacity. Individual machines in the cluster are referred to as **Data Nodes**. A file can be made of several blocks, and they are not necessarily stored on the same machine; the target machines which hold each block are chosen randomly on a block-by-block basis. Thus access to a file may require the cooperation of multiple machines, but supports file sizes far larger than a single-machine DFS; individual files can require more space than a single hard drive could hold.

Starting HDFS:

Now we must format the file system that we just configured:
user@namenode:hadoop$ bin/hadoop namenode -format
This process should only be performed once. When it is complete, we are free to start the distributed file system:
user@namenode:hadoop$ bin/start-dfs.sh

This command will start the NameNode server on the master machine (which is where the start-dfs.sh script was invoked). It will also start the DataNode instances on each of the slave machines. In a single-machine "cluster," this is the same machine as the NameNode instance. On a real cluster of two or more machines, this script will ssh into each slave machine and start a DataNode instance.

Interacting With HDFS:

This section will familiarize you with the commands necessary to interact with HDFS, loading and retrieving data, as well as manipulating files. This section makes extensive use of the command-line.

The bulk of commands that communicate with the cluster are performed by a monolithic script named bin/hadoop. This will load the Hadoop system with the Java virtual machine and execute a user command. The commands are specified in the following form:
user@machine:hadoop$ bin/hadoop *moduleName -cmd args...*

The *moduleName* tells the program which subset of Hadoop functionality to use. *-cmd* is the name of a specific command within this module to execute. Its arguments follow the command name.

Two such modules are relevant to HDFS: **dfs** and **dfsadmin**. Their use is described in the sections below.

Shutting Down HDFS:

If you want to shut down the HDFS functionality of your cluster (either because you do not want Hadoop occupying memory resources when it is not in use, or because you want to restart the cluster for upgrading, configuration changes, etc.), then this can be accomplished by logging in to the NameNode machine and running:

```
someone@namenode:hadoop$ bin/stop-dfs.sh
```

This command must be performed by the same user who started HDFS with bin/start-dfs.sh.

USING HDFS IN MAPREDUCE

The HDFS is a powerful companion to Hadoop Map Reduce. By setting the fs.default.name configuration option to point to the Name Node (as was done above), Hadoop Map Reduce jobs will automatically draw their input files from HDFS. Using the regular File Input Format subclasses, Hadoop will automatically draw its input data sources from file paths within HDFS, and will distribute the work over the cluster in an intelligent fashion to exploit block locality where possible.

HDFS provides a decommissioning feature which ensures that this process is performed safely. To use it, follow the steps below:

***Step 1: Cluster configuration**:* If it is assumed that nodes may be retired in your cluster, then before it is started, an *excludes file* must be configured. Add a key named dfs.hosts.exclude to your conf/hadoop-site.xml file. The value associated with this key provides the full path to a file on the Name Node's local file system which contains a list of machines which are not permitted to connect to HDFS.
***Step 2: Determine hosts to decommission**:* Each machine to be decommissioned should be added to the file identified by dfs. hosts. exclude, one per line. This will prevent them from connecting to the Name Node.
***Step 3: Force configuration reload**:* Run the command bin/hadoop dfsadmin -refreshNodes. This will force the Name Node to reread its configuration, including the newly-updated excludes file. It will decommission the nodes over a period of time, allowing time for each node's blocks to be replicated onto machines which are scheduled to remain active.
***Step 4: Shutdown nodes**:* After the decommission process has completed, the decommissioned hardware can be safely shut down for maintenance, etc. The bin/hadoop dfsadmin -report command will describe which nodes are connected to the cluster.
Step 5: Edit excludes file again: Once the machines have been decommissioned, they can be removed from the excludes file. Running bin/hadoop dfsadmin –refresh Nodes again will read the excludes file back into the Name Node, allowing the Data Nodes to rejoin the cluster after maintenance has been completed, or additional capacity is needed in the cluster again, etc.

Using the Map Reduce Plug in For Eclipse:

An easier way to manipulate files in HDFS may be through the Eclipse plug in. In the DFS location viewer, right-click on any folder to see a list of actions available. You can create new subdirectories, upload individual files or whole subdirectories, or download files and directories to the local disk.

If /user/hadoop-user does not exist, create that first. Right-click on the top-level directory and select "Create New Directory". Type "user" and click OK. You will then need to *refresh* the current directory view by right-clicking and selecting "Refresh" from the pop-up menu. Repeat this process to create the "hadoop-user" directory under "user."
Now, prepare some local files to upload. Somewhere on your hard drive, create a directory named "input" and find some text files to copy there. In the DFS explorer, right-click the "hadoop-user" directory and click "Upload Directory to DFS." Select your new input folder and click OK. Eclipse will copy the files directly into HDFS, bypassing the local drive of the virtual machine. You may have to refresh the directory view to see your changes. You should now have a directory hierarchy containing the /user/hadoop-user/input directory, which has at least one text file in it.

Partitioning Data:

"Partitioning" is the process of determining which reducer instance will receive which intermediate keys and values. Each mapper must determine for all of its output (key, value) pairs which reducer will receive them. It is necessary that for any key, regardless of which mapper instance generated it, the destination partition is the same. If the key "cat" is generated in two separate (key, value) pairs, they must both be reduced together. It is also important for performance reasons that the mappers be able to partition data independently -- they should never need to exchange information with one another to determine the partition for a particular key.

Hadoop uses an interface called *Partitioner* to determine which partition a (key, value) pair will go to. A single partition refers to all (key, value) pairs which will be sent to a single reduce task. Hadoop MapReduce determines when the job starts how many partitions it will divide the data into. If twenty reduce tasks are to be run (controlled by the JobConf.setNumReduceTasks()) method), then twenty partitions must be filled.
The *Partitioner* defines one method which must be filled:

```
public interface Partitioner<K, V> extends JobConfigurable
{
  int getPartition(K key, V value, int numPartitions);
}
```

The getPartition() method receives a key and a value and the number of partitions to split the data across; a number in the range [0, numPartitions) must be returned by this method, indicating which partition to send the key and value to. For any two keys k1 and k2, k1.equals(k2) implies getPartition(k1, *, n) == getPartition(k2, *, n).

The default *Partitioner* implementation is called HashPartitioner. It uses the hashCode() method of the key objects modulo the number of partitions total to determine which partition to send a given (key, value) pair to.

CONCLUSIONS

In this paper, we argue that fundamental risks arise from sharing physical infrastructure between mutually distrustful users, even when their actions are isolated through machine virtualization as within a third-party cloud compute service.

However, having demonstrated this risk the obvious next question is "what should be done?". There are a number of approaches for mitigating this risk.

First, cloud providers may obfuscate both the internal structure of their services and the placement policy to complicate an adversary's attempts to place a VM on the same physical machine as its target. For example, providers might do well by inhibiting simple network-based co-residence checks. However, such approaches might only slow down, and not entirely stop, a dedicated attacker. Second, one may focus on the side-channel vulnerabilities themselves and employ blinding techniques to minimize the information that can be leaked. This solution requires being confident that all possible side-channels have been anticipated and blinded. Ultimately, we believe that the best solution is simply to expose the risk and placement decisions directly to users. A user might insist on using physical machines populated only with their own VMs and, in exchange, bear the opportunity costs of leaving some of these machines under-utilized. For an optimal assignment policy, this additional overhead should never need to exceed the cost of a single physical machine, so large users—consuming the cycles of many servers—would incur only minor penalties as a fraction of their total cost. Regardless, we believe such an option is the only foolproof solution to this problem and thus is likely to be demanded by customers with strong privacy requirements.

REFERENCES

[1]. O. Acıi¸cmez, ¸C. Kaya Ko ç, an d J.P. Seifert. On th e power of simple branch prediction analysis. IACR Cryptology ePrint Archive, report 2006/351, 2006.

[2]. O. Acıi¸cmez, ¸C. Kaya Ko¸c, and J.P. Seifert. Predicting secret keys via branch prediction. RSA Conference Cryptographers Track – CT-RSA '07, LNCS vol. 4377,pp. 225–242, Springer, 2007.

[3]. O.Acıi¸cmez.Yet another microarchitectural attack:exploiting I-cache. IACR Cryptology ePrint Archive, report 2007/164, 2007.

[4]. O. Acıi¸cmez, and J.P. Seifert. Cheap hardware parallelism implies cheap security. Workshop on Fault Diagnosis and Tolerance in Cryptography – FDTC '07, pp. 80–91, IEEE,2007.

[5]. Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/

[6]. Amazon Web Services. Auto-scaling Amazon EC2 with Amazon SQS. http://developer.amazonwebservices.com/ connect/entry.jspa?externalID=1464

[7]. Amazon Web Services. Creating HIPAA-Compliant Medical Data Applications with Amazon Web Services. White paper, http://awsmedia.s3.amazonaws.com/AWS_ HIPAA_Whitepaper_Final.pdf, April 2009.

[8]. Amazon Web Services. Customer Agreement.http://aws.amazon.com/agreemet/

[9]. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris,A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003.

[10]. D. Bernstein. Cache-timing attacks on AES. Preprint available at http://cr.yp.to/papers.html#cachetiming,2005.

[11]. DentiSoft. http://www.dentisoft.com/index.asp

[12]. D. Grunwald and S. Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. International Symposium on Microarchitecture – MICRO '02, pp. 409–418, IEEE, 2002.

[13]. D. Hyuk Woo and H.H. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. Workshop on Chip Multiprocessor Memory Systems and Interconnects, 2007.

[14]. W-H. Hu, Reducing timing channels with fuzzy time. IEEE Symposium on Security and Privacy, pp. 8–20, 1991.

[15]. W-H. Hu, Lattice scheduling and covert channels. IEEE Symposium on Security and Privacy, 1992

[16]. Thomas Ristenpart,Eran Tromer , Hovav Shacham, Stefan Savage" Hey, You, Get Off of My Cloud:Exploring Information Leakage in Third-Party Compute Clouds" *CCS'09,* November 9–13, 2009, Chicago, Illinois, USA. Copyright 2009 ACM 978-1-60558-352-5/09/11