Cloud security

CS642: Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu
Cloud computing

NIST: Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.
A simplified model of public cloud computing

Users run Virtual Machines (VMs) on cloud provider’s infrastructure

- **Multitenancy** (users share physical resources)
- Virtual Machine Manager (VMM) manages physical server resources for VMs
- To the VM should look like dedicated server

Owned/operated by cloud provider

Virtual Machine Manager (VMM)
Trust models in public cloud computing

Users must trust third-party provider to

- not spy on running VMs / data
- secure infrastructure from external attackers
- secure infrastructure from internal attackers
Trust models in public cloud computing

Users must trust third-party provider to
not spy on running VMs / data
secure infrastructure from external attackers
secure infrastructure from internal attackers

Threats due to sharing of physical infrastructure?

Your business competitor
Script kiddies
Criminals
...
A new threat model:

Attacker identifies one or more victims VMs in cloud

1) Achieve advantageous placement via launching of VM instances

2) Launch attacks using physical proximity

Exploit VMM vulnerability    DoS    Side-channel attack
1 or more targets in the cloud and we want to attack them from same physical host

Launch lots of instances (over time), with each attempting an attack

Can attackers do better?
Outline of a more damaging approach:

1) **Cloud cartography**
   - Map internal infrastructure of cloud
   - Map used to locate targets in cloud

2) **Checking for co-residence**
   - Check that VM is on same server as target
     - Network-based co-residence checks
     - Efficacy confirmed by covert channels

3) **Achieving co-residence**
   - Brute forcing placement
   - Instance flooding after target launches

4) **Location-based attacks**
   - Side-channels, DoS, escape-from-VM

Placement vulnerability: attackers can knowingly achieve co-residence with target
Case study with Amazon’s EC2

1) given no insider information
2) restricted by (the spirit of) Amazon’s acceptable use policy (AUP)
   (using only Amazon’s customer APIs and very restricted network probing)

We were able to:

- Pick target(s)
- Choose launch parameters for malicious VMs
- Each VM checks for co-residence
- Cross-VM side channel attacks to spy on victim’s computational load
- Frequently achieve advantageous placement

“Cloud cartography”
Some info about EC2 service (at time of study)

- Linux-based VMs available
- Uses Xen-based VM manager

User account

- 3 “availability zones” (Zone 1, Zone 2, Zone 3)
- 5 instance types (various combinations of virtualized resources)

<table>
<thead>
<tr>
<th>Type</th>
<th>gigs of RAM</th>
<th>EC2 Compute Units (ECU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.small (default)</td>
<td>1.7</td>
<td>1</td>
</tr>
<tr>
<td>m1.large</td>
<td>7.5</td>
<td>4</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>c1.medium</td>
<td>1.7</td>
<td>5</td>
</tr>
<tr>
<td>c1.xlarge</td>
<td>7</td>
<td>20</td>
</tr>
</tbody>
</table>

1 ECU = 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor

Limit of 20 instances at a time per account.
Essentially unlimited accounts with credit card.
Our experiments indicate that internal IPs are **statically assigned** to physical servers.

Co-residence checking via Dom0: only hop on traceroute to co-resident target.
Cloud cartography

Pick target(s)

Choose launch parameters for malicious VMs

3 “availability zones”
(Zone 1, Zone 2, Zone 3)

5 instance types
(m1.small, c1.medium, m1.large, m1.xlarge, c1.xlarge)

User account
Cloud cartography

Pick target(s) → Choose launch parameters for malicious VMs

- 3 "availability zones" (Zone 1, Zone 2, Zone 3)
- 5 instance types (m1.small, c1.medium, m1.large, m1.xlarge, c1.xlarge)

User account

- Launch parameters

Graph showing account A and B with internal IP addresses.
Associate to each /24 an estimate of Availability zone and Instance Type

External IP → DNS → Internal IP → Availability zone/24 → Instance Type

Mapping 6,577 public HTTP servers running on EC2 (Fall 2008)
Achieving co-residence

“Brute-forcing” co-residence

Attacker launches many VMs over a relatively long period of time in target’s zone and of target type

Experiment:

1,686 public HTTP servers as stand-in “targets” running m1.small and in Zone 3 (via our map)

1,785 “attacker” instances launched over 18 days

Each checked co-residence against all targets using Dom0 IP

Results:

78 unique Dom0 IPs

141 / 1,686 (8.4%) had attacker co-resident

Sequential placement locality lowers success

Lower bound on true success rate
Achieving co-residence

Instance flooding near target launch abuses parallel placement locality

Launch many instances in parallel near time of target launch
Achieving co-residence

Instance flooding near target launch abuses parallel placement locality

Launch many instances in parallel near time of target launch

Experiment:

Repeat for 10 trials:

1) Launch 1 target VM (Account A)

2) 5 minutes later, launch 20 “attack” VMs (alternate using Account B or C)

3) Determine if any co-resident with target using Dom0 IP

4 / 10 trials succeeded
Achieving co-residence

Instance flooding near target launch abuses parallel placement locality

How long is parallel placement locality good for?

Experiment:

40 “target” VMs (across two accounts)
20 “attack” VMs launched hourly
Achieving co-residence

Instance flooding near target launch abuses parallel placement locality

What about commercial accounts?

Free demos of Internet appliances powered by EC2

2 attempts

1st – coresident w/ 40 VMs

2nd – 2 VMs coresident w/ 40 launched

Several attempts

1st – coresident w/ 40 VMs

Subsequent attempts failed
Checking for co-residence

How do we know Dom0 IP is valid co-residence check?

Use simple covert channel as ground truth:

VM1
Sender transmits ‘1’ by frantically reading random locations
Sender transmits ‘0’ by doing nothing

Hard disk

VM2
Receiver times reading of a fixed location

Covert channels require control of both VMs: we use only to verify network-based co-residence check
Checking for co-residence

Experiment

Repeat 3 times:
1) 20 m1.small Account A
2) 20 m1.small Account B
3) All pairs w/ matching Dom0 → send 5-bit message across HD covert channel

Ended up with 31 pairs of co-resident instances as indicated by Dom0 IPs

Result: a correctly-received message sent for every pair of instances

During experiment also performed pings to:
* 2 control instances in each zone
* co-resident VM

<table>
<thead>
<tr>
<th>Zone</th>
<th>Control</th>
<th>Median RTT (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zone 1 Control 1</td>
<td></td>
<td>1.164</td>
</tr>
<tr>
<td>Zone 1 Control 2</td>
<td></td>
<td>1.027</td>
</tr>
<tr>
<td>Zone 2 Control 1</td>
<td></td>
<td>1.113</td>
</tr>
<tr>
<td>Zone 2 Control 2</td>
<td></td>
<td>1.187</td>
</tr>
<tr>
<td>Zone 3 Control 1</td>
<td></td>
<td>0.550</td>
</tr>
<tr>
<td>Zone 3 Control 2</td>
<td></td>
<td>0.436</td>
</tr>
<tr>
<td>Co-resident VM</td>
<td></td>
<td>0.242</td>
</tr>
</tbody>
</table>

RTT times also indicate co-residence
So far we were able to:

- Pick target(s)
- Choose launch parameters for malicious VMs
- Each VM checks for co-residence
- Frequently achieve advantageous placement

This shouldn’t matter if VMM provides good isolation!
Violating isolation

- Hard drive covert channel used to validate Dom0 co-residence check already violated isolation
- Degradation-of-Service attacks
  - Guests might maliciously contend for resources
  - Xen scheduler vulnerability
- Escape-from-VM vulnerabilities
- Side-channel attacks
Cross-VM side channels using CPU cache contention

1) Read in a large array (fill CPU cache with attacker data)
2) Busy loop (allow victim to run)
3) Measure time to read large array (the load measurement)
Cache-based cross-VM load measurement on EC2

Repeated HTTP get requests

Running Apache server

Performs cache load measurements

3 pairs of instances, 2 pairs co-resident and 1 not
100 cache load measurements during HTTP gets (1024 byte page) and with no HTTP gets

Instances co-resident

Instances co-resident

Instances NOT co-resident
Cache-based load measurement of traffic rates on EC2

3 trials with 1 pair of co-resident instances:
1000 cache load measurements during
0, 50, 100, or 200 HTTP gets (3 Mbyte page) per minute for ~1.5 mins
Performance Loss from Contention

- **Performance Degradation (%)**
  - CPU: 0%
  - Net: 200%
  - Disk: 500%
  - Cache: 600%

**Local Xen Testbed**
- **Machine**: Intel Xeon E5430, 2.66 Ghz
- **CPU**: 2 packages each with 2 cores
- **LLC Size**: 6MB per package
Resource Freeing Attacks (RFAs)

• Goal:
  – Reduce performance loss from contention

• Intuition:
  – Performance suffers from contention for a target resource
  – Introducing new workload on a victim can shift their usage away from target
Ingredients for a successful RFA

- **Shift** resource away from the **target resource**

- **Create a bottleneck** on an essential resource used by the **victim without affecting the beneficiary**

- **Freeing up the target resource**

![Diagram](https://via.placeholder.com/150)
Example RFA: Network bandwidth

• *Victim* runs Apache webserver hosting static and dynamic content (CGI pages)
• *Beneficiary* also runs Apache webserver hosting static content
• Contending for network bandwidth
Example RFA: Network bandwidth

• *Helper* sends CPU-intensive CGI requests
• Creates CPU *bottleneck* on victim
• *Frees up* bandwidth
  – Increasing beneficiary’s share of bandwidth from 50 to 85%
Example RFA: Cache contention

- *Victim* runs Apache webserver hosting static and dynamic content (CGI pages)
- *Beneficiary* runs cache-sensitive workload
- Contending for cache
Example RFA: Cache contention

Figure 6: Cumulative runtime distribution of (top) the web server domain (with load 2,000 rps) and (bottom) the LLCProbe domain under both no RFA and with RFA 320 in pinned core case.

Figure 7: Offered vs. observed load on web server with varying RFA intensities when all the VMs float across all cores.

Figure 8: Normalized performance (baseline runtime over runtime) for SPEC workloads on our local testbed for various RFA intensities. All values are at a web server request rate of 3000 rps.

When we pin LLCProbe and the web server to different packages (no shared cache) but let Dom0 float, LLCProbe still experiences interference. At a load of 2000 rps on the web server, LLCProbe suffered a 78% degradation in performance just due to Dom0's interference. The RFA we explore can only alleviate contention from Dom0 by forcing a drop in the web server's foreground traffic rate (by exhausting its VM's CPU allocation as shown in Figure 7).

Finally, we analyze a spectrum of SPEC benchmarks. Each SPEC benchmark is run three times with an idle webserver, an active web server, and an active web server with various RFA where all the VMs (including Dom0) float across all cores. Figure 8 depicts the normalized performance of seven benchmarks under no RFA and intensities of 320 and 640. That is, the reported fractions are computed as $t'/t$ where $t$ is the average runtime (request latency is computed and used for SPECjbb) and $t'$ is the average baseline performance when no traffic is sent to the victim. All benchmarks benefit from the RFA, with the general trend that cache-sensitive benchmarks (as indicated by a larger drop in performance relative to the baseline) achieve more gains from the RFA. For example, the 640 RFA increases normalized performance of SPECjbb from 0.91 to 0.97, a 6 percentage point improvement in performance and a 66.5% reduction in harm due to contention. The smallest improvement occurs with hmmer, which shows only a 1.1 percentage point improvement because it only suffers a performance loss of 1.6% without the RFA. Across all the benchmarks, the 640 RFA achieves an average performance improvement of 3.4 percentage point and recovers 55.5% of lost performance. These improvements come largely from the ability of the RFA to reduce the request rate of the victim web server.

5.2 Evaluation on EC2

The above experiments clearly indicate that RFAs can provide substantial gains in a controlled setting. To verify that the attacks will also work in a noisier, more realistic setting, we turn to Amazon's...
Experiments on EC2

Arranged for co-resident placement of m1.small instances from accounts under our control

Pair of co-resident instances used as stand-ins for victim and beneficiary

<table>
<thead>
<tr>
<th>Machine</th>
<th>#</th>
<th>Machine</th>
<th>#</th>
<th>Machine</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>E5507-1</td>
<td>4</td>
<td>E5507-4</td>
<td>3</td>
<td>E5507-7</td>
<td>2</td>
</tr>
<tr>
<td>E5507-2</td>
<td>2</td>
<td>E5507-5</td>
<td>2</td>
<td>E5507-8</td>
<td>3</td>
</tr>
<tr>
<td>E5507-3</td>
<td>2</td>
<td>E5507-6</td>
<td>2</td>
<td>E5507-9</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 9: Summary of EC2 machines and number of co-resident m1.small instances running under our accounts.
Demonstration on Amazon EC2

- **MCF**: cache bound
- **Apache**: interrupts/data pollute cache

![Chart showing performance comparison between MCF and Apache with and without RFA.]
What can cloud providers do?

1) Cloud cartography
   - Random Internal IP assignment

2) Checking for co-residence
   - Isolate each user’s view of internal address space
   - Hide Dom0 from traceroutes
   - Allow users to opt out of multitenancy

3) Achieving co-residence
   - Improved performance isolation

4) Side-channel information leakage
   - Resource-freeing attacks
   - Hardware or software countermeasures to stop leakage [Ber05, OST05, Page02, Page03, Page05, Per05]
   - Improved performance isolation

Amazon provides dedicated instances now. They cost a lot more.

Amazon doesn’t report Dom0 in traceroutes anymore.
Untrusted provider

• A lot of work aimed at untrustworthy provider
• Attestation of cloud:
  – Homealone: use L2 cache side-channels to detect presence of foreign VM
  – RAFT: Remote Assessment of Fault Tolerance to infer if data stored in redundant fashion
  – Keep data private: searchable or fully-homomorphic encryption
More on cache-based physical channels

Prime+Trigger+Probe combined with differential encoding technique gives high bandwidth cross-VM covert channel on EC2

See [Xu et al., “An Exploration of L2 Cache Covert Channels in Virtualized Environments”, CCSW 2011]

Keystroke timing in experimental testbed similar to EC2 m1.small instances
More on cache-based physical channels

Prime+Trigger+Probe combined with differential encoding technique gives high bandwidth cross-VM covert channel on EC2
See [Xu et al., “An Exploration of L2 Cache Covert Channels in Virtualized Environments”, CCSW 2011]

Keystroke timing in experimental testbed similar to EC2 m1.small instances
More on cache-based physical channels

Prime+Trigger+Probe combined with differential encoding technique gives high bandwidth cross-VM covert channel on EC2

See [Xu et al., “An Exploration of L2 Cache Covert Channels in Virtualized Environments”, CCSW 2011]

Keystroke timing in experimental testbed similar to EC2 m1.small instances
More on cache-based physical channels

Prime+Trigger+Probe combined with differential encoding technique gives high bandwidth cross-VM covert channel on EC2

See [Xu et al., “An Exploration of L2 Cache Covert Channels in Virtualized Environments”, CCSW 2011]

Keystroke timing in experimental testbed similar to EC2 m1.small instances

We show that cache-load measurements enable cross-VM keystroke detection

Keystroke timing of this form might be sufficient for the password recovery attacks of [Song, Wagner, Tian 01]