Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds

Thomas Ristenpart* Eran Tromer† Hovav Shacham* Stefan Savage*

*Dept. of Computer Science and Engineering
University of California, San Diego, USA
{tristenp,hovav,savage}@cs.ucsd.edu

†Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, USA
tromer@csail.mit.edu

ABSTRACT

Third-party cloud computing represents the promise of outsourcing as applied to computation. Services, such as Microsoft's Azure and Amazon's EC2, allow users to instantiate virtual machines (VMs) on demand and thus purchase precisely the capacity they require when they require it. In turn, the use of virtualization allows third-party cloud providers to maximize the utilization of their sunk capital costs by multiplexing many customer VMs across a shared physical infrastructure. However, in this paper, we show that this approach can also introduce new vulnerabilities. Using the Amazon EC2 service as a case study, we show that it is possible to map the internal cloud infrastructure, identify where a particular target VM is likely to reside, and then instantiate new VMs until one is placed co-resident with the target. We explore how such placement can then be used to mount cross-VM side-channel attacks to extract information from a target VM on the same machine.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: UNAUTHORIZED ACCESS

General Terms

Security, Measurement, Experimentation

Keywords

Cloud computing, Virtual machine security, Side channels

1. INTRODUCTION

It has become increasingly popular to talk of "cloud computing" as the next infrastructure for hosting data and deploying software and services. In addition to the plethora of technical approaches associated with the term, cloud computing is also used to refer to a new business model in which core computing and software capabilities are outsourced *on demand* to shared third-party infrastructure. While this model, exemplified by Amazon's Elastic Compute Cloud (EC2) [5], Microsoft's Azure Service Platform [20], and Rackspace's Mosso [27] provides a number of advantages — including economies of scale, dynamic provisioning, and low capital expenditures — it also introduces a range of new risks.

Some of these risks are self-evident and relate to the new trust relationship between customer and cloud provider. For example, customers must trust their cloud providers to respect the privacy of their data and the integrity of their computations. However, cloud infrastructures can also introduce non-obvious threats from *other customers* due to the subtleties of how physical resources can be transparently shared between *virtual machines* (VMs).

In particular, to maximize efficiency multiple VMs may be simultaneously assigned to execute on the same physical server. Moreover, many cloud providers allow "multi-tenancy" — multiplexing the virtual machines of disjoint customers upon the same physical hardware. Thus it is *conceivable* that a customer's VM could be assigned to the same physical server as their adversary. This in turn, engenders a new threat — that the adversary might penetrate the isolation between VMs (e.g., via a vulnerability that allows an "escape" to the hypervisor or via side-channels *between* VMs) and violate customer confidentiality. This paper explores the practicality of mounting such cross-VM attacks in existing third-party compute clouds.

The attacks we consider require two main steps: placement and extraction. Placement refers to the adversary arranging to place their malicious VM on the same physical machine as that of a target customer. Using Amazon's EC2 as a case study, we demonstrate that careful empirical "mapping" can reveal how to launch VMs in a way that maximizes the likelihood of an advantageous placement. We find that in some natural attack scenarios, just a few dollars invested in launching VMs can produce a 40% chance of placing a malicious VM on the same physical server as a target customer. Using the same platform we also demonstrate the existence of simple, low-overhead, "co-residence" checks to determine when such an advantageous placement has taken place. While we focus on EC2, we believe that variants of our techniques are likely to generalize to other services, such as Microsoft's Azure [20] or Rackspace's Mosso [27], as we only utilize standard customer capabilities and do not require that cloud providers disclose details of their infrastructure or assignment policies.

Having managed to place a VM co-resident with the target, the next step is to extract confidential information via a cross-VM attack. While there are a number of avenues for such an attack, in this paper we focus on side-channels: cross-VM information leakage due to the sharing of physical resources (e.g., the CPU's data caches). In the multi-process environment, such attacks have been shown to enable extraction of RSA [26] and AES [22] secret keys. However, we are unaware of published extensions of these attacks to the virtual machine environment; indeed, there are significant practical challenges in doing so.

We show preliminary results on cross-VM side channel attacks, including a range of building blocks (e.g., cache load measurements in EC2) and coarse-grained attacks such as measuring activity burst timing (e.g., for cross-VM keystroke monitoring). These point to the practicality of side-channel attacks in cloud-computing environments.

Overall, our results indicate that there exist tangible dangers when deploying sensitive tasks to third-party compute clouds. In the remainder of this paper, we explain these findings in more detail and then discuss means to mitigate the problem. We argue that the best solution is for cloud providers to expose this risk explicitly and give some placement control directly to customers.

2. THREAT MODEL

As more and more applications become exported to third-party compute clouds, it becomes increasingly important to quantify any threats to confidentiality that exist in this setting. For example, cloud computing services are already used for e-commerce applications, medical record services [7, 11], and back-office business applications [29], all of which require strong confidentiality guarantees. An obvious threat to these consumers of cloud computing is malicious behavior by the cloud provider, who is certainly in a position to violate customer confidentiality or integrity. However, this is a known risk with obvious analogs in virtually any industry practicing outsourcing. In this work, we consider the provider and its infrastructure to be trusted. This also means we do not consider attacks that rely upon subverting a cloud's administrative functions, via insider abuse or vulnerabilities in the cloud management systems (e.g., virtual machine monitors).

In our threat model, adversaries are non-provider-affiliated malicious parties. Victims are users running confidentiality-requiring services in the cloud. A traditional threat in such a setting is direct compromise, where an attacker attempts remote exploitation of vulnerabilities in the software running on the system. Of course, this threat exists for cloud applications as well. These kinds of attacks (while important) are a known threat and the risks they present are understood.

We instead focus on where third-party cloud computing gives attackers *novel* abilities; implicitly expanding the *attack surface* of the victim. We assume that, like any customer, a malicious party can run and control many instances in the cloud, simply by contracting for them. Further, since the economies offered by third-party compute clouds derive from multiplexing physical infrastructure, we assume (and later validate) that an attacker's instances might even run on the same physical hardware as potential victims. From this vantage, an attacker might manipulate shared physical resources (e.g., CPU caches, branch target buffers, network queues, etc.) to learn otherwise confidential information.

In this setting, we consider two kinds of attackers: those who cast a wide net and are interested in being able to attack *some* known hosted service and those focused on attacking a particular victim service. The latter's task is more expensive and time-consuming than the former's, but both rely on the same fundamental attack.

In this work, we initiate a rigorous research program aimed at exploring the risk of such attacks, using a concrete cloud service provider (Amazon EC2) as a case study. We address these concrete questions in subsequent sections:

- Can one determine where in the cloud infrastructure an instance is located? (Section 5)
- Can one easily determine if two instances are co-resident on the same physical machine? (Section 6)
- Can an adversary launch instances that will be co-resident with other user's instances? (Section 7)
- Can an adversary exploit cross-VM information leakage once co-resident? (Section 8)

Throughout we offer discussions of defenses a cloud provider might try in order to prevent the success of the various attack steps.

3. 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, OpenSolaris 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 traceroutes.

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

[1]We will limit our subsequent discussion to the Linux kernel. The same issues should apply for other guest operating systems.

(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.compute-1.amazonaws .com, internal IP 10.252.146.52, and internal name domU-12-31-38-00-8D-C6.compute-1.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.

Note that we focus on the United States region — in the rest of the paper EC2 implicitly means this region of EC2.

4. NETWORK PROBING

In the next several sections, we describe an empirical measurement study focused on understanding VM placement in the EC2 system and achieving co-resident placement for an adversary. To do this, we make use of network probing 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, hping, and wget 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 hping to perform *TCP SYN* traceroutes, which iteratively sends TCP SYN packets with increasing time-to-lives (TTLs) until no ACK is received. Both TCP connect probes and SYN traceroutes 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 discuss the legal, ethical and contractual issues around such probing in Appendix A).

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.

5. CLOUD CARTOGRAPHY

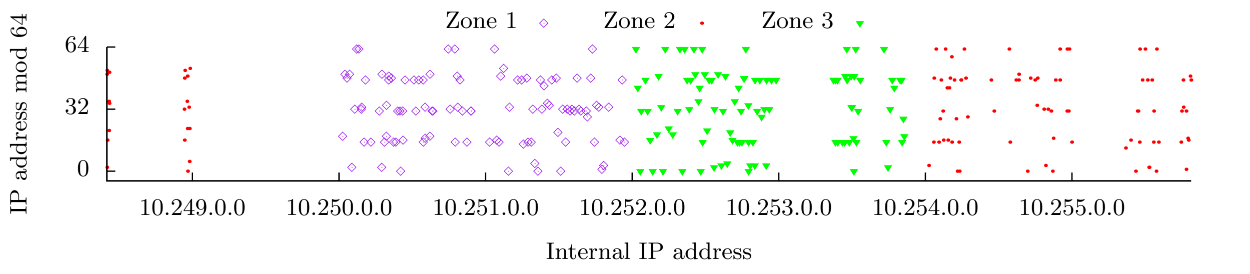
In this section we 'map' the EC2 service to understand where potential targets are located in the cloud and the instance creation parameters needed to attempt establishing co-residence of an adversarial instance. This will speed up significantly adversarial strategies for placing a malicious VM on the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

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

Surveying public servers on EC2. Utilizing WHOIS queries, we identified four distinct IP address prefixes, a /16, /17, /18, and /19, as being associated with EC2. The last three contained public IPs observed as assigned to EC2 instances. We had not yet observed EC2 instances with public IPs in the /16, and therefore did not include it in our survey. For the remaining IP addresses (57 344 IP addresses), we performed a TCP connect probe on port 80. This resulted in 11 315 responsive IPs. Of these 9 558 responded (with some HTTP response) to a follow-up wget on port 80. We also performed a TCP port 443 scan of all 57 344 IP addresses, which resulted in 8 375 responsive IPs. Via an appropriate DNS lookup from *within* EC2, we translated each public IP address that responded to either the port 80 or port 443 scan into an internal EC2 address. This resulted in a list of 14 054 unique internal IPs. One of the goals of this section is to enable identification of the instance type and availability zone of one or more of these potential targets.



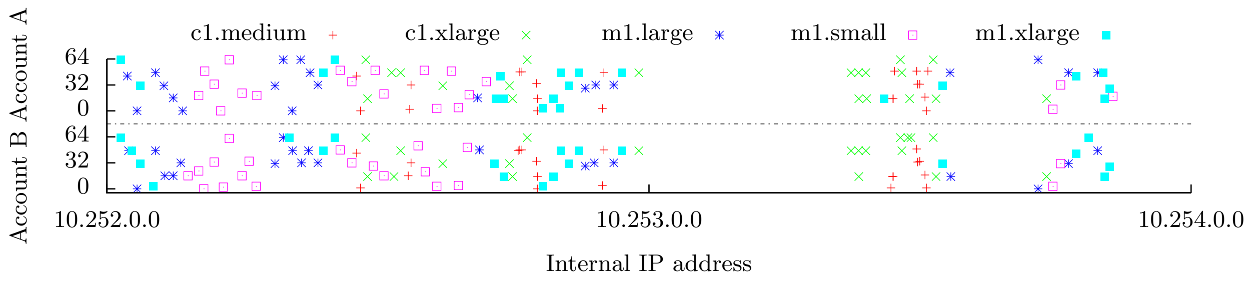


Figure 1: (Top) A plot of the internal IP addresses assigned to instances launched during the initial mapping experiment using Account A. (Bottom) A plot of the internal IP addresses of instances launched in Zone 3 by Account A and, 39 hours later, by Account B. Fifty-five of the Account B IPs were repeats of those assigned to instances for Account A.

Instance placement parameters. Recall that there are three availability zones and five instance types in the present EC2 system. While these parameters could be assigned independently from the underlying infrastructure, in practice this is not so. In particular, the Amazon EC2 internal IP address space is cleanly partitioned between availability zones (likely to make it easy to manage separate network connectivity for these zones) and instance types within these zones also show considerable regularity. Moreover, different accounts exhibit similar placement.

To establish these facts, we iteratively launched 20 instances for each of the 15 availability zone/instance type pairs. We used a single account, call it "Account A". The top graph in Figure 1 depicts a plot of the internal IP address assigned to each of the 300 instances, partitioned according to availability zone. It can be readily seen that the samples from each zone are assigned IP addresses from disjoint portions of the observed internal address space. For example, samples from Zone 3 were assigned addresses within 10.252.0.0/16 and from discrete prefixes within 10.253.0.0/16. If we make the assumption that internal IP addresses are statically assigned to physical machines (doing otherwise would make IP routing far more difficult to implement), this data supports the assessment that availability zones use separate physical infrastructure. Indeed, none of the data gathered in the rest of the paper's described experiments have cast doubt on this conclusion.

While it is perhaps not surprising that availability zones enjoy disjoint IP assignment, what about instance type and accounts? We launched 100 instances (20 of each type, 39 hours after terminating the Account A instances) in Zone 3 from a second account, "Account B". The bottom graph in Figure 1 plots the Zone 3 instances from Account A and Account B, here using distinct labels for instance type. Of the 100 Account A Zone 3 instances, 92 had unique /24 prefixes, while eight /24 prefixes each had two instances, though of the same type. Of the 100 Account B instances, 88 had unique /24 prefixes, while six of the /24 prefixes had two instances each. A single /24 had both an m1.large and an m1.xlarge instance. No IP addresses were ever observed being assigned to more than one instance type. Of the 100 Acount B IP's, 55 were repeats of IP addresses assigned to instances for Acount A.

A fuller map of EC2. We would like to infer the instance type and availability zone of any public EC2 instance, but our sampling data is relatively sparse. We could sample more (and did), but to take full advantage of the sampling data at hand we should take advantage of the significant regularity of the EC2 addressing regime. For example, the above data suggests that /24 prefixes rarely have IPs assigned to distinct instance types. We utilized data from 4 499 instances launched under several accounts under our control; these instances were also used in many of the experiments described in the rest of the paper. These included 977 unique internal IPs and 611 unique Dom0 IPs associated with these instances.

Using manual inspection of the resultant data, we derived a set of heuristics to label /24 prefixes with both availability zone and instance type:

- All IPs from a /16 are from the same availability zone.
- A /24 inherits any included sampled instance type. If there are multiple instances with distinct types, then we label the /24 with each distinct type (i.e., it is ambiguous).
- 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.
- 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. (For example, 10.250.8.0/24 contained Dom0 IPs associated with m1.small instances in prefixes 10.254.9.0/24 and 10.254.10.0/24.) 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. Figure 7 in the appendix graphs the result of mapping these servers.

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 traceroutes (both discuss more in the next section) to help "triangulate" on a victim.

6. 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

(1) matching Dom0 IP address,

(2) small packet round-trip times, or

(3) 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 traceroute 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. (Note that m1.small instances are reported by CPUID as having two CPUs each with two cores and these EC2 instance types are limited to 50% core usage, implying that one such machine could handle eight instances.)

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.

We performed three independent trials. These generated, in total, 31 pairs of instances for which the Dom0 IPs were equal. The internal IP addresses of each pair were within 7 of each other. Of the 31 (potentially) co-resident instance pairs, 12 were 'repeats' (a pair from a later round had the same Dom0 as a pair from an earlier round).

The 31 pairs give 62 ordered pairs. The hard-drive covert channel successfully sent a 5-bit message for 60 of these pairs. The last two failed due to a single bit error each, and we point out that these two failures were not for the same pair of instances (i.e. sending a message in the reverse direction succeeded). The results of the RTT probes are shown in Figure 2. The median RTT for co-resident instances was significantly smaller than those to any of the controls. The RTTs to the controls in the same availability zone as the probe (Zone 3) and victim instances were also noticeably smaller than those to other zones.

Discussion. From this experiment we conclude an effective false positive rate of zero for the Dom0 IP co-residence check. In the rest of the paper we will therefore utilize the

	Count	Median RTT (ms)
Co-resident instance	62	0.242
Zone 1 Control A	62	1.164
Zone 1 Control B	62	1.027
Zone 2 Control A	61	1.113
Zone 2 Control B	62	1.187
Zone 3 Control A	62	0.550
Zone 3 Control B	62	0.436

Figure 2: Median round trip times in seconds for probes sent during the 62 co-residence checks. (A probe to Zone 2 Control A timed out.)

following when checking for co-residence of an instance with a target instance we do not control. First one compares the internal IP addresses of the two instances, to see if they are numerically close. (For m1.small instances close is within seven.) If this is the case, the instance performs a TCP SYN traceroute to an open port on the target, and sees if there is only a single hop, that being the Dom0 IP. (This instantiates the Dom0 IP equivalence check.) Note that this check requires sending (at most) two TCP SYN packets and is therefore very "quiet".

Obfuscating co-residence. A cloud provider could likely render the network-based co-residence checks we use moot. For example, a provider might have Dom0 not respond in traceroutes, might randomly assign internal IP addresses at the time of instance launch, and/or might use virtual LANs to isolate accounts. If such precautions are taken, attackers might need to turn to co-residence checks that do not rely on network measurements. In Section 8.1 we show experimentally that side-channels can be utilized to establish co-residence in a way completely agnostic to network configuration. Even so, inhibiting network-based co-residence checks would impede attackers to some degree, and so determining the most efficient means of obfuscating internal cloud infrastructure from adversaries is a good potential avenue for defense.

7. 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). We offer two adversarial strategies that make crucial use of the map developed in Section 5 and the cheap co-residence checks we introduced in Section 6. 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. Our experiments show that this feature (combined with the ability to map EC2 and perform co-residence checks) represents an exploitable placement vulnerability: measurements show that the strategy achieves co-residence with a specific (m1.small) instance *almost half the time.* As we discuss below, an attacker can infer when a victim instance is launched or might even trigger launching of victims, making this attack scenario practical.

Towards understanding placement. Before we describe these strategies, we first collect several observations we initially made regarding Amazon's (unknown) placement algorithms. Subsequent interactions with EC2 only reinforced these observations.

A single account was never seen to have two instances simultaneously running on the same physical machine, so running n instances in parallel under a single account results in placement on n separate machines. No more than eight m1.small instances were ever observed to be simultaneously co-resident. (This lends more evidence to support our earlier estimate that each physical machine supports a maximum of eight m1.small instances.) While a machine is full (assigned its maximum number of instances) an attacker has no chance of being assigned to it.

We observed strong *placement locality.* Sequential placement locality exists when two instances run sequentially (the first terminated before launching the second) are often assigned to the same machine. Parallel placement locality exists when two instances run (from distinct accounts) at roughly the same time are often assigned to the same machine. In our experience, launched instances exhibited both strong sequential and strong parallel locality.

Our experiences suggest a correlation between instance density, the number of instances assigned to a machine, and a machine's affinity for having a new instance assigned to it. In Appendix B we discuss an experiment that revealed a bias in placement towards machines with fewer instances already assigned. This would make sense from an operational viewpoint under the hypothesis that Amazon balances load across running machines.

We concentrate in the following on the m1.small instance type. However, we have also achieved active co-residence between two m1.large instances under our control, and have observed m1.large and c1.medium instances with co-resident commercial instances. Based on the reported (using CPUID) system configurations of the m1.xlarge and c1.xlarge instance types, we assume that these instances have machines to themselves, and indeed we never observed co-residence of multiple such instances.

7.1 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 from Section 5. 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. (This restriction is arbitrary. It only makes the task harder since it cut down on the number of potential victims.) This left 6 577 servers. We targeted Zone 3 and m1.small instances and used our cloud map to infer which of the servers match this zone/type. This left 1 686 servers. (The choice of zone was arbitrary. The choice of instance type was due to the fact that m1.small instances enjoy the greatest use.) We collected data from numerous m1.small probe instances we launched in Zone 3. (These instances were also used in the course of our other experiments.) The probes were instrumented to perform the cheap co-residence check procedure described at the end of Section 6 for all of the targets. For any co-resident target, the probe performed a wget on port 80 (to ensure the target was still serving web pages). The wget scan of the EC2 servers was conducted on October 21, 2008, and the probes we analyzed were launched over the course of 18 days, starting on October 23, 2008. The time between individual probe launches varied, and most were launched in sets of 20.

We analyzed 1 785 such probe instances. These probes had 78 unique Dom0 IPs. (Thus, they landed on 78 different physical machines.) Of the 1 686 target victims, the probes achieved co-residency with 141 victim servers. Thus the "attack" achieved 8.4% coverage of the target set.

Discussion. We point out that the reported numbers are conservative in several ways, representing only a lower bound on the true success rate. We only report co-residence if the server is still serving web pages, even if the server was actually still running. 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 (i.e. by being taken down and then relaunched) were not detected, even when we in fact achieved co-residence with them. We could have corrected some sources of false negatives by actively performing more internal port scans, but we limited ourselves to probing ports we knew to already be serving public web pages (as per the discussion in Section 4).

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. (For example, in the data above, the first 510 launched probes had already achieved co-residence with 90% of the eventual 141 victims covered.) This suggests that fuller coverage of the target set could require many more probes.

7.2 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. (Examples include scalr [30] and RightGrid [28]. See also [6].) We believe clever adversaries can find many other practical realizations of this attack scenario.

The rest of this section is devoted to quantifying several aspects of this attack strategy. We assess typical success rates, whether the availability zone, attacking account, or the time of day has some bearing on success, and the effect of increased time lag between victim and attacker launches. Looking ahead, 40% of the time the attacker (launching just 20 probes) achieves co-residence against a specific target instance; zone, account, and time of day do not meaningfully impact success; and even if the adversary launches its instances two days after the victims' launch it still enjoys the same rate of success.

In the following we will often use instances run by one of our own accounts as proxies for victims. However we will also discuss achieving co-residence with recently launched commercial servers. Unless otherwise noted, we use m1.small instances. Co-residence checks were performed via comparison of Dom0 IPs.

The effects of zone, account, and time of day. We start with finding a base-line for success rates when running probe instances soon (on the order of 5 minutes) after victims. The first experiment worked as follows, and was repeated for each availability zone. A victim account launched either 1, 10, or 20 instances simultaneously. No sooner than five minutes later, a separate attacker account requested launch of 20 instances simultaneously. The number of collisions (attacker instances co-resident with a victim instance) are reported in the left table of Figure 3. As can be seen, collisions are quickly found for large percentages of victim instances. The availability zone used does not meaningfully affect co-residence rates.

We now focus on a single availability zone, Zone 1, for the next experiment. We repeated, at three different time periods over the course of a day, the following steps: A single victim instance was launched. No more than 5 minutes later 20 probe instances were launched by another account, and co-residence checks were performed. This process was repeated 10 times (with at least 5 minutes in between conclusion of one iteration and beginning of the next). Each iteration used a fresh victim; odd iterations used one account and even iterations used another. The right table in Figure 3 displays the results. The results show a likelihood of achieving co-residence as 40% — slightly less than half the

	# victims v	# probes p	coverage
Zone 1	1	20	1/1
	10	20	5/10
	20	20	7/20
Zone 2	1	20	0/1
	10	18	3/10
	20	19	8/20
Zone 3	1	20	1/1
	10	20	2/10
	20	20	8/20

	Account		
Trial	A	B	Total
Midday (11:13 – 14:22 PST)	2 / 5	2 / 5	4/10
Afternoon (14:12 – 17:19 PST)	1 / 5	3 / 5	4/10
Night (23:18 – 2:11 PST)	2 / 5	2 / 5	4/10

Figure 3: (Left) Results of launching p probes 5 minutes after the launch of v victims. The rightmost column specifies success coverage: the number of victims for which a probe instance was co-resident over the total number of victims. (Right) The number of victims for which a probe achieved co-residence for three separate runs of 10 repetitions of launching 1 victim instance and, 5 minutes later, 20 probe instances. Odd-numbered repetitions used Account A; even-numbered repetitions used Account B.

time a recently launched victim is quickly and easily "found" in the cloud. Moreover, neither the account used for the victims nor the portion of the day during which the experiment was conducted significantly affected the rate of success.

The effect of increased time lag. Here we show that the window of opportunity an attacker has for launching instances is quite large. We performed the following experiment. Forty victim instances (across two accounts) were initially launched in Zone 3 and continued running throughout the experiment. These were placed on 36 unique machines (8 victims were co-resident with another victim). Every hour a set of 20 attack instances (from a third account) were launched in the same zone and co-residence checks were performed. These instances were terminated immediately after completion of the checks. Figure 4 contains a graph showing the success rate of each attack round, which stays essentially the same over the course of the whole experiment. (No probes were reported upon for the hours 34–43 due to our scripts not gracefully handling some kinds of EC2-caused launch failures, but nevertheless reveals useful information: the obvious trends were maintained regardless of continuous probing or not.) Ultimately, co-residence with 24 of the 36 machines running victim instances was established. Additionally, probes were placed on all four machines which had two victim instances, thus giving three-way collisions.

The right graph in Figure 4 shows the cumulative number of unique Dom0 IP addresses seen by the probes over the course of the experiment. This shows that the growth in the number of machines probes were placed on levels off rapidly — quantitative evidence of sequential placement locality.

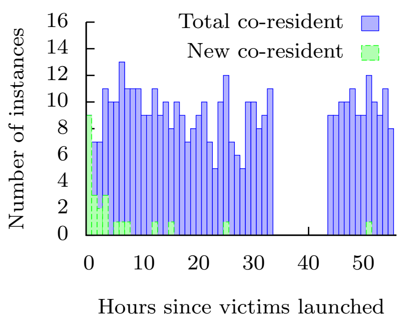
On targeting commercial instances. We briefly experimented with targeted instance flooding against instances run by other user's accounts. RightScale is a company that offers "platform and consulting services that enable companies to create scalable web solutions running on Amazon Web Services" [28]. Presently, they provide a free demonstration of their services, complete with the ability to launch a custom EC2 instance. On two separate occasions, we setup distinct accounts with RightScale and used their web interface to launch one of their Internet appliances (on EC2). We then applied our attack strategy (mapping the fresh instance and then flooding). On the first occasion we sequentially launched two rounds of 20 instances (using a single account) before achieving co-residence with the RightScale instance. On the second occasion, we launched two rounds of 38 instances (using two accounts). In the second round, we achieved a three-way co-residency: an instance from each of our accounts was placed on the same machine as the RightScale server.

rPath is another company that offers ready-to-run Internet appliances powered by EC2 instances [29]. As with RightScale, they currently offer free demonstrations, launching on demand a fresh EC2 instance to host systems such as Sugar CRM, described as a "customer relationship management system for your small business or enterprise" [29]. We were able to successfully establish a co-resident instance against an rPath demonstration box using 40 instances. Subsequent attempts with fresh rPath instances on a second occasion proved less fruitful; we failed to achieve co-residence even after several rounds of flooding. We believe that the target in this case was placed on a full system and was therefore unassailable.

Discussion. We have seen that attackers can frequently achieve co-residence with specific targets. Why did the strategy fail when it did? We hypothesize that instance flooding failed when targets were being assigned to machines with high instance density (discussed further in Appendix B) or even that became full. While we would like to use network probing to better understand this effect, this would require port scanning IP addresses near that of targets, which would perhaps violate (the spirit of) Amazon's AUP.

7.3 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.



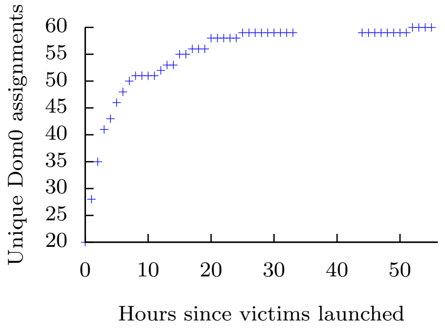


Figure 4: Results for the experiment measuring the effects of increasing time lag between victim launch and probe launch. Probe instances were not run for the hours 34–43. (Left) "Total co-resident" corresponds to the number of probe instances at the indicated hour offset that were co-resident with at least one of the victims. "New co-resident" is the number of victim instances that were collided with for the first time at the indicated hour offset. (Right) The cumulative number of unique Dom0 IP addresses assigned to attack instances for each round of flooding.

8. 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. We have experimentally investigated the first two on running EC2 instances. 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 (e.g., [10, 22, 26]) 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 (in the case of EC2) unknown load from other instances and a fortuitous choice of CPU configuration (e.g. no hyperthreading). 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 [1, 2, 3, 12], DRAM memory bus [21], CPU pipelines (e.g., floating-point units) [4], scheduling of CPU cores and timeslices, disk access [16], etc. We have implemented and measured simple covert channels (in which two instances cooperate to send a message via shared resource) using memory bus contention, obtaining a 0.006bps channel between co-resident large instances, and using hard disk contention, obtaining a 0.0005bps channel between co-resident m1.small instances. In both cases no attempts were made at optimizing the bandwidth of the covert channel. (The hard disk contention channel was used in Section 6 for establishing co-residence of instances.) Covert channels provide evidence that exploitable side channels may exist.

Though this is not our focus, we further observe that the same resources can also be used to mount cross-VM performance degradation and denial-of-service attacks, analogously to those demonstrated for non-virtualized multiprocessing [12, 13, 21].

8.1 Measuring cache usage

An attacking instance can measure the utilization of CPU caches on its physical machine. These measurements can be used to estimate the current load of the machine; a high load indicates activity on co-resident instances. Here we describe how to measure cache utilization in EC2 instances by adapting the Prime+Probe technique [22, 32]. We also demonstrate exploiting such cache measurements as a covert channel.

Load measurement. We utilize the Prime+Probe technique [22, 32] to measure cache activity, and extend it to the following Prime+Trigger+Probe measurement to support the setting of time-shared virtual machines (as present on Amazon EC2). The probing instance first allocates a contiguous buffer B of b bytes. Here b should be large enough that a significant portion of the cache is filled by B. Let s be the cache line size, in bytes. Then the probing instance performs the following steps to generate each load sample:

(1) **Prime**: Read B at s-byte offsets in order to ensure it is cached.

(2) **Trigger**: Busy-loop until the CPU's cycle counter jumps by a large value. (This means our VM was preempted by the Xen scheduler, hopefully in favor of the sender VM.)

(3) **Probe**: Measure the time it takes to again read B at s-byte offsets.

When reading the b/s memory locations in B, we use a pseudorandom order, and the pointer-chasing technique described in [32], to prevent the CPU's hardware prefetcher from hiding the access latencies. The time of the final step's read is the load sample, measured in number of CPU cycles. These load samples will be strongly correlated with use of the cache during the trigger step, since that usage will evict some portion of the buffer B and thereby drive up the read time during the probe phase. In the next few sections we describe several applications of this load measurement side channel. First we describe how to modify it to form a robust covert channel.

Cache-based covert channel. Cache load measurements create very effective covert channels between cooperating processes running in different VMs. In practice, this is not a major threat for current deployments since in most cases the cooperating processes can simply talk to each other over a network. However, covert channels become significant when communication is (supposedly) forbidden by information flow control (IFC) mechanisms such as sandboxing and IFC kernels [34, 18, 19]. The latter are a promising emerging approach to improving security (e.g., web-server functionality [18]), and our results highlight a caveat to their effectiveness.

In the simplest cache covert-channel attack [15], the sender idles to transmit "0" and frantically accesses memory to transmit "1". The receiver accesses a memory block of his own and observes the access latencies. High latencies are indicative that the sender is evicting the receiver's data from the caches, i.e., that "1" is transmitted. This attack is applicable across VMs, though it tends to be unreliable (and thus has very low bandwidth) in a noisy setting.

We have created a much more reliable and efficient cross-VM covert channel by using finer-grained measurements. We adapted the Prime+Trigger+Probe cache measurement technique as follows. Recall that in a set-associative cache, the pool of cache lines is partitioned into associativity sets, such that each memory address is mapped into a specific associativity set determined by certain bits in the address (for brevity, we ignore here details of virtual versus physical addresses). Our attack partitions the cache sets into two classes, "odd sets" and "even sets", and manipulates the load across each class. For resilience against noise, we use *differential coding* where the signal is carried in the difference between the load on the two classes. Noise will typically be balanced between the two classes, and thus preserve the signal. (This argument can be made rigorous by using a random-number generator for the choice of classes, but the following simpler protocol works well in practice.)

The protocol has three parameters: a which is larger than the attacked cache level (e.g., $a = 2^{21}$ to attack the EC2's Opteron L2 cache), b which is slightly smaller than the attacked cache level (here, $b = 2^{19}$), and d which is the cache line size times a power of 2. Define *even addresses* (resp. *odd addresses*) as those that are equal to $0 \bmod 2d$ (resp. $d \bmod 2d$). Define the class of *even cache sets* (resp. *odd cache sets*) as those cache sets to which even (resp. odd) addresses are mapped.

The sender allocates a contiguous buffer A of a bytes. To transmit "0" (resp. 1) he reads the even (resp. odd) addresses in A. This ensures that the one class of cache sets is fully evicted from the cache, while the other is mostly untouched.

The receiver defines the difference by the following measurement procedure:

(1) Allocate a contiguous buffer B of b bytes

(2) Sleep briefly (to build up credit with Xen's scheduler).

(3) **Prime**: Read all of B to make sure it's fully cached.

(4) **Trigger**: Busy-loop until the CPU's cycle counter jumps by a large value. (This means our VM was preempted by the Xen scheduler, hopefully in favor of the sender VM.)

(5) **Probe**: Measure the time it takes to read all even addresses in B, likewise for the odd addresses. Decide "0" iff the difference is positive.

On EC2 we need to deal with the noise induced by the fact that each VM's virtual CPU is occasionally migrated between the (m1.small) machine's four cores. This also leads to sometimes capturing noise generated by VMs other than the target (sender). Due to the noise-cancelling property of differential encoding, we can use a straightforward strategy: the receiver takes the average of multiple samples for making his decision, and also reverts to the prime stage whenever it detects that Xen scheduled it to a different core during the trigger or probe stages. This simple solution already yields a bandwidth of approximately 0.2bps, running on EC2.

8.2 Load-based co-residence detection

Here we positively answer the following question: can one test co-residence without relying on the network-based techniques of Section 6? We show this is indeed possible, given some knowledge of computational load variation on the target instance. This condition holds when an adversary can actively cause load variation due to a publicly-accessible service running on the target. It might also hold in cases where an adversary has a priori information about load variation on the target and this load variation is (relatively) unique to the target.

Consider target instances for which we can induce computational load — for example, an instance running a (public) web server. In this case, an attacker instance can check for co-residence with a target instance by observing differences in load samples taken when externally inducing load on the target versus when not. We experimentally verified the efficacy of this approach on EC2 m1.small instances. The target instance ran Fedora Core 4 with Apache 2.0. A single 1 024-byte text-only HTML page was made publicly accessible. Then the co-residence check worked as follows. First, the attacker VM took 100 load samples. (We set $b = 768 * 1024$ and $s = 128$. Taking 100 load samples took about 12 seconds.) We then paused for ten seconds. Then we took 100 further load samples while simultaneously making numerous HTTP get requests from a third system to the target via jmeter 2.3.4 (a utility for load testing HTTP servers). We set jmeter to simulate 100 users (100 separate threads). Each user made HTTP get requests as fast as possible.

The results of three trials with three pairs of m1.small instances are plotted in Figure 5. In the first trial we used two instances known to be co-resident (via network-based co-residence checks). One can see the difference between the load samples when performing HTTP gets and when not. In the second trial we used a fresh pair of instances co-resident on a different machine, and again one can easily see the effect of the HTTP gets on the load samples. In the third trial, we used two instances that were not co-resident. Here the load sample timings are, as expected, very similar. We emphasize that these measurements were performed on live EC2 instances, without any knowledge of what other instances may (or may not) have been running on the same machines. Indeed, the several spikes present in Trial 2's

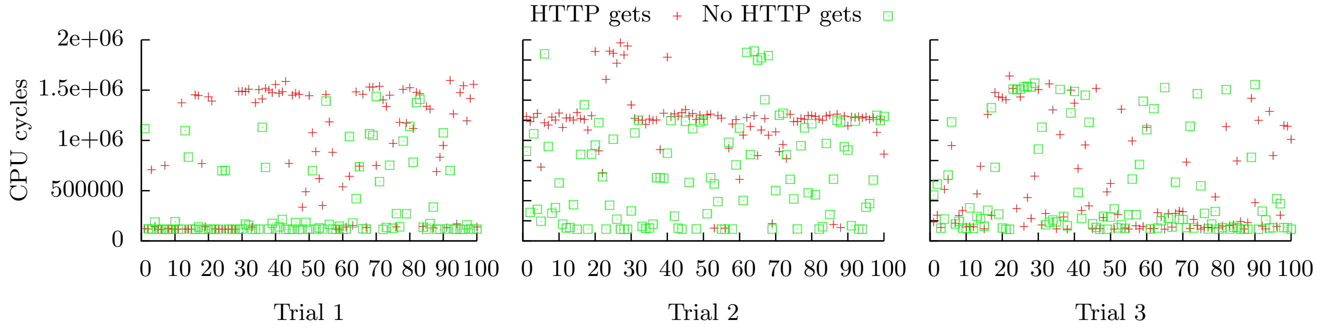


Figure 5: Results of executing 100 Prime+Trigger+Probe cache timing measurements for three pairs of m1.small instances, both when concurrently making HTTP get requests and when not. Instances in Trial 1 and Trial 2 were co-resident on distinct physical machines. Instances in Trial 3 were not co-resident.

load measurements were likely due to a third co-resident instance's work load.

8.3 Estimating traffic rates

In theory load measurement might provide a method for estimating the number of visitors to a co-resident web server or even which pages are being frequented. In many cases this information might not be public and leaking it could be damaging if, for example, the co-resident web server is operated by a corporate competitor. Here we report on initial experimentation with estimation, via side channel measurements, of HTTP traffic rates to a co-resident web server.

We utilized two m1.small instances, as in the trials discussed above. We then instructed one of the instances to perform four separate runs of 1 000 cache load measurements in which we simultaneously (1) sent no HTTP requests, (2) sent HTTP requests at a rate of 50 per minute, (3) 100 per minute, or (4) 200 per minute. As before we used jmeter to make the requests, this time with 20 users and the rate maintained across all users. Taking 1 000 load measurements takes about 90 seconds. The requested web page was a 3 megabyte text file, which amplified server load per request compared to a smaller page. We repeated this experiment three times (with the same instances). The graph in Figure 6 reports the mean load samples from these three trials, organized according to traffic rate. Note that among the 12 000 samples taken, 4 were extreme outliers (2 orders of magnitude larger than all other samples, for reasons unclear); we omitted these outliers from the calculations.

Figure 6 shows a clear correlation between traffic rate and load sample. This provides evidence that an attacker might be able to surreptitiously estimate traffic rates in some cases.

8.4 Keystroke timing attack

In this section we describe progress on the use of cache-based load measurements as a means for mounting keystroke timing attacks [31]. In such an attack, the adversary's goal is to measure the time between keystrokes made by a victim typing a password (or other sensitive information) into, for example, an SSH terminal. The gathered inter-keystroke times (if measured with sufficient resolution) can then be used to perform recovery of the password. In prior work [31], the attacker was assumed to have a network tap to time packet arrivals. In third-party compute clouds, we can replace network taps with co-residence and load measurements:

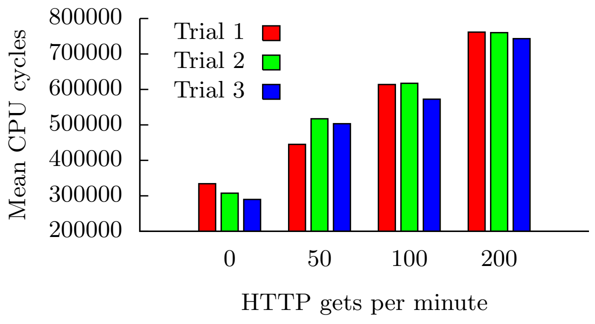


Figure 6: Mean cache load measurement timings (over 1 000 samples) taken while differing rates of web requests were made to a 3 megabyte text file hosted by a co-resident web server.

when the user of a co-resident instance types sensitive information into the shell, the malicious VM can observe the keystroke timings in real time via cache-based load measurements. Our hypothesis is that, on an otherwise idle machine, a spike in load corresponds to a letter being typed into the co-resident VM's terminal. We experimentally validated this hypothesis on an EC2-like virtualized environment, described next.

Experimental setup. For what comes below, we ran our experiments not on EC2 but on a local testbed, running a configuration of Opteron CPUs, Xen hypervisor and Linux kernels that is very similar (to the extent that we could discern) to that used to host EC2 m1.small instances — but with the VMs pinned to specific cores. EC2, in contrast, occasionally migrates the instances' virtual CPUs between the machine's four cores. We refer to our testbed as the *pinned Xen machine*.

The pinned Xen machine avoids two technical complications in conducting the attacks on EC2: it ensures that the machine is completely idle other than the test code, and it allows us to know the relation between VMs involved in the attack (i.e., whether they timeshare the same core, are assigned to two cores in the same physical chip, or neither). Machines in EC2 might indeed be idle (or at least have one idle core, which suffices for our attack) during a non-negligible fraction of their time, especially during off-peak hours; and the assignment of virtual CPUs changes often enough on EC2 that any desired combination will be occasionally achieved. That we were able to establish reliable covert channels via the cache in EC2 already testifies to this. A patient attacker might just wait for the requisite condition to come up, and detect them by adding redundancy to his transmission.

Note also that the pinned Xen machine is by itself a realistic setup for virtualized environments. In fact, in a cloud data center that never over-provisions CPU resources, pinning VMs to CPUs may improve performance due to caches and NUMA locality effects. We thus feel that these attacks are of interest beyond their being progress towards an attack within EC2.

Keystroke activity side channel. We utilize the Prime+Trigger+Probe load measurement technique to detect momentary activity spikes in an otherwise idle machine. In particular, we repeatedly perform load measurements and report a keystroke when the measurement indicates momentarily high cache usage. Further analysis of *which* cache sets were accessed might be used to filter out false positives, though we found that in practice it suffices to use simple thresholding, e.g., reporting a keystroke when the probing measurement is between 3.1μs and 9μs (the upper threshold filters out unrelated system activity).

We have implemented and evaluated this attack on the pinned Xen machine, with variants that exploit either L1 or L2 cache contention. The attacking VM is able to observe a clear signal with 5% missed keystrokes and 0.3 false triggers per second. The timing resolution is roughly 13ms. There is also a clear difference between keys with different effects, e.g., typing a shell command vs. pressing Enter to execute it. While the attacker does not directly learn exactly which keys are pressed, the attained resolution suffices to conduct the password-recovery attacks on SSH sessions due to Song et al. [31].

The same attack could be carried over to EC2, except that this measurement technique applies only to VMs that time-share a core. Thus, it can only reliably detect keystrokes during periods when EC2's Xen hypervisor assigns the attacker and victim to the same core. Assuming uniformly random assignment, this is about 25% of the time (and changes at most every 90ms, typically much slower). Statistical analysis of measurements might be used to identify periods of lucky allocation. We conjecture that measuring not just the overall momentary load, but also the use of individual cache associativity sets during the trigger stage, might further help identify the target VM.

8.5 Inhibiting side-channel attacks

One may focus defenses against cross-VM attacks on preventing the side-channel vulnerabilities themselves. This might be accomplished via blinding techniques to minimize the information that can be leaked (e.g., cache wiping, random delay insertion, adjusting each machine's perception of time [14], etc.). Countermeasures for the cache side channels (which appear to be particularly conducive to attacks) are extensively discussed, e.g., in [23, 24, 10, 26, 25, 22]. These countermeasures suffer from two drawbacks. First, they are typically either impractical (e.g., high overhead or nonstandard hardware), application-specific, or insufficient for fully mitigating the risk. Second, these solutions ultimately require being confident that *all* possible side-channels have been anticipated and disabled — itself a tall order, especially in light of the deluge of side channels observed in recent years. Thus, at the current state of the art, for unconditional security against cross-VM attacks one must resort to avoiding co-residence.

9. 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.

Acknowledgments

Eran Tromer thanks Ron Rivest and Saman Amarasinghe for discussion of potential attacks. This work was supported by NSF grants NSF-0433668, CNS-0808907, CNS-0524765, CNS-0627779 and NSF-0829469, AFRL grant FA8750-08-1-0088, and partially supported by a MURI grant administered by the Air Force Office of Scientific Research. Views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFRL, NSF, the U.S. Government or any of its agencies.

10. REFERENCES

[1] O. Acıiçmez, Ç. Kaya Koç, and J.P. Seifert. On the power of simple branch prediction analysis. IACR Cryptology ePrint Archive, report 2006/351, 2006.
[2] O. Acıiçmez, Ç. Kaya Koç, 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. Acıiçmez. Yet another microarchitectural attack: exploiting I-cache. IACR Cryptology ePrint Archive, report 2007/164, 2007.
[4] O. Acıiçmez, 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/agreement/`

[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] P. Karger and J. Wray. Storage channels in disk arm optimization. *IEEE Symposium on Security and Privacy*, pp. 52–56, IEEE, 1991.

[17] O. Kerr. Cybercrime's scope: Interpreting 'access' and 'authorization' in computer misuse statutes. *NYU Law Review*, Vol. 78, No. 5, pp. 1596–1668, November 2003.

[18] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. ACM Symposium on Operating Systems Principles (SOSP), 2007.

[19] M. Krohn, and E. Tromer. Non-Interference for a Practical DIFC-Based Operating System. IEEE Symposium on Security and Privacy, 2009.

[20] Microsoft Azure Services Platform. `http://www.microsoft.com/azure/default.mspx`

[21] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. *USENIX Security Symposium*, pp. 257–274, 2007.

[22] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. *RSA Conference Cryptographers Track (CT-RSA) 2006*, 2006.

[23] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical report CSTR-02-003, Department of Computer Science, University of Bristol, 2002. Available at `http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=1000625`.

[24] D. Page. Defending against cache-based side-channel attacks. *Information Security Technial Report*, vol. 8 issue. 8, 2003

[25] D. Page. Partitioned cache architecture as a side-channel defence mechanism. IACR Cryptology ePrint Archive, report 2005/280, 2005.

[26] C. Percival. Cache missing for fun and profit *BSDCan 2005, Ottawa*, 2005.

[27] Rackspace Mosso. `http://www.mosso.com/`

[28] RightScale. `http://rightscale.com/`

[29] rPath. `http://www.rpath.com`

[30] scalr. `http://code.google.com/p/scalr/`

[31] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and SSH timing attacks. *10th USENIX Security Symposium*, 2001.

[32] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, available online, July 2009.

[33] Xen 3.0 Interface Manual. Available at `http://wiki.xensource.com/xenwiki/XenDocs`.

[34] N. B. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. *Symposium on Operating Systems Design and Implementation (OSDI), 2006*

APPENDIX

A. LEGAL, ETHICAL, AND CONTRACTUAL OBLIGATIONS

Network probing is central to our study and while we note that remote probing of Internet servers and devices is a common technique, both in practice and in the networking and security research literature, it is also sometimes controversial. While we cannot hope to cover this controversy in its full complexity, we attempt to sketch its broad outlines as it applies to our own work.

In this discussion, we separate between legal obligations under statute, ethical concerns, and our contractual obligations under the customer agreement provided by Amazon, considering each in turn.

In the United States, the prevailing statute concerning interacting with computer systems is the Computer Fraud and Abuse Act (CFAA) which requires, roughly speaking, that computer system access must be authorized. As with many such statutes, the wording is quite broad and there is considerable ambiguity in the terms authorization and access (we refer the reader to Kerr [17] for an elaboration on these complexities and associated legal decisions). We are unaware of any case law covering the topic of network probing writ large, however on the more controversial issue of external "port scanning" (that is, scanning a range of networks ports, absent explicit permission, particularly in search of potential network-accessible vulnerabilities) we are informed by Moulton v VC3 (2000). This decision provided that port scanning, by itself, does not create a damages claim (i.e., that direct harm must be shown to establish damages under the CFAA).

However, we are also sensitive to the ethical issues resulting from the *perception* of a threat, especially when no greater good is achieved, and thus we are careful to restrict our network probes to services that are designed to be *publicly facing*— TCP port 80 (the standard port for HTTP Web service) and TCP port 443 (the standard port for HTTPS Web service). We operate under the assumption that providing a service designed to be accessed by the public is an implicit authorization to do so (in the same sense that having a doorbell provides an implicit authorization to ring it). We believe that the counterfactual theory, that addresses themselves are private and that it is unethical to visit a Web server absent an explicit advance invitation to do so, seems difficult to reconcile with how much of the Internet actually works. Finally, we should be clear that we make no attempt to interact with these sites beyond establishing their presence and, in some cases, downloading the public home page they export; there were no vulnerabilities searched for, discovered, or exposed through our measurements and we implicitly respect any access control

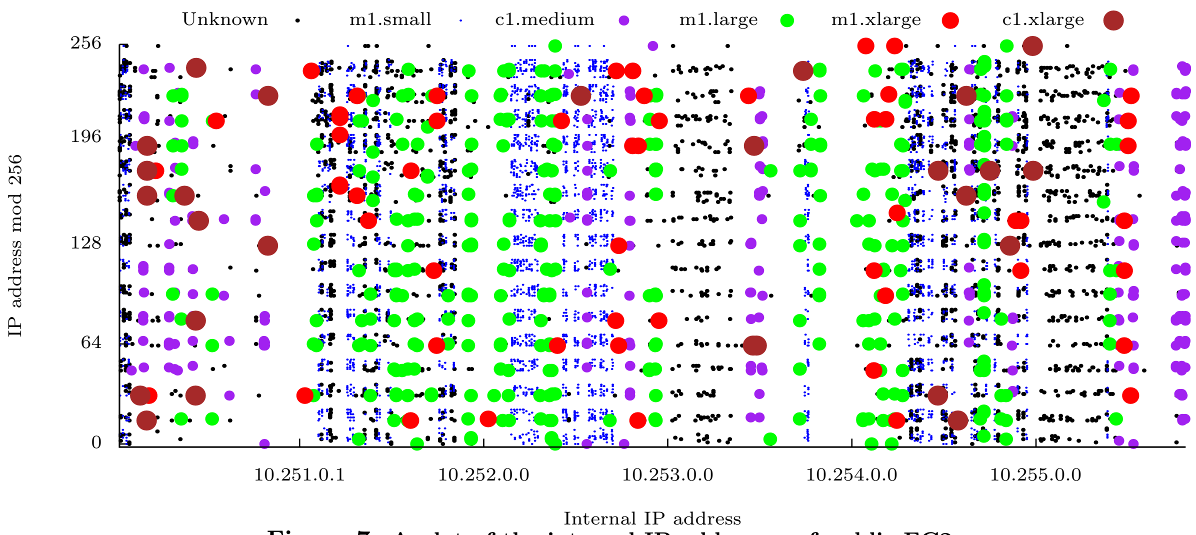


Figure 7: A plot of the internal IP addresses of public EC2 servers.

mechanisms put in place.

Our internal probes originate from within the Amazon EC2 service (i.e., from instances under our control). As such they are additionally subject to contractual obligations under Amazon's terms of service [8] at the time of this study. Particularly salient in the AWS Web Services Agreement is subsection 5.5.5 on the use of the Network in the EC2 service: "You may make network connections from Amazon EC2 hosted servers to other hosts only with the permission and authorization of the destination hosts and networks". While, unlike the CFAA, this seems clearer on the topic of *access* (making a network connection) it remains ambiguous concerning what constitutes *authorization*. However, some guidance can be found in the subsequent examples of unacceptable traffic provided in the same subsection, including "Unauthorized probes and port scans for vulnerabilities." and "Web crawling which is not restricted in a rate so as not to impair or otherwise disrupt the servers being crawled." The first example makes it clear that creating network connections to discover *vulnerabilities* is circumscribed, but the second indicates that connecting to Web servers is implicitly authorized (since this is what Web crawlers do) so long as they do not impair their function. Thus, we operate under the interpretation that connecting to Web servers from within EC2 is implicitly authorized so long as we are not disrupting them nor attempting to discover vulnerabilities therein (which we do not).

Finally, we wish to be clear that we made no attempt to disrupt, impair or acquire private information from any customer or client of the EC2 service. Thus, any "attacks", as described in this paper, are mounted between EC2 instances under our control (either directly, or via a third-party service) and should have in no way impacted any third party.

B. INSTANCE DENSITY AND PLACEMENT

We conducted a TCP connect scan on the EC2 address space for ports 80 and 443, and translated these to a list of internal IP addresses using DNS lookups from within the cloud. (This was a separate scan from the one discussed in Section 5, but performed in the same manner.) We repeated the following ten times. Twenty "victim" instances were launched. These instances then determined a (very loose) lower-bound on the number of co-resident instances by determining the number of co-resident servers from the public servers list. These instances were left running while 20 further probe instances were launched. Each probe checked whether it was co-resident with one of the victims. Figure 8 displays the results. The average (over 10 iterations) mean number of co-resident servers for the victims for which a probe was co-resident was 1.2. The average mean number of co-resident servers for the victims for which no probe was co-resident was 1.7. This suggests a slight bias towards assignment of new instances to lightly loaded machines. We expect that with better measurement techniques one would see an even stronger bias, however we avoided measurement techniques that could be seen as having violated (the spirit of) Amazon's AUP.

	Found		Missed	
Iteration	count	mean	count	mean
1	18	1.22	2	2
2	16	1	4	1.75
3	17	1.18	3	1.33
4	8	1.13	12	1.75
5	18	1.44	2	1.5
6	12	1.33	8	1.63
7	17	1.29	3	1.66
8	11	0.91	9	1.66
9	16	1.31	4	1.75
10	16	1.18	4	2.25

Figure 8: Instance density averages for runs of 20 victim instances and then 20 probe instances, over 10 iterations. The 'count' columns specify (respectively) the number of victims which were found by (co-resident with) a probe or missed by the probes. The 'mean' columns report the average number of other instances running on the victim instances' physical machines (before the probes were launched).