J Supercomput (2016) 72:961–984
DOI 10.1007/s11227-016-1627-9

Sibling virtual machine co-location confirmation and avoidance tactics for Public Infrastructure Clouds

John O'Loughlin[1] · Lee Gillam[1]

Published online: 10 February 2016

Abstract Infrastructure Clouds offer large scale resources for rent, which are typically shared with other users—unless you are willing to pay a premium for single tenancy (if available). There is no guarantee that your instances will run on separate hosts, and this can cause a range of issues when your instances are co-locating on the same host including: mutual performance degradation, exposure to underlying host failures, and increased threat surface area for host compromise. Determining when your instances are co-located is useful then, as a user can implement policies for host separation. Co-location methods to date have typically focused on identifying co-location with another user's instance, as this is a prerequisite for targeted attacks on the Cloud. However, as providers update their environments these methods either no longer work, or have yet to be proven on the Public Cloud. Further, they are not suitable to the task of simply and quickly detecting co-location amongst a large number of instances. We propose a method suitable for Xen based Clouds which addresses this problem and demonstrate it on EC2—the largest Public Cloud Infrastructure.

Keywords Virtualisation · Cloud computing · Xen · Co-location · Security and performance

✉ John O'Loughlin
john.oloughlin@surrey.ac.uk

Lee Gillam
l.gillam@surrey.ac.uk

[1] Department of Computer Science, University of Surrey, Guildford GU2 7XH, UK

1 Introduction

Infrastructure Clouds offer compute resources for rent on-demand, typically on a per hour basis [1]. One of the most popular offerings is the virtual server, which is the mainstay of providers of Infrastructure Clouds such as Amazon, Google and Microsoft. Customers can rapidly acquire running virtual servers (known as instances), use them for as long as required, then release them when no longer needed, with the equivalent resource then available for use by other customers.

The ability for a user to scale their infrastructure up and down as required is referred to as elasticity, or 'elastic infrastructure'. In such environments, instances may be relatively short lived and applications designed for Cloud deployments do not, or certainly should not, rely on the permanence of any particular instance.

Infrastructure Clouds support elasticity through the use of hypervisors, such as Xen and KVM, which can logically partition a physical server into multiple virtual servers on request. This allows a provider to share each physical host between multiple customers, and any given physical host may well have instances from multiple customers co-located on it at any given time. Providers argue that through multi-tenancy they can achieve high utilization rates [2], although they do not publish the rates achieved, and resulting cost savings can be passed onto customers, who in turn are more likely to use Cloud services, with resultant increases in utilization. Amazon refers to this as a 'virtuous circle', and it is clear that multi-tenancy with high occupancy rates is at the heart of the Cloud Infrastructure business model.

However, multi-tenancy does raise various concerns for users of such systems, of which security and performance are key. For *security*, such concerns include hypervisor breakouts, whereby hypervisor security is compromised and a guest virtual machine can run arbitrary code with the same privileges as the hypervisor on the underlying host. All virtual machines co-located on a host compromised in this manner are now vulnerable to attack from the malicious virtual machine. Even without a breakout, certain shared resources, such as L2 caches and network cards, may be exploitable for various purposes. The L2 cache in particular is vulnerable to information leakage whereby one virtual machine can extract information from another, with demonstrations of this vulnerability including the extraction of encryption keys. For *performance*, one such concern is noisy neighbours, where performance degradation occurs for co-located instances when the (legitimate and not necessarily malicious) resource consuming actions of co-locating instances coincide in the shared components of the hardware—for example, requiring a large memory bandwidth.

Multi-tenant environments also offer the potential for *targeted attacks*. Whilst targeted attacks themselves are not new, with Distributed Denial of Service (DDoS) and brute force SSH attacks commonplace on the public Internet, Clouds offer a new vector for them: it may be possible to *place* an instance on the same physical host as the target instance. Of course, before such an attack can be carried out, the first requirement is that of detecting co-location, i.e. determining whether your instance is on the same host as the intended target—which is assumed to be owned by another user.

However, for the majority of users, identifying co-location amongst their *own* instances may be of even more use. Sibling instances[1] that are co-located may be undesirable for at least the following reasons:

1. They may degrade the performance of each other when running compute bound workloads.
2. They are all vulnerable to failure, or degradation, of the underlying host. This complicates the building of redundant services on top of Infrastructure Clouds.
3. They are all vulnerable to the same noisy neighbours.
4. There is more exposure to a security compromise on a single host.

If a user can detect when their siblings instances are co-locating, they could put in place a *host separation policy* to ensure that instances that should not be on the same host can be detected and corrected for. Such a policy would involve starting additional instance of the desired type and only putting into deployment those instances that are not co-locating, and could be incorporated into instance deployment systems.

To date, methods for identifying co-location with a targeted virtual machine, assumed to be owned by another user, include (1) simple network probes (2) watermarking network flows and (3) measuring shared cache usage. These techniques can of course be applied to detecting co-locating siblings. However, as we discuss in detail in Sect. 2, these methods are either no longer viable or are as yet unproven on the Public Cloud. There is a need, then, for a simple test that can address the sibling co-location problem in elastic deployments. We list our test requirements as follows:

1. Simple to implement
2. Can quickly determine non co-location
3. Can determine co-location with a high degree of assurance
4. Scales with the size of the deployment being tested

In this paper, we address these requirements by exploring a trace derived from data exported by the Xen hypervisor [3]—domain ids (domids). We demonstrate how an arbitrary minimum distance between co-locating domids may be introduced, and we use this as a watermark for determining co-location. We also suggest values for this based on measured rates of domid increases.

The rest of the paper is structured as follows: in Sect. 2 we review literature relevant to co-location, in particular we review extent work on performance and security issues as well previuous techniques for determining co-location. In Sect. 3 we discuss Xen domains and in particular the domain identification (domid) generation process. In Sect. 4 we demonstrate co-location detection in sole tenancy, which serves to validate and link the methods and ideas discussed in Sect. 3 to the multi-tenant case. Detection in the multi-tenant environment is described and results discussed in Sects. 5–8. In Sect. 9 we consider how we might address the same problem for containers, which are gaining popularity for workload packaging on the Public Cloud. Section 10 considers possible detection of previous locations—and the implications this has for targeted attacks as well as so-called instance seekers. In Sect. 11 we consider costs involved for

[1] We will refer to instances started by the same user as *sibling instances* in the remainder of this paper.

implementing a host separation policy, and finally, in Sect. 12 we present a summary, conclusions and future work.

2 Related work

2.1 Performance issues

In a virtualized environment, no matter whether Public Cloud or on-premise or remotely hosted data centre, the ability for one instance to degrade the performance of other co-located instances is well documented [4] and is referred to as 'noisy neighbours'. Intel identifies the primary cause of noisy neighbours as the sharing of resources, such as the L2 cache, which cannot be partitioned [5]; that is, there is no mechanism to limit how much of the resource an instance may consume. Consequently, it is possible for some instances to use such resources disproportionately, to the detriment of others.

Xen itself can exacerbate the noisy neighbor problem by the way it accounts for CPU time used by instances. Xen uses weights and CPU limits to allocate CPU time to instances on the same host, and two instances which are assigned the same CPU weight should obtain the same amount of CPU time as each other. Instances perform I/O operations (e.g. reading data from disk) by making hyper-calls to Xen, so Xen will perform a significant amount of work on behalf of instances running I/O bound workloads. However, the CPU time that Xen spends doing this is not accounted for in the instance's use of the CPU [6]. Consequently, an instance running an I/O bound workload can obtain a disproportionate amount of CPU time at the expense of co-located instances.

In multi-tenant environments, identifying if the performance of a running task is being affected by a noisy neighbor, and therefore likely to take longer to complete than expected, is difficult. On their production clusters, Google [4] attempt to detect tasks which are likely to be performing poorly—and also the noisy neighbor responsible! They do so by measuring a task's cycles per instruction (CPI), i.e. the number of cycles required to execute an instruction, on a per second basis. In this way they build a CPI distribution for the task. Due to architectural differences between CPU models, a task has a CPI distribution per CPU model. During a task's execution they compare the CPI measurements with the known CPI distribution and search for outliers. Here, an outlier is defined as being more than 2 standard deviations from the mean, and if more outliers are detected than the CPI distribution predicts, then performance of the task is likely to be poor. The protagonist, i.e. the noisy neighbour, is identified by correlating co-locating instances' CPU usage with the increase in CPI outliers for the victim. That is, if task A's use of the CPU repeatedly coincides with the occurrence of an outlier for task B, then task B is identified as a noisy neighbor. If a task is identified as being a noisy neighbour, its CPU usage may be throttled, or indeed it may even be terminated.

Google manages dynamic CPU allocation for tasks by packaging them into a Linux Container [7], which is an Operating Systems virtualisation technology, similar to Solaris Zones [8] and BSD jails [9]. These containers share the underlying OS, but are provided with their own namespaces. Processes started within the same container can see each other through standard process listing, but will not show processes started in other containers. Resource allocation to containers is managed via control groups, (cgroups) [10] which allows a tasks CPU usage to be managed dynamically. As well as throttling CPU usage for noisy neighbors, Google also allows tasks to use more than their allotted CPU usage if the resource is available.

The ability to identify noisy neighbors, and dynamically manage resource allocation, is crucial to Google as its web searches are latency sensitive. That is, each web search starts a number of tasks, all of which must complete within a fixed time. Previously, all tasks ran for the specified time, however the work done by any task which did not complete in time was not included in the results, and hence were a waste of resources. Now these tasks can be identified and terminated.

On a Public Cloud, avoiding being your own noisy neighbor has a performance and cost benefit. Public Clouds operate a ‘pay as you use’ billing model: on Amazon’s EC2, instances are rented by the hour, whilst on GCE and Azure the billing is per minute. Tasks whose performance is being degraded will likely take longer to complete than expected (based on past performance), and this may lead to higher costs.

2.2 Security issues

The recent spate [11] of hypervisor vulnerabilities, resulting in the large scale reboot of many Public Clouds, including EC2, GoGrid and Rackspace, has heightened the awareness of hypervisor vulnerabilities and their potential impact on large numbers of customers. At the time of writing there is limited evidence of any such breakouts in the wild. However, as we have already noted, security breaches in multi-tenant environments do not necessarily require a compromised hypervisor and may be achieved through the use of shared resources, such as the L2 cache, as we now discuss.

The problem of extracting information between co-locating virtual machines has been investigated by a number of authors. In [12], the sharing of an L2 cache between VMs was shown to be a possible vulnerability when it was demonstrated that one VM may extract cryptographic keys from another VM on the same host. Such an attack is known as an access driven side channel attack. Particularly noteworthy is the fact that the attack was demonstrated on an SMP system. In this case the challenge of core migrations i.e. the scheduling of a VM onto different cores during its lifetime, as would be encountered in a Cloud environment, needs to be overcome.

The vulnerability of a shared cache relies, in part, on exploiting hypervisor scheduling. Methods to increase the difficulty of successfully using such attacks are under development [13] and are already being integrated into Xen. Whilst such work mitigates fine-grained attacks, other attacks that seek to obtain a large share of the L2 cache are considered viable. In this case then, the intention is to be a noisy neighbour.

2.3 Co-location techniques

The potential to extract information between co-locating virtual machines has led to work on *targeted attacks* in the Cloud, where an attacker seeks to co-locate with a specific target. The attacker identifies the target via a publically available service (such as http) that they are running, and from this the IP address of the service can be determined. The initial identification that a target is running on a Public Cloud comes from comparing this IP address with known IP ranges used by Public Clouds. Next, the attacker must also attempt to determine the target's location within the Cloud i.e. the Region and Zone. Having done so, the attacker then speculatively launches instances within the same location in the hope that one of those instances is started on the same host as the target.

Such a targeted attack requires techniques for determining co-location with the target before the attack can be launched successfully. We classify techniques developed to date as:

1. Simple Network Based Probes
2. Network Flow Watermarking
3. Cache Avoidance

In [14], a number of network based probes were proposed including (1) ping round trip time and (2) common IP address of dom0. The latter technique works as follows: suppose we have two instances, A and B. In instance B we have a service running that any machine on the internet may connect to (for example an HTTP service). Instance A will launch a traceroute using a TCP probe against the open port on instance B. Note that, as ICMP echo replies (pings) may be disabled, the technique requires a running service on B. The traceroute command will output the list of IP addresses that have responded to the tcp probe. The last IP address to respond is the address of the service. On Xen, the penultimate response is from the Xen hypervisor, which is the dom0. In this way, instance A may determine the IP address of the dom0 which is running and managing instance B. Similarly, instance A can launch such a probe against *itself*, and determine the IP address of the dom0 it is running on. If the two IP addresses are the same then the instances are running on the same host.

To test the veracity of these methods they also use access timings of shared drives. No details are provided of the type of drive being used (local or network) or how the disk is being shared. Whilst access times to shared drives may potentially be used for detecting co-locating siblings, there are a number of issues not discussed that demand further investigation. Perhaps most important, is the widely reported variation in disk read/write timings on EC2 [1], which clearly needs to be accounted for in any test that uses such timings as a method for detection. Perhaps unsurprisingly, dom0 no long responds in a traceroute, as we and others [15] have confirmed and so the method is no longer viable.

Whilst simple network probes no longer work, the observation that instances on the same host most likely share the network card has led to the development [15] of a technique which allows one instance to inject a watermark into the network flow of another instance (on the same host). This time we require instances A, B and C. As before, instance B is the target, and must be running a publically accessible network service. Instance C is the control instance, whilst instance A is being tested for co-location with B. As B is running an accessible service, instance C connects to this service and maintains an open connection with it. In doing so instance C establishes a network flow from instance B to itself. Similarly, instance C establishes a network flow from itself to instance A. If A and B are on the same host then they are (most likely) sharing a network interface; that is, both their network flows to instance C are through the same physical network card. The aim then is to exploit this shared physical network card, with instance A being able to inject a watermark into the flow from B to C. If it is possible for instance A to do this, then A and B are (most likely) on the same host.

This technique was demonstrated on a variety of stand-alone virtual systems. It is notable as the only hypervisor agnostic test we have found, as all others seek to exploit properties of hypervisors—as indeed we ourselves do. However, as the authors state, there a number of defenses against watermarking in place in Public Clouds, and in particular on EC2, and to date the authors have been unable to successfully implement the test on a Public Cloud.

In [16] a technique is developed for detecting the presence of *any* other instances on the host. The aim of the work is to detect sole tenancy violations. With sole tenancy instances a provider makes a guarantee that the instance will not co-locate with instances owned by another user, however multiple sibling sole-tenancy instances may co-locate. To detect violations, a cache avoidance strategy is used whereby an instance avoids use of the L2 cache, and the resulting cache speed is measured after doing so. By detecting variation in cache timings, an instance detects if it is sharing the cache with other instances on the same host. To avoid use of the cache requires modifications to the kernel running the guest, which is technically challenging, and has a performance overhead.

In a sole tenancy environment—and assuming no violations of the tenancy agreement—multiple sibling instances can co-ordinate their use and avoidance of the cache and in doing so can detect each other. However, it is not clear if this technique can be extended to a multi-tenancy environment, where we expect the presence of other instances. As such the technique is unproven in this case.

In summary, neither simple network probes nor network flows watermarking co-location tests work on EC2 due to measures already in place, whilst cache avoidance in the multi tenancy environment remains unproven and technically challenging. This amplifies the need for simple, and workable, methods.

3 Xen domain identifiers (domids)

The Xen system is a widely deployed hypervisor in Infrastructure Cloud systems, and is in use at Amazon, Rackspace, IBM and GoGrid, amongst others. The Xen system consists of the Xen *hypervisor* together with a *privileged* VM called domain 0 or *dom0*. Xen is a bare-metal hypervisor, started by the BIOS, which in turn starts *dom0*. The d*om0* is a *privileged* VM and can directly access hardware such as network cards and local disk storage. *Dom0* provides a management interface for the Xen system, from which system administrators can launch and manage the life cycle of VMs. These VMs are unprivileged domains and are referred to as *domUs*.

The Xen hypervisor is responsible for scheduling VM CPU time, managing memory, and handling interrupts. On an x86 CPU, *dom0* privilege escalation is provided by running *dom0* in ring 1, whilst the Xen hypervisor runs in ring 0 (and the unprivileged

VMs, *domUs*, run in ring 3). *DomUs* gain access to hardware devices such as disks and network cards via calls to *dom0,* commonly referred to as hyper-calls.

Upon creation, each domain is given a UUID, which serves as a unique identifier amongst a deployment of multiple Xen systems; that is, it uniquely identifies a domain amongst the set of all domains across the Xen systems. For example, on EC2 the UUID assigned to a new instance will (in theory) be unique to that instance, at least within the Region in which it was launched.

In addition, a newly launched domain is assigned a domain identifier, referred to as the domid. This uniquely identifies domains on the physical server only. On EC2, instances on the same physical server will have different domids. However, these may well clash with domids for instances on other hosts. The domid is a 16 bit integer and allocation starts at 1 and is monotonically increasing with Xen assigning the next available domid, 2, 3 and so on.

When domid 65536 has been assigned, a ‘wraparound’ occurs, and Xen assigns the next available domid starting again from 1 to allocate new—but only available—domids: allocation will again occur in a monotonically increasing manner, but may be interrupted if an existing domain already has the next number in the sequence as a domid. For example, if the last assigned domid was 900, but existing domains have domids of 901 and 902, then the next available domid is 903. Domids do not, however, survive an underlying host reboot, and in this case the next available domid is reset to 1.

Generally, then, instances that are started one after the other will obtain consecutive domids. On EC2, therefore, we would typically expect co-locating instances that are started at the same time to have consecutive domids—or, with other requests also being satisfied, quite close to each other.

Xen domids have a rather interesting property, and one which will be crucial to us later: an instance can obtain a new domid simply by rebooting. Upon rebooting, an instance will obtain the next available domid. It is not possible to predict what this will be as its depends upon a number of factors, including the number of new instances that have started on the same host, the number of other instance reboots that have occurred, and whether or not the underlying host itself has been rebooted. In the later case, a host reboot is likely detectable since running instances will also be rebooted and the new domids will be assigned in the lower end of the range 1–65536. Of course, if an instance has an existing domid at the lower end then it may not be possible to distinguish between a host reboot and an unexpected instance reboot.

In the case where there is only one instance on a host and it is guaranteed that there are no co-locating instances, then the instance can cause its domid to monotonically increment simply by repeated reboots. That is, if the current domid is Y, then the only activity on the host which can cause the next available domid to increment is the activity of the instance itself. In this case, a reboot produces a new domid of $Y + 1$, a second reboot increments the domid to $Y + 2$, and after $k > 0$ reboots the instance's domid is $Y + k$. If a wraparound occurs then the value of domid is $Y + k + 1$ modulo 65537. Note that we need to include a +1 on our expression as the domid of 0 is already assigned to the dom0. For example, if an instance's current domid is 65536, then a reboot will produce a new domid of (65536 + 1 + 1) modulo 65537 which is 1.

Sibling instances on the same host, and in the guaranteed absence of non-siblings, can coordinate their reboots, and in doing so can use this to increment the next available domid by at least some agreed fixed amount each time. That is, instance A can increment the next available domid by $k > 0$ and this is observable by a sibling instance B, as A and B have the next available domid in common. In turn, B can also increment by at least a fixed amount, and again this should then be observable by A.

In multi-tenant environments, we can increment in this fashion by at least a fixed amount and not an exact amount. That is, an instance which reboots itself $k > 0$ times causes the next available domid to increment by at least k. There may well be instances other than the siblings on the host, which of course also share the next available domid, and whose activity causes it to increment, as well as new instances being launched on the same host. Before we consider the more general case of detecting co-locating siblings in a multi-tenant environment we first consider the case of detecting in a sole-tenancy environment.

4 Detecting sibling co-location in single tenancy instances

Some Cloud providers, such as Amazon, offer single-tenancy instances, also known as dedicated instances. Single-tenancy instances are guaranteed not to co-locate with any other users' instances. This does come at an increase in costs. Although the host may not be shared with others users, it is entirely possible that siblings can co-locate. Indeed, it would seem more efficient from Amazon's point of view to co-locate single-tenancy siblings.

In this section we show how we can detect co-location of a pair of sibling single tenancy instances. We exploit the properties of domids discussed in Sect. 3, as well as the guaranteed absence of any other instances on the hosts which the siblings are running on. We begin by explaining how domids are obtained. (Note: In this section all instances under discussion are single tenancy instances.)

4.1 Obtaining the domid

A user does not have administrative access to Xen on EC2 (or indeed any Public Cloud). However, we can determine an instance's domid via Xenstore. Xenstore [17] is a data area exported from dom0 to domUs, the interface of which is a pseudo file system which can be mounted on /proc/xen within a guest. This is analogous to the /proc and /sys pseudo file systems in Linux which provide an interface for user space processes to the Linux kernel. Under a standard Xen system, a domain can extract information such as the domids of all the running domains and the CPU weightings assigned to them. As one would expect, on EC2 the data exported to the instances via Xenstore is restricted, and does not allow a domain to obtain any information other than about itself. However, it is particularly useful for our purposes that a domain can obtain its own domid.

A user can obtain the domid with the following steps:

1. Install the xen-utils package

2. mount the /proc/xen filesystem with the following command: *mount –t xenfs none /proc/xen*
3. Run the command: *sudo xenstore-read* domid

4.2 Single tenancy sibling co-location detection

Consider a pair of single-tenancy sibling instances in the same AZ. Further, suppose the user has no other running instances apart from this pair. Note that as we have single tenancy, the instances are either on the same host as each other i.e. are co-locating, or each is running on a host entirely free of co-locating instances.

Suppose that the current domids of the two instances are X and Y respectively, and that $X > Y$, and we refer to these instances as A and B respectively.

Suppose we reboot the instance with the higher domid, A, k times where $0 < k < 65536$, whilst ensuring that instance B is not rebooted during this period. We can of course ensure this since they are siblings. Further, assume that a wraparound did not occur for instance A with respect to its new domid. In this case then, instance A's new domid is $X + k$. On the host A is running on, the next available domid is either $X + k + 1$, or 1 if A has domid 65536. For instance B, there are only *two choices* for its next available domid, which correspond to it co-locating with A or not. If B is not co-locating with A, then the next domid it will obtain is $Y+1$, with $Y+1 < X+k+1$ since $Y < X$. If it is co-locating, then next available domid B can obtain is $X + k + 1$ or 1. To determine co-location, we simply reboot B and obtain its new domid.

Now, suppose a wraparound did occur while we rebooted A $k > 0$ times. As stated in Sect. 3, its new domid is $Z = X + k + 1$ modulo 65537. Now, the next available domid on the host A is running on $Z + 1$. If B is not co-locating with A, then upon a reboot B will obtain a new domid of $Y + 1 \leq 65536$, since $Y < X \leq 65536$. If B is co-locating with A then its next domid will be $Z + 1$. Again, to determine co-location, we simply reboot B and obtain its new domid.

For the rest of this paper, when discussing instance reboots, we assume for the sake of simplicity that a wraparound does not occur, and note that an extension of the argument being made is possible to the wraparound case.

4.3 Investigating co-location in single tenancy

We investigate single tenancy on Amazon's EC2, and we start by launching four pairs of m4.large instances in the US-East-1 Region. For each pair was launched from the same request as single-tenancy instances, and further each pair was launched in a different availability zone (AZ): us-east-1a, us-east-1b, us-east-1c and us-east-1e. For each pair we recorded the domids, and these are shown in the Table 1.

We can observe that the domids for our pairs are all consecutive. Instances started consecutively on the same host, and in the absence of any other instance launching activity, would of course have consecutive domids. This is a first strong hint that the pairs of instances may be co-locating. For each of the pairs, we then rebooted the instance with the higher domid four times, and ensured the instance with the lower domid was not rebooted. As per the co-location test described, if we reboot the instance

Table 1 Domid pairs

AZ	Domid pairs
A	35, 36
B	59, 60
C	21, 22
E	47, 48

with the lower domid, we would now expect its domid will either increment by 1 or by 5. We reiterate that because we are in a single tenancy environment these should be the only choices available. If the increment is by 1, then the instances cannot be co-locating, since this value is less than the domid of the other instance. If it has incremented by 5, then as the only activity on the host that could have caused this is the other instance, they must be co-locating. For all four pairs of instances above, the domid of the second instance was incremented by five each time, and this was true for further five pairs of m4.large instances. This kind of incrementality was also seen for pairs of sole-tenancy m4.xlarge instances and also for pairs of sole-tenancy m4.2xlarge instances.

Next we launched three single tenancy instances, one of each type of m4.large, m4.xlarge and m4.2xlarge. Using our domid test we determined that the instances were not co-locating, and we suspect that this was entirely due to the fact that they are of different types. We kept these three instances running and launched an additional three single tenancy instances, again one each of m4.large, m4.xlarge and m4.2xlarge, in the same AZ. Using our domid test we find the new m4.large was co-located with the existing m4.large, and similarly for the m4.xlarge and m4.2xlarge. That is, we find that instances of the same type appear to co-locate, whilst instances of different types do not, even though the types are part of in the same instance family—the M4 class. Finally we did not find any m4.10xlarge types co-locating, this type has 40 vCPUs and from the Amazon documentation we know that 1 vCPU is assigned one hardware hyper thread, and so the m4.10xlarge is running on a host with at least 20 real cores, and 40 hyper-threaded cores. The lack of co-location may point to the m4.10xlarge consuming an entire host. Our attempts to co-locate a smaller instance type with the m4.10xlarge did not succeed.

As this discussion shows, it may be possible to make inferences regarding VM allocation policies by using co-location detection methods, and this could prove a useful tool for researchers in this area wishing to investigate VM allocation at hyper-scale.

We now wish to extend co-location detection via domids to the multi-tenant environment. We begin by collecting domids to see if we can find evidence of potential co-location, and in particular we look for consecutive (or close) domids as instances started on the same host within a short period would have this.

5 Collecting domids in multi tenant instances

Having collected domids in single tenancy instances, and shown how we can use domids to determine co-location, we now turn our attention to the multi-tenant case.

Table 2 Consecutive domids

Seq	Domids
1	563, 564, 565, 566, 567 and 568
2	723, 724, 725, 726, 727, 728 and 729
3	752, 753, 754, 755, 756, 757 and 758

We begin by collecting domids for multi-tenant instances and discuss the degree to which they hint at co-location in this case. Using an Ubuntu precise 12.04 AMI, we can readily launch 20 m1.small instances as a single request in the Region US-East-1, in AZ *us-east-1b*. Each instance gets xenstore-utils installed, and has the exported Xen store file system mounted on /proc/xen. In this setup, it is then possible to obtain an instance's domid, *uuid* and *cpuid*.

In Table 2, we list 20 domids obtained from just such a setup (on 07/10/2014), which are readily organised into three sequences of consecutive domids. For all instances, the CPU model was an E5-2651.

The simplest explanation for these consecutive domids is that the 20 instances are allocated to just three hosts. It may also be possible that these sequences are obtained simply by chance across a large number of hosts that are churning VMs at similar rates, and we discuss this possibility in Sect. 5.

The AZ *us-east-1b* appears homogeneous (just one CPU model) for the account we were using. To simplify concerns further, we instead examine domids in *us-east-1a* as this provides heterogeneous hosts. This helps to improve clarity over co-location since instances with consecutive domids on *different CPU models* are clearly not co-located, and so here the consecutive domids are more likely to indicate co-locating instances—unless, of course, *cpuid* and domid values are spoofed.

We ran 5 requests, with 20 instances per request, on Amazon's spot market for us-east-1b. Of the 100 instances started, 3 were reclaimed and so we have results for just 97 instances. As before, we determine the domid, *uuid* and *cupid.* After this information was obtained, the instances were released. Each request was made at a different time over a 2 days period, from 07/10/2014 to 08/10/2014. In Table 3, below, we list only the sequences with consecutive domids found in each request, together with the instance CPU models—one of E5645, E5507 or E5-2650.

3 out of 5 of the requests evidence consecutive domids with E5645 CPUs, and all three contain at least two such sequences. The most common pattern is of two consecutive domids, and the longest sequence is 7. We note consecutive domids in request 1 of 1671 and 1672, with different CPU models—E5645 and E5-2650 respectively—which clearly cannot be co-located (unless, again, the *cpuid* is spoofed). In request 1, it would appear that 10 of 20 instances are not host separated, in request 3 this is 14, and in request 4 it is 12.

6 Necessary conditions for co-location in multi-tenant instances

In a single tenancy environment, we can express exact conditions for co-location using domids. This is done using the properties of how domids are generated, and as we

Table 3 Domids from multiple time-separated requests

Request, date and time	CPU model	Consecutive domids
1: 07/10/2014: 17:05	E5645	242, 243, 244
	E5645	469, 470
	E5645	1499, 1500
	E5645 + E5-2650	1671, 1672
	E5-2650	2627, 2628, 2629
2: 07/10/2014: 17:58		None
3: 07/10/2014: 21:57	E5645	250, 251, 252, 253, 254, 255,256
	E5507	732, 733
	E5645	1501, 1502
	E5-2650	2630, 2631, 2632
4: 08/10/2014: 10:25	E5645	263, 264, 265, 266
	E5645	501, 502
	E5645	1505, 1506
	E5-2650	2637, 2638, 2639, 2640
5: 08/10/2014: 21:50		None

are in a single tenancy environment we do not have to consider the possible effects non-sibling instances may have on the next available domid. In this section we extend this to multi-tenant environments, taking into account the potential presence of other instances on the host.

Suppose we launch two (multi-tenant, not sole-tenant) instances A and B at the same time, what conditions must be satisfied if the instances are co-located? First, their underlying host must have the same CPU model. Secondly, the values of their domids must be sufficiently close to each other.

For the second condition, we note that we do not require that the domids be consecutive but should be sufficiently close to each other. How close depends upon the amount of domid increasing activity that can occur between the instances being started. In turn, this depends in large part on how many instances a host can have running concurrently on it. Suppose a host supports up to k instances, and that two sibling instances A and B are scheduled onto the host. From our previous work, based on launching close to 10,000 instances on EC2, we estimate that instances started from the same request typically launch within a minute of each other at most, which offers just a minute of opportunity for domid increasing activity on the host between the two sibling instances being launched. We consider two cases at the extremity:

1. The maximum number of instances a host can run is given by '$n > 0$'. The host has $n - 2$ instances already running when A and B are scheduled onto it. Further, suppose all these instances are rebooted in between A and B launching. This would set the domid distance between A and B to at least $n - 2$. Now, from our experiments we estimate that it takes between 60 and 90 s from initiating a reboot of an instance to it becoming available again so that a user can log into it and

reboot again. Therefore the majority of the existing instances could only reboot once in the time period. Note that the 60–90 s is predominately taken up with OS shutting down services, followed by the OS restart, during which time the domain is active. This then leaves little, if any, time for other instances, in addition to the existing $n - 2$ to start.

2. The host had no instances running on it. Between starting A and B, another request was satisfied and this resulted in the launch of $n - 2$ instances. *EC2* states that boot times for instances with EBS root volumes is in the region of 60–90 s, which is in agreement with data we have collected.Further, instance termination can take up to a minute. Therefore there is no time for one of the $n - 2$ instances to have been started and terminated and an additional instance started before B is launched.

From the discussion above, if a host can support n instances, then n is a good bound for the maximum distance between the domid of two sibling instances launching on the host from the same request. We could of course take multiples of n and increase the bound, and the larger we set the bound the less likely sibling instances, launched from the same request on the same host, will have a distance between their domids greater than this. From our empirical data, we would suggest that a bound of $2n$ is highly unlikely to be exceeded, and would perhaps indicate some somewhat pathological behavior on the host.

We can estimate values of n, i.e. the number of instances a host can support, from the CPU model and from the vCPU to CPU core allocation. On EC2 (and indeed on GCE, HP Helion and others) all current generation instance types schedule one vCPU as one hardware hyper thread. In these cases, we can obtain a hyper thread count from the CPU specification, and we assume a dual socket configuration. For previous generation instances, such as the m1.small in Sect. 5, we have to do more work. We have previously shown [18] that m1.small instances on EC2 may be backed by 6 different CPU models, including the AMD 2218 and the Intel Xeon E5-2651—the oldest and newest CPU models respectively. The former is a dual core CPU, so a host with dual socket can have at most 4 cores. The latter, however, has 10 cores per socket and dual socket would have 20 cores. Further, if hyper threading is enabled—common practice on EC2 on CPU models which support it—the core count rises to 40. Finally, the configured ratio of vCPUs to physical cores determines k. As a rule of thumb we will take n to be two times the core count of a CPU, and times again by 2 if the CPU supports hyper threading.

Note that as we are estimating the maximum number of vCPU that may be available for instances, and further that the maximum number of concurrent instances running corresponds to this. In practice, our estimate of n by this method is an over estimate in most cases, since most instance types will have more 1 vCPU. That is, if a host has 40 vCPUs, it may support up to 40 concurrent instances each with 1 vCPU or 20 instances with 2 vCPUs and so on.

In Table 4 below, we list the six models we identified (to the time of writing) as backing m1.small instances, together with a domids range based on the above reasoning:

We state our necessary conditions for sibling co-location as:

1. The instances have the same CPU model

Table 4 CPU model and domid range for siblings

CPU model	Domid range
AMD 2218	8
Intel Xeon E5430	16
Intel Xeon E5507	16
Intel Xeon E5645	48
Intel Xeon E5-2650	64
Intel Xeon E5-2651	80

2. The instances have domids within $2n$ of each other, where n is the number of concurrent instances we have inferred to be supportable by the CPU model

Whilst the two conditions are necessary for co-location, they are *not sufficient*. It is entirely possible that the instances have been allocated to hosts, with the same CPU, whose next available domids are close simply by chance. In this case then the instances can satisfy the conditions but not be co-locating. Indeed, this becomes more likely if the hardware platform and configuration is identical, and if the churn rate of VMs is similar. In fact, we have already seen an example in Table 3, batch 1, of instances with domids of 1671 and 1672, but which had different CPU models—and so were not co-locating.

Whilst emphasis has to date been on instances launched from the same request, we can drop this restriction and consider two instances A and B started at any time. By rebooting them both they would obtain new domids, and we can now consider them as equivalent to newly launched instances with regards to their new domids.

We note that the conditions here give a very quick test for *non co-location*. This allows a user to separate instances between non co-located and *candidates* for being co-located. This fulfills the first two requirements for our co-location testing, as stated in Sect. 1.

We therefore need to address the question of the likelihood that non-co-locating instances have domids near to each other, and we consider this in the next section.

7 Domids and the almost birthday problem

The question of how likely are two instances with close domids to be co-locating is similar to the well known ‘birthday’ and ‘almost birthday’ problems. The birthday problem can be stated like this: How many people do we need in a room in order for there to be a 0.5 chance that at least 2 people will share the same birthday? In this case the answer is 23. As we are interested in near domids our problem is more akin the ‘almost birthday problem’: In a room of 23 people how likely is it to have at least one pair of consecutive birthdays? An analytic solution to this is presented in [19], with the answer 0.89.

Monte Carlo methods can be used to tackle the birthday problems stated above. We can assume that a birthday is equally likely to fall on any day in the year. We then generate random samples, of size 23, drawn from the uniform distribution. For each sample we record a success if there is a matching (or consecutive, depending upon the problem of interest) birthday. The number of successes divided by the number of trials is then the estimate of the probability.

We note that the assumption that birthdays are uniformly distributed is not entirely accurate and that seasonal variations do exist. However, the uniform distribution does provide a good approximation.

Can we apply such methods to estimate the probabilities of instances having consecutive, or near, domids by chance—and not because they are necessarily co-locating? An immediate requirement is a reasonable approximation for the distribution of domids across hosts. In theory, a domid is in the range [1, 65536], however we have so far only observed domids within a restricted range. Further, the domid distribution is likely CPU dependent to some degree. CPUs with more cores, such as the E5-2651, will likely increment domids at a different rate to the E5645, as they can run more instances.

We could assume that the range of domids for hosts with the same CPU model is equally likely to be between the observed minimum and maximum. Applying this to the E5645, that would be between 252 and 20,708. Using a Monte Carlo simulation, we find that 20 non co-located instances, placed on randomly selected hosts with E5645 CPUs, will have at least one pair of consecutive domids with a probability of 0.009. That is, approximately 1 in 100 batches of 20 instances would have at least one pair with consecutive domids.

However, it is not obvious that we can model the problem in a manner similar to the birthday problems. Consider for example, a power failure in one portion of a data centre resulting in a large number of E5645 hosts being rebooted. In this case then, we initially have a large number of E5645 hosts with small domids. Instances allocated to these hosts would have a far greater chance of consecutive, or near, domids than our estimate would imply. Whereas birth dates do not tend to change in such a way.

Indeed, it is not clear that the domid range should be well approximated by any statistical distribution. Further, the VM allocation mechanisms in use, by which instances are assigned to hosts, are not advertised, may well produce domid ranges whereby near domids are more likely, and perhaps considerably so, than our assumptions would allow for. However, developing a model to accurately represent domid distribution across hosts is beyond the scope of this paper, so we do not rely on purely statistical arguments and instead look for further evidence for co-location, which we describe in the next section.

8 Co-location testing in multi-tenant instances

We have already seen that when an instance is rebooted it acquires a new domid. This will be the number of new instances started on the host plus the number of instance reboots. This observation allows us to add an additional condition: suppose, then, that we have two instances on hosts with the same CPU model. If they have identical domids they are not on the same host. Therefore, suppose that the instances' domids are different but within a host's domid range (from Table 4). We denote the lower domid by m, and refer to the instance with this domid by A. We refer to the higher domid by n and the instance with this domid by B. Upon rebooting A, its new domid must, simply, be greater than the domid of B, if not they are on different hosts.

We now state this as a third necessary condition for co-location:

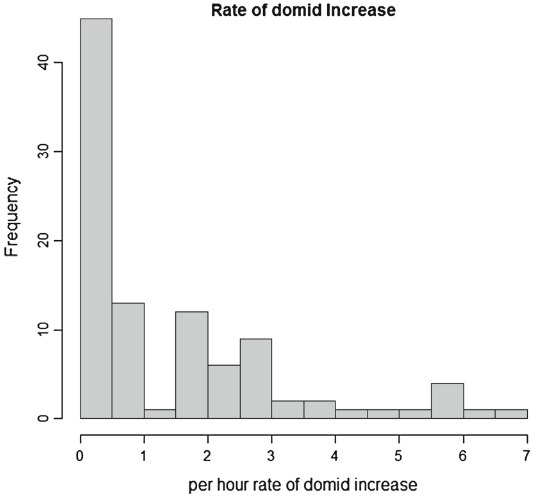
1. A and B are instances with domids (m, n) respectively, where $m < n$. If A and B are co-locating, then upon rebooting A, its new domid, p, must satisfy $p > n$.

Of course, we still do not have a sufficient condition—instances may satisfy the above by chance. However, a user is free to reboot their instances as often as they like. So we can strengthen the condition as follows:

(a) Reboot the instance A, which has domid p, an additional k times. Upon rebooting instance B, it will obtain a new domid, q, that must satisfy $q > p + k$.

Proceeding in this way we can use the actions of A to introduce a minimum distance to the next available domid for B, and vice versa. Again this may be satisfied by chance. It may the case that whilst we were rebooting instance A k times, there were sufficient domid increasing events on the host B is running on, so that the condition is satisfied. However we can introduce a fixed minimum distance which must always be satisfied, and repetition of this test increases confidence that A and B are indeed co-locating and we are not observing the minimum domid distance injection merely be chance.

The question now is: What value should we set k? As noted, if the domid increase on host B is high then we may have a false positive—even with repeated tests. The value of k then should be chosen on the basis of likely domid increase. We determine rate of domid increase we launched 100 m3.medium instances for a period of 3 h and determined the domid for each instance at the start and at the end of the period. From this we can calculate the rate of domid increase on the hosts the instances are running on. Below we present summary statistics as well as a histogram for our results.



We find then that on average we have a per hour rate of domid increase less than 2, and indeed for 50 % of our instances it is less than 1. We note that we would expect

Table 5 Per hour rate of domid increase

Mean	SD	Median	Min	Max
1.56	1.64	0.67	0.34	6.67

our tests based on obtaining domids and reboots to take considerably less than 1 h. Therefore, based on the above, if we set a value of k of 7 (larger than maximum per hour rate of increase) we could perform the co-location test in less than 10 min (Table 5).

Finally, we note that a user may have long running instances, and want to know if newly started instances are co-located with any long running instance. In this case, a long running instance's domid is likely not representative of the current domids available from the host due to requests and reboots in the intervening period. In this case, rebooting the long running instance will update its domid, and bring the domid into range of new instances, allowing for further confirmatory tests to be run.

We now state our test for co-location as follows: two instances, A and B, chosen because they have domids, m and n, such that $m < n$ are likely co-locating if they satisfy the following:

1. Same CPU model
2. Values of domids are in range (by Table 4). That is, $n - m \leq k$ where k is the CPU domid range in Table 4.
3. Upon rebooting instance A, its new domid satisfies $p > n$.

If 3 is satisfied, then we strengthen the condition as follows:

3a. Upon rebooting instance A a further k times, a reboot of B results in a new domid, q satisfying $q > p + k$.

We reiterate that (3a) can be carried out as many times as the user wishes, for any value of k, which should be chosen on the basis of rate of domid as described.

To test this, we use two pairs of instances, the first pair which obtained domids (7635, 7638) respectively, and the second pair (9536, 9538). As the first pair of instances were on E5-2650 hosts (condition 1), and have close domids (condition 2) they are good candidates for co-location. However, upon rebooting the instance with domid 7635, its new domid was 7636, and so cannot be co-locating with the instance with domid 7638 (due to condition 3). For the second pair, again both with CPU model of E5-2650 (condition 1) when rebooting the instance with domid 9536, its new domid was 9539, and so greater than 9538 (condition 3). We rebooted this instance a further 5 times and after the last reboot its domid was 9544. We then rebooted the instance with domid 9538, after which its domid was 9545 (condition 3a). This more strongly suggests co-location, and we note again that a user is of course free to set the domid distance to any value they like by rebooting (we set to 6), and to repeat as many times as they wish.

9 Detecting past locations

In addition to some degree of co-location, we also observe that instances started from later requests appear to be scheduled onto the same hosts as earlier ones. This observation is also based on domids, as we explain now.

In request 1 we obtain instances with domids 1499 and 1500, and both have E5645 CPUs. In request 3 we obtain instances with domids of 1501 and 1502, and in request 4 we have 1505 and 1506—again all E5645. One explanation is that these instances were scheduled onto just one host. As another example, we have the domids 2627, 2628 and 2629 in request 1, followed by 2630, 2631 and 2632 in request 3 and then followed by 2637, 2638, 2639 and 2640 in request 4. All of the instances were running on a host with a E5-2650 CPU, so could again have been scheduled onto just one same host.

In a follow up experiment, we launched 100 instances and found 4 consecutive domids. We terminated these instances, and 5 min later started another 100 instances (5 of which were reclaimed). The domids in the two sets ranged between 759 and 7292. Comparing domids in the first set to the second, we found a remarkable 51 domids in the first set with consecutive domids in the second set, 27 domids in the first set with a ‘plus 2’ in the second, 7 at ‘plus three’ and 1 at ‘plus 4’. The likelihood of our second set of instances being on a completely different set of hosts to the first, but having domids so close to the first set would appear to be small.

Running 3 further requests, again of size 100, we find the same behavior of later instances appearing to be scheduled on to previously used hosts. This is also not just a feature of either on-demand or spot instances, as we observe this for both. Indeed, when running a batch of spot instances after a batch of on-demand, we again observe such behavior, suggesting that requests are being satisfied from the same resource pool.

It is unclear whether this might be a temporal or spatial issue. In the former, it may simply be the case that whilst there is a large amount of available resource, instances started shortly after earlier ones are scheduled back on to previously obtained hosts. In the latter, it may be that a user is restricted to a subset of the available resources. We know that EC2 is vast in scale, with 28 AZs, most of which comprise at least 2 data centres—with the largest AZ having 6—and each data centre houses between 50,000 and 80,000 physical servers [20]. For each user, an AZ identifier, such as us-east-1a, relates to some pool of resources out of which requests are served. It is possible that AZ identifiers may map to a data centre in an AZ, or indeed to some rather smaller subset thereof.

Recycling of resources has the clear potential to impact on a user’s ability to separate co-locating instances. In this case, a user may be interested in the number of attempts needed, and so the cost, to ensure separation. Perhaps more intriguingly, if a user is restricted to a subset of resources then launching a targeted attack against them on EC2 would be much harder—you would only be able to target users that you share the same resource partition with. With sufficient data, it may be possible to answer these questions, and also estimate the size of resource pool available for use. From this, one might also estimate a likely number of people with whom the resource pool is shared, and could use this number to suggest the risk of security and performance issues arising.

Finally, given the well established problem of performance variation due to the heterogeneous [21–23] nature of Public Clouds, there has been interest in so-called 'instance seeking' or 'deploy and ditch' strategies [24,25]. The assumption behind these strategies is that a poorly performing instance can be released and a new, better performing one, found. However, as the performance of an instance is determined by the hosts it is running on, such strategies are rather less likely to produce performance gains in the face of resource recycling.

10 Application to containers

Workload and application packing via container technology such as Docker [26] is receiving increasing attention with companies such as Google deploying all of their workloads with them on their internal systems. The popularity is being driven by the ability to isolate workloads and their dependencies and apply resource controls to them via cgroups without, at least in private systems, the overhead of a hypervisor.

Unsurprisingly, there is interest in using these technologies on the Public Cloud. However as containers do not (currently) provide strict security isolation it is not recommended to use containers in a multi-tenant environment. On the Public Cloud then, container services such as Amazons Container Service and Google Container Engine are built on top of virtual machines which are provisioned from existing Infrastructure Clouds. Issues of performance, availability and redundancy that affect virtual machines in general will then also apply to workloads packaged in Docker and run in virtual machines. With Docker applications co-located on a VM, better separation of VMs could become attractive.

A user can use domid testing with the instances they have provisioned to enforce any desired workload separation. If a user is using workload scheduling services, such as Kubernetes [27] then they should be aware that such services make no distinction between scheduling Docker workloads onto virtual machine or physical machines. It may, however, be possible to extend such services to make them virtual machine aware and enable them to make scheduling decisions which take into account virtual machine co-location. Kubernetes uses a distributed key value store to store meta-data about hosts/virtual machines onto which Docker workloads are scheduled. We would suggest that this seems a suitable place to mark a host as physical or virtual, and in the latter case store the CPU model and domid (when used on Xen based systems).

11 Host separation policy costs

By being able to identify co-locating siblings a user can put in place a host separation policy. Such a policy would specify which sibling instances are allowed to co-locate and which are not. When starting a, potentially large, number of instances, a user can obtain their CPU model and domid with no extra costs. From this information alone, and using the necessary conditions for co-location, as described in Sect. 6, a user can quickly filter the instances into those that are *not* co-locating, and those that may be co-locating, and require further testing for confirmation.

In the case of single tenancy instances, to determine whether or not a pair of instances are co-located simply requires one reboot—that of the instance with the lower domid (as discussed in Sect. 4). Co-location can be determined within the time taken to reboot, approximately 60–90 s, with no extra costs involved.

For multi tenancy instances, we require more instances reboots as we seek to put a minimum distance to the next available domid, and further, we the more we perform our test the greater our confidence that instances are co-locating. There are no direct costs associated with this; however there is a time delay—the time taken to perform the tests. For a user who has two instances that may be co-locating, and whose policy states they should be separated, is now faced with the following choices: either, reboot a sufficiently large number of times be confident of detecting co-location, or start additional instances and in the hope of obtaining the required number that are not co-locating. In the later case there is a cost implication, as each instance on EC2 has a minimum rental period of one hour. In the former case, there is the time delay in detecting co-location to the required confidence, which may or may not be acceptable. This trade off will depend upon the nature of work being done in the instances.

In future, providers may well offer host separation as a service. We note that in order to do so for all users likely requires additional physical resource, which incurs additional cost to the provider, which would be passed on to the user. A comparison of provider costs for host separation compared to a user implementing their own polices can be made.

12 Conclusions and future work

Identifying when sibling instances are co-locating is beneficial to users in a number of situations:

1. Co-located instances may degrade the performance of each other when running compute bound workloads.
2. Co-located instances are all vulnerable to failure, or degradation, of the underlying host.
3. Co-located instances are all vulnerable to other noisy neighbours.
4. Co-located instances imply is a greater exposure to a security compromise on a single host.

Determining co-location is challenging, particularly so on Public Clouds. The approach we have presented in this paper is based on information provided from Xen, which is currently the dominant hypervisor technology used in Public Infrastructure Clouds. Xenstore provides an interface for domains to obtain information such as domids and *uuids*. However, as would be expected, on EC2 the interface is restricted so a domain can only obtain information about itself. But the domid is still very useful for our purposes. In addition to EC2, we have also been able to obtain domids from instances running on GoGrid and Rackspace, both of which use Xen. On a standard Xen system, domids are assigned consecutively when starting domains and are not recycled—except when the range itself cycles. Instances are assigned the next available (new) domid when rebooted. Domids also do not survive host reboots, which resets the next available domid to 1.

These characteristics of domids allow for the formulation of the simple test for co-locating sibling instances as described, based on the same CPU model and close domids (per Table 4 for the various CPU models we have observed backing m1.small instance types). It is still, as we have elaborated, possible that such instances have close domids simply by chance, and indeed we have seen such examples. Simulation methods could be employed to determine the likelihood of this, but assumptions regarding the distribution of domids are required, the validity of which is difficult to establish. Whilst nearness hints at co-location, further evidence is required.

Further evidence is provided by the observation that one instance can restrict the possible range of values for another instance's domid—simply via rebooting itself and so increasing the next available domid value. The second instance, upon a reboot, can then in turn restrict possible domid values for the first instances. This process can be repeated as often as a user chooses, and at the domid distance the user chooses (the reboot value), and therefore each time this is done the probability that this happens by chance decreases. Further, this is not limited to instances started close to each other in time, but can be used when any pair of instances is suspected of co-locating.

We should be clear that whilst passing the tests described in Sect. 8 decreases the likelihood that the instances are not co-locating, increasingly so when repeated, we cannot say for certain that the instances are co-locating. From a pragmatic point of view, a user must balance the risk of having co-located instances with the cost of (determining and then ensuring) separation.

Determining such costs may be difficult as there appears to be a degree of recycling of resources, as described in Sect. 6. This also has an immediate and significant consequence for the probability of success in carrying out a targeted side channel attack on a Public Cloud. Indeed, from our work here, we find the chance of intentionally co-locating with sibling instances to be fairly small. Co-locating with any intended target would therefore be more unlikely still, if it is indeed possible at all. We also note the impacts for so-called 'performance seekers', whereby a user releases back underperforming instances in the hope of acquiring better performing new instances. A user may simply be paying to obtain resources they have already had.

In summary, our test is simple to implement and works on Linux, Windows and FreeBSD Operating Systems, with the appropriate Xenstore client tool, and satisfies the following criteria, as stated in Sect. 1:

1. Simple to implement
2. Can quickly determine non co-location
3. Can determine co-location with a high degree of assurance
4. Scales with the size of the deployment being tested

In future we intend to use our co-location technique to measure performance correlation for co-locating instances on EC2. This will allow use to quantify performance risk on EC2, with the aim of pricing SLAs which include QoS terms for performance.

Further, we will investigate KVM and VMware hypervisors, with the aim of developing a similar technique as to one described here.

References

1. Armbrust M et al (2009) Above the clouds: a Berkeley view of cloud computing. Technical report EECS-2008-28, EECS Department, University of California, Berkeley
2. Amazon Web Services (2015) What is cloud computing. http://aws.amazon.com/what-is-cloud-computing/. Accessed 16 July 2015
3. Linux Foundation (2013) The Xen Project. http://www.xenproject.org/. Accessed 16 July 2015
4. Zhang X et al (2013) CPI^2: CPU performance isolation for shared compute clusters, EuroSys 13. In: Proceedings of the 8th ACM European Conference on Computer Systems
5. Intel (2014) http://www.intel.com/content/dam/www/public/use/en/documents/white-papers/intel-saa-performance-white-paper.pdf. Accessed 16 July 2015
6. Takemura C, Crawford L (2009) The book of Xen. No Starch Press, San Francisco, CA
7. Linux Containers https://linuxcontainers.org/. Accessed 16 July 2015
8. Oracle (2105) Oracle solaris zones. http://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm
9. FreeBSD, Jails https://www.freebsd.org/doc/handbook/jails.html. Accessed 16 July 2015
10. Menage P, CGROUPS https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt. Accessed 16 July 2015
11. Xen Project Xen security announcement. http://wiki.xenproject.org/security_announcements. Accessed 16 July 2015
12. Zhang Y et al (2012) Cross-VM side channels and their use to extract private keys. In: Proceedings of the 2012 ACM conference on computer and communications security, pp 305–316
13. Lui F et al (2014) Mitigating cross-VM side channel attacks on multiple tenants cloud platform. J Comput 9(4):1005–1013
14. Ristenpart T et al (2010) Hey you get off my cloud. In: Proceedings of the 16th ACM conference on computer and communications security, pp 199–212
15. Bates A et al (2014) On detecting co-resident cloud Instances using network flow watermarking techniques. Int J Inf Secur 13(2):171–189
16. Zhang Y et al (2011) Home alone: co residency detection in the cloud via side channel analysis. In: Proceedings of 2011 IEEE symposium on security and privacy, pp 313–328
17. Xenstore (2014) http://wiki.xen.org/wiki/XenStoreReference. Accessed 16 July 2015
18. O'Loughlin J, Gillam L (2014) Performance evaluation for cost-efficient public infrastructure cloud use. In: Altmann J et al(ed) GECON 2014, LNCS 8914, pp 133–145
19. Dasgupta A (2004) The matching, birthday and strong birthday problem: a contemporary review. J Stat Plan Inference 130:377–389
20. Vanian J (2014) https://gigaom.com/2014/11/12/amazon-details-how-it-does-networking-in-its-data-centers/. Accessed 16 July 2015
21. Osterman S et al (2010) A performance analysis of EC2 cloud computing services for scientific computing. Cloud computing. Lecture notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, vol 34, pp 115–131
22. Yelick K et al (2011) The Magellan report on cloud computing for science. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Magellan_Final_Report.pdf
23. Phillips S et al (2011) Snow white clouds and the seven dwarfs. In: Proceedings of the IEEE international conference and workshops on cloud computing technology and science (Nov. 2011), pp 738–745
24. Ou Z et al (2012) Exploiting hardware heterogeneity within the same instance type of Amazon EC2. In: Presented at 4th USENIX workshop on hot topics in cloud computing, Boston, MA

25. Farley B et al (2012) More for your money: exploiting performance heterogeneity in public clouds. In: Proceedings of the 3rd ACM symposium on cloud computing. Article no. 20
26. Docker (2015) https://www.docker.com/. Accessed 16 July 2015
27. Google (2015) Kubernetes. http://kubernetes.io/. Accessed 16 July 2015