Addressing Issues of Cloud Resilience, Security and Performance through Simple Detection of Co-locating Sibling Virtual Machine Instances

John O'Loughlin and Lee Gillam
Department of Computing, University of Surrey, Guildford, GU2 7XH, Surrey, U.K.

Keywords: Virtualisation, Xen, Cloud Computing, Co-location, Security, Performance.

Abstract: Most current Infrastructure Clouds are built on shared tenancy architectures, with resources shared amongst large numbers of customers. However, multi tenancy can lead to performance issues (so-called "noisy neighbours") and also brings potential for serious security breaches such as hypervisor breakouts. Consequently, there has been a focus in the literature on identifying co-locating instances that are being affected by noisy neighbours or suggesting that such instances are vulnerable to attack. However, there is limited evidence of any such attacks in the wild. More beneficially, knowing that there is co-location amongst your own Virtual Machine instances (siblings) can help to avoid being your own worst enemy: avoiding your instances acting as your own noisy neighbours, building resilience through ensuring host-based redundancy, and/or reducing exposure to a single compromised host. In this paper, we propose and demonstrate a test to detect co-locating sibling instances on Xen-based Clouds, as could help address such needs, and evaluate its efficacy on Amazon's EC2.

1 INTRODUCTION

Infrastructure Clouds offer compute resources for rent on-demand, typically on a per hour basis (Armbrust et al, 2009). 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. Infrastructure Clouds use virtualisation technologies such as Xen and KVM to offer physical servers as (often, multiple) virtual servers. Customers can rapidly acquire virtual servers, use them for as long as required, then release them back to the provider when no longer needed, with the equivalent resource then available for use by other customers.

At any given time, a physical server in a Public Cloud could be running virtual servers, also referred to as instances, for a number of different users (customers). From the user's perspective, shared tenancy raises various concerns, of which security and performance are key. For security, one particular concern is hypervisor breakouts, where hypervisor security can be compromised and a resulting privilege escalation can be used to obtain data from other customers' instances. For performance, one such concern is noisy neighbours, where performance degradation occurs in one instance due to the (legitimate and not necessarily malicious) resource consuming actions of another.

In such cases, the concern tends to focus on the security or performance impact from *other* users. Consequently, research has tended to be focused on identifying vulnerable instances, or hiding from potential attackers. However, identifying co-locating instances may be of even more use for the majority of users with respect to their *own* instances. We will refer to instances started by the same user, irrespective of when, as *sibling instances* in the remainder of this paper.

Sibling instances that are co-located on the same host may be undesirable for 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.
3. They are all vulnerable to other noisy neighbours.
4. There is a greater exposure to a security compromise on a single host.

Determining co-location is difficult, and to date, no simple methods have been proposed that would reliably allow for such detection. This paper aims to

O'Loughlin J. and Gillam L..
Addressing Issues of Cloud Resilience, Security and Performance through Simple Detection of Co-locating Sibling Virtual Machine Instances.
DOI: 10.5220/0005485000600067
In *Proceedings of the 5th International Conference on Cloud Computing and Services Science* (CLOSER-2015), pages 60-67
ISBN: 978-989-758-104-5

address this problem by exploring one kind of trace left by virtualization on Xen based Clouds - domain ids (*domids*). The rest of this paper is structured as follows: In section 2 we review relevant related work to offer background to the problem; in section 3, we discuss the Xen hypervisor and the generation of domain ids, and in section 4 we discuss the results of domids collected from a small sample (100) of virtual servers in the Amazon Cloud, and use this as a basis for tests for co-location in section 5. In section 6 we use domids collected from further samples to demonstrate likely recycling of resources. Finally, in section 7 we present our conclusions, and future directions of this work.

2 RELATED WORK

The ability for one instance to degrade the performance of other co-located instances is well known, and is referred to as noisy neighbours. Intel identifies the primary cause of the problem as the sharing of resources, such as the L2 cache, which cannot be partitioned (Intel, 2014); that is, there is no mechanism to limit how much of the resource an instance may consume. Consequently, it is possible for instances to use such resources disproportionately, to the detriment others.

The standard metric for compute performance is execution time. Identifying if a running task is likely to suffer from poor performance i.e. need increased execution time, is difficult. On their production clusters, Google detects likely poor performance by repeatedly measuring a task's cycles per instruction (CPI), i.e. the number of cycles required to execute an instruction, and comparing with the known CPI distribution (Zhang et al, 2013). If more outliers (defined as more than 2 standard deviations from the mean) are detected than expected, then performance of the task is likely to be poor. The protagonist, i.e. the noisy neighbour, is identified by correlating other instances' CPU usages with the increase in CPI outliers for the victim.

On a Public Cloud, information about when an instance is scheduled for CPU time by the hypervisor is only available to the provider, and is not subsequently made available to customers. As such, it is not possible to precisely state when an instance is running or not. A coarser approach would be to attempt to correlate instance performance using compute benchmarks. Such an approach would likely require a minimum number of co-located instances on a given host in order to be successful, and so this already requires co-location to be knowable, and there is the potential to miss a small degree of co-location per host.

The problem of extracting information between co-locating virtual machines has been investigated by a number of authors. In (Zhang et al, 2012) the sharing of an L2 cache between VMs was shown to be 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 VMs onto different cores during its lifetime, as would be encountered in a Cloud environment, needs to be overcome. However, the demonstration was on a standalone Xen system rather than on a Public Cloud.

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 (Lui, Ren and Bai, 2014), and indeed, are already being integrated into Xen. Whilst such work mitigates fine grained attacks, denial of service attacks, which seek to obtain a large share of the L2 cache, are considered viable.

This has led to work on targeted attacks in the Cloud, whereby an attacker seeks to co-locate with a specific target. This requires methods for determining co-location with the target before the attack can be launched. In (Ristenpart, Tromer, Shacham and Savage, 2010) a number of network based probes have been proposed, for example ping trip time and common IP address of dom0. In order 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.

However, as the authors state, a provider can easily obfuscate network based probes and this already appears to the case. From our experiments we can confirm this. Whilst access times to shared drives may potentially be used for detecting co-locating siblings, there are a number of issues not discussed that need require further investigation. Perhaps most importantly, is the widely reported variation in disk read/write timings on EC2 (Armbrust et al, 2009), which clearly needs to be accounted for in any test that proposes to use them.

In (Bates et al, 2013) watermarking of network flows is proposed and demonstrated on a variety of stand- alone virtual systems. However, as the authors state, there a number of defences against watermarking in place in Public Clouds, and in particular on EC2, which prevented them from successfully using the tests.

In (Zhang, Juels, Oprea and Reiter, 2011) a cache avoidance strategy is used so that instances

can co-ordinate their use (or avoidance) of the L2 cache and measure resulting cache use. This, then, is a basis for detecting co-locating siblings. The method is applicable to Xen-based Clouds but requires modification of how the guest OS kernel manages memory, and has a performance overhead when cache use is turned off. Such an approach is technically challenging, as it involves kernel changes, and this is likely beyond the capabilities of most Cloud users.

In summary, neither simple network probes nor network flows watermarking co-location tests work on EC2 due to measures in place, whilst cache avoidance technically challenging. There is a need for simple methods then.

3 THE XEN HYPERVISOR

The Xen system (Xen, no date) 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*.

Each domain is given two identifiers, a *domid* and a UUID. The UUID is 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 it was launched in.

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 is monotonically increasing - Xen assigns the next available domid. This means that instances that are started one after the other will obtain consecutive *domids*. On EC2, therefore, we would expect co-locating instances, started at the same time, to have consecutive *domids* – or, with other requests also being satisfied, being quite close to each other.

Xen *domids* have a rather interesting property, and one which will be crucial to us later: an instance can increase its own *domid* simply by rebooting. An instance's new *domid* will be its old *domid* plus the number of instances that have started on the same host since it was last rebooted, plus the number of reboots that have occurred. Domids do not, however, seem to survive an underlying host reboot, and in this case the next available *domid* is reset to 1.

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 (Xenstore, 2014) 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*.

In the next section we present the results of *domids* collected on EC2 via Xenstore from some 120 instances.

4 COLLECTING DOMIDS

We can initially collect domids from instances launched on EC2, and examine the extent to which these hint at co-location. 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 1, below, 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.

Table 1: 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

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 section 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 day period, from 07/10/2014 to 08/10/2014. In Table 2, 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.

Table 2: Domids from Multiple Time-Separated Requests.

Request, Date & Time	Consecutive Domids and CPU Model
1 07/10/2014 17:05	242,243,244 – E6545 469,470 – E5645 1499,1500 – E5645 *1671, 1672 – E5645 + E5-2650* 2627, 2628, 2629 – E5-2650
2 07/10/2014 17:58	None
3 07/10/2014 21:57	250, 251, 252, 253, 254, 255 ,256 – E5645 732, 733 – E5507 1501, 1502 – E5645 2630, 2631, 2632 – E5-2650
4 08/10/2014 10:25	263, 264, 265, 266 – E5645 501,502 – E5645 1505, 1506 – E5645 2637, 2638, 2639, 2640 – E5-2650
5 08/10/2014 21:50	None

3 out of 5 of the requests evidence consecutive *domids* with E5645 CPUs, and all three contain at least 2 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.

5 CO-LOCATION TEST

Based on the discussion and results in section 4 we can state that for any pair of instances the following initial conditions must be satisfied if the instances are more likely to be co-located:

1. Same CPU model
2. Values of *domids* are sufficiently close to each other

For the second condition, we do not require that the *domids* be consecutive but should be sufficiently close to each other. In order to understand why consider the following: two sibling instances are scheduled onto the same host, but in between them being launched an existing instance is rebooted. In this case then, they will not have consecutive domids but the *domids* will differ by (at least) 2. We discuss how close is 'sufficiently close' later in this section.

Whilst the two conditions listed above are necessary for co-location, they are not sufficient. It is entirely possible that the instances have been allocated to hosts whose next available domids were within the *domid* distance simply by chance. Indeed, this becomes more likely if the hardware platform and configuration is identical, and if the churn rate of VMs is the same. In fact, we have already seen an example in batch 1 of instances with *domids* of 1671 and 1672 that had different CPU models.

For the second condition, closeness of the domids depends in large part on how many instances a host has been configured to support. If a host supports k instances, then any instances started within a short period of time on the host would likely have their domids within k of each other. We cannot state this for certain, since it's possible that within that period (1) a number of instances on the host were rebooted (2) a number of instances were terminated, and a number more were started.

We also cannot state the value of k for a host with certainty, since it depends on its CPU model, the CPU configuration, how many sockets the host has, and the degree of over commitment. As an example, we have previously shown (blind ref, no

date) 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 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 (as is common practice on EC2), the core count rises to 40. Finally, the configured ratio of vCPUs to physical cores determines k. As EC2 does not advertise socket count, and only specifies vCPU to cores for some instance type, as a rule of thumb we will take 'close' to be 2 times the core count of a CPU, and times again by 2 if the CPU supports hyper threading.

In table 3 below we list the 6 models we have identified to date as backing m1.small instances together with a domids closeness range based on the above reasoning:

Table 3: CPU model and Domid Range.

CPU Model	Domid Range (m1.small only)
AMD 2218	4
Intel Xeon E5430	8
Intel Xeon E5507	8
Intel Xeon E5645	24
Intel Xeon E5-2650	32
Intel Xeon E5-2651	40

We are naturally led to the question of the likelihood that non-co-locating instances have *domids* near to each other. This question 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 (Dasgupta, 2004), 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 the 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 20708. 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 then 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, which are not advertised, may well produce *domid* ranges whereby near *domids* are more likely, and perhaps considerably so, than our assumptions would allow for. As such, 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 now.

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 both on hosts with the same CPU model. If they have identical domids they are not on the same host. Suppose that the instances' *domids* are different and

within a host's *domid* range (from Table 3). 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.

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

3. A and B are instances with *domids* (m,n) respectively, where m < n. If A and B are co-locating, then upon rebooting B, 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 the like. So we can strengthen the condition as follows:

a. Reboot the instance A, which has *domid* p, k times. When rebooted, the instance with domid n will obtain a new *domid*, q, that must satisfy q > p + k.

Whilst again this may be satisfied by chance, further repetition should lead to greater confidence as co-locating instances will satisfy these conditions

To test this, we used 2 pairs of instances, the first pair with 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 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.

To now, we only considered instances started within a short space of time of each other. 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 3). That is, n –m <= k where k is the CPU domid range in Table 3.
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.

6 RECYCLED RESOURCES

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 minutes 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 to 80,000 physical servers (Vanian, 2014). 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 (Osterman et al, 2010, Iosup, Nezih and Dick, 2011) nature of Public Clouds, there has been interest in so-called 'instance seeking' or 'deploy and ditch' strategies (Farley et al, 2012, Zhuang, Liu, Ou and Arberer, 2013). 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.

7 CONCLUSIONS

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 simple 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. 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 3 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 section 5 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 section 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. Future work is largely aimed at further exploration and confirmation of the ideas discussed in this paper. In particular, we would like to be able to identify behaviours of instances that can be detected by others as would confirm co-location, without incurring the effort involved with rewriting (for Linux) kernel memory management features to spot avoidance of shared cache use, and further ensuring that any such observation are not due to chance.

REFERENCES

Armbrust, M. et al. (2009) "Above the clouds: a Berkely view of cloud computing". Technical Report EECS-2008-28, EECS Department, University of California, Berkeley.

Intel, (2014) [Online]. Available at: www.intel.com/content/dam/www/public/use/en/documents/white-papers/intel-saa-performance-white-paper.pdf. [Accessed on 02/01/2015]

Zhang, X. et al. (2013) CPI^2: CPU performance isolation for shared compute clusters, Proc of EuroSys 2013, pp 379-391.

Zhang, Y. et al. (2012) Cross-VM Side Channels and their use to Extract Private Keys, Proc of the 2012 ACM Conference on Computer and communications Security, pp305-316.

Ristenpart, T. Tromer, E. Shacham, H. Savage, S. (2010) Hey you get off my Cloud, Proc of the 16th ACM Conference on Computer and communications Security, pp199-212.

Bates, A. et al (2013) On Detecting Co-resident Cloud Instances using Network Flow Watermarking Techniques, International Journal of Information Security, Vol 13, Issue 2, pp 171-189.

Lui, F. Ren, L. Bai, H. (2014) Mitigating Cross-VM Side Channel Attacks on Multiple Tenants Cloud Platform, Journal of Computers, Vol 9, No 4, pp1005-1013.

Zhang, Y. Juels, A. Oprea, A. Reiter, M.K. (2011) Home Alone: Co residency detection in the cloud via side channel analysis, Proc 2011 IEEE Symposium on Security and Privacy, pp313-328.

Xen, (no date) [Online]. Available at: www.xenproject.org [Accessed: 08/02/2015].

Xenstore, (2014) [Online]. Available at: http://wiki.xen.org/wiki/XenStoreReference [Accessed: 08/02/2015].

Blind Ref, no date:

Dasgupta, A. (2004) The Matching, Birthday and Strong Birthday Problem: A Contemporary Review, Journal of Statistical Planning and Inference 130, pp377-389, 2004.

Vanian, J., 2014. [Online]. Available at: https://gigaom.com/2014/11/12/amazon-details-how-it-does-networking-in-its-data-centers/ [Accessed: 08/02/2015].

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, pp115-131.

Iosup, A. Nezih, Y. and Dick, E. (2011) On the performance variability of production cloud services. In Cluster, Cloud and Grid Computing (CCGrid), 2011.

Farley, B. et al. (2012) "More for your money: exploiting performance heterogeneity in Public Clouds", in Proc. of the Third ACM Symposium on Cloud Computing, article no. 20.

Zhuang, H. Liu, X. Ou, Z. Arberer, A. (2013) "Impact of Instance Seeking Strategies on Resource Allocation in Cloud Data Centres", in Proc. Of the IEEE Sixth International Conference on Cloud Computing, pp27-34.