A Comprehensive Study of Co-residence Threat in Multi-tenant Public PaaS Clouds

Weijuan Zhang[1,2,3,4], Xiaoqi Jia[1,2,3,4], Chang Wang[2,3,4], Shengzhi Zhang[5], Qingjia Huang[2,3,4](✉), Mingsheng Wang[1,2], and Peng Liu[6]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China
{zhangweijuan,jiaxiaoqi,wangmingsheng}@iie.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Key Laboratory of Network Assessment Technology, IIE, CAS, Beijing, China
{wangchang,huangqingjia}@iie.ac.cn
[4] Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China
[5] School of Computing, Florida Institute of Technology, Melbourne, USA
zhangs@fit.edu
[6] College of Information Sciences and Technology, The Pennsylvania State University, State College, USA
pliu@ist.psu.edu

Abstract. Public Platform-as-a-Service (PaaS) clouds are always multi-tenant. Applications from different tenants may reside on the same physical machine, which introduces the risk of sharing physical resources with a potentially malicious application. This gives the malicious application the chance to extract secret information of other tenants via side-channels. Though large numbers of researchers focus on the information extraction, there are few studies on the co-residence threat in public clouds, especially PaaS clouds. In this paper, we in detail studied the co-residence threat of public PaaS clouds. Firstly, we investigate the characteristics of different PaaS clouds and implement a memory bus based covert-channel detection method that works for various PaaS cloud platforms. Secondly, we study three popular PaaS clouds Amazon Elastic Beanstalk, IBM Bluemix and OpenShift, to identify the co-residence threat in their placement policies. We evaluate several placement variables (e.g., application type, number of the instances, time launched, data center region, etc.) to study their influence on achieving co-residence. The results show that all the three PaaS clouds are vulnerable to the co-residence threat and the application type plays an important role in achieving co-residence on container-based PaaS clouds. At last, we present an efficient launch strategy to achieve co-residence with the victim on public PaaS clouds.

Keywords: PaaS cloud · Co-resident · Memory bus · Co-residence threat · Multi-tenant

1 Introduction

Cloud computing develops rapidly in recent years and public PaaS cloud plays an important role in the cloud service market. Report from Synergy Research Group [1] shows that across six key cloud services and infrastructure market segments, operator and vendor revenues of 2015 had grown by 28% on an annualized basis. The report also says public IaaS (Infrastructure-as-a-Service)/PaaS services had the highest growth rate of 51%, nearly twice the average level. The global market for PaaS is projected to reach US $7.5 billion by 2020 [2]. PaaS cloud provides the environment to rapidly develop and deploy applications for the developers. It saves the developers' effort to build up the complicated environment each time and bypasses the maintenance of the underlying infrastructure and services. The public PaaS clouds are usually multi-tenant due to the resource consolidation needs of cloud service providers. PaaS cloud often leverages OS-based techniques such as process protect mechanisms or Linux containers (LXC) to isolate tenants, which is a light-weighted virtualization and takes less resources compared to hypervisor-based techniques common in IaaS clouds.

However, multi-tenancy in public clouds enables co-resident attacks [3]. If an attacker has successfully launched an instance[1] co-resident with the victim, i.e., on the same physical machine, the attacker can then implement attacks to break the logical isolation between tenants and extract secret information from the victim. One of the most notable attacks is the side-channel attack that breaks the virtualization isolation boundary by actively monitoring shared resource usage, e.g., utilizing performance degradation [4,5] to influence the victims, or using Last-Level-Caches (LLCs), local storage disks or memory bus [3,6–10] to obtain useful information.

To successfully implement co-resident attack, there are two main steps: the first step is to achieve co-residence with the victim, which includes a launch strategy (to follow some policies when the attacker creates instances) together with co-residence detection (to detect whether the launched instances have achieved co-residence with the victim); the second step is the information extraction. Existing researches focus on the second step of the attack, that is how to exploit the shared resources to steal other tenants' secret information, such as private key [11,12] or password reset link [10]. Recently, researchers have begun to study the effort made by the adversary to attain co-residence with victim instances. Varadarajan et al. [8] investigated the placement vulnerabilities of three IaaS clouds (Amazon EC2, GCE and Microsoft Azure) and quantitatively evaluated their susceptibility to co-location attacks. Zhang et al. [9] gave a measurement study on the co-residence threat inside Amazon EC2.

For the multi-tenant public PaaS clouds however, little has been done to investigate the potential co-residence issue. Although Varadarajan et al. [8] briefly discussed the co-residence problem of the PaaS cloud Heroku in their

[1] In IaaS cloud, "instance" typically refers to an instantiated VM. While in this paper, it refers to a service unit provided to the tenants by the PaaS cloud providers, which is usually an application development/runtime environment.

work, no proof-of-concept prototype has been proposed. Zhang et al. [10] simply tested the PaaS cloud DotCloud and OpenShift to show it is possible to accomplish co-residence but did not give further analysis on co-residence threat, since the paper mainly focused on the information extraction step of the co-resident attack. To the best of our knowledge, this is the first study to systematically assess the co-residence threat of multi-tenant public PaaS clouds.

Since the instance isolation mechanisms (e.g., container-based or application-based) in PaaS clouds can be different from the ones (VM-based) in IaaS clouds, we believe that the findings about the co-residence threat of IaaS clouds cannot be simply "borrowed" to summarize the characteristics of PaaS specific co-residence problem. Also, the placement variables that may influence co-residence in PaaS clouds are not completely the same with IaaS clouds. For example, the application type, which is a new feature and may play a key role in co-residence analysis does not exist in VM-based IaaS clouds. What's more, due to the relative small size of the service unit, the co-residence threats in PaaS clouds cannot be the same with the ones in IaaS clouds. All in all, we believe the PaaS specific co-residence threats have unique characteristics yet to be uncovered. This work seeks to provide new understanding about these unique characteristics.

In this paper, we study the co-residence threat of three popular public PaaS clouds Amazon Elastic Beanstalk [13], IBM Bluemix [14] and OpenShift [15]. A memory bus contention based covert channel is implemented to detect co-residence. The main contributions of our paper are: (1) We investigate the isolation mechanism of three popular PaaS clouds and identify the co-residence threat in their placement policies. We find that, for the container-based PaaS clouds, the application type (e.g., Python or Node.js) plays an important role in achieving co-residence. (2) We test several other placement variables, such as the number of the instances, time launched and data center region, to study their influence on achieving co-residence in PaaS clouds. (3) According to the experimental results, we propose a launch strategy to achieve co-residence with the victim using least effort.

The remaining of the paper is organized as follows. Section 2 presents the background and the problem statement. Section 3 proposes the co-residence detection technique used in our tests as well as the experimental methodology. In Sect. 4 we discuss the experimental results. Section 5 describes the related work and Sect. 6 concludes the paper.

2 Problem Statement

2.1 Public PaaS Clouds

PaaS is a virtualization based cloud that hosts numerous customer programs in the same machine simultaneously to reduce the overall costs. Since public PaaS clouds are usually multi-tenant, isolation between tenants is essential for the security. Two of the common isolation mechanisms are VM-based isolation and container-based isolation used in a variety of PaaS systems. Some PaaS clouds give each customer a separate IaaS VM instance, e.g., the Amazon Elastic

Beanstalk, thereby leveraging the isolation offered by modern virtualization. However, some PaaS clouds utilize the container to isolate different instances. A container is always a group of processes that are isolated from other groups via distinct kernel namespaces and resource allocation quotas (so-called control groups or cgroups). A popular open-source project, Docker [16], which has been adopted by several PaaS offerings (e.g., OpenShift, IBM Bluemix, etc.), is built on top of facilities provided by the Linux kernel and does not require a complete operating system (OS).

In the PaaS service model, cloud provider offers the customers various execution environments (e.g., PHP, Ruby, Node.js, Java, etc.). The customers can upload the applications' executables or source code to the environment, which is deployed in a provider-managed OS. This OS may run on a physical machine or within a guest VM on a public IaaS platform such as Amazon EC2. Before uploading an application, the customer should first apply for a corresponding execution environment. For example, if the customer wants to deploy a Python application to the PaaS clouds, she should first apply for a Python instance on the cloud and then push her source code into the instance. The host OS manages the source usage of all the instances deployed on it.

2.2 Motivation

There are mainly three reasons motivating us to study the co-residence threat of PaaS clouds: (1) Due to the different cloud architectures, the placement policies of the container-based PaaS cloud may be different from the IaaS clouds. We have a key observation that containers of the same type usually boot from the same image while for VMs, each VM must have a separate image. This means that application types could play a critical role in achieving co-residence. (2) The co-residency security problems of the PaaS cloud are more serious than the IaaS cloud. The service unit (e.g., a container or simply an application) of the PaaS cloud is much smaller than that of the IaaS cloud in most cases, so there are more chances for instances in the PaaS cloud to achieve co-residence. (3) The weak isolation mechanisms between PaaS cloud instances introduce more security problems. For example, in a docker container, the attacker is able to gain information of the entire system, such as modules, interrupts, memory usage and etc. These information can be used as logical side-channels for co-residency detection. Even worse, two containers of the same application type may boot from the same image, which gives the attacker the chance to launch the flush-reload attacks [10,11]. In summary, it is necessary to study the co-residence threat of the PaaS clouds independently rather than simply follow the previous research results of the IaaS clouds.

The placement policies of the cloud determine how hard it is to achieve co-residence. In this paper, we plan to study the co-residence threats residing in the placement policies of PaaS clouds from the following aspects: (1) How much effort is needed to achieve co-residence with a single target or a set of targets in both container-based and VM-based PaaS clouds? Is it cheap or expensive? (2) Does application type really play a critical role in container-based PaaS clouds' co-residence threat analysis? (3) How do the control knobs (number of the instances, launch time, etc.) identified by IaaS co-residence threat analysis influence the PaaS cloud? (4) Is there any chance for the attacker to achieve co-residence with the victim using less effort?

2.3 Threat Model

To achieve co-residence with the victim there are two steps: a launch strategy to create attacker's instances and a co-residence detection. We do not consider the following information extraction attack after attaining co-residence. The focus of our work is to search for the launch strategies that an adversary can follow to increase the chance of co-residency with the victim instances. We assume the victim instances provide external service interfaces to the customers and the attacker has normal right to use the public PaaS clouds just like any other regular customers. Also, we assume the cloud providers and the cloud platforms are trusted.

3 Experimental Methodology

3.1 Co-residence Detection

Memory Bus Contention Based Co-residence Detection. We adopt the memory bus contention technique [7] to detect co-residence in this paper. The contention of the memory bus is used as a covert channel. If one of the instance locks the memory bus regularly, it slows down other instances of using the bus, such as fetching data from the DRAM. Processors always lock the memory bus through atomic memory operations. However, modern x86 processors support atomic memory operations and maintain their atomicity using cache coherence protocols, which may not need to lock the memory bus. But when an atomic memory operation extends across two cache-lines, the x86 processor will lock the memory bus [17]. We utilize this feature to implement the detection to ensure the detection works on different CPU architectures.

In our implementation, the memory bus covert channel is between a *lock process* and a *probe process*. The two processes run in separate instances. The *lock process* creates a memory buffer and uses pointer arithmetic to force atomic operations on unaligned memory addresses, which will cause atomic operations across two cache-lines regularly. This indirectly locks the memory bus even on all modern processor architectures [7]. The *probe process* creates a memory buffer, accesses it frequently, and measures the time taken to access the memory. Before accessing the buffer, it first flushes the memory using the *clflush*[2] instruction. The *clflush* instruction evicts the specific memory line from all the cache hierarchy, including the L1, L2 and the shared LLC. This ensures the following probe operation will hit the memory and use the memory bus. Thus, if the memory

[2] The clflush instruction takes a virtual address as the operand and will flush all cachelines with the corresponding physical address out of the entire cache hierarchy.

bus is locked, the *probe process* will take longer time to finish probing the buffer memory. Otherwise, the time is shorter. We combine the memory bus contention and the *clflush* instruction to make the detection more accurate. It works even when the cache size of the machine is unknown.

Threshold. We use a *threshold* to determine when the change in the *probe process* performance indicates co-residency. In each test, we run the *probe process* in one instance and keep the *lock process* idle at first. The performance measured by this run is the *baseline* performance without contention. Then the *probe process* and the *lock process* are run together. We test the public PaaS cloud Amazon Elastic Beanstalk, IBM Bluemix, OpenShift as well as a local machine. The configurations of the physical machines are shown in Table 1 and the test results are shown in Table 2.

In order to measure the effectiveness of the memory bus covert channel we run tests in our local machine. The result shows that the blocking of the memory bus can significantly slowdown the *probe process* of accessing the memory. The performance degradation is as high as 5.4x (Table 2). The local hardware architecture has multi sockets (Table 1). We didn't pin the process to a particular CPU or core. We run as many as 100 samples in the test to let the processes have the chance to run on the same socket or on different sockets. The results do not demonstrate obvious difference. That means the detection method works even when the co-resident instances are running on cores on different sockets, which is also concluded by Varadarajan et al. [8].

Also, across these hardware configurations (Table 1) on the public clouds, we repeat the test for 100 times and find no obvious difference either. We observed a performance degradation of at least 3.2x (Table 2) compared to not running memory locking process (i.e., a baseline). The following tests in this paper are started with a conservative threshold of 4x for Amazon Elastic Beanstalk, 3.5x for Bluemix and 2.5x for OpenShift to minimize false positives.

Table 1. Machine configurations.

Cloud provider	Machine architecture	Clock (GHz)	LLC (MB)	Cores/CPU	Socket
Local machine	Intel xeon E5-4610	2.40	15	6	2
Elastic beanstalk	Intel xeon E5-2670 v2	2.50	25	10	2
Bluemix	Intel xeon E5-2690 v3	2.60	30	12	2
OpenShift	Intel xeon E5-2670 v2	2.50	25	10	2

Reducing Noise. The sources of noise come from the neighboring instances of the attacker or victim. Any noise could affect the performance of the *probe process* with and without the block signal and result in misdetection. To reduce the noise, we switch between with and without the block signal in each test and compare the difference to determine co-residency. Also, we take 20 samples of each measurement and only when the time difference is stable all the time, the instances can be detected as co-resident.

Table 2. Memory probing tests with/without the block process. *Times* represent the ratio of co-residency time to the baseline. The time unit is 10^7 CPU cycles.

Cloud provider	Isolation	Baseline	Co-resident	Times
Local machine	Process	112	610	5.45
Elastic beanstalk	Xen VM	109	556	5.46
Bluemix	Docker container	121	516	4.26
OpenShift	Docker container	115	372	3.23

3.2 Experimental Design

The success of the co-residence attack refers to the fact that there is at least one attack instance achieves co-residence with the victim instances. There are several placement variables such as cloud provider, application type, number of the instances, time launched and data center region that may influence the success of the attack. We study the effect of each placement variable on the premise of the other variables.

When studying the effect of application types, we choose Python and Node.js applications for OpenShift and Amazon Elastic Beanstalk, while for the Bluemix, we use two different docker images to launch containers. When studying the effect of number of instances, we varies the number of victim instances as well as the number of attack instances. Since many clouds support auto scaling to ensure load balance when running an application, e.g., a web server, the attacker has the chance to enforce the victim to launch more instances through increasing the workload of the application.

The delay from the time when the victim has finished launching its instances to the time when the attacker begins to launch instances is used to define the *time interval* between attacker and victim. For example, if the attack instances are created exactly at the time when the victim has finished his instances' creation, the time interval is 0; if the attack instances are created 1 h later after the victim instances are created, the time interval is 1 h. We use the default instance sizes in our tests, that is t2.small on Amazon Elastic Beanstalk, small gear on OpenShift and docker container on Bluemix, because co-residence detection doesn't need too much resource. The default data center regions are: us-east-1 for OpenShift, us-west-1a for Amazon Elastic Beanstalk, US South for Bluemix, unless otherwise noted.

We use the APIs of the PaaS clouds to implement auto test. Each cloud has the CLI tools running in Linux, e.g., *eb* for Amazon Elastic Beanstalk, *rhc* for OpenShift, *cf* for Bluemix. We used a single, local Intel Core i5-3470 machine to launch instances, log instance information and run the co-residency detection test suite. We crafted several scripts to implement auto creation, deletion and test of the instances. All these experiments were conducted over 2 months between April 2016 to June 2016.

4 Co-residence Threat Study

In this section, we present our measurement on placement and quantification of achieving co-residence in PaaS cloud Amazon Elastic Beanstalk, IBM Bluemix and OpenShift. At first, we test the three clouds for the threat of co-residence (Sect. 4.1). Then we investigate whether the application type plays an important role in achieving co-residence in container-based PaaS clouds (Sect. 4.2). Next up, we study how the other placement variables influence co-residence attacks (Sects. 4.3, 4.4 and 4.5) and finally summarise our findings (Sect. 4.6).

4.1 The Effort Taken to Achieve Co-residence with a Particular Target

In this section, we test the effort needed by an adversary to obtain co-residence with a single victim instance. After a target victim instance was launched, the adversary launched attack instances one-at-a-time sequentially until one obtained co-residence with the victim as indicated by the detection method. The attack instances and the victim instance have the same application type. In this test, there is a chance that the attack will never succeed if the machine where the victim instance runs has reached its upper limit. We treat the result as *valid* only if the co-residence is achieved within 200 attack instances (the valid rate with varying thresholds will be studied in the future). We repeat the trials until ten valid results are obtained. Table 3 shows the number of attack instances needed to obtain co-residence with a single victim. From the experiment results we can see that, co-residence phenomenon with one single victim commonly happens on all the three popular PaaS clouds. In Bluemix, only a few attack instances are needed to obtain co-residence. The results also show that Bluemix has the smallest variance of the results, while Amazon Elastic Beanstalk and OpenShift's variance is relatively higher, which means higher randomness in the number of attack instances.

Table 3. Distribution of the number of attack instances that are needed to obtain co-residence with a single victim. Results are ten times of test on each cloud. The time interval is 0.

Cloud provider	Value of ten tests	Mean	S.D	Min	Median	Max
Bluemix	1 5 2 2 4 2 2 4 1 2	2.5	1.35	1	2	5
OpenShift	37 37 15 11 36 18 17 32 27 31	26.1	9.97	11	29	37
Amazon	56 37 13 53 23 44 4 8 15 21	27.4	18.82	4	22	56

4.2 Effect of the Type of Instances

The victim instance could have the same application type with the attacker (all Python) or different (e.g., victim instance is Node.js and attack instances are Python). We study how the application types influence co-residence in this part.

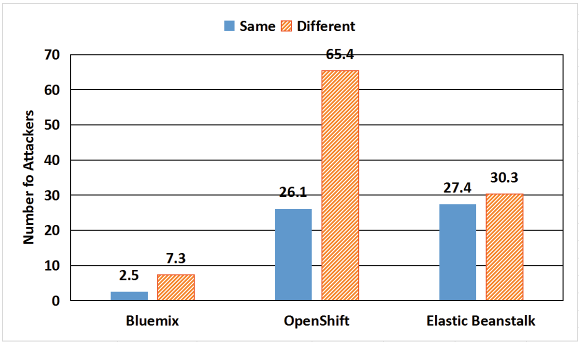


Fig. 1. The average number of attack instances needed to obtain co-residence with a single victim. *Same* means the victim and the attackers are the same application type. *Different* means the victim and the attackers are different application types. Results are the average of ten tests on each cloud. The time interval is 0.

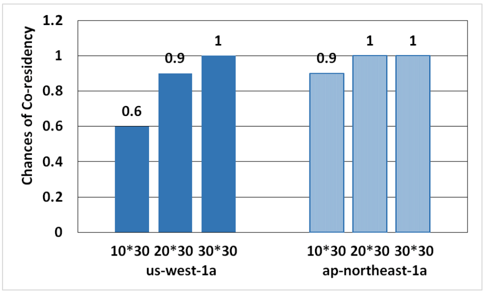
Bluemix and OpenShift are container-based PaaS clouds while Amazon Elastic Beanstalk is a VM based PaaS cloud. Figure 1 shows the average number of attack instances needed to obtain co-residence with a single victim. From the experiment results we can see that, for Bluemix and OpenShift, it is obviously easier to achieve co-residence with the victim using the same application type than different application types. While for the Amazon Elastic Beanstalk, there is no obvious difference. The reason should be the different isolation mechanisms between instances. This indicates that using the instances of the same application type as the victim to attack will increase the chance of co-residence in container-based PaaS clouds.

Bluemix has the weakest resistance to co-resident attack of the three clouds. Sometimes only one attack instance is enough. Since it is too easy for Bluemix to achieve co-residence, it can be predicted that in the following experiments (ten victim instances or more), the co-residency achievement will be much easier. So we will not talk about the Bluemix in the following experiments any more.

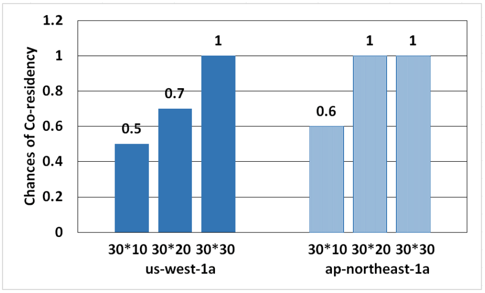
4.3 Effect of the Number of Instances

We observe the placement behavior varying the number of victim and attacker instances in this part. Intuitively, we expect the chances of co-residence to increase with the increasing number of attack or victim instances. All the tests use the same application type (Python) and the time interval is 0. The experiment results are the average of ten times tests.

At first, we keep all the placement variables constant including the number of attack instances (fixed to 30) and then vary the number of victim instances (10, 20, 30) to observe how the number of victim instances influences the results of co-residency. As is shown in Fig. 2-(a), for both OpenShift and Amazon Elastic Beanstalk we observe that, the more victim instances, the higher co-residency chance. Similarly, we also see an increase in the chances of co-residency with



(a) Vary the number of victim instances



(b) Vary the number of attack instances

Fig. 2. Chances of co-residency with varying number of instances. (a) Fixing the number of attack instances to 30 and varying the number of victim instances; (b) Fixing the number of victim instances to 30 and varying the number of attack instances. All these results are from one data center region (OpenShift: us-east-1, Amazon Elastic Beanstalk: us-west-1a).

increasing number of attack instances (10, 20, 30) across both of the cloud providers (as shown in Fig. 2-(b)). So we can conclude that, the chance of co-residency increases as the increase of the number of attacker or victim instances.

4.4 Effect of the Time Interval

In this section, we want to find the answer to the questions that how quickly an attacker should launch her instances after the victim instances are launched, and whether there is any increase in chance associated with the time interval or whether the result can help an adversary to design better launch strategies? We launch 20 victim instances at the beginning and after every certain time delay (e.g., 0 h, 1 h, 2 h, etc.), we launch 20 attack instances and do the co-residency detection and then remove them. We keep the other placement variables constant and repeat the test for ten times.

Figure 3 shows the chances of co-residency with varying delays between victim and attack instance launches. The experimental results have no obvious regularity. For OpenShift, the success rate reaches maximum at time interval 0 h, 1 h and 8 h while minimum at time interval 32 h. For Amazon Elastic Beanstalk,

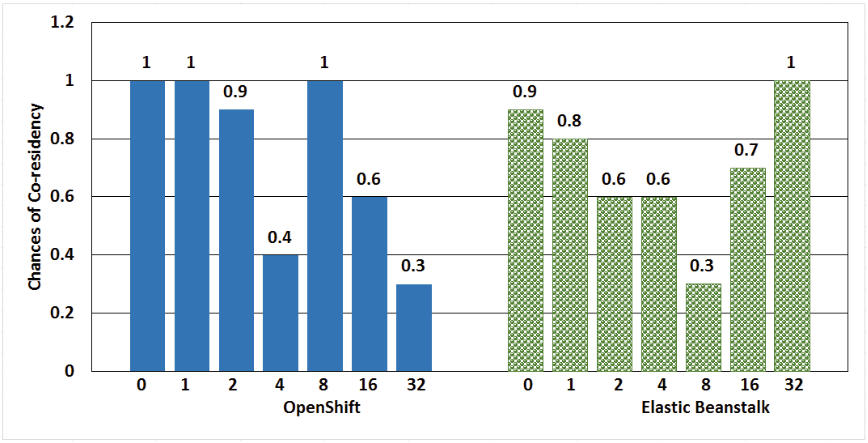


Fig. 3. Chances of co-residency with varying delays between victim and attacker instances launches. All these results are from one data center region (OpenShift: us-east-1, Amazon Elastic Beanstalk: us-west-1a).

the success rate reaches maximum at time interval 32 h, while minimum at time interval 8 h. With smaller interval (within 1 h), the adversary has relatively high chance to attack successfully. We did not find the chance of co-residency drops to zero during the detection. We speculate that the reason could be that some neighboring instances on the victim's machine were terminated though some may be created.

4.5 Effect of the Data Center Region

We only compared the different regions of Amazon Elastic Beanstalk (the us-west-1a and ap-northeast-1a) in this section since OpenShift is built on Amazon EC2 and we believe they have the same regularity. All the tests use the same application type (Python) and the delay is 0. The experiment results are the average of ten times tests. Ap-northeast-1a is less popular than us-west-1a and has relatively fewer machines, so we expect higher success rate. Figure 4 shows that, no matter at any circumstances, attack instances in ap-northeast-1a have the same or more chance of achieving co-residence with the victim instances than the ones in us-west-1a, as we expected.

4.6 Summary of the Co-residence Threat

Through the experiment results we can find that: (1) All three clouds examined by us show weak resistance against co-resident attacks. In the Bluemix PaaS cloud, only a few attack instances are needed to obtain co-residence with a particular target. Even in the worse observed cases, one hundred attack instances are always sufficient in Amazon and OpenShift. This indicates the cost of such co-resident attack can be really low. In fact, we spend only a little money for

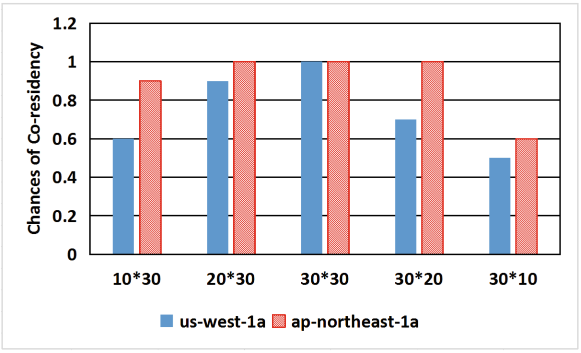


Fig. 4. Chances of co-residency with varying number of instances in different regions of Amazon Elastic Beanstalk.

Amazon Elastic Beanstalk during the tests. For the other two PaaS clouds, we can finish all the tests for free by applying a number of accounts. (2) Application types really play a critical role in achieving co-residence in container based PaaS clouds. When an adversary uses the instances of the same application type as the victim to attack in container-based PaaS clouds Bluemix and OpenShift, the success rate is higher. (3) In the study of the other placement variables that may influence co-residence possibility we find that, increasing the number of victim instances or attack instances will help increase the chance of co-residency in clouds OpenShift and Amazon. Besides, though in the long term of time delay we did not find any regularity in the chance of achieving co-residence, we find that in the delay within 1 h the adversary has relatively higher chance to attack successfully. Also in the less popular region of Amazon with less physical servers, the attacker has more chance achieving co-residence with the victim.

Therefore, to improve the success rate in co-resident attacks, the adversary should launch instances of the same type as the victim and try to increase the number of victim or attack instances. Also a smaller region and shorter time interval will help the attack. The adversary can first trigger a scale-up event on target victim by increasing its workload, which will cause more victim instances to launch. Afterwards, the adversary can launch multiple instances and may observe some of them achieve co-residence with the newly launched victim instances.

4.7 Discussion

Although the test is done in a way that we control both the attack instances and the target instances, we believe the results can reflect the real placement policy of the PaaS clouds. Also, the co-residence detection can be implemented without controlling the victim instances. For example, the attacker can run *lock process* in the attack instance and then trigger the victim to access the memory by requesting large size web pages from the target web application as described in [8]. Through analyzing the response time of the web requests the attacker can infer whether he has achieved co-residence with the victim instance or not.

What's more, there are other placement variables that we haven't test in this paper, such as the time of the day, the size of the instances, the threshold of valid results (mentioned in Sect. 4.1) and other PaaS cloud platforms. We plan to investigate their effects on co-resident attack in our future work.

5 Related Work

Co-residence Detection. Techniques for co-residency detection have been studied by many pioneers. They categorize these techniques into two classes: the *side-channel* detection and *covert-channel* detection [8].

The side-channel detection is done by the attacker without the help of the victim. There are network based side-channel detection method, for example, network round-trip timing side-channel was used by Ristenpart et al. [3] to detect co-residency, Bates et al. [18] proposed a side-channel for co-residency detection by causing network traffic congestion in the host NICs from attacker-controlled VMs. There are also time based side-channel detection, e.g., Zhang et al. [19] developed a system called HomeAlone to enable VMs to detect third-party VMs using timing side-channels in the last level caches, Varadarajan et al. [8] used a timing side-channel based on memory bus blocking to detect co-residency when study placement vulnerability in different clouds.

Unlike the side-channel detection, the covert-channel detection needs the victim's cooperation. Ristenpart et al. [3] use coarse-grained covert-channels in CPU caches and hard disk drives for co-residency confirmation. Xu et al. [6] established covert-channels in shared LLC between two colluding VMs in the public clouds. Zhang et al. [10] also use the LLC as the covert-channel to detect co-residency on PaaS clouds. There are also researches [7–9] exploited memory bus as a covert-channel on modern x86 processors, in which the sender issues atomic operations on memory blocks spanning multiple cache lines to cause memory bus locking or similar effects on recent processors. Inci et al. [20] compared three co-residence detection methods LLC software profiling, LLC covert channel and memory bus locking for their efficiency on detecting co-location and showed that the LLC software profiling technique worked no matter with or without the cooperation from the victim. We use the memory bus as the covert-channel in our paper, by adding a *clflush* instruction to the receiver to ensure the always hitting of the memory without thinking about the cache size.

Co-residence Threat Studies. The co-residence problem was first proposed by Ristenpart et al. [3], which showed that a malicious cloud tenant may place one of his VMs on the same machine as a target VM. Their study was followed by Xu et al. [6] and further extended by Herzberg et al. [21]. Xu et al. [9] investigated how Amazon EC2 evolved in VM placement, network management, and Virtual Private Cloud (VPC), conducted a systematic measurement study of co-residence threats in Amazon EC2. Varadarajan et al. [8] studied placement vulnerabilities in the context of VPC on EC2, as well as on Azure and GCE. There are also studies about new VM placement policies, Han et al. [22,23] and

Azar et al. [24], which are used to defend against placement attacks. However, all of these researches focus on IaaS clouds placement problems.

Varadarajan et al. [8] have briefly mentioned the placement problem of the PaaS clouds. They take Heroku as an example in their work and use logical side-channel to determine co-residency. Zhang et al. [10] also simply tested the PaaS cloud DotCloud and OpenShift to show it is not too difficult to accomplish co-residency. However, none of them gave further analyse of the co-residence threat in public PaaS clouds, which is just our purpose of this paper.

6 Conclusion

In this paper we studied the co-residence threat of the multi-tenant public PaaS clouds. We analyzed the characteristics of different public PaaS clouds and implemented a memory bus based covert-channel co-residency detection method. Using the detection method we investigated three popular public PaaS clouds Bluemix, OpenShift and Amazon Elastic Beanstalk to identify the potential co-residence threat. We find that it is straightforward and cost-efficient to achieve co-residence on the three public PaaS clouds. It even costs no money to achieve co-residence in Bluemix and OpenShift. Also application type plays a critical role in container-based PaaS clouds' co-residence threat analysis. Finally, based on our finding, we presented the strategy to achieve co-residence with least effort.

Acknowledgement. We would like to thank Zeyi Liu and the anonymous reviewers for their insightful and detailed comments. This paper was supported by National Natural Science Foundation of China (NSFC) under Grant No. 61100228 and the project Core Electronic Devices, High-end Generic Chips and Basic Software (No. 2015ZX01029101-001). Peng Liu was supported by NSF CNS-1422594 and NSF SBE-1422215.

References

1. 2015 Review Shows $110 billion Cloud Market Growing at 28% Annually (2016). https://www.srgresearch.com/articles/2015-review-shows-110-billion-cloud-market-growing-28-annually. Accessed 3 Mar 2016
2. Platform as a service - a global strategic business report (2015). http://www.strategyr.com/MCP-7070.asp. Accessed 3 Mar 2016
3. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud,: exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 199–212. ACM (2009)
4. Varadarajan, V., Kooburat, T., Farley, B., Ristenpart, T., Swift, M.M.: Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 281–292. ACM (2012)
5. Zhou, F., Goel, M., Desnoyers, P., Sundaram, R.: Scheduler vulnerabilities and attacks in cloud computing. arXiv preprint arXiv:1103.0759 (2011)

6. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of L2 cache covert channels in virtualized environments. In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, pp. 29–40. ACM (2011)
7. Wu, Z., Xu, Z., Wang, H.: Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In: Presented as Part of the 21st USENIX Security Symposium (USENIX Security 12), pp. 159–173 (2012)
8. Varadarajan, V., Zhang, Y., Ristenpart, T., Swift, M.: A placement vulnerability study in multi-tenant public clouds. In: 24th USENIX Security Symposium (USENIX Security 2015), pp. 913–928 (2015)
9. Xu, Z., Wang, H., Wu, Z.: A measurement study on co-residence threat inside the cloud. In: 24th USENIX Security Symposium (USENIX Security 2015), pp. 929–944 (2015)
10. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-tenant side-channel attacks in paas clouds. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 990–1003. ACM (2014)
11. Yarom, Y., Falkner, K.: Flush+reload: a high resolution, low noise, L3 cacheside-channel attack. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 719–732 (2014)
12. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: 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. ACM (2012)
13. Amazon. https://aws.amazon.com/cn/. Accessed 23 Apr 2016
14. Bluemix. https://new-console.ng.bluemix.net/. Accessed 23 Apr 2016
15. Openshift. https://developers.openshift.com/. Accessed 23 Apr 2016
16. Docker. https://www.docker.io/. Accessed 23 Apr 2016
17. Guide, P.: Intel® 64 and IA-32 architectures software developers manual (2011)
18. Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., Butler, K.: Detecting co-residency with active traffic analysis techniques. In: Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop, pp. 1–12. ACM (2012)
19. Zhang, Y., Juels, A., Oprea, A., Reiter, M.K.: Homealone: co-residency detection in the cloud via side-channel analysis. In: 2011 IEEE Symposium on Security and Privacy (SP), pp. 313–328. IEEE (2011)
20. Inci, M.S., Gulmezoglu, B., Eisenbarth, T., Sunar, B.: Co-location detection on the cloud (2016)
21. Herzberg, A., Shulman, H., Ullrich, J., Weippl, E.: Cloudoscopy: services discovery and topology mapping. In: Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop, pp. 113–122. ACM (2013)
22. Han, Y., Alpcan, T., Chan, J., Leckie, C.: Security games for virtual machine allocation in cloud computing. In: Das, S.K., Nita-Rotaru, C., Kantarcioglu, M. (eds.) GameSec 2013. LNCS, vol. 8252, pp. 99–118. Springer, Heidelberg (2013). doi:10.1007/978-3-319-02786-9_7
23. Han, Y., Chan, J., Alpcan, T., Leckie, C.: Virtual machine allocation policies against co-resident attacks in cloud computing. In: 2014 IEEE International Conference on Communications (ICC), pp. 786–792. IEEE (2014)
24. Azar, Y., Kamara, S., Menache, I., Raykova, M., Shepard, B.: Co-location-resistant clouds. In: Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security, pp. 9–20. ACM (2014)