Stalling Live Migrations on the Cloud

Ahmed Atya[*], *Azeem Aqil*[*], *Karim Khalil*[*]
Zhiyun Qian[*], *Srikanth V. Krishnamurthy*[*] *and Thomas F. La Porta*[†]
[*]*University of California, Riverside,* [†]*The Pennsylvania State University*
{afath001,aaqil001,karimk,zhiyunq,krish}@cs.ucr.edu, tlp@cse.psu.edu

Abstract

Live migration is commonly employed by cloud providers for performance reasons (e.g., ensuring load balancing). Recently, migration has been considered as a countermeasure against cloud-based side-channel attacks. In this paper, we discover an attack using which an adversary can effectively stall a live migration; this can not only hurt performance but also hurt the usage of virtual machine (VM) migration as a defense against cloud-based side channel attacks. Specifically, we discover a KVM vulnerability which, if exploited by a co-resident attacker, can suspend or stall the live migration time by up to 3x in some scenarios. The attacker can also delay her own VM migration, indefinitely to ensure sustained co-residency. The attacks that we propose are essentially based on increasing the volume of dirty pages and creating bus contention, leading to delaying the migration process. We show that this approach does not cause significant interference to side channel attacks such as the Flush+reload attack, which the attacker can continue to carry out in parallel. In fact, the success rates of the Flush+reload can increase by about 100 % (when the defender invokes migrations), if a stalling attack is simultaneously launched.

1 Introduction

Live Virtual Machine (VM) migration is commonly employed by cloud providers and datacenter administrators for performance reasons. Physical machine maintenance and service scalability are the two most common reasons for migrating a virtual machine [1].

On the other hand, recent papers have touted VM migration as a viable, general defense mechanism to counter cloud based side-channel attacks [2, 3]. Specifically, most side-channel attacks that target the leakage of secret information from a victim VM rely on co-residing the attack VM on the same physical machine as the victim VM. Such information could be a login password, an encryption key or simply a session token. The severity of leakage of such secret information is subsequently manifested in various ways such as gaining access to the victim VM or session hijacking. The thesis in [2] and [3] is that by decreasing the co-residency times between the attacker and victim VMs, migration drastically decreases the potency of side-channel attacks. In other words, the information leakage rates are drastically reduced.

In this paper, we test this hypotesis. Specifically, we ask whether it is possible for an attacker to pre-emptively stall or suspend live migrations in order to defeat the objectives in [2, 3]. In fact, an attack that stalls a VM migration may not only have effects on security, but also have negative ramifications on performance. In addressing the above question, we find a hypervisor vulnerability that gives a co-residing attack process the ability to stall migrations while simultaneously carrying out a cloud-based side-channel attack targeting information leakage (e.g., Flush+Reload [4]).

The key property that is exploited in such an attack is that live migrations are designed so as to minimize the penalty of disrupting ongoing customer computations or services [1, 5]. Thus, a migration is completed in several stages and in each stage, the objective is to minimize the degree of synchronization necessary between the original VM (on an old physical machine) and the new VM (now housed on a different physical machine). If the attacker can somehow disrupt this synchronization process (e.g., by increasing the number of dirty pages as we discuss later), the VM migration may take much longer time to complete. This in turn implies that the attacker VM is now able to co-reside with the victim VM for longer, and thus, is able to enjoy more sustained information leakage.

To illustrate the impact of such an attack, consider the work in [6], where the authors propose a Last Level Cache (LLC) side-channel attack. This attack supposedly requires only a few tens of minutes to be successful (≈ 27 mins). Thus, prolonging migrations by the order of minutes can significnatly improve the chances of this side-channel attack succeeding. In the worst case scenario, if the network provider/administrator takes more severe measures, such as forcefully suspending VMs, it could potentially hurt the performance of benign VMs when false positives are incurred.

In this paper, we showcase the possibility of an attack that targets the stalling of the live migration of a VM. We call this attack *the stalling attack*. To do so, the requirement is that the adversarial VM co-resides on the same physical machine as the victim VM. The attack exploits a vulnerability in the live migration process of the hypervisor (specifically the KVM hypervisor).

Our contributions are as follows;

- We discover a vulnerability in the live migration implementation of the KVM hypervisor and design a side-channel attack that exploits this vulnerability to stall or suspend the migration process.
- We extensively evaluate the proposed attack in different scenarios. We demonstrate the effectiveness of the stalling attack in aiding other side channel attacks that target information leakage.

Our results show that the stalling attack can prolong VM migrations by over a factor of 3X. This in turn drastically improves the likelihood of success with a simultaneously launched Flush+reload attack; in fact, given a sufficient number of attack attempts, this success rate increases by well over 100 % in some cases compared to the case where there is no simultaneous stalling attack (how and why are discussed later).

Scope: We note that we only study the question "how an attacker can suspend or prolong a migration process?" but not "how an attacker can detect a live migration process?". Prior efforts, such as [7], discuss how an attacker can detect an ongoing migration. We assume that an attacker will continuously carry out the stalling attack after co-residing with a victim VM in order to guarantee that neither the victim VM nor the attacker VM are migrated during performing the side-channel attack

2 Background and Related Work

Virtual machine migration is a powerful tool that introduces greater flexibility in cloud environments where VMs need not be tied down to physical machines. Since VMs are stored as files in a system, they can be easily transferred from one machine to another in a similar way, for instance via network file transfer, as long as they are halted. This naturally incurs large down times wherein a VM must first be completely shut down, then transferred and restarted at the target machine. Live VM migration [8, 1] , on the other hand, refers to the concept of transferring a VM while it is still running so that the downtime is as small as possible.

While, different cloud providers may have different, and often proprietary, protocols for live migration, the underlying principles are the same. The objective is to transfer the VM from a source machine to a target machine. The migration process starts off by transferring memory pages belonging to the VM from the source machine to the target machine while the VM is still running. During this process, any pages that change (commonly referred to as pages becoming dirty) have to be retransmitted over the network. Then, the migration process halts the VM on the source machine, transfers the rest of the pages and finally restarts the VM at the target physical machine. The point at which the VM is halted and transmission of dirty pages stopped depends on the hypervisor used. For example, Xen [9] has an upper limit on the number of dirty pages that are transmitted, while KVM [10] keeps transmitting dirty pages as long as some exist.

2.1 Related Work

There has been recent work towards demonstrating the usefulness of live VM migration [11, 3, 12]. Live migration is used for load balancing, resource management, and security. Recent work has demonstrated that live migration is a very effective defense against side channel attacks [2, 3]. However, very little work has been done to evaluate the security of the live migration process itself.

In [7], the authors identify methods to discover that a migration is ongoing. They also propose techniques to hide traffic patterns in migrations.

The first work that demonstrated the need for secure migration was [13]. The authors identified three classes of threats to the migration process, namely, threats to the control plane, threats to the date plane and threats to the migration module. The control plane includes the communication mechanism employed by the virtual machine monitor, which is the entity that orchestrates the migration. On the other hand, the data plane is where the actual migration occurs. While the authors identified three different sources of threats, they only experimentally validated two attacks using the data plane and the migration module.

The first attack targets the data plane and is a memory manipulation attack which makes use of a malicious intermediary node. On the testbed in [13], the migrating VM data is relayed using a malicious node. The malicious node replaces certain pages as they are forwarded to their destination. The authors successfully managed to replace pages belonging to the `ssh` process so that the victim VM accepted all incoming `ssh` connections after migration is complete.

The second attack is more general in that it's implications are not limited to VM migration. The authors identified several heap overflow vulnerabilities in the Xen hypervisor that were present in the code responsible for handling migration. These vulnerabilities allowed the authors to run arbitrary code and thus allowed use of malicious code to completely compromise the hypervisor.

Other works do not demonstrate working examples of attacks to VM migration. They briefly mention how some attacks *could* be employed against live migration. For e.g., the authors in [14] briefly mention that time-of-check to time-of-use (TOCTTOU) [15] [16] attack and replay attacks can be used if the migration process is not sufficiently protected. There has also been a host of work that introduce frameworks or techniques for live VM migration and discuss the need for secure migration [16] [17] [18]. However, this need has not been substantiated by the presence of real implementable attacks that can delay or stall the migration process. In contrast, our work demonstrates how an attacker can stall live VM migrations. In particular, to the best of our knowledge, our work is the first to showcase how a stalling attack can be used in conjunction with a side channel attack to greatly increase the attack surface of the side channel attack.

3 Threat Model

We assume an active attacker that controls multiple VMs running on the cloud. In addition, we assume that the attacker has computational resources (e.g., servers or other VMs) external to the cloud. We impose no restrictions on the computational resources available to the attacker. We also assume that the victim VM has some publicly accessible service via which the attacker can interact with the VM (e.g., the victim hosts a web service).

We assume that the attacker has already co-resided with the victim VM (previous work, such as [19, 3], showed the possibility of the co-residency process wherein an attacker can can verify the co-residency of its own VM with a victim's VM). We make no assumptions on the number of other VMs co-residing with the attacker VM on the same physical machine. Upon co-residing with the victim, we assume that the attacker is launching a side channel attack. For ease of exposition we focus on flush and reload attack [20].

In carrying out the stalling attack, we assume that the attacker has two main objectives. First, it seeks to suspend or delay the victim's VM from being migrated. Second, considering that the cloud service provider can potentially migrate the attacker's (co-resident) VM as an alternative, the second goal of the attacker is to prevent her own VM, which is co-residing with the victim VM

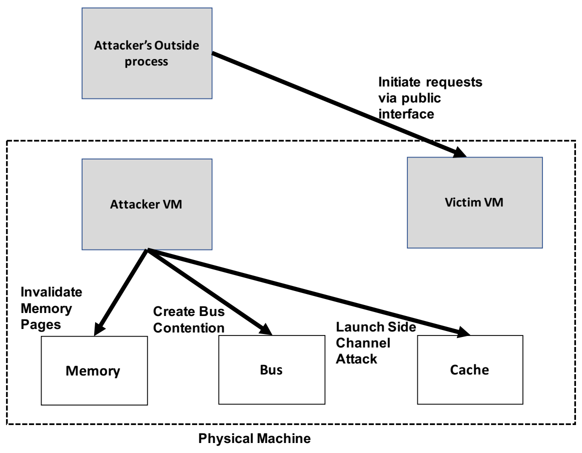

Figure 1: Overview of the attack.

on the same physical machine, from being migrated to another physical machine.

Cloud Provider Migration Policies: Different cloud providers may have different reasons and policies for triggering a live VM migration. These policies will determine the effectiveness of any attack that aims to influence a migration. Broadly speaking, we consider three provider policies for live VM migrations; (i) maintenance policy, (ii) security policy, and (iii) hybrid policy. We believe that these three migration policies correspond to the most popular reasons why cloud providers might want to migrate VMs. At a high level, with the maintenance policy, VMs are migrated for maintenance reasons such as faults or software updates. If the security policy is in force, VMs are migrated periodically to alleviate potential malicious activities such as side channel attacks. The hybrid policy is simply a combination of both the security and maintenance policies. We provide details on how we implement these policies (based on prior papers) in Section 5.

4 Live Migration Stalling Attack

As discussed earlier, a key factor that determines how quickly a VM can be migrated, is the rate at which memory pages become dirty. Recall that dirty pages need to be retransmitted over the network. The stalling attack exploits the key observation that if the page dirtying (also called invalidation) rate is greater than the available network bandwidth, the process can potentially be suspended indefinitely.

To that end, the stalling attack attempts to do two things viz., (i) intentionally attempt to dirty memory pages, and (ii) cause bus contention to slow the rate of transfer of dirty pages. Consequently, the stalling attack is comprised of two steps.

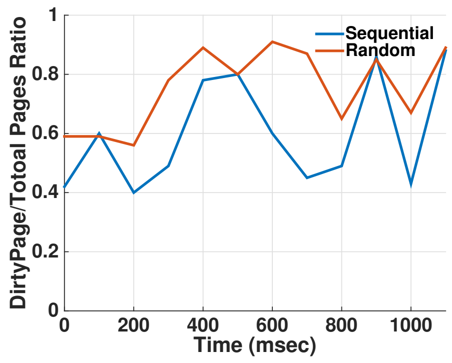

Figure 2: The ratio between dirty and total number of pages over time.

First, the attacker engineers memory access requests that have a high chance of invalidating memory pages. The attacker uses different techniques to invalidate its own pages and that of the victim VM (recall that the attacker wants to delay the migration of both its own VM and that of its victim). Second, by inducing bus contention, the attacker attempts to delay the processes of reading and writing memory pages, which consumes transmission bandwidth (contributes to overhead). While he ultimate goal of the attacker is to invalidate all the memory pages of the victim all the time, note that the effectiveness of the attack and how long the attacker can successfully stall a migration depend on how the migration policy in effect is implemented. Given the above high level strategy, the stalling attack is comprised of three main steps. 1) Invalidate memory pages of the attacker VM 2) Invalidate memory pages of the victim VM and 3) Induce bus contention. Figure 1 provides a bird's eye view of how the attack functions. Not only are the 3 steps listed above shown, the figure also depicts the attacker simultaneously launching a side channel attack. As previously mentioned and as will be showcased later, the stalling attack can be used increase the effectiveness of cloud-based side channel attacks.

In the Xen hypervisor, there is a cap (maximum amount) on how many VM memory pages can be transferred from the old physical machine to the new one. After those many pages are transferred, a timeout is initiated and the VM is forcefully shut down and migrated (i.e., a regular migration instead of a live migration is invoked). In this case, the down time can be high (of the order of the regular migration time, i.e., without live migration). However, for the KVM hypervisor, there is no timeout or a limit on the maximum number of page transfers (by default). Hence theoretically, a stalling attack can prolong a VM live migration process infinitely.

4.1 Invalidating the attacker VM's memory pages

We first briefly explain how memory pages are managed on the Linux operating system [21]. Then, we explain how to leverage page management knowledge to quickly invalidate the attacker VM's memory pages and reduce the read and write throughputs significantly.

Linux maintains a three level page table regardless of the underlying architecture. The Page Global Directory (PGD) represents the physical page frame. Each active PGD points to an array of Page Middle Directory (PMD) entries. In turn, each entry in the PMD points to a Page Table Entry (PTE) that actually points to the user data. A PTE contains protection and status bits summarized in Table 1.

To programmatically invalidate all pages of the memory, we set the PAGE_DIRTY bit for the PTE items. A simple way to achieve this is to modify the first byte from each memory page. There are multiple ways one can access and modify pages in the memory. The simplest method is to sequentially access and modify the pages. Another method is to access the pages in random order. The linux OS uses various tricks to optimize memory and it is not clear which method can be more effective (more details on this later). We test these two memory access and modification methods.

For each of the memory access methods, we scan the entire memory and modify one byte from each page. To determine which access and modification pattern gives a better page dirtying rate, we measure the number of dirty pages once every 100 msec (via `cat /proc/meminfo | grep Dirty`). Then, we compute the ratio of the number of dirty pages to the total number of pages in the memory (via `cat /proc/meminfo | grep PageTables`). In Fig. 2, we compare random and sequential memory access and modification methods. We plot the ratio between the number of dirty pages to the total number of pages over a period of 1.2 seconds. The figure shows that random access and modification results in a larger page dirtying rate on average.

To understand why, notice that the greater the number of dirty pages in memory, the greater the odds of dirty pages being retransmitted over the network to the target physical machine. If memory pages are accessed sequentially, the operating system correctly predicts that pages once accessed, will not be accessed again for a while and writes them out to disk, which also makes the memory page 'clean'. This represents a disadvantage for the attacker because the migration process periodically checks memory for new dirty pages and sequential access reduces the number of dirty pages in memory. On the other hand, randomly accessing memory keeps the OS from correctly predicting page access patterns and

Bit	Function
PAGE_DIRTY	Whether a page is modified or not
PAGE_ACCESSED	Whether a page is retrieved or not
PAGE_PRESENT	Whether a page is resident in memory or swapped out
PAGE_PROTNONE	Indicate that a page is resident but not accessible
PAGE_USER	Whether a page is accessible from user space
PAGE_RW	Whether a page can be modified

Table 1: Protection and Status bits for a Page Table Entry

delays writing pages to disk (or turning dirty pages to clean pages).

4.2 Invalidating the victim VM's memory pages

The memory pages invalidation steps described in the previous subsection only works if the attacker owns the address space associated with the VM. In other words, the above approach cannot be directly applied to dirty the victim VM's pages. Here, as indicated in Section 3, we assume that the victim VM is running some kind of service that exports a public interface. We then send random requests (specific to the service) to the service, periodically. For example, for a web service hosting documents, periodically random documents are requested. Again, the randomness ensures a high rate of dirty pages.

We wish to point out that an attacker who intends to launch a side channel attack (as in [20]) one can expect that it already knows what type of service is hosted by the victim VM. If not, the attacker can perform a port scan to figure out which service is running on the victim VM. It then requests various objects randomly (as discussed above); these requests are expected to adversely affect the victim VM's memory state.

Later, we validate the approach in a variety of scenarios in Section 5 where the victim VM is running web-based and non web-based services. We choose different services to cover some of the most popular types of services in production today.

4.3 Creating bus contention

To amplify the delay in migrating a VM to a new physical machine, the last part of the attack involves creating contention on the bus (normally, two virtual machines that share a CPU also share the CPU bus). The purpose of creating bus contention is to delay the read and write memory operations, since it takes longer to transfer dirty pages around the system when bus contention exists. A similar idea was proposed in [22] as a method to check for co-residency. In particular, a cache locking mechanism is implemented to guarantee coherency across the same area in memory, if that common area in memory is simultaneously being modified by different processors.

To create bus contention, the attacker VM allocates a chunk of memory that is larger than the size of the last level cache. Next, the attacker VM misaligns the memory access pointer by adding an offset to it. The value of the offset can be anything less than the size of the memory word (for 64bit architectures this is 4bytes). The issuance of an unaligned, atomic access operation (such as a read, or the XADD operations for x86 processors) causes the locking of the memory bus [22]. If unaligned access happens across cache lines, a more dramatic slow down is observed due to double cache misses. A significant increase (around 3x) in the memory access time is observed by the attacker as well as the victim VMs given that both VMs share the same memory bus (validated experimentally).

4.4 Stalling attack in conjunction with side-channel attack

One of the core strengths of the stalling attack is that it can be carried out in conjunction with a side-channel attack to greatly increase the attack surface of the side-channel attack. We use the popular Flush+Reload side-channel attack to illustrate in this work [4].

In brief, Flush+Reload attacks the last level of cache (LL3). The attacker monitors the cache lines in order to deduce whether a specific instruction is executed in a given time slot. The attack is composed of two steps. First, the attacker flushes instructions from the cache (for example, by using the `clflush` command). Then, the attacker accesses the flushed cache line and measures the time it took do so. The measured time information gives an indication of whether the instruction was executed. If the victim has accessed the cache line, the load time will be much shorter than if the victim had not. This is because the cache line would need to be loaded from memory in the latter case. The more time that an attacker VM is able to co-reside with the victim VM on the same physical machine, the more likely it is that this side-channel attack will be successful.

Using the stalling attack in conjunction. It has been argued that VM migration is a powerful defense mechanism against such side channel attacks, because it can interrupt the flow of side channel attacks (disrupt co-residency). Side channel attacks are generally very time consuming (Flush+reload typically take at least 30 mins to complete) and an attacker, armed with our stalling attack, can greatly increase the likelihood of a prolonged co-residency and thus, in turn magnify the chances of the side channel attack being successful. The attacker does this by simultaneously launching both the side channel attack and the stalling attack.

There are a few factors to consider when launching these attacks together. First, as has been mentioned before, the aim of the stalling attack is to invalidate pages in memory. Given that there will be dirty pages in memory, a natural question to ask if the stalling attack will have any implicit effect on Flush+Reload. Since the stalling attack does not target the cache, we do not expect there to be any overlap between the attacks in terms of cache contention (the attacker VM can only dirty its own pages in memory). However, the bus contention that the stalling attack creates will slow down page loads from memory. As such, we expect that Flush+Reload will take longer to complete when a stalling attack is underway. However, we will later show that this overhead is not too large and that the advantage of launching the attacks together (i.e., the Flush+Reload will have more time to execute and complete successfully) outweighs the cost in terms of the overhead.

The second question to consider is whether Flush+Reload affects the stalling attack. Again, since they target different components (one the memory, the other the cache) we do not expect the Flush+Reload to affect the stalling attack (as verified in our experiments).

5 Implementation and Evaluations

In this section, we describe our experimental setup in detail and then evaluate the effectiveness of the stalling attack.

5.1 Implementation details

To the best of our knowledge, VM migration is not a feature available to clients of today's cloud service providers. This necessitates a private cloud over which we have administrative control.

Our private cloud consists of two Cisco 20-Port gigabit switches, 13 Servers (11 DELL and 2 HP), and 9 DELL machines to initiate server requests. This cloud can host up to 280 (512 MB, 1 GHZ) VMs, 140 (1 GB, 1 GHZ) VMs or 70 (2 GB, 1 GHZ) VMs, simultaneously. These three different VM configurations are equivalent to t2.nano, t2.micro and t2.small on EC2 [23], respectively. We run the KVM hypervisor [24] on top of Ubuntu 14.04 [25]. All VMs run either Centos 7 [26] or Ubuntu 15 images. We use Apache CloudStack [27] as the cloud management and provisiioning tool. Live migration is carried out by using virt-manager tool (KVM + QEMU).

Victim VM's host different types of services, such as, Taiga [28], ownCloud [29] and MediaWiki Server [30]. Taiga is an open-source project manager software that involves a mix of CPU, disk, and memory workloads. ownCloud is an open-source file hosting service (resembles Dropbox) that involves memory and disk intensive workloads. (ownCloud is Disk intensive). MediaServer is an open-source wikipage server that involves a mix of CPU, disk, and memory workloads. By using such a diverse set of services, we consider workloads that cover CPU, memory, and disk, equally. Finally, to simulate legitimate background traffic, we use 9 hosts that interact with the services running on the victim VM.

For each experimental scenario (discussed later), there is one attacker VM and one victim VM that co reside on a single physical machine. The attacker also controls an outside process that interacts with the victim VM via its web service interface. For each attack, the attacker performs the stalling attack via the 3 steps detailed in Section 4. The attacker VM runs two processes; one that runs the bus contention procedure described earlier, and the other that runs the memory invalidation procedure. Both are implemented in C++. The outside process also simultaneously interacts with the victim VM to invalidate its pages.

We get the code of the flush+reload attack from [31]. To increase the odds of Flush+reload succeeding, the attacker simultaneously launches the stalling attack in conjunction with Flush+reload.

Because physical machines in data centers are shared between multiple tenants, we also spawn other VM's on the same physical machine (between 2 and 4) and assign random jobs to them.

Migration Policies: Recall from section 3 that we consider 3 migration policies (or reasons for initiating a migration), namely maintenance policy, security policy and hybrid policy.

With the maintenance policy, the provider migrates VMs for maintenance reasons. (for e.g., performance, failures, or regular software updates). There have been many works that study hardware and software failures [32]. We use the Markovian model used and evaluated in [33] to simulate hardware and software faults and trigger a live VM migration. Specifically, whenever the model predicts a fault/failure, we attempt to migrate all VM's that are running on the physical machine that faulted. The input parameters needed for the model are probabil-

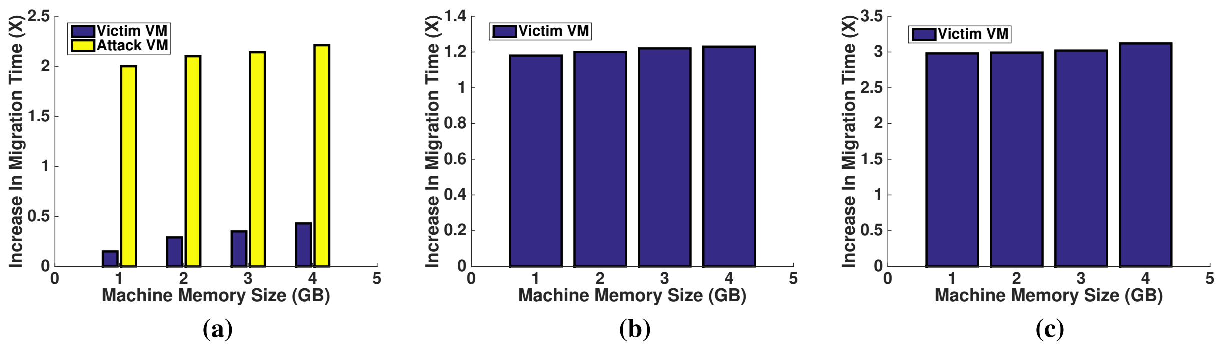

Figure 3: (a) The increase in migration time using memory invalidation. (b) The increase in migration time using bus contention. (c) The increase in migration time using both memory invalidation and bus contention.

ities of failure for hardware and software faults. We use independent failure rates of 0.005 failures every minute.

With the security policy, according to Nomad [2], migrating VMs periodically could significantly decrease the probability of a side-channel attack being successful. The authors in [3] determined that between one to two hours was the ideal period for VM's to be migrated. Consequently, for the security policy, we attempt to migrate all VM's every 90 minutes (starting from whenever a specific VM is started).

In the hybrid mode, we assume the security policy is on. However, VMs are still susceptible to errors. Thus, VM's are migrated if either a fault (error) occurs, or the periodic migration due to the security policy is triggered.

5.2 Attack Efficacy

In this section, we demonstrate the efficacy of the proposed attack at stalling live VM migration.

Fig. 3 shows how the time taken to migrate a VM increases when the attack method is (a) only memory invalidation, (b) only bus contention or (c) a combination of memory invalidation and bus contention. By soley invalidating memory pages, the attacker stalls the migration of the attack VM by ≈ 2 x. On the other hand, the effect on the victim VM is limited. Using bus contention, the attacker VM is totally suspended, while the victim VM sees a 1.2x increase in migration time (Fig.3(b)). By combining both techniques, the victim VM time increases by ≈ 3x as shown in Fig.3(c). An interesting thing to note about Fig 3 is that the time for Flush+reload increases with the size of the victim's memory. To understand why, recall that by probing the cache, Flush+reload tries to guess what instructions were executed by the victim VM. Operating systems reduce load times by trying to keep the most used instructions in memory. This way, it is likely that an instruction can be directly loaded from memory into the cache instead of being loaded from the disk. The larger the memory size, the greater the number of instructions it can hold. Consequently, larger memory sizes mean that a larger number of different instructions can potentially be loaded into the cache from memory. In such a scenario the attacker has to guess which instruction was executed from a much larger set of candidates. The probability of the attacker successfully guessing which instruction was executed is hence lower implying that the attackers needs more time, on average, to be successful.

In Fig. 4, we investigate how the migration time for the victim changes with the number of VM's sharing a physical machine. We only attempt to migrate one VM and we use one VM to launch the attack. We randomly vary the load on each legitimate machine. A slight increase in the average migration time is observed when the number of VM increases. However, the standard deviation increases significantly which indicates that adding more VMs increases the uncertainty in effectiveness of the attack.

We also note that when the other tenant VMs are dormant (i.e., less utilized), the attack effect is less pronounced because the time slice (equivalently, the attack window) to invalidate memory and cause bus contention is smaller. On the other hand, active tenants (i.e., highly utilized VMs) involuntary help the attacker by invalidating the memory. The bus contention effect induced by the attack is aggravated due to the natural contention between VM's.

In fig 5, we investigate how the attack characteristics change when the number of attack VM's increase. We fix the number of VMs sharing the same physical machine to five. Then, we vary the attack VM from one to four. We attempt to migrate only one machine. We also fix the load for other tenants to 100 rps (requests per second). Fig. 5 shows that increasing the number of attack VMs dramatically increases migration time (e.g, by up to 4-5 folds). In addition, it can be seen that by decreasing

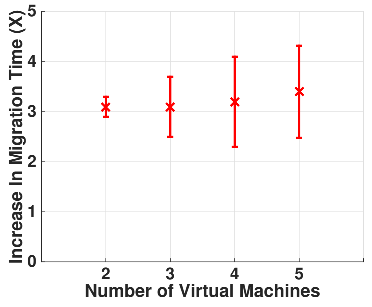

Figure 4: Increase in migration time vs. number of VMs sharing the same physical machine.

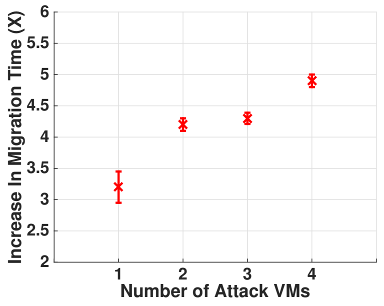

Figure 5: Increase in migration time vs. number of Attack VMs sharing the same physical machine. Total number of VMs = 5.

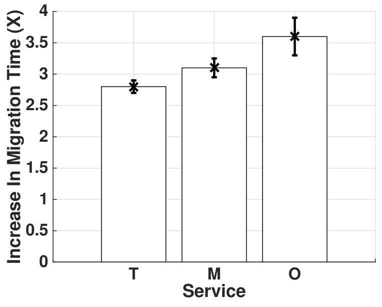

Figure 6: Increase in migration time for different services. (T) Taiga, (M) MediaWiki and (O) ownCloud.

the number of tenants and simultaneously increasing the number of attack VMs, the attack is more focused and the uncertainty shrinks to less than 10%.

Next, we demonstrate that the live migration stalling attack works consistently across different services. Here, we launch the attack against victim VMs running the three different services mentioned above. We set the rps to 500. For Taiga, we send a request to fetch the task associated with multiple projects. For MediaServer, we request different wiki pages. Finally, for ownCloud, we request different files with sizes varying from 1K to 1M. Fig. 6 shows a greater migration time for ownCloud. This is due to the fact that ownCloud is a memory hungry service.

The large variation seen in ownCloud is because the size of the web response varies greatly. Taiga and MediaServer both host static web pages that are largely the same in size and so these two services exhibit lower variance.

5.3 Flush+Reload and Suspension Attack

In this section, we investigate the efficacy of the proposed stalling attack when combined with a side-channel attack, namely, the Flush+Reload attack [4]. We use a version of Flush+Reload that was implemented in [31]. We compare the effectiveness of Flush+Reload when it is carried out independently as well as when it is used in conjunction with the proposed live migration stalling attack to show how the stalling attack increases the attack surface of Flush+Reload.

With regards to the side channel attack, we assume that the victim VM generates an 8 bit key [4]. The co-resident attacker tries to infer what that key is using Flush+Reload. The Flush+reload is considered to have succeeded if the attacker correctly deduces the victim's 8 bit key prior to the victim VM being migrated.

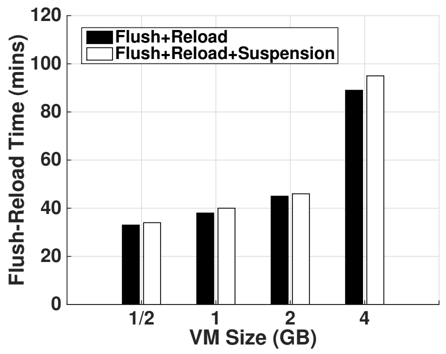

Figure 7: Average time to launch Flush+Reload with and without stalling.

Capturing the interactions between the attacks. First we show that the stalling attack (in the absence of real migrations) does not cause a significant impact on the Flush+Reload efficiency. We vary the memory size of a VM between 512 and 4096 MB. We limit the number of VMs sharing a physical machine to two (one attacker and one victim VM). We compare how long it takes the Flush+Reload attack to complete when launched with and without the stalling attack. Fig. 7 shows that using the stalling attack in conjunction with Flush+Reload slightly prolongs the completion time and incurs a penalty. This overhead can be attributed to the bus contention process which delays memory pages reloading (This can be thought of slightly slowing down everything running on the machine). However, note that this is not high, and as discussed earlier (and shown next) is insignificant compared to the increase in the efficiency of the flush and reload attack in the presence of stalling.

Improved chance of success with Flush+reload. Next, we seek to examine if the success rate of the Flush+Reload attack improves, and if so the extent to

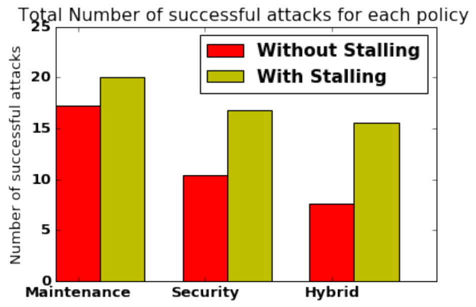

Figure 8: The total number of successful attacks, with and without the stalling attack, for each migration policy

which it does, when it is launched with a stalling attack. In this experiment, we migrate the victim VMs according to the different migration policies that were described earlier. The Flush+Reload attack is considered to have been successful if it correctly determines the 8 bit victim key (as discussed earlier). We repeat the experiment a total of 240 times. We plot the total number of successful attacks for each policy in Figure 8. We see that the stalling attack improves the odds for the Flush+Reload to succeed dramatically. With the maintenance migration policy, the stalling attack improves the likelihood of the side-channel succeeding by approximately 17%. With the security migration policy, the stalling attack improves the odds by 62%. Finally, when the defender uses the hybrid policy, the stalling attack improves the chance of Flush+reload succeeding by 105%. This indicates that the stalling attack can be a powerful attack that can be used in conjunction with a cloud-based side channel attack to thwart VM migration as a defense (as proposed in [2, 3]).

6 Discussion and Defense

In this section we suggest some migration policies that can be potentially used to thwart the stalling attack.

Recall that one of the main principles behind the stalling attack is to dirty memory pages because they will need to be re-transmitted over the network. If the page dirtying rate is greater than the network bandwidth then the migration can be stalled indefinitely.

At first glance, it would seem that the stalling attack can be trivially thwarted if there is a cap on the maximum number of pages that are transferred over the network before the migration process forcibly stops and migrates the target VM. In fact, this is the exact approach that is adopted by Xen. However, while this approach is successful in the sense that it does mitigate the stalling attack, it defeats the purpose of a *live* VM migration because the fallback is to initiate a regular VM migration.

We suggest two different migration policies. Assume the scenario where a VM needs to migrated from a source server to a target server. At the start of the migration, pages in memory are transferred to the target machine as usual. We propose a time (or resource) budget, similar to what is implemented in Xen. When the budget is exhausted, we propose that the VM be started early at the target server with an incomplete set of pages. The rest of the pages should be fetched on-demand (Note that on-demand transfer of pages has been suggested before to optimize VM migrations [34]). If an attack is under-way, the VM will be started at another server and the attacker's affect minimized.

Another simpler approach that we suggest is for the maximum amount of compute resources be allocated to the migrating VM. Such an approach is similar to existing defenses against DoS attacks where resources are diverted towards the victim of attack. As long as the compute resources allocated to the migrating VM are greater than what the attacker has, the migration can be expected to be successful.

7 Conclusion

In this paper, we construct a live migration stalling attack that exploits a KVM vulnerability to suspend or delay live VM migrations. The stalling attack is composed of two main steps: (a) invalidation of memory pages to increase the dirty page rate, and (b) creation of bus contention to induce delays. We show through extensive experiments, on our in-house cloud testbed, that the stalling attack increases victim VMs migration time by up to 3x. Potentially, the migration time for the attack VM can be stalled infinitely. More importantly, we show that side-channel attacks, such as the Flush+Reload attack, can capitalize on live migration stalling. By launching a composite attack which includes both stalling and Flush+Reload, the probability of success of the side-channel attack is improved by up to 30% in some scenarios.

Acknowledgment: The effort described in this article was partially sponsored by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Cooperative Agreement W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon.

References

[1] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[2] Soo-Jin Moon, Vyas Sekar, and Michael K Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1595–1606. ACM, 2015.

[3] Ahmed Atya, Zhiyun Qian, Srikanth V. Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa Marvel. Malicous co-residency on the cloud: Attacks and defense. In *Infocom*. IEEE, 2017.

[4] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, August 2014.

[5] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Cloud Computing*, pages 254–265. Springer, 2009.

[6] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy (S&P 2015)*, 2015.

[7] Stefan Achleitner, Thomas La Porta, Patrick McDaniel, Srikanth V. Krishnamurthy, Alexander Poylisher, and Constantin Serban. Stealth migration: Hiding virtual machines on the network. In *Infocom*. IEEE, 2017.

[8] Pradip D Patel, Miren Karamta, MD Bhavsar, and MB Potdar. Live virtual machine migration techniques in cloud computing: A survey. *International Journal of Computer Applications*, 86(16), 2014.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[10] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[11] Yi Zhao and Wenlong Huang. Adaptive distributed load balancing algorithm based on live migration of virtual machines in cloud. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pages 170–175. IEEE, 2009.

[12] Mayank Mishra, Anwesha Das, Purushottam Kulkarni, and Anirudha Sahoo. Dynamic resource management using virtual machine migrations. *IEEE Communications Magazine*, 50(9), 2012.

[13] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Empirical exploitation of live virtual machine migration. In *Proc. of BlackHat DC convention*. Citeseer, 2008.

[14] Fengzhe Zhang, Yijian Huang, Huihong Wang, Haibo Chen, and Binyu Zang. Palm: security preserving vm live migration for systems with vmm-enforced protection. In *Trusted Infrastructure Technologies Conference, 2008. APTC'08. Third Asia-Pacific*, pages 9–18. IEEE, 2008.

[15] Matt Bishop, Michael Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.

[16] William S. McPhee. Operating system integrity in os/vs2. *IBM Systems Journal*, 13(3):230–252, 1974.

[17] Kai Hwang and Deyi Li. Trusted cloud computing with secure resources and data coloring. *Internet Computing, IEEE*, 14(5):14–22, 2010.

[18] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 101–110. ACM, 2009.

[19] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 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*, pages 199–212. ACM, 2009.

[20] Yuval Yarom and Katrina E Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

[21] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.

[22] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security 15) Washington, DC*, pages 913–928, 2015.

[23] Amazon EC2. T2 Instance Requirements. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html`, 2016.

[24] Kernel Virtual Machine. `http://www.linux-kvm.org/page/Main_Page`, 2014.

[25] Trusty Tahr. Ubuntu 14.04.3 LTS. `http://releases.ubuntu.com/14.04/`, 2014.

[26] Centos 7. `https://www.centos.org/`, 2014.

[27] Apache CloudStack. Open Source Cloud Computing. `https://cloudstack.apache.org/`, 2016.

[28] Taiga. `https://taiga.io/`, 2016.

[29] ownCloud. `https://owncloud.org/`, 2016.

[30] MediaWiki. `https://www.mediawiki.org/wiki/MediaWiki`, 2016.

[31] Daniel Ge, David Mally, and Nick Meyer. Implementation of the FLUSH+RELOAD side channel attack. `https://github.com/DanGe42/flush-reload`, 2016.

[32] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):10, 2010.

[33] CD Lai, Min Xie, Kim-Leng Poh, Yuan-Shun Dai, and P Yang. A model for availability analysis of distributed software/hardware systems. *Information and software technology*, 44(6):343–350, 2002.

[34] Constantine P Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. *ACM SIGOPS Operating Systems Review*, 36(SI):377–390, 2002.