Co-residency Attacks on Containers are Real

Sushrut Shringarputale
Microsoft
sushring@microsoft.com

Patrick McDaniel
The Pennsylvania State University
mcdaniel@cse.psu.edu

Kevin Butler
University of Florida
butler@ufl.edu

Thomas La Porta
The Pennsylvania State University
tfl12@psu.edu

ACM Reference Format:
Sushrut Shringarputale, Patrick McDaniel, Kevin Butler, and Thomas La Porta. 2020. Co-residency Attacks on Containers are Real. In *2020 Cloud Computing Security Workshop (CCSW'20), November 9, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3411495.3421357

Abstract

Public clouds are inherently multi-tenant: applications deployed by different parties (including malicious ones) may reside on the same physical machines and share various hardware resources. With the introduction of newer hypervisors, containerization frameworks like Docker, and managed/orchestrated clusters using systems like Kubernetes, cloud providers downplay the feasibility of co-tenant attacks by marketing a belief that applications do not operate on shared hardware. In this paper, we challenge the conventional wisdom that attackers cannot confirm co-residency with a victim application from inside state-of-the-art containers running on virtual machines. We analyze the degree of vulnerability present in containers running on various systems including within a broad range of commercially utilized orchestrators. Our results show that on commercial cloud environments including AWS and Azure, we can obtain over 90% success rates for co-residency detection using real-life workloads. Our investigation confirms that co-residency attacks are a significant concern on containers running on modern orchestration systems.

1 Introduction

Cloud computing adoption has grown exponentially in the last decade, with revenue of $175 billion in 2018 [19] and $206.2 billion projected in 2019. Several diverse industries have begun migrating workloads to the cloud including healthcare, financial services, technology, and insurance [19]. Dependence on cloud services continues to increase as all companies, from small startups to large multi-national corporations and governments, store and process their data on the cloud. The data handled by many of these organizations can be highly sensitive. In most cases, customers run their code on the same physical machine as others (a state called *co-residency*). Those other customers could include potential adversaries, and this co-residency exposes a major attack surface. Despite the isolation between processes and virtual machines, shared access to hardware resources introduces security and privacy concerns for tenants running on the same hardware.

Several recent events have shown that such vulnerabilities are being exploited by adversaries. Reported attacks against data-centers include NordVPN's data-center breach, which took place on a third-party provider, leading to leakage of TLS keys [34]. Misconfiguration by Tesla of a Kubernetes cluster led to a cryptojacking attack and exposed secret access keys to an AWS account, leading to attackers launching crypto-currency mining software on Tesla's compute resources [43]. In cases where misconfiguration was not the issue, a rarely disclosed datacenter breach showed a targeted attack where the adversaries employed Trojan malware to compromise websites hosted on a Central-Asian datacenter [33]. Cases like these show that attackers are employing sophisticated mechanisms to attack data-centers with specific targets. As providers patch misconfiguration vulnerabilities, the adversaries are bound to turn towards attacks on the virtualization layer. Zero-day vulnerabilities have been publicly exploited in virtualization software in controlled environments [37], which are bound to make their way into actual exploits. To locate targets then, we suspect that attackers will implement co-residency detection on cloud environments as a major attack vector.

So far, Cloud Service Providers (CSPs) have adopted hypervisors, e.g., *KVM*, container runtimes, e.g., *Docker*, and orchestration systems, e.g., *Kubernetes* to streamline access to cloud resources. CSPs depend on low-layer abstractions such as hypervisors and container sandboxes [6] to provide security from co-residency attacks. The processes themselves run using abstractions that elide a shared-hardware view: providing an illusion that the VM and processes share no part of the hardware with others. A study by Eder [23] claims that the combination of container-based isolation and hypervisor-based virtualization may provide an improved security model. The direction of engineering and research in the industry indicates that this belief is widespread. The development of orchestration systems regularly downplays the potential impact of co-residency. CSPs are introducing multi-tenant, managed Software-as-a-Service implementations of container orchestrators that make running applications easier for customers [4, 44]. However, these implementations share the cluster-level abstractions between tenants, making co-residency more likely.

The growth of such orchestration systems has fundamentally changed the way cloud platforms manage virtualized computation.

ACM ISBN 978-1-4503-8084-3/20/11...$15.00
https://doi.org/10.1145/3411495.3421357

In these systems, containers become the fundamental unit of computation, rather than customers managing discrete VMs per application. The marketing literature for containers are largely driven by security. They are rapidly poised to replace VMs, made evident by the development of Kata containers [1], etc., which claim to have the security of VMs while maintaining low overhead.

Research by Atya et al. [14, 15], Bazm et al. [18] have confirmed that virtual machines are vulnerable to co-residency attacks. The structure of containers induced speculation that similar vulnerabilities also exist for containers, even when running within virtual machines and orchestration systems. For example, Gao et al. [26] showed that the way containers mount the `procfs` pseudo file system exposes many exploitable side-channels. Indeed, Google also suggests that containers should not be trusted as a security boundary due to the lower isolation provided by the containers [12]. Moreover, the authors suggest that the ideal isolation boundary should include a hypervisor to be the most secure boundary possible. Research has still not shown however, whether this secures against co-residency attacks. [1]

This paper is the first to challenge the conventional wisdom that containers within VMs and orchestration systems are secure against co-residency attacks. We adapt the co-residency watermarking attack [17] which uses hardware side-channels on VMs to establish whether an adversary is co-resident with its target. We show that the attack is also successful when deployed against containers running on virtual machines, even within commercially deployed orchestration systems.

To analyze the effectiveness of the attack, we perform experiments on real-life workloads on a variety of system configurations. These range from structured environments (*Local*, *Local Separated*, *Quiet Cloud*, *Noisy Cloud*) to real-world clouds, e.g., AWS[7], and Azure[8]. We measure the sensitivity of the attack to architecture, load, orchestrator selection, and adversarial activity. Using modern orchestrator implementations (*Docker Swarm (DS)*, *Kubernetes (K8)*, *Helios*), we show that variations in these choices have little effect on the success of such an attack. Our evaluation yielded results with 90% success rates on commercial cloud environments.

Finally, we analyze how inconspicuous an adversary can be during the attack and show that an adversary can detect co-residency with little work, making them much harder to detect.

We make the following observations in this paper:

- We empirically confirm that systems running on containers within virtual machines are vulnerable to co-residency attacks, with over 90% detection rate on cloud systems.
- We demonstrate that co-residency detection can tolerate background noise at a 70% success rate, as long as it does not exceed hardware capacity.
- We show that changes to the architecture and choice of orchestrator reduce the fidelity of detection by at most 10%, demonstrating little effect on adversarial success.
- We analyze the amount of adversarial work necessary for the adversary to be successful in detecting a target and confirm that an adversary can be successful by generating interference without reaching the hardware capacity, around 70% of the capacity in one of our experiments.

2 Background

Orchestration platforms have ushered in a new way of virtualizing execution on shared cloud systems. They provide a way for administrators to deploy their applications at a cluster-level abstraction. The primary goal of orchestration platforms is to orchestrate the placement, scaling, and lifecycle of applications in a distributed manner, providing high availability and scalability. This removes the effort needed to plan how the applications run. As a result, large scale applications are adopting these strategies because fast scaling, cost-effective usage, and lower downtimes help make their systems much more reliable. Applications have been growing in size and complexity, requiring layers of authentication, web-capabilities, integration with other platforms, caching, and storage. To make these applications easier to maintain, engineers are developing and deploying them as smaller units, or *micro-services*, that can be scaled and managed independently.

Google's Borg project led to the development of the Kubernetes orchestrator [46] which helps automate assigning containers to physical machines (nodes). Also, the platform performs virtualization at the compute, device, and network levels to simplify application development. Other developers have invested significant effort into such systems, e.g., Docker Swarm [21], Spotify's Helios [42], and Apache Mesos [13].

2.1 Prior work

Varadarajan et al. [45] describe performance covert-channel co-residency detection as: If an adversary is co-resident with the victim and is generating contention that leads to measurable performance degradation for the victim, it can be measured by the adversary to detect co-residency. Prior works have used various methods to analyze this drop in performance on various attack surfaces. When generating contention in the memory bus [45], the authors compared the means and medians of the distributions, taking into account the noise of the system when the adversary was not active. Building on this methodology, when detecting co-resident VMs in VPC enabled clouds, Xu et al. [49] thresholded the latency of a traceroute to a possible candidate VM. Finally, Bates et al. [17] generated contention in the NIC and measured the difference in distributions of the travel time for packets that were watermarked and ones that were not. They measured the difference in these distributions using the Kolmogorov-Smirnov statistical test, confirming co-residency if the test rejected the null hypothesis. All of these attacks used virtual machines as the sole isolation boundary; our work adapts these in containerized environments and shows that such isolation is still not enough to prevent co-residency attacks. We also show that a simpler detection strategy (Mean Square Error) is sufficient to prove co-residency in most realistic systems.

2.2 Containers

Micro-services demand that each running service can start and stop quickly (scale), have a small footprint, and automate easily.

[1]Despite this, Alibaba cloud claims that their Container Service for Kubernetes "Ensures end-to-end application security and provides fine-grained access control" [9], while IBM Cloud claims âĂIJHighly secure environment for production workloads with built-in container-level securityâĂİ [10]. These statements are representative of the security-related claims about containers made by cloud service providers.

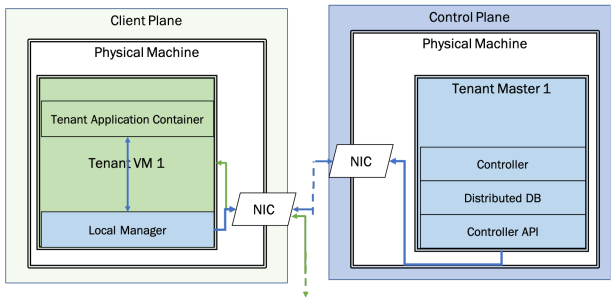


Figure 1: Components of orchestration platform

This makes virtual machines a very poor choice. Creating a VM per application would require a very long setup time, requiring a new initialization of the operating system per service instance, leading to much higher resource usage. To solve this problem, the industry is rapidly switching over to a new type of process-isolated runtime for applications. This runtime uses Linux namespaces, control groups, and the Union File system (OverlayFS [22]) to make the application system-agnostic as well as easy to spin-up and isolate. This runtime defines the process as a *container*, which contains all the libraries necessary to run the application. By building the blueprint for this ahead of time, the spin-up time and resource usage for containers tend to be very low [32]. Due to these reasons, containers are becoming the standard unit of computation within micro-services.

2.3 Orchestrator implementation

Most orchestrators only differ in architectural structures (See Figure 1). The core concept includes a coordinator application (Master) that communicates in a distributed way with all the physical nodes (VMs) running the client's applications. The coordinator uses a distributed database that stores the latest state of the cluster of client containers. The state includes which containers need to run, how many instances of a container should be active, which node they should run on, and other metadata. All these administrating pieces together constitute the *control-plane* of the cluster.

The control-plane is responsible for continuously monitoring the cluster, so it implements health checks that verify the states of all container instances using periodic HTTP calls. The control-plane also implements a cluster network to allow containers to communicate with each other. Each client node runs a local manager that the coordinator communicates with. The local manager manages the container runtime and keeps track of the running containers and their health. Finally, it provides the firewall rules and port mappings necessary to implement the cluster network subnet needed by the containers running on that node. Utilizing the health checks, if a container is in a non-ideal state, the control-plane takes steps to fix that. In most cases, the control-plane is managed by the cloud-provider and is physically separated from client applications.

We looked into the source of three orchestrators listed above: Docker Swarm, Kubernetes, and Helios. DS and K8 expose HTTP APIs for communication between the coordinator and nodes, initiated by the coordinator. Thus, the majority of computation happens there. On the other hand, Helios implements a poll-based paradigm, where each node queries the state database directly to compute the action needed for the cluster to reach the "goal state". Since

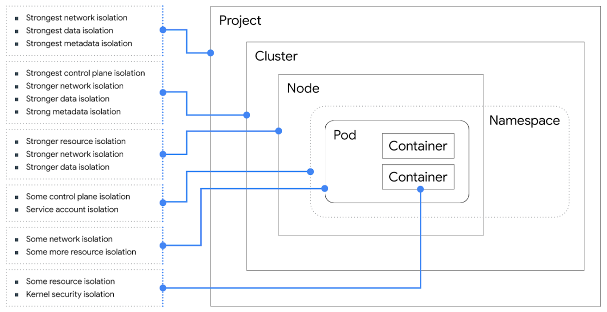

Strongest network isolation
Strongest data isolation
Strongest metadata isolation
Strongest control plane isolation
Stronger network isolation
Stronger data isolation
Strong metadata isolation
Stronger resource isolation
Stronger network isolation
Stronger data isolation
Some control plane isolation
Service account isolation
Some network isolation
Some more resource isolation
Some resource isolation
Kernel security isolation
Project
Cluster
Node
Namespace
Pod
Container
Container

Figure 2: Isolation provided by all layers in K8 [12]

the orchestrators are critical to the operation of the cluster, best practices suggest that they are separated from the actual compute nodes by administrators [24].

2.4 Virtual machines and isolation

CSPs such as Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform, etc., provide services to their customers with a set of guarantees (i.e., , service level agreements). While it is evident to customers (i.e., *tenants*), that they are sharing hardware with other tenants, the cloud platform also provides certain isolation guarantees for code running on these systems. More specifically, the operating system provides guarantees between processes that ensure that the memory access is isolated. Access Control mechanisms in the operating system provide isolation at file and device levels. Similarly, hypervisors provide isolation to virtual machines running complete operating systems. These isolation guarantees are supposed to ensure that tenants' code will not be adversely affected by others from a security, performance, and privacy perspective. Since each layer of isolation describes a new attack surface, each layer must be properly secured.

The Kubernetes (K8) project describes the isolation provided at each layer, seen in Figure 2. K8 wraps the containers in an abstraction called the Pod (a collection of processes), which are isolated from other processes by a Linux namespace. The VM running these is called the node. In this paper, we focus on the isolation guarantees provided by the node layer and below. At this layer, all the contention is generated in the hardware of the physical machine, however, orchestration platforms claim that multiple layers of isolation, including Kernel security, network, resource, and data isolation exist among all the layers below the Node.

2.5 Co-residency Attacks

A co-residency attack considers an active, malicious adversary that has no affiliation to the cloud provider [39]. The adversary appears as an innocuous tenant to the CSPs with no elevated privileges on the actual machine. They access the standard interface for launching instances, same as the victim, are free to launch as many VM instances on the cloud service as they want (barring some limits for specific CSP), and can choose the hardware configuration the instances will have. This hardware configuration is applied at the VM level, however, certain CSPs tend to restrict the type of VM instance that can be placed on a specific type of machine [5]. This means that the adversary can choose from a subset of physical machines in some cases. For example, if an adversary decides to use *m2.large* instances on AWS, they will most likely be placed on machines specified for *m2.large* instances [45]. As a result, co-resident placement becomes much simpler. Zhang et al. [51] shows that influencing co-tenancy with a 90% probability takes only 20 instances on a chosen instance type.

The victim is a legitimate cloud tenant performing security-sensitive operations using hardware shared with the adversary. The victim has the same capabilities for launching instances as the adversary. Both members in this model trust the cloud provider and have little to no insight into the placement policies of the datacenter. Additionally, the adversary gains little information from metadata provided to the instance such as the IP address. While this information was useful in older attacks for cloud cartography [39], this is no longer viable [45] on current cloud architectures due to the introduction of Virtual Private Clouds, where all user VMs are launched in a logically isolated section of the cloud, giving the user the ability to pick their IP addresses [49].

Instead, the adversary uses one of the available side channels to gain unauthorized access to some information belonging to the victim. This can range from knowledge of existence [17, 39], to application data [30], to cryptographic keys [52]. Gao et al. [26] show the extent of channels that can be used by containers on cloud providers, including benign information such as power usage.

Most of these studies looked at the isolation provided by virtual machines on clouds, assuming that each application runs within its virtual machine. However, containers and orchestrators seem to provide an added abstraction layer that may improve the isolation guarantees. Indeed Eder [23] claims that the addition of containerization-based isolation within hypervisor-based virtualization may be an improved security model. The direction CSPs have taken with their development of orchestration services seem to certainly imply that this is the case [4, 44]. For example, managed orchestration clusters support multiple tenants on an identical set of virtual machines, vastly increasing the possibility of co-residency. Honig and Porter [29] claim that the KVM hypervisor is secure enough to protect from side-channel attacks since the vulnerabilities in libraries are patched by the developers quickly. Allclair and Kaczorowski [12] from Google suggest that the layers of isolation in Kubernetes generate enough security for containers on VMs and orchestrators. Recent research by Atya et al. [14] shows that this is certainly not the case for VMs.

Sources intuit that containerized isolation is not secure, even while running within virtual machines and orchestrators. Edwards et al. [24] recently completed a security review of all systems within the Kubernetes architecture and outlined some of the major attack vectors and best practices relating to clusters running on Kubernetes. They note that multi-tenant clusters are a major attack surface for orchestrators today. However, their definition of multi-tenancy differs from ours in the following way: they define a multi-tenant cluster as an orchestrated collection of containers deployed on compute nodes that have containers deployed by multiple tenants. This may induce co-resident multi-tenancy, but the attack surface looks at access control and over-privilege as the main issues to protect from. Our definition deals with applications that are assigned the same physical node to containers deployed by two unrelated tenants on *different clusters*. As a result, the attack surface we investigate deals with adversaries circumventing the combined isolation guarantees provided by the hypervisor and container runtime using some side-channels.

3 Co-resident Watermarking Attack

The goal of this study is to ascertain that containers running on virtual machines are also vulnerable to co-residency attacks, regardless of system configuration. To achieve this goal, we picked a representative attack that requires a reasonably low setup cost. The attack attempts to determine if the adversary's container is running on a VM that is co-resident to their victim's container. We then analyze the various sensitivities of the attack to the architecture, system load, orchestrator choice, and adversarial activity.

For measuring architectural sensitivity, we run the attack on a quiescent setup where only the adversary and the victim are active. With this setup, we can show that the attack is possible and ensure that the measurements are affected by external interference. After this, the attack is run on a cloud environment to analyze the effect of other tenants on the success of the attack. Similar to realistic cloud environments, we analyze the effect of other load in the system to the attack. In addition to the victim and adversary, we analyze the attack's success when other processes and tenants are demanding a high amount of CPU and network bandwidth.

The third sensitivity we measure is the choice of orchestrator. We show that the presence and choice of an orchestrator plays a minimal role in increasing resiliency against the attack. Finally, we vary the amount of activity the adversary performs, i.e., quantity and duration of interference generated, and measure how deceptive an adversary can be while successfully detecting the co-resident victim.

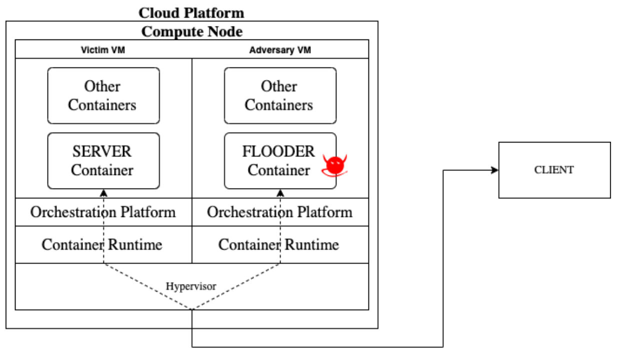


Figure 3: Attack setup: `FLOODER` and `SERVER` are co-resident while the `CLIENT` communicates with them externally

3.1 Attack Phase

We use the co-resident watermarking attack [17], which works against public-facing web servers to detect when a malicious VM is running co-resident to the server. The setup for this attack is similar to the one implemented for Tor traffic analysis [38]. For this paper, we found that the watermarked network-flow detection attack was sufficient as it runs very quickly (<2 minutes for each run) and has a higher success rate than other attacks. In our setup, we have two applications running inside containers that play the following roles: the `SERVER` and the `FLOODER` as shown in Figure 3.

```
set p as period
set d as duration
notify_client()
loop for d seconds   // On sequence
    notify_client()
    loop p seconds:  // Watermarking Sequence (On-WM)
        send_burst_packet()
    end loop
    notify_client()  // Notify end of watermarking sequence
    sleep(10-p)      // Idling Sequence (On-Idle)
    notify_client()  // Notify end of idling sequence
end loop
notify_client()      // Notify Start of off sequence
loop while client.active()
    sleep(p)         // Watermarking Sequence (Off-WM)
    notify_client()  // Notify end of watermarking sequence
    sleep(10-p)      // Idling Sequence (Off-Idle)
    notify_client()  // Notify end of idling sequence
end loop
```

Algorithm 1: Flooder implementation

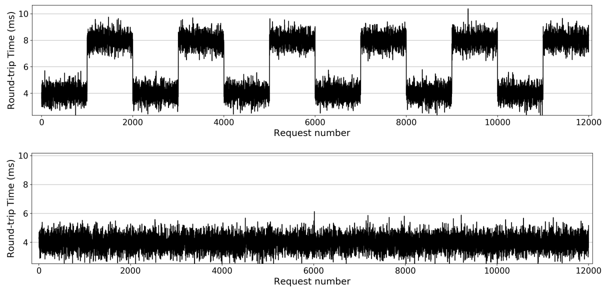


Figure 4: Expected trace of RTTs: The trace shows a square wave (top) when the FLOODER causes interference (*On* sequence), while a near constant RTT when the FLOODER is not causing interference (bottom) (*Off* sequence)

The third application, the CLIENT runs on an external machine and connects over the network to both the SERVER and the FLOODER.

The SERVER is the modeled victim in this attack and has no knowledge of the adversary. It listens for web requests, which for simplicity were restricted to one type, where the SERVER sends a 10 MB file in chunks of 8 KB.

The CLIENT is controlled by the adversary and is aware about the co-residency detection attack in progress. It maintains communication with the adversary that is potentially co-resident with the SERVER, and it measures round trip time (RTT) for requests made to the SERVER. The CLIENT continues to make requests to the SERVER until the adversarial container i.e., FLOODER notifies completion.

Finally, the FLOODER is a container that generates interference in the network interface card (NIC) of the physical machine by flooding it with large amounts of data. To minimize the interference from the hypervisor, the FLOODER signals the CLIENT, runs for period p, signals the CLIENT, and idles for the next $10 - p$ seconds. The code for FLOODER is provided in Algorithm 1.

The FLOODER runs in two sequences: *On* and *Off*. During the *On* sequence, the FLOODER runs in a loop for d seconds. In this loop, the first p seconds are the watermarking sequence (*On-WM*), while the idling sequence (*On-Idle*) is made up of the following $10 - p$ seconds. During *On-WM*, the FLOODER generates large bursts of traffic sent out of the machine, approaching the capacity of the NIC. Due to the generated contention in accessing the NIC, we expect that the Round Trip Time (RTT) for the CLIENT's requests should increase.

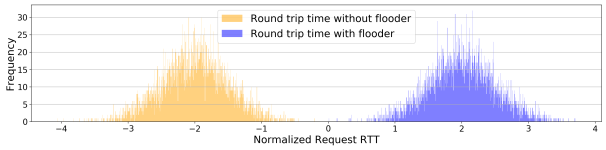


Figure 5: Expected histogram of RTTs: The histograms show clear separation when the FLOODER causes interference (*On* sequence)

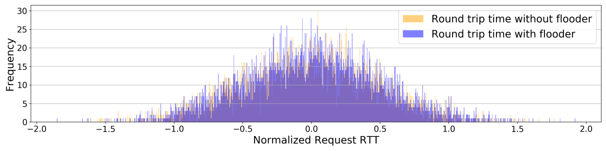


Figure 6: Expected histogram of RTTs: The histograms look identical when the FLOODER is not causing interference (*Off* sequence)

The FLOODER signals the CLIENT the end of the *On* sequence and goes into the *Off* sequence. This sequence is useful to obtain a model of the environmental noise (load l), which is factored out during the co-residency detection phase. During this sequence, the CLIENT makes the same RTT observations with signals from the FLOODER. The FLOODER sleeps for p seconds (instead of actually sending packets), which is the *Off-WM* sequence. After notifying the client, the FLOODER sleeps again for $10 - p$ seconds, which is the *Off-Idle* sequence. We expect that the RTTs should be nearly identical during the entire *Off* sequence. The expected RTT trace for both the *Off* and *On* sequences is shown in Figure 4. Using the collected RTT trace, the CLIENT analyzes whether the FLOODER is actually co-resident with the SERVER.

3.2 Co-residency Detection Phase

For detection, we found that applying the KS-test to round trip times does not work. Unless the measured values generate near ideal distributions (Figure 6), the KS-test has a very high false-positive rate due to the high variance in RTTs. For this paper, it was sufficient to use a simple distribution distance metric such as the Mean Square Error (MSE) between the histograms for *On-WM* and *On-Idle*. We calculate the MSE as follows: for each data point in the RTT histogram (Y) when then FLOODER was active, we average the squared difference with the corresponding RTT histogram ($\hat{Y}$) value without the FLOODER active. Histograms where the FLOODER has generated enough interference should be separate, as demonstrated in Figure 5. In contrast, when the FLOODER is not co-resident, the distributions will be nearly identical, as shown in Figure 6. Thus, the MSE value should be high when the FLOODER is co-resident and close to zero otherwise.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \quad (1)$$

The final piece of detection is generating the threshold MSE value. We need to determine the threshold to identify which configurations are co-resident. In the case of the ideal clean square wave, that is trivial. Any MSE value greater than zero should be a successful detection since there will be no effect generated when the adversary is not co-resident. However, during our evaluation, we found that as the system gets noisier, the distributions for the *On-WM* and *On-Idle* sequences get closer together while the distributions for *Off-WM* and *Off-Idle* show pseudo-separation. This makes it difficult to determine a ground-truth threshold value for the MSE.

To determine and eliminate the effect of this background noise, we use a model of the background noise by obtaining an RTT trace when the FLOODER is inactive (*Off* sequence). This technique is similar to the technique used when determining co-residency based on memory bus access contention [45]. Using the *Off* sequence, we generate a baseline MSE value for the system without the activity of the FLOODER. Since the *Off* sequence is obtained immediately after, it should provide reasonable measurement of the background noise. We subtract this value (MSE_{off}) from the (MSE_{on}) and use the result to generate the correct threshold.

$$aMSE = MSE_{on} - MSE_{off} \tag{2}$$

Since the model of the background noise may change on each system we evaluate, we run a controlled experiment where we assume that the adversary has access to the SERVER and can force co-residency. Using the *aMSE*value from these runs, we generate the Receiver Operating Characteristic (ROC) curve, find the Pareto point on the curve given by Equation 3, and obtain the optimal threshold value ($aMSE_0$) for that system configuration.

$$aMSE_0 = argmax(TruePositive - FalseNegative) \tag{3}$$

4 Evaluation

As part of the evaluation, we try to answer the following:

- Is co-residency detection possible when running containers on virtual machines?
- Is co-residency detection affected when containers run on varying system configurations and how?
- Do orchestrators play a role in the success of co-residency detection? Which components of orchestration software play the largest role?
- Do system load and architecture affect success metrics?
- How deceptive can an adversary be before co-residency detection is not possible? What is the least amount of data needed by the adversary for the attack to succeed?
- Is the signal obtained from the side-channel affected by duration of contention?
- Does network-watermarking co-residency detection react to the vendor of orchestrator? If yes, why?

For each measured sensitivity, we establish the experimental variables show in Table 1.

Variable Name	Symbol	Description
Period of sequence	p	How long the flooding sequence (*On-WM*) lasts
Detection threshold	$aMSE_0$	The threshold to detect co-residency
Load	l	The load in the system on the shared hardware
Orchestrator choice	o	The chosen orchestrator
Duration of run	d	How long each run takes

Table 1: Experimental Variables

4.1 Experimental setup

We performed our experiments with three different setups: the first emulated a small, simple cloud environment, the second mimicked a realistic cloud provider, and the third used commercial cloud providers with dedicated hosts. We chose not to run any of the sensitivity analysis on the commercial cloud provider due to ethical reasons, ensuring that uninvolved third parties were not affected.

4.1.1 *Simple setup*

Local Our simple setup involved just two physical nodes capable of running workloads. We set up two orchestration clusters: one for the target container (SERVER) and the other controlled by the adversary. Each cluster had its coordinating application, i.e., Master, running on the same node, meaning that the Master node VM for each cluster was allowed to run client containers directly. This allowed us to generate a footprint for the amount of computation and IO interference generated by the Master, which is a significant piece in any orchestration system. Since the role of the Master is to simply designate a specific node (including itself) to run the container, perform health checks, and virtualize access to other portions of the system, we expected that once an application has been successfully assigned to a node, i.e., scheduled, the orchestration system should generate minimal interference in the actual containers' performance.

4.1.2 *Realistic setup*

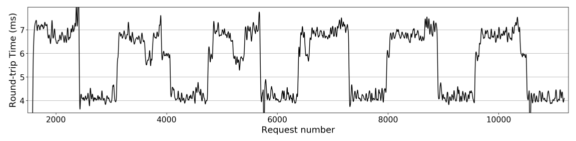
Local Separated Running client containers on the Master node does not reflect realistic cloud system setups. Cloud providers generally handle the setup for the Master node(s) and keep them separate from the user's applications [3]. To mimic this, we separated the Master nodes and CLIENT onto different physical machines. With this, we were able to isolate the victim and adversary with no other system-level interference (barring the OS and hypervisor).

In such a setup, the only piece of software from the orchestration platform that runs on the same machine as the application is the node manager. This software only responds to health-check requests and ensures that the application container is healthy. We expect lesser interference from the orchestration here and the attack should be much more robust.

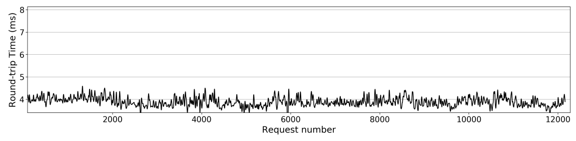
Quiet Cloud We created the same setup as above in a realistic, cloud-like environment. The Master node ran on separate hardware, with only the local manager running alongside the client applications. On this setup, other tenants did not share the physical hardware with the SERVER and FLOODER, however, other tenants were active in the local network, generating more noise than the *Local and Local Separated* setups.

Commercial Cloud We tested the feasibility of this attack on two of the biggest commercial cloud providers, Amazon Web Services (AWS) [7] and Microsoft Azure [8]. We used dedicated hosts to ensure that (1) The contention generated was not affected by any tenant not involved in the experiment and likewise, (2) the attack did not affect any real tenants. For these reasons, we also chose not to perform any sensitivity analysis on the commercial cloud platforms, instead, we use the *Quiet Cloud* environment controlled by us.

4.1.3 *Non-quiescent environment*



(a) Measured RTT during a flooding sequence: The flooding sequence generates a square wave watermark that confirms co-residency



(b) Measured RTT when adversary is not co-resident: The flooding sequence does not generate a square wave watermark

Noisy Cloud We also ran the attack in a setup that includes a lot more system noise on the victim. To emulate this, we used the popular benchmarking tool Yahoo! Cloud Serving Benchmark (YCSB) [20] to continuously benchmark Memcached, a fast, persistent key-value store. This generates many disk-IO and network calls on the system. Within this environment, we kept the Master node separate to ensure that the load in the system is predictable.

In addition to the YCSB benchmark, we use a realistic server trace of traffic as background noise to evaluate how realistic load in the system will affect the detection rates. We use a recent trace (April 2019) from the MAWI dataset [41] to mimic background traffic. We adjust the multiplier on the trace replay to increase the contention in the NIC due to noise.

The local setups (*Local* and *Local Separated*) for experiments used two Dell Precision T7600 machines with identical internal configurations. They had 32 GB RAM, six-core Intel Xeon E5-2630 processor, Ubuntu 18.04, and were connected to a local network over Ethernet with a Cisco Linksys E1550 router. The NIC for the target machine had a 1 Gbps Ethernet connection. A third machine, a 2013 Mac Pro was used as a packet sink and existed on the local network, but took no computational part in the actual attack. The KVM hypervisor was the virtualization layer, which is the standard in industry [2].

The experiments were repeated on a cloud-like cluster (*Quiet Cloud* and *Noisy Cloud*) which contains sixteen compute nodes. Each node is a Dell PowerEdge M620 machine running eight-core Intel Xeon E5-2609 processors with 64 GB RAM. All nodes were running Ubuntu 18.04 Server and were connected to the cluster network. Each node communicated through a central NIC for the cluster, which had a bandwidth of 1 Gbps. Any node that required virtualization used the KVM hypervisor like above. The cluster is currently in use by other researchers to run various workloads and applications, making it similar to an Infrastructure-as-a-Service (IaaS) cloud service. Other tenants of the cluster were accessing some nodes on the same shared network, leading to slightly higher background noise than *Local Separated* and *Local*.

4.2 Co-residency Detection

Recall that the job of the FLOODER was to generate interference in the network interface (NIC) of the physical machine by flooding it with large amounts of data. In this experiment, we fixed the value of

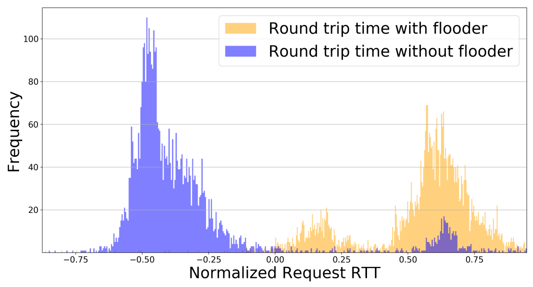


Figure 8: Normalized RTT histogram for a flooding sequence: The dark histogram has clear separation from the light, determining co-residency

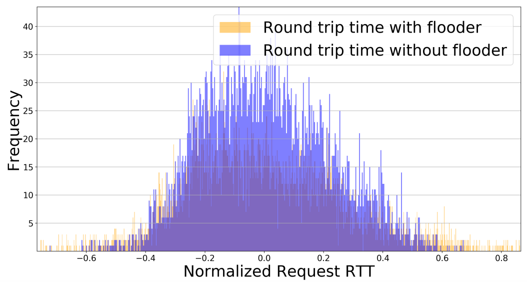


Figure 9: Normalized RTT histogram for a control sequence: The dark histogram has no separation from the light, showing no adversarial interference

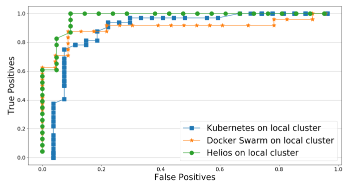
the period p to 5 seconds. The setup was on the *Local Separate* setup. The result of this can be seen in Figure 7a. Every time the FLOODER ran, the measured RTT from the SERVER increased as expected. This sequence continued for a minute, after which it signaled the CLIENT and went into the *Off* sequence, where it continued the above cycle without actually sending any data. During this period, the RTTs show little to no pattern as seen in Figure 7b.

During the co-residency detection, we compared the histograms of the two sequences. Figure 8 shows a clear separation in the mean RTTs. The histogram during the control sequence, as seen in Figure 9, has nearly no separation and shows that the adversary is not co-resident. For each setup, we calculated the $aMSE_0$ threshold value from the ROC curves using Equation 3, seen in Figure 10.

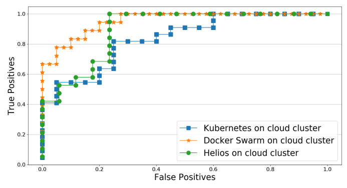
4.2.1 *Local and Local Separated*

We ran this experiment 20 times on each setup and created the confusion matrices for all setups and orchestrators. This section presents and compares the success rates for detection.

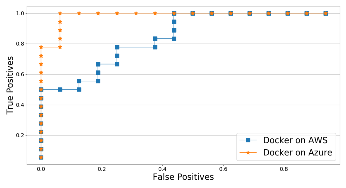
Initially, the local setup was very simple. The orchestrator's coordinator shared the physical node with the application containers, the effect of which is described in much more detail in section 4.5. Confusion matrices for this run are presented in Table 2. As the orchestrator's coordinator is generally managed by the cloud provider, so the effect of the coordinator should be extracted away from the evaluation. The interference generated by the coordinator seemed to be reacting to external traffic in the system. As a result, when the FLOODER was active during *On-WM*, the RTTs took much longer, generating very high success values. While this shows that the attack is possible, the results seem unrealistically high.



(a) ROC curves for controlled experiment across different orchestrators on the local cluster



(b) ROC curves for controlled experiment across different orchestrators on the cloud cluster



(c) ROC curves for controlled experiment across different commercial clouds

Figure 10: ROC curves across different system configurations

		Configuration							
		Local		*Local Separated*		*Quiet Cloud*		*Noisy Cloud*	
		True Positive	False Positive	True Positive	False Positive	True Positive	False Positive	True Positive	False Positive
Orchestrator	K8	100	38	94	15	91	35	35	36
	DS	100	22	92	9	72	5	58	29
	Helios	100	21	100	10	88	18	45	47

Table 2: Successful detection rates across orchestrators and configurations

Once we separated the coordinators onto different machines for the realistic setup, the noise in the system became extremely low and provided realistic success rates for detection. Table 2 describes the confusion matrices generated for each of the orchestrators.

Due to the Master containers being co-resident, we saw that the ideal MSE threshold value determined for this setup was able to get great separation between true positives and true negatives. As a result, the prediction rate here was extremely high. While running the containers co-resident, the response to the noise generated by the Master was extremely high on Helios and Docker, creating very high MSE values for most runs. The same effect was not observed when the Master containers were moved to an independent node. The MSE values for co-residency detection were in a much tighter range, leading to a higher number of false positives and lower success rates. However, even in this state, the success rates were reasonably high - close to 95% for all orchestrators. This detection was carried out with an $aMSE_0$ value of 25.

Summary: Co-residency detection is possible with high accuracy and precision on containers within virtual machines, even when deployed on container orchestration systems.

4.2.2 Cloud Environment The first experiment ran on a semi-quiescent network (*Quiet Cloud*) where the network backbone was only shared at a switch level and no VM or user shared the CPU and RAM allocated for this experiment. The success rates for this run are presented in the confusion matrices in Table 2. Success rates on all three orchestrators on this cloud-like environment are high and show that co-residency attacks are significant.

Commercial Clouds: We ran the same experiment on two commercial clouds: Amazon Web Services and Microsoft Azure. Our results are listed in Table 3, calculated with a threshold value 0. The primary difference between these providers is that AWS uses a modified KVM [2] hypervisor while Azure uses the proprietary Windows Azure Hypervisor. The differences between these cause variations in how the FLOODER's activity affects the priority of the SERVER.

The reason for higher success rates on commercial clouds is that the allocated system had high CPU and memory availability (96 vCPUs and 384 GB memory), but only a small amount of that was used by each VM. As a result, the only contention generated is in the target's NIC making the attack's signature much clearer to the adversary. Regardless, the success rates for both providers make it clear that commercial cloud systems are highly vulnerable to co-residency attacks.

	AWS	Azure
True Positive	80	94
False Positive	25	6

Table 3: Detection rates on commercial cloud systems

Noisy Cloud: The second run ran the YCSB benchmark in the SERVER container in the background. This generated load that made the system non-quiescent. The results of this run are described in the confusion matrices in Table 2.

The addition of the noise on this setup generates an expected drop in the success rates. Because this drop varies, it shows that the orchestrators are variably sensitive to noise in the environment.

Finally, we also investigated the reasons behind the failures during detection on the previous setups. Some platforms seemed to have a much higher failure rate than others. In general, we found that during the cases when the detector failed, the FLOODER was throttled by the KVM scheduler. In such cases, the CPU time that the FLOODER VM ran for was about 20x higher than the SERVER VM.

In our initial solution to this problem on the local cluster, we added a cooldown to the FLOODER so that it may gain back scheduling priority, but since the SERVER was also inactive during those times, it had little effect. To create more realistic CPU usage by the server, we chose to add more CPU usage to the SERVER's VM by making it read from /dev/urandom. As a result, the FLOODER VM's priority with the scheduler increased, giving much better results.

Summary: Co-residency detection within virtual machines is possible in cloud-like environments. Background noise in the system has a significant effect on the success of co-residency detection.

4.2.3 Non-quiescent setup To verify that IO and network interference from virtualization or the master plays a role, we also ran the

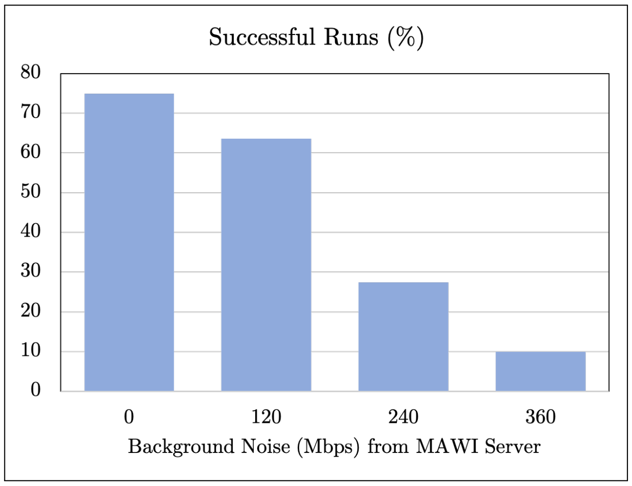


Figure 11: Success rates with variable background noise: Detection has high success rates while system load l is within hardware network capacity

setup on the *Noisy cloud* while a separate application issued a lot of IO and network calls on the SERVER container. We used the Memcached benchmark tool from the YCSB benchmarking tool [20] for this purpose, which added a high level of load on the system. The result here seems to build on the above postulation. The additional network and disk usage make K8 and DS react more than Helios. Kubernetes seems especially sensitive to such load in the system. The success values shown in Table 2 are very low. This shows that the attack is fairly sensitive to system noise.

Realistic noise setup The YCSB benchmark generates very high noise in the NIC since it makes requests as fast as possible. The successful detections in this scenario are low since the amount of traffic generated in the NIC is over the capacity of the NIC, leading to thrashing. We confirm that this is the case by running the same experiment with a recent (April 2019) traffic trace from the MAWI dataset [41]. The results of this experiment are displayed in Figure 11. When the overall load l in the system is within the capacity of the NIC, we find that detection is quite successful. As l passes this value, we find that successful detection gets much harder.

Summary: Background noise in the system significantly affects the success of the attack. However, this is only the case when l exceeds the capacity provided by the hardware. Any background noise under that limit is tolerated and does not reduce detection success rates significantly.

4.3 Adversarial deception

On the *Quiet Cloud*, we controlled the amount of interference generated by the adversary for several runs of the attack and evaluated the number of times the attack succeeded. Figure 12 shows the successes the adversary obtained when flooding at varying rates. This result presents an interesting finding. As flooding reaches 1 Gbps (network capacity), the success rates seem to flatten out. As the FLOODER's activity approaches network capacity, the NIC starts to thrash, making it difficult for the system to keep up. At this point, the adversary generates the highest contention possible. This result

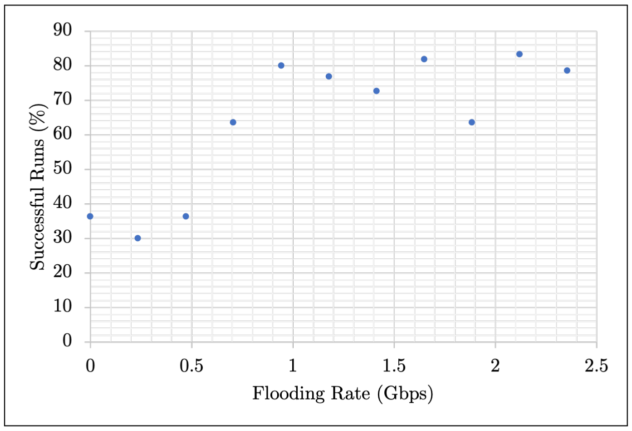


Figure 12: Success rates at varying rates of flooding

concurs with the previous section: as the background noise generates enough contention to generate thrashing in the NIC, the success of the attack reduces. However, if the adversary can prevent the system to reach such a state, the amount of activity needed by the adversary can be quite reasonable. Based on this result, we can conclude that the adversary can have significant success at detection while only using around 70% of network capacity for the node in a quiescent environment.

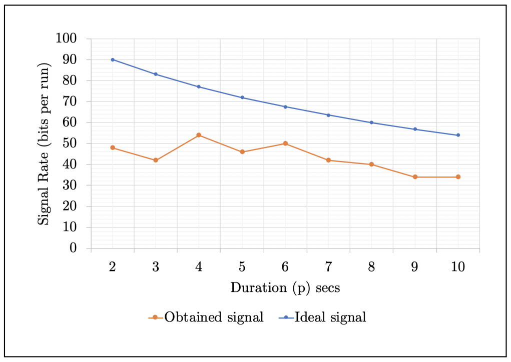
Summary: An adversary can be fairly deceptive when generating contention in the shared resource, requiring close to 70% of network capacity for high successes on our setup.

4.4 Flooding period

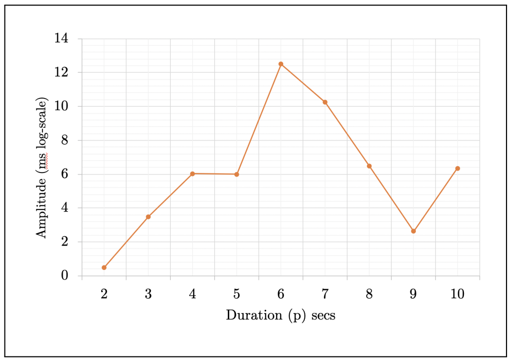
We evaluate if the period of flooding has any sensitivity to the attack. This experiment also provides us with a measure of covertness that the adversary can employ. The technique used for this experiment differs slightly from the previous co-residency detection mechanism. Rather than looking at the overall RTT trace, we consider each flooding cycle (*On-WM* and *On-Idle*). In such a cycle, we define that the *On-WM* sequence will generate a 1− bit if there is separation generated from the baseline. A combined sequence of *On-WM* and *On-Idle* will then generate a pair of one and zero bits. An ideal trace with perfect contention will generate a maximum of d/p bits. We then measure the cycles that generated separation; any cycle that does is counted as two bits of signal.

To measure the separation, we use the difference of mean RTT between *On-WM* and *On-Idle*. If the difference is positive, that is a detectable pair of bits. The results for this are shown in Figure 13a. Each data point shows the average signal over eight runs. We vary p and measure the signal we find in a number of runs. This shows that for variations in flooding duration, there is not a lot of signal lost.

We note here the importance of the *On-Idle* portion of the cycle. During our experiments we found that at certain flooding frequencies, the flooder's activity generates trashing in the kernel queues causing delays during the *On-Idle* sequence to rise significantly enough to generate incorrect data. To alleviate this issue, we increased the duration of the *On-Idle* sequence to 15 seconds giving enough time for the system to revert to a stable state. Thus, the

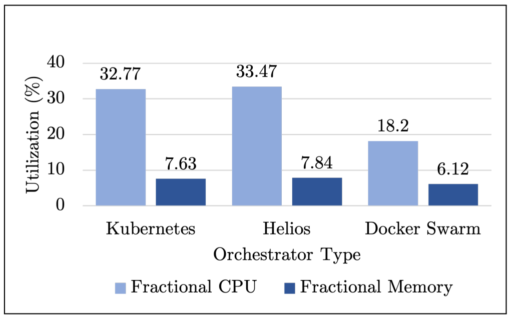


(a) Flooding duration p vs Signal rate: The duration of flooding has very little effect on the amount of signal lost.

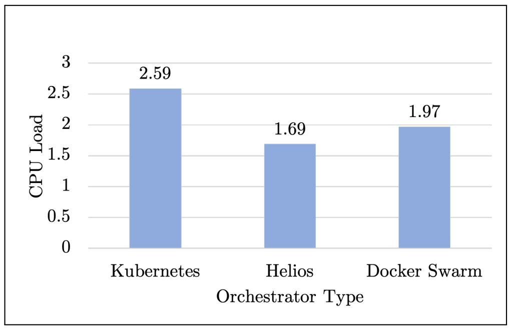


(b) The amplitude of separation between *On-WM* and *On-Idle* vs p. Longer duration runs have lower amplitude as the system can recover from the start of flooding, making it more covert.

Figure 13: Sensitivity analysis of flooding sequence period



(a) Orchestrators have varying CPU usage yielding varying resiliency due to hypervisor scheduling



(b) CPU Load: A lower load shows that the orchestrator performs more CPU bound computation than IO

Figure 14: System performance variation based on orchestrator

On-Idle sequence is not only important while generating the signal, but it is also necessary to restore the state of the system and improve the success of the adversary in the following cycle.

To analyze the covertness of the attack, we consider the mean amplitude of the generated separation. The more delay the adversary generates, the easier it is to detect. Figure 13b shows the average amplitude vs flooding duration (p). The amplitude is much higher in runs with lower p. For runs with a higher p, the amplitude of separation is much less detectable. As a result, we empirically conclude that the ideal period to maximize signal while minimizing detectability is 6-7 seconds.

Finally, we also considered results by varying the overall duration d, however, we did not see any significant difference in the results. All experiments here use $d = 2$ minutes.

Summary: Empirically, the ideal flooding period which maximizes the signal and covertness is 6-7 seconds.

4.5 Orchestrator Comparison

4.5.1 Local When running the same experiment on three orchestration systems, we see that the effect of the FLOODER is clearly visible in Figure 15. The RTT graph shows a clear square wave that coincides with the FLOODER's interference. Some outlier RTT values drop to zero for requests that were lost on the network, but we can conclude that orchestrators do not prevent such attacks.

However, the successes for each orchestrator seem to differ on each system. To understand why this is the case, we compared the system performance while running the application containers with the Master on the same node (*Local* setup). We found that all systems had differing CPU and memory usage on average. Figure 14a shows a comparison for each of these metrics. The architecture of these systems provides a better understanding of why this is the case.

Kubernetes and Docker Swarm both use a centralized database (e.g., Etcd) to store the state of the cluster. All possible Master nodes maintain a consistent state of the cluster in the database instances they run. Each database instance communicates with others to maintain this consistency. Whenever the Master realizes that the state of the cluster is less than ideal, for example, when a new application is deployed on the cluster, an application crashes, etc., the Master node picks a client node to run the application container on and enforces this by making an HTTP request to the node's manager. Additionally, to understand the state of the cluster, the Master node sends a heartbeat, health-check request to all client nodes (including itself). The Master itself also runs

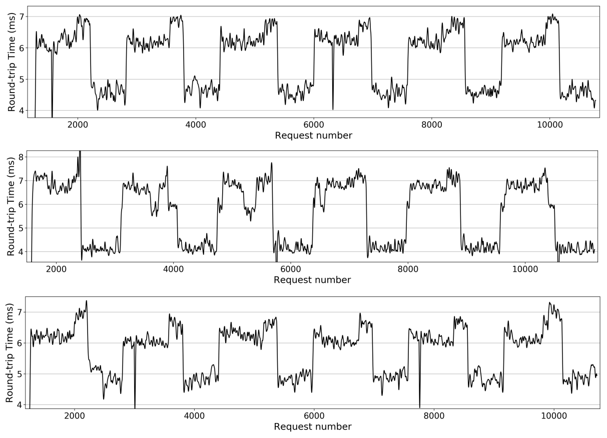


Figure 15: RTT histogram for a flooding sequence across K8 (top), Helios (middle), and DS (bottom): Square wave is visible on all three platforms

virtualization for IO, DNS, etc. While Docker Swarm and Kubernetes are similar in architecture, their differences arise from the actual implementation of the software used. By exploring the source code for K8 [25], Helios [42], and DS [21], we were able to draw the following conclusions: DS has a much simpler connection to talk to the Docker runtime while K8 implements many more abstractions to ensure container-runtime agnosticism. As a result, DS has much lower CPU, memory, and IO usage than K8.

Helios on the other hand uses a completely different architecture. It maintains an Apache Zookeeper cluster instance that both the Master and client nodes connect to. The Master stores whatever changes are necessary to the cluster and the clients continuously poll Zookeeper on a KeepAlive TCP connection. Whenever they notice a change, they implement these into their state and communicate it back to Zookeeper. Due to this reason, Helios uses more CPU and memory while it queries the Zookeeper instances for data, but spends less time waiting on IO. Thus, the CPU load when Helios runs is much lower than its competitors.

Summary: The inclusion of orchestrators does not have any major impact on the success of co-residency detection.

4.5.2 Local Separated By separating the Master and the client nodes, we were able to isolate the interference generated in the system by the Master node. While the data was useful, it did not represent a realistic setup. In such a setup, we can now compare the realistic effect an orchestrator will have on the success of co-residency detection. Any effect by the orchestrator will exist whether or not the attack is currently in progress. Hence, we used the RTTs generated during the *Off* sequence for each orchestrator as well as a baseline measurement where no orchestrator was present. We compare the average RTT for each run on the orchestrator and compare the average for each run during the baseline. We can compare the averages using a simple statistical T-test. The T-statistic and p-value values for each are presented in Table 4.

For each orchestrator, we reject the null hypothesis suggesting that the orchestrator does have some effect on the RTTs. This means that the interference generated by the orchestrator will raise the baseline RTTs, possibly generating a higher number of false positives. However, all orchestrators only raise the baseline by at most $0.55ms$. Compared to the RTT increase generated by the `FLOODER` (≈4 ms), the increase by the orchestrator is minimal. Finally, it is interesting to note that the rise in the baseline is lower in Helios than the other orchestrators, which, due to the poll-based implementation, sees comparatively more CPU than IO usage.

	K8	Helios	DS
P-value	0.00013	$3.9 * 10^{-9}$	$5.29 * 10^{-6}$
T statistic	4.198	7.922	5.395
Mean RTT gain (ms)	0.51	0.36	0.55

Table 4: T-test statistic values for each orchestrator

Summary: The orchestrators have some effect on the success of co-residency detection, but the effect is minimal and not enough to be claimed secure against co-residency attacks.

From our experiments, we highlight a few key takeaways:

- Co-residency detection attacks are feasible on containers running within virtual machines. Additionally, orchestration platforms play a minimal role in the success of attacks. These results were verified with a simple setup as well as mimicked and real cloud environments.
- Adding load to the attack surface reduces the success of co-residency attacks. However, this is only the case when the load exceeds the hardware capacity of the system.
- The attacker can be fairly deceptive in their attack and only needs about 70% of the bandwidth over ten seconds to detect co-residency.

5 Discussion and Future Work

5.1 Effect of noise

While we found that added noise to the system reduces the success of the attacks, that is not necessarily the end of the road for co-residency detection. The detection success rates we generated were for one-shot detection: this means that the adversary runs the attack once with duration d seconds. Based on the data obtained in that run, the adversary detects whether or not they are co-resident. However, modifying the detection mechanism can help improve the accuracy of detection. If the adversary were to increase d, the background noise would be more likely to normalize and improve detection. Alternately, if the adversary used a combination of runs to detect co-residency, they could define a successful detection when more than 50% of the runs return positive results. This could be a significant boost in the detection success.

5.2 Why does this attack generalize?

Bazm et al. [18], Singh and Somani [40], Zhang et al. [51] performed comprehensive studies on the various types of side-channel attacks. They demonstrated that VMs and containers are both individually vulnerable to the entire class of co-residency attacks. These include last-level cache attacks [28], network topography and instance placement [45], memory bus attacks [48], and network interface cards [17]. Each of these attacks follows a very specific pattern:

- The attack surface is some shared hardware resource that may or may not be virtualized. Despite the status of virtualization, there is no isolation guarantee provided by the software for such shared hardware.
  For example, in case of the attack on the shared cache during encryption/decryption, the attack relies on access to shared memory pages in the encryption libraries.
  During both the Ristenpart et al. [39] placement attack and the knowledge of existence [17] attack used in this paper, the NIC is the shared hardware resource under contention. Despite the kernel scheduler's best efforts, when the attacker demands access to the NIC for transmitting data, it takes away bandwidth from the victim, delaying its transmissions.
- Once the attacker has some form of shared hardware, the attacks attempt to generate some sort of contention while accessing it. As a result, the original execution of the victim takes a latency hit, which the adversary can measure and glean information from.
- Since the software that enables and performs this access does not play a significant role in isolating access to the hardware, each of these attacks is made possible.

The addition of another abstraction layer (orchestrator) to the applications has no major impact on how the hardware is isolated between tenants. Since noise in the system seems to reduce detection success, this may be implemented as a defense. The results in this paper show that while it is possible for the orchestrators to add noise to the system during the attack and make it slightly less effective, they don't have any effect on whether the hardware is completely isolated to prevent co-residency attacks altogether.

Note that while we show that this attack works quite well in various environments, there is no claim of optimality. The work here shows that co-residency attacks are still a reasonable threat in the cloud-computing domain, even in commercial settings using modern technologies like containers and orchestrators. We acknowledge and encourage that other techniques for detecting co-residency and exploiting side-channels in hardware exist, which achieve the same result more covertly. Indeed, the co-residency detection technique that uses probing instead of flooding [14] could achieve the same results with a more covert footprint. Other techniques like traffic shaping, previously used for stealth VM migration [11], could be implemented with flooding to make the traffic indistinguishable from a real source of heavy traffic like a streaming service. These techniques should be considered in future work.

Other work should consider the effects of newer container runtimes like Kata and gVisor [1, 27] that utilize hardware-level isolation between processes. Since these runtimes are focused on security and isolation, it presents an interesting challenge to either show that co-residency attacks are viable against such setups or evaluate how these systems provide strong enough isolation to prevent the effects of side channels.

5.3 Defenses

While the attack analysis here is a good representation of co-residency attacks on containers, there are some caveats when the correlations may not apply. Since this attack depends on watermarking network traffic, the `CLIENT` must be able to communicate with the same server constantly. In cases when the web-server sits behind a load balancer, the `CLIENT`'s successive requests may get routed to other server instances and the attack may be unsuccessful.

For other attacks, specific defenses should also work. Blocking access to the `clflush` instruction for VMs seems to solve last-level cache attacks [50]. Duplicating pages instead of sharing them across VMs will be an effective defense against cache-based side-channel attacks [31]. Zhang et al. [51] suggests periodically flushing cache lines to prevent all `FLUSH+RELOAD` attack variants. Finally, adding noise to the specific attack surface may be a possible defense. However, since the success of the attack is only affected when hardware capacity is reached, such a defense is not encouraged since it will adversely affect the performance of client applications.

To implement defenses that encompass a wider range of attacks, moving target defenses [14, 16, 36] are successful. Since orchestrators often manage a large number of nodes for a cluster, performing container migration on these nodes will be easier than migrating the entire VM as suggested by Atya et al. [14]. There are multiple benefits to implementing migrations of applications at the container level. The migration cost for a virtual machine is very high since a new operating system instance needs to be instantiated at each migration. As a result, migrations cannot be made frequently. Additionally, migrating VMs generate enough detectable traffic for an adversary to potentially follow the target to the new location [11]. Due to the lightweight nature of containers, if the persistent data used by an application is handled properly, there may potentially be no trace of a migrated container. The orchestration platform may also implement a hybrid migration strategy, where VMs are also migrated, but less frequently.

Since the orchestration components are closely linked to the running kernel, we also suggest implementing migration triggered through VM-monitoring as described by [35, 47]. Such a solution would further reduce the cost incurred by the migration while providing stronger security against co-residency attacks.

6 Conclusion

In this paper, we showed that containers running on virtual machines are susceptible to co-residency attacks. We analyzed the attack on systems varying in architecture, load, and orchestrators and showed that this plays a minimal role in how they affect resiliency to co-residency based attacks. Our analysis showed that while orchestrators may add some amounts of noise to the system, there is no evidence that enough interference is provided by the orchestrators to make the system significantly and provably secure to network-based co-residency detection attacks. Any differential addition to the resiliency of the system comes at a performance cost that may not be worth paying. Cloud customers that have secure computing requirements should not depend on the orchestration platforms to provide enough protection from co-residency attacks.

Acknowledgements

This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). 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 Combat Capabilities Development Command 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 here on.

This work was partially supported by the National Science Foundation grant CNS-1815883, the Air Force Office of Scientific Research award number FA9550-19-1-0169, and the Semiconductor Research Corporation.

Author 1's contribution in this paper was conducted while they were affiliated with The Pennsylvania State University. The views presented in this publication are not endorsed or sponsored by Microsoft corporation.

References

[1] 2019. About Kata Containers: Kata Containers. https://katacontainers.io/
[2] 2019. Amazon EC2 FAQs - Amazon Web Services. https://aws.amazon.com/ec2/faqs/
[3] 2019. Amazon EKS - Managed Kubernetes Service. https://aws.amazon.com/eks/
[4] 2019. Cluster multi-tenancy | Kubernetes Engine Documentation | Google Cloud. https://cloud.google.com/kubernetes-engine/docs/concepts/multitenancy-overview
[5] 2019. EC2: Dedicated Hosts. https://aws.amazon.com/ec2/dedicated-hosts/getting-started/
[6] 2019. GKE Sandbox | Kubernetes Engine | Google Cloud. https://cloud.google.com/kubernetes-engine/sandbox/
[7] 2020. amazon web services (aws) - cloud computing services. https://aws.amazon.com/
[8] 2020. Cloud computing services (azure). https://azure.microsoft.com/
[9] 2020. Container Service for Kubernetes (ACK). https://www.alibabacloud.com/product/kubernetes?spm=a3c0i.239195.3156523820.138.7af912bd31X52F
[10] 2020. Kubernetes Service - Overview. https://www.ibm.com/cloud/container-service/
[11] S. Achleitner, T. L. Porta, P. McDaniel, S. V. Krishnamurthy, A. Poylisher, and C. Serban. 2017. Stealth migration: Hiding virtual machines on the network. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057195
[12] Tim Allclair and Maya Kaczorowski. 2018. Google Cloud Platform Blog. https://cloud.google.com/blog/products/gcp/exploring-container-security-isolation-at-different-layers-of-the-kubernetes-stack
[13] Apache. 2015. Apache Mesos. https://github.com/apache/mesos. , 9 pages. https://github.com/apache/mesos
[14] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V. Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa M. Marvel. 2019. Catch Me if You Can: A Closer Look at Malicious Co-Residency on the Cloud. *IEEE/ACM Transactions on Networking* 27, 2 (4 2019), 560–576. https://doi.org/10.1109/TNET.2019.2891528
[15] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V. Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa Marvel. 2017. Malicious co-residency on the cloud: Attacks and defense. In *Proceedings - IEEE INFOCOM*. https://doi.org/10.1109/INFOCOM.2017.8056951
[16] Mohamed Azab and Mohamed Eltoweissy. 2016. MIGRATE: Towards a Lightweight Moving-Target Defense Against Cloud Side-Channels. *Proceedings - 2016 IEEE Symposium on Security and Privacy Workshops, SPW 2016* (2016), 96–103. https://doi.org/10.1109/SPW.2016.28
[17] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. 2012. Detecting Co-residency with Active Traffic Analysis Techniques. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop ({CCSW} '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2381913.2381915
[18] Mohammad Mahdi Bazm, Marc Lacoste, Mario Südholt, and Jean Marc Menaud. 2017. Side-channels beyond the cloud edge: New isolation threats and solutions. In *2017 1st Cyber Security in Networking Conference, CSNet 2017*, Vol. 2017-Janua. IEEE, 1–8. https://doi.org/10.1109/CSNET.2017.8241986
[19] Louis Columbus. 2016. Roundup Of Cloud Computing Forecasts And Market Estimates, 2016 - Forbes. http://www.forbes.com/sites/louiscolumbus/2016/03/13/roundup-of-cloud-computing-forecasts-and-market-estimates-2016/
[20] Brian F Cooper. 2010. Yahoo ! Cloud Serving Benchmark. , 19 pages.
[21] Docker. 2019. Docker Swarm. https://github.com/docker/swarm. https://github.com/docker/swarm
[22] Docker Documentation. 2017. Overlayfs storage driverâĂŤdocker documentation.
[23] Michael Eder. 2016. Hypervisor-vs. Container-based Virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* 1 (2016).
[24] Stefan Edwards, Dominik Czarnota, and Robert Tonic. 2019. *Kubernetes Security Whitepaper*. Technical Report. Trail of Bits. https://github.com/trailofbits/audit-kubernetes/blob/master/reports/KubernetesWhitePaper.pdf
[25] Cloud Native Foundation. 2019. Kubernetes. https://kubernetes.io
[26] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. 2017. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. *Proceedings - 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017* (6 2017), 237–248. https://doi.org/10.1109/DSN.2017.49
[27] Google. 2019. gVisor. https://gvisor.dev/
[28] Daniel Gruss, ClÃľmentine Maurice, and Klaus Wagner. 2015. Flush + Flush : A Stealthier Last-Level Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 279–299.
[29] Andy Honig and Nelly Porter. 2017. Google Cloud Platform Blog. https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext
[30] Taylor Hornby. 2016. Side-Channel Attacks on Everyday Applications: Distinguishing Inputs with FLUSH+RELOAD. In *BlackHat USA 2016*.
[31] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 299–319.
[32] H. Kang, M. Le, and S. Tao. 2016. Container and Microservice Driven Design for Cloud Infrastructure DevOps. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 202–211. https://doi.org/10.1109/IC2E.2016.26
[33] Denis Legezo. 2018. LuckyMouse hits national data center to organize country-level waterholing campaign. https://securelist.com/luckymouse-hits-national-data-center/86083/
[34] Daniel Markuson. 2019. Why the NordVPN network is safe after a third-party provider breach. https://nordvpn.com/blog/official-response-datacenter-breach/
[35] Preeti Mishra, Emmanuel S Pilli, Vijay Varadharajan, and Udaya Tupakula. 2017. Out-vm monitoring for malicious network packet detection in cloud. In *2017 ISEA Asia Security and Privacy (ISEASP)*. IEEE, 1–10.
[36] Soo-Jin Moon, Vyas Sekar, and Michael K Reiter. 2015. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *Proceedings of the 22nd acm sigsac conference on computer and communications security*. ACM, 1595–1606.
[37] MorteNoir1. 2018. VirtualBox Zero-Day exploit. https://github.com/MorteNoir1/virtualbox$_e$1000$_0$day
[38] S. J. Murdoch and G. Danezis. 2005. Low-cost traffic analysis of Tor. In *2005 IEEE Symposium on Security and Privacy (S P'05)*. 183–195. https://doi.org/10.1109/SP.2005.12
[39] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 199–212. https://doi.org/10.1145/1653662.1653687
[40] Gulshan Kumar Singh and Gaurav Somani. 2018. *Cross-VM Attacks: Attack Taxonomy, Defense Mechanisms, and New Directions*. Springer International Publishing, Cham, 257–286. https://doi.org/10.1007/978-3-319-97643-3$_8$
[41] CSL Sony and Kenjiro Cho. 2000. Traffic data repository at the WIDE project. In *Proceedings of USENIX 2000 Annual Technical Conference: FREENIX Track*. 263–270.
[42] Spotify. 2019. Helios. https://github.com/spotify/helios. https://github.com/spotify/helios
[43] RedLock CSI Team. 2018. Lessons from the Cryptojacking Attack at Tesla. https://redlock.io/blog/cryptojacking-tesla
[44] Eddy Truyen, Dimitri Van Landuyt, Vincent Reniers, Ansar Rafique, Bert Lagaisse, and Wouter Joosen. 2016. Towards a container-based architecture for multi-tenant SaaS applications. In *ARM 2016 - 15th Workshop on Adaptive and Reflective Middleware, colocated with ACM/IFIP/USENIX Middleware 2016*. 1–6. https://doi.org/10.1145/3008167.3008173
[45] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. [n.d.]. A Placement Vulnerability Study in Multi-Tenant Public Clouds.
[46] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems, EuroSys*

2015. Bordeaux, France. https://doi.org/10.1145/2741948.2741964

[47] Omar Abdel Wahab, Jamal Bentahar, Hadi Otrok, and Azzam Mourad. 2017. Optimal load distribution for the detection of VM-based DDoS attacks in the cloud. *IEEE Transactions on Services Computing* (2017).

[48] Zhenyu Wu, Zhang Xu, and Haining Wang. 2015. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Transactions on Networking* 23, 2 (4 2015), 603–615. https://doi.org/10.1109/TNET.2014.2304439

[49] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co-residence Threat inside the Cloud. *24th USENIX Security Symposium (USENIX Security 15)* (2015), 929–944. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu

[50] Yuval Yarom and Katrina Falkner. 2014. Flush + Reload : a High Resolution , Low Noise , L3 Cache Side-Channel Attack. In *USENIX Security 2014*, Vol. 1. 1–14. https://doi.org/Report2013/448, 2013

[51] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Mingsheng Wang, and Peng Liu. 2016. A comprehensive study of co-residence threat in multi-tenant public PaaS clouds. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9977 LNCS. 361–375. https://doi.org/10.1007/978-3-319-50011-9{-28

[52] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the ACM Conference on Computer and Communications Security*. 990–1003. https://doi.org/10.1145/2660267.2660356