THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Interference-based Identification of Cloud Tenant Operations

Alexander Lumsden

Allow Others to View: **Yes**
Hard Copy Printed: **Yes**

This report is submitted in partial fulfilment of the requirements for the degree of Bachelor of Computing and Mathematical Sciences with Honours (BCMS(Hons)) at The University of Waikato.

Abstract

Hosting applications in a public cloud enables use of an on-demand pool of resources and an isolated environment without having to purchase or configure the infrastructure. In a public cloud, organisations can have their deployed applications running simultaneously alongside other organisations workloads while executing on the same underlying hardware resources. This is the reason isolation is paramount in a cloud environment. This project aims to determine if it is possible from a tenant perspective to break through this isolation and infer a co-located tenant application type or functionality by monitoring performance interference within a containerized cloud environment. The goal of the project is to provide evidence that performance interference can cause a leaky side channel to be exposed and sensitive data can be extracted from it. The project proposes a technique to measure the magnitude of performance interference and uses a set of applications to collect data by profiling the environment. This data is then used to train machine learning models in order to predict the co-located tenant application.

After evaluating the techniques used in the investigation the results found that monitoring the performance interference was able to break the isolation between tenants, as co-located applications were able to be predicted using the machine learning models. This confirmed that performance interference can be used to extract privileged information. However, the reliability of this approach was not confirmed as the evaluation methods yielded a 49% prediction accuracy rate on the test dataset created.

Acknowledgements

I would like to thank both of my supervisors Panos Patros and Vimal Kumar for their support during this project.

Vimal for always giving a different perspective on topics discussed during the course of the project, giving me knowledgeable advice on the next steps to take and always trying to keep me on track.

Panos for always showing enthusiasm towards the project, inspiring me to look deeper into problems to find better solutions and guiding me to make the correct decisions throughout the investigation.

A special mention also to the other ORCA lab members for always taking the time to help me out if I had any questions or was just looking for a second opinion.

Contents

List of Figures

List of Algorithms

List of Tables

1 Introduction

In 2017, 75% of enterprises utilized cloud computing in one form or another, with 32% running on public clouds (Weins (2017)). Fast forward to the present and 91% of enterprises now utilize cloud computing with as many as 84% incorporating public clouds (Weins (2019)).This increasing trend in the adoption of cloud utilization and the number of companies currently heavily invested in cloud resources is a clear illustration of the importance of having cloud resources that are guaranteed to be secure, private and predictable.

Hosting data and deploying applications to cloud infrastructure can have multiple benefits including financial savings, performance advantages, increasing accessibility and flexibility (Catteddu (2009)). However, cloud adoption does not come without associated risks, this cloud platform presents a new form of attack surface and vulnerabilities that previously did not exist when data and applications were hosted with private, on-premises infrastructure (Zissis and Lekkas (2012)). In a public cloud environment, multiple independent organisations with deployed data and applications can be executing on the same physical hardware simultaneously. This environment requires the cloud provider to ensure isolation of each organisations workload and obfuscate any activity from all other co-located workloads.

Previously, malicious actors have compromised public cloud environments and used them as launch surfaces for exploitation. Attackers have taken advantage of the fact underlying hardware is shared between applications, as given by the Spectre vulnerability (Kocher et al. (2019)) that broke the isolation between applications by exploiting a vulnerability in modern processors and allowing a program to read privileged data. Another example was the Meltdown vulnerability (Lipp et al. (2018)) that exploited a vulnerability in a processors Out-of-Order execution protocol to leak privileged data. Both vulnerabilities are examples of side channel attacks in a cloud environment.

This investigation aims to identify and explore the viability of another form of side channel attack in a cloud environment. Specifically, if monitoring and measuring the interference caused by another workload can leak information regarding what application is executing in the cloud environment. If a co-located application can be determined using this side channel leak, then it overcomes the isolation between cloud tenants. This research focuses on cloud environments that utilize containerization as the technique to isolate tenant workloads. The investigation also takes the perspective of a malicious cloud tenant to evaluate the feasibility of this attack occurring without any

knowledge of the cloud environment.

This research proposes and investigates a technique to measure performance interference using knowledge of the underlying hardware organisation at the systems level. Investigation into feature extraction using timeseries analysis and applying extracted data to machine learning models. Hence, the goal of this research is to provide a proof of concept that monitoring performance interference can allow a leaky side channel to emerge and sensitive data can be extracted from a tenants perspective.

The main contributions of this research are techniques to measure and observe performance interference, using them identify a co-located workload and the experiments to evaluate these technique's. Six independent applications designed to induce contention and measure interference in a specific resource are developed using the techniques. A Docker Image is created that represents a containerized version of the six independent applications and is designed to be applied in a Kubernetes context as a Kubernetes job. A minor contribution includes a python program that can be used to automatically deploy and then delete Kubernetes Pods using the Kubernetes API, this program was used to test the independent applications. Each of the artifacts has been open-sourced and can be accessed at `https://github.com/Alxlmsdn/Interference_Identification_Sniffers`.

1.1 Structure of Dissertation

Section 2 of this dissertation defines the key concepts and technologies that were used throughout the investigation, also listing key areas focused on for the research. Following this background information, the related work in the areas focused on is explored, the published research include works that explore similar concepts and that used different approaches or technologies to solve closely related research questions. The research question of this investigation is also compared to each similar approach to distinguish it and convey its difference to previous work.

Section 3 details the approach that was taken to carry out this research and provides explanations for both the architecture and methodology choices taken in the investigation.

Section 4 provides technical details regarding the development and implementation of the applications proposed in this research. Implementation details include algorithms proposed, metrics for data collection, languages and libraries used. This section also details the verification of the applications to

investigate comparison between theoretical and practical performance.

Section 5 explores the evaluation phases of the applications and research question. The section is split evaluation phase sections, detailing each evaluation phases purpose, methodology and results. This section discusses the success of the approach taken and reflects back to the initial research question of the investigation.

Section 6 concludes the dissertation, stating the contributions of the investigation and the potential future work to expand on this research.

2 Background

This chapter of the dissertations details the technologies focused on in the investigation and the related works that have explored related problems or approaches.

2.1 Concepts & Technology

The following section describes in detail the technologies, concepts and techniques that are utilized and focused on in this investigation.

2.1.1 Co-resident Clouds

A Co-resident cloud is the environment that this investigation focuses on. Evaluation of the techniques used in the research are performed in a multitenant scenario.

Currently, the major cloud computing service providers—Amazon Web Services[1], Microsoft Azure[2], IBM Cloud[3] and Google Cloud Platform[4] (Dignan (2019))—all offer core services that adhere to the multitenancy cloud model. The multitenancy model corresponds to the concept of multiple customers sharing resources; resources can include physical hardware, such as the CPU, memory and off core resources like network I/O. The key concepts driving the multitenancy model are resource utilization and a strive towards efficient use of resource capacity. Allocating only a single application per host can lead to the underutilization of resources because many applications do not necessarily require the use of all available resources constantly (Al-Jahdali et al. (2014)). Being able to allocate multiple applications to the same hardware resources increases the chance that they will constantly be in demand, resulting in a higher level of utilization and improving the cost efficiency of running the resources. The improved resource efficiency is vital to enable cloud service providers to keep computing service costs low and entice enterprises to adopt cloud computing over setting up their own computing infrastructure.

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/en-us/
[3] https://www.ibm.com/cloud
[4] https://cloud.google.com/

Co-residency applies to each of the core service models provided; Infrastructure as a service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS is where customers have control over the resource management of the platforms they require (e.g. Virtual Machines (VMs)) and have full control over the applications running on that infrastructure. PaaS allows customers to only have control over what applications they have executing in the cloud. Resource management is abstracted with load balancing and scaling handled by the provider, and this means customers have little idea of where and what kind of platform on which their applications are running. Finally, SaaS is where all aspects are abstracted, the customer interacts with an existing software application designed for a specific task or purpose (Krebs et al. (2014); Sareen (2013)). Co-residency in this service is of the form where multiple customers data could be stored in the same infrastructure e.g. database tables.

2.1.2 Containers

This research investigates cloud environments that utilize containers as the isolation technique between tenants.

In the realm of cloud computing and the idea of multitenancy, isolation between tenants is paramount to successful operation. The idea of isolation is that each tenants' application, which resides on a physical host, is under the impression that only it has access to all the resources. In reality, there is a high chance that multiple other applications are running parallel to the application. Aside from computing resources like the CPU and memory, the disk also needs isolation as customers rely on a high degree of trust that only their applications can access their potentially sensitive data. Virtualisation systems are used to achieve this isolated environment, and the most widely adopted techniques are to use VMs and containers (Sareen (2013)).

VMs are an abstraction of the entire host on top of which they run, a virtualized instance of a machine. Each VM has its own Operating System (OS) and thus can have its own system binaries and libraries completely separate from the host OS (Rosenblum and Garfinkel (2005)). A VM interacts with the host machine via a hypervisor layer that provides virtualization of the physical hardware, and it then uses these virtual resources that correspond to a subset of the physical hardware (Figure 1). It is this hypervisor layer and the VM that provide an isolated environment, and because the VM has its own OS, the customer application executing within sees only this guest

OS and processes within it.

In contrast, the systems that this project focuses on are containers, and they provide an abstraction of only the operating system. A container still causes an application running in it to believe it is the only process executing on the machine; however, it does not use a guest OS as VMs do, but instead each container has its own set of namespaces and resources are allocated using the cgroups namespace. Because each container has its own set of namespaces, it abstracts the container to be a separate machine running on the host and therefore isolated from any other processes or containers running on the same host OS (Bernstein (2014)). This type of separation still allows each container to have its own independent libraries, tools and runtimes for an application. A container engine is used to create, manage and destroy containers and executes in the host OS environment.

Due to containers not requiring their own guest OS, they are smaller in size than a corresponding VM and thus have less overhead. Containers are rising in popularity because in comparison to VMs, start-up times are faster, and more containers can concurrently run on a physical machine (Joy (2015)).

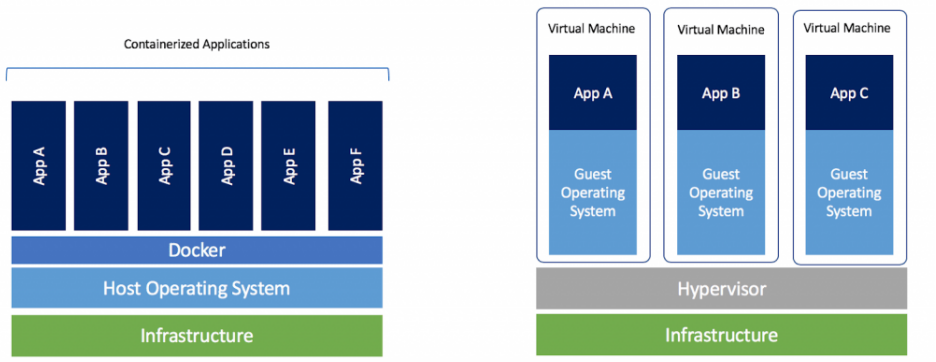


Figure 1: Containers vs VM Architecture (Source: Docker)

2.1.3 Kubernetes & Docker

The cloud environment used in this investigation is created using Kubernetes to manage Docker containers.

Because the utilization of container technology is increasing, a popular system coupling of container and container orchestration software has emerged. Currently, the most widely used system comprises of Docker[5] as the container engine and Kubernetes[6] for the orchestration of the Docker containers (Weins (2019)). Docker provides the OS-level virtualization and creates these isolated units of software that represent a container. These containers run on top of the Docker engine and are based on Linux containers (LXC). In Docker, containers that are not running are represented by Docker Image files. An image file contains an ordered collection of root filesystem changes and systems, tools, dependencies, libraries and applications the container will have at runtime.

In a cloud environment, these containers need managing, and therefore Kubernetes is used with Docker for this purpose. Management of containerized workloads orchestrates the computing, networking and storage infrastructure on behalf of the user. Kubernetes facilitates services such as container deployment, scaling, load balancing and monitoring.

The architecture of Kubernetes is as follows: At the highest level, there is a cluster manager known as the master; this is responsible for managing an entire cluster, where a cluster is one or more machines known as worker nodes. Pods are the smallest unit of deployment in Kubernetes and each Pod is a group of one or more containers. Each node in a cluster contains the services necessary to run Kubernetes pods, and these services include the container runtime (Docker), the kubelet to manage Pods, and kube-proxy for managing networking.

2.1.4 Interference

Interference is the key concept driving this investigation, it is used to gather data regarding tenant applications.

As stated in the co-resident section, a key principle of cloud computing is isolation between tenants' applications when they are sharing resources on a single machine. Interference is something that can put this isolation at risk and breaking the isolation between tenants could lead to the leaking of sensitive data, degradation of performance or denial of service to tenant applications (Joshi et al. (2017)). The interference focused on in this project is

[5]https://www.docker.com/

[6]https://kubernetes.io/

internal interference at the tenant level, where the isolation boundary is broken due to the behaviour or execution characteristics of a co-resident tenant. For performance interference, this characteristic is high resource contention between tenants. Contention can be regarding core resources like CPU utilization or off-core resources like Network I/O. If there is significant performance interference, the effects observed are most likely to be performance degradation, decreased throughput or increased tail latency.

2.1.5 Hardware Resources

Performance interference is due to multiple tenants competing or requesting for limited resources simultaneously. The resources that will be the focus of this project and to investigate their impact on interference will be:

CPU: The central processing unit of a machine, responsible for carrying out the instructions of an application. The CPU performs arithmetic, logic and controlling operations. In a cloud environment, this unit can be shared between each tenant, where the tenants are guaranteed a share/slice of the CPU processing time. Commonly, there are multiple CPUs within a node and each CPU can have multiple cores. The number of CPU cores represents the number of processes that can be performed concurrently.

Memory Capacity: This corresponds to a machines available RAM, and this resource needs to be partitioned among the tenants just as the CPU does. The memory partitions need to have hard boundaries between tenant processes because often sensitive data will reside in the memory that only the intended tenant should access.

Both of the above resource types can have specific values requested and limited in Kubernetes for each container in a pod. A request guarantees the specific amount of the resource is available at any point in time, whereas a limit will allow a container to use a higher amount of resources up to the limit only if resources are available, i.e., not used by other co-resident containers. In this case of containers, the cgroups tool is used to implement these requests and limits.

L1 Instruction Cache: This cache layer in the CPU is responsible for holding information about the instructions the CPU has to perform. Along

with the L1 data cache, these are the fastest areas of memory in a machine and consequently the smallest.

L1 Data Cache: Alongside the instruction cache is the data cache. This cache area holds data that the CPU will use to perform a set instruction on.

L2 Data Cache: This is next cache layer in the CPU after the level 1 cache and the second fastest area of memory. The L2 cache has a larger capacity than the L1 data cache.

Last Level Cache: This cache layer is the final, slowest and the largest areas of the CPU cache. This level is the last level where, if evicted the data is swapped back into the systems memory off the CPU.

Data is swapped from the main memory to the L1 cache and then swapped out to L2 and finally the last level (depending on how many cache levels the CPU has). When the CPU is trying to locate data on which to perform an operation, it looks in the order from L1 through to the main memory.

Network Bandwidth: Cloud-based applications are most often needed to be accessed externally, accepting requests then performing a service according to the request before returning a response. The operations of accepting requests and returning responses all make use of the network interface. Like the previous resource types, heavy utilization, such as a flood of requests to the same physical address, can cause saturation of the network bandwidth because the network cannot accept any more incoming requests until it has finished with previous ones. This concept also applies to virtual networks, which Kubernetes utilizes to route requests to specific pods on a node and because external responses will at some point need to use the physical network.

Disk Bandwidth: Refers to the amount of data that the system can simultaneously read and write from the primary storage system. i.e. hard disk, SSD. Like the above resources, this disk I/O has the possibility of becoming saturated if a large number of read or write operations are made to it simultaneously.

2.1.6 Timeseries Data

Timeseries data was the type of data collected when monitoring interference and used for tenant prediction.

Timeseries data represents a series of datapoints that are collected over time. Each datapoint in the series is, therefore, indexed according to its value in time respective to the other values in the series (Brillinger (1981)). In these series, time is often the independent variable, and commonly the time values are equidistant in the series. For this project, the time values are not equidistant as they are dependent on resource performance. There are commonly two categories of timeseries data: univariate and multivariate. Univariant timeseries is where there is only one time-dependant variable in the series, whereas a multivariate series contains multiple time-dependant variables. Common reasons for collecting data in a timeseries structure is it gives the opportunity to perform timeseries analysis or forecasting. Timeseries forecasting involves using a model that utilizes the historical or previous values in the time series to predict future values of the timeseries variables. Timeseries analysis is observing all the data points gathered within the timeseries and using methods to extract useful statistics, features or characteristics from the data (Brillinger (1981)). Timeseries analysis in this project allows the extraction of key attributes from resource data to be extracted and used as input to a supervised machine learning model for prediction of unknown tenant tasks. Hence, timeseries forecasting is not used in this research because predicting resource usage is not the aim of the research.

2.1.7 Machine Learning

In order to predict a co-located tenant application, machine learning was used to learn characteristics from the timeseries data.

Machine learning can be considered a subset of the larger topic of artificial intelligence; it leverages mathematical models built from sample data. Using these models, which are trained on known data, predictions or decisions can be made when provided with new data. These models are most commonly derived from statistical models and are used to identify patterns in the known data (Harrington (2012)). The underlying concept of machine learning is to enable a system to make a decision or prediction given data without explicitly providing instructions according to data values, i.e., the model learns with minimal external input. Within machine learning there are two main types of

tasks, these are supervised and unsupervised learning. Supervised learning is where the model is built using a set of training data where the input and also the desired output (prediction) are known. The model then uses this data to learn patterns or relationships between the inputs and outputs. Using this trained model, it can then make a prediction of the output on input data it has not been trained with nor knows the true desired output. In contrast, unsupervised learning involves training the model on data where the inputs are given but the desired output (decision) is not. The aim of using such a model is to identify structure in the data, like clusters or implicit grouping of data. This project employs supervised learning models for the purpose of identifying tenants running application.

2.1.8 Performance & Side Channel Attacks

The aim of this investigation was to uncover if interference allows sensitive information to be extracted. If successful, the gained information could be used to launch attacks on co-located applications.

The concept of performance interference can be exploited to become a tool for launching attacks against cloud tenants. Resource Freeing Attacks (RFAs) are a type of performance attack directed at degrading a victim tenants performance, these attacks specifically apply in a multitenant cloud environment because they involve a co-located adversary tenant and victim tenant ()Bazm et al. (2017); Delimitrou and Kozyrakis (2017). The adversary tenant will cause the victim to yield its resources for use of the adversary, and this is carried out by causing contention in a bottleneck resource used by the victim, causing the scheduler to give priority to the adversary. This type of attack degrades the victim tenant application performance and can also cost them financially because the application is running on a public cloud provider.

If an adversary has prior knowledge of a tenants operations that can be used to better degrade a victims performance, this attack is also classed as a side channel attack. A side channel attack is any kind of attack that is based on information gained from the implementation of a system rather than a weakness in the implementation (Godfrey and Zulkernine (2014); Bazm et al. (2017)). Hence, a side channel represents a technique used by a system to extract data that by design, it should not be able to access.

2.2 Related Work

Among the topics highly researched in the field of multitenant clouds is the effects of performance interference between co-resident VMs at the tenant level, and consequently techniques to improve the isolation of co-resident VMs. However, because the adoption of containerized systems in a cloud environment has only recently gained traction in the last 5 years, there are fewer publications that investigate the interference between containers and the levels of isolation between them. Currently, to the best of the authors knowledge, no research involving multi-tenancy has been applied to a container orchestration system (Kubernetes).

2.2.1 Performance Interference

Many of these studies regarding tenant isolation and interference focus from different perspectives and motivations, such as focusing from a cloud providers stance. Shue et al. (2012) investigated co-located VM tenant interference caused by contention in the underlying storage infrastructure, the authors presented Pisces, a system that reduces the contention using weighted shares and queuing. Li et al. (2019) presented PINE, a strategy to deal with contention in storage infrastructure similar to Pisces (Shue et al., 2012). The difference between PINE and Pisces was Pisces categorized applications according to their type (latency or throughput sensitive) and containers were the virtualization technique studied. Krebs et al. (2014) proposed an approach to isolate and allocate Quality of Service (QoS) standards to co-located SaaS tenants using resource demand estimation and request admission for each request, allowing the system to control the performance of each SaaS user.

The following works analysed interference using performance hardware counters, and while this provided accurate data about the system, the perspective of which this research takes is from a tenant, and a tenant will be unlikely to have access to such counters. Koh et al. (2007) analysed performance interference in VMs by monitoring statistics in the hypervisor and clustered applications that generated specific types of interference. Novaković et al. (2013) presented DeepDive, a system that also uses low-level system information from hardware counters to monitor performance interference. Deepdive used an early warning component located in the hypervisor of the host, if interference is detected the system will clone the victim VM into a sandbox environment to compare and pinpoint the cause of the interference.

2.2.2 Side Channel Attacks

Numerous studies have been conducted into side channel vulnerabilities and attacks in virtualized environments. Bazm et al. (2019, 2017) investigated the current threats within virtualized environments by evaluating currently known Side Channel Attacks (SCAs) and presenting the idea of a distributed SCA for distributed clouds. The SCAs that Bazm et al. (2019, 2017) investigated were focused on extracting cryptographic secrets through timing attacks and cache-based attacks, and the paper proposed mitigation strategies for each. Wu et al. (2014) further investigated cache-based covert channel attacks and used Amazon EC2 to evaluate the effectiveness of a proposed high-bandwidth approach to SCAs. The above research targets a specific piece of information to extract from a victim e.g. AES/DES keys. In contrast, this research aims to investigate the side channel that performance interference potentially opens, determining if sensitive information regarding a tenants operation tasks can be extracted.

Examples of the affect side channel attacks can have on target systems are shown by the Spectre and Meltdown vulnerabilities, both exploit hardware protocols and allow a malicious application to steal data from memory which it is designed to be isolated. Spectre affects modern processors that perform branch prediction, and is based on the side effects of speculative execution (a common method to hide memory latency in a processor), the vulnerability is able to manipulate another process into revealing its own data, including sensitive information (Kocher et al. (2019)). Meltdown breaks the fundamental isolation between user applications and the OS, it exploits a race condition that allows a process to bypass privilege checks designed to isolate a user application from accessing OS data and other processes (Lipp et al. (2018)).

2.2.3 Similar Approaches

Below are further key publications that relate to this project, containing related techniques that can be adapted for this research involving containerized environment.

Patros et al. (2016) investigated the impact of applying specific cloud burners in a VM environment, these burners were apart of a tenant application to stress target resources such as the CPU, Cache, resident memory, Disk I/O and network I/O. Using these cloud burners, the authors observed the effects of performance interference by comparing the slowdown of a tenant

in the presence of a resource-intensive tenant. This was done by proposing a metric to measure resource slowdown and intensiveness. Using the proposed metric and cloud burners together, they observed the effectiveness of scaling tenants both horizontally and vertically. The CPU shares algorithm was used both when the target tenant was running alone and when it was in the presence of a resource hungry tenant. In conclusion they found that scaling out (horizontally) does not improve performance when the tenant is already utilizing 100% of the VM resources or when CPU is not the bottlenecking resource. Performance may worsen if the resource being stressed is a non-scaled-out resource such as the network. Applications with internal bottlenecks only improve their performance when scaled out, not up.

This project touches on the aspects that Patros et al. (2016) mentioned in their future work, which was to investigate the effect of targeting more types of resources, as the aim is to determine the level of interference across a range of hardware resources. The authors proposed a set of cloud burners to investigate the effects on the performance of a tenant and in contrast, this project looks at investigating what information can be inferred from such interference. Hence, using the interference metrics as a profiling technique, the goal is to investigate the possibility of grouping and organising tenants according to their resource intensiveness. This grouping would lead to a novel performance orientated classification method for containerized environments.

Xavier et al. (2015) Investigated the level of performance isolation between LXC containers running disk intensive workloads. The separate disk intensive workloads used in the experiments were DBMS Oracle and MySQL, and in each case the benchmark was SysAdmin and Swingbench respectively. Each experiment used 2 containers, each allocated half of the total resources. The first container was running either of the workloads and the other an idle container with no application running to first test the benchmark in isolation. To test performance isolation the second container then ran a stressing application that utilized all its allocation of a specific resource including CPU, Memory and Disk I/O. Results showed that the Memory stressing application caused up to a 38% degradation of performance, Disk I/O nearly 20% degradation and CPU 5% degradation. In comparison, a VM environment under the same circumstances had a lot greater level of performance isolation. The authors concluded by recommending that disk and memory intensive workloads should not be in containers on the same physical host. Instead, performance interference was minimised when I/O and CPU intensive workloads are consolidated on the same machine.

The above paper focused purely on the performance effects of performance interference, similar to Patros et al. (2016). What the two papers have in common is that they both focus at a tenant level but take a holistic view of the overall system. i.e. they have all the information about each tenant in the environment. They also focus on the behaviour of what could be considered the 'victim' tenant, in contrast, this project takes the perspective of an adversary tenant e.g. the cloud burner.

Ristenpart et al. (2009) investigated approaches that could lead to the introduction of new vulnerabilities in a public cloud environment by using Amazon's EC2 (VM based IaaS platform) as a case study for the paper. The approaches the authors explored and proposed involved: Generating a mapping of Amazons internal cloud structure where EC2 instances are hosted; determining if it was possible to place a provisioned VM with a target VM; determine if it was possible to confirm co-residency between VMs; finally, if cross-VM side channel attacks are feasible. The authors mapped the internal cloud structure of Amazon by observing both the internal and external IP addresses given to EC2 instances and using network tools to observe RTT's and hops of request packets. Using these tools also allowed the authors to determine if two VMs are co-located on the same machine. This was achieved by comparing the first hop a packet takes on each VM, this first hop was the privileged VM responsible for managing all customer VMs on a host. Thus, a network-based co-residency detection method was proposed. Following the effectiveness of co-residency detection and assuming a victim and adversary tenant are co-located; the authors investigated CPU cache-based side channel attacks to extract information from the victim. Their research yielded a cache-based co-residency detection to compliment the network-based method. The authors also proposed a load-detecting approach by measuring the back pressure from the CPU-cache usage.

Within the paper, the authors acknowledge that there are multiple resources that could be used to implement side channel inference attacks, but their research focused on the CPU-cache. This project aims to investigate side channel inference attacks using a greater range of resources to potentially increase the attack surface. The majority of the approaches used by the authors in Ristenpart et al. (2009), prior knowledge of the victim tenant was necessary or at the very least had a publicly accessible port (such as a webserver would have). This project takes the position of an adversary tenant with no prior knowledge of co-located tenants. Ristenpart et al. (2009) also purely studies the above approaches using VMs.

Joshi et al. (2017) presented *Sherlock*, a lightweight mechanism designed to detect performance interference in containerized cloud services. The goal of *Sherlock* was to identify and quantify the level of performance interference experiences by a cloud deployed web service. The authors designed the system to be deployed by a cloud customer and does not require any knowledge of the system it runs non knowledge of any co-located tenants. *Sherlock* accounts for this by first running the web service in a sandbox environment with artificial load on the service, this allows the calibration of expected performance. When the web service is deployed into a containerized cloud environment, it uses the same metrics used in calibration to determine if any deviation due to performance interference occurs. If a deviation was measured, this represents the magnitude of the interference. The metrics and calibration method *Sherlock* use are based off the fact that the Last-Level Cache (LLC) of a CPU is shared between each tenant on a physical host. Using this knowledge, Sherlock monitors both CPU utilization and throughput to gather statistics in pairs and plot them in a 2-dimensional space to create an expected performance curve. The authors state that *Sherlock* achieved an accuracy of between 89% and 98.4% for identifying interference. This accuracy metric is calculated by correctly identifying a data point as being affected by interference within the correct time interference was applied.

This project aims to take a similar perspective as Joshi et al. (2017) as no prior knowledge of any tenants co-located should be required. However, *Sherlock* is limited as it only applies to detecting interference for web services, the authors acknowledge that it could be extended to further types of applications and this would require monitoring of different resources and metrics. Sherlock also only accounts for the interference caused from the LLC, this project aims to investigate further hardware resources for evidence of interference.

Delimitrou and Kozyrakis (2017) presented a practical system *Bolt*, this system uses performance interference to measure resource pressure of co-located VM tenants and used machine learning to classify the type of application running in the victim VM. *Bolt* focused on 10 resources and used micro benchmarks to apply artificial load to a specific resource. Using the micro benchmarks, *Bolt* steadily increases load until it encounters pressure from another VM (by comparing the benchmark as if it were in isolation) and notes at what utilization the pressure was encountered. Using these measurements, they were feed into an online machine learning model that is a hybrid recommender to classify the type of application and characteristics

(like dominant resource) running on the victim. Employing a shutter mode, *Bolt* could also distinguish between multiple co-located tenants. The period required for classification is stated as around 2-5 seconds for profiling and milliseconds for prediction. Using *Bolt*, multiple types of security attacks were able to be launched to degrade the performance of the identified tenants, such as Denial of Service (DoS) and Resource Freeing Attacks (RFA). Using the knowledge from *Bolt*, mitigation and detection techniques against these attacks were rendered useless. Isolation techniques like thread pinning and cache partitioning were used to observe the effect on the accuracy of *Bolt*; it was found that each technique slightly reduced the accuracy, but not technique was able to mitigate Bolt making correct classifications.

This research aims to closely replicate the results achieved by *Bolt* and aspects of the authors methodology, with the difference of focusing on containers and using Kubernetes as the cloud environment. The use of machine learning was significant in *Bolts* accuracy so this will also be incorporated as a part of the project. *Bolts* authors reveal very limited details regrading the architecture, design and implementation of the system and this makes it difficult to understand just how Bolt makes its predictions. *Bolt* only uses 4 datapoints collected from the profiling stage that are used as input to their machine learning model, this is interpreted as a limitation of *Bolt* and this project aims to uncover a more comprehensive snapshot of the system.

3 Approach

Corresponding to the research question of determining if a co-located tenant application or workload can be inferred within a containerized cloud environment using performance interference as a medium to extract this confidential information. This research aims to take the perspective of a malicious cloud tenant and investigate the viability of this approach occurring in a public cloud, where little or no knowledge of the system is required prior. The first step of the approach in investigating the research objective was to lay out the key aspects that needed to be addressed within the investigation. These aspects in sequential order consisted of: First, how to induce performance interference within the underlying hardware infrastructure; determine if this performance interference could be observed and if so, how to measure it; using the gathered data to determine if a co-located tenant is present or executing; finally, how to use the data to identify operations of that tenant. Thus, the main goals this research aimed to satisfy were confirming that performance interference can cause a leaky side channel in a containerized cloud environment and the information extracted can be used to infer what should be confidential information in a virtualized environment.

3.1 Design

A goal of this investigation into the above research question was to produce an artefact in the form of a portable and modular application. This application would be capable of being executed within a cloud environment and return a prediction on a co-located tenant task or workload type as a result, or the option to export the raw performance interference data. This information can then be used by another application, one that takes advantage of gaining this knowledge of a tenant. Knowledge that should be confidential to the co-located tenant itself as its activity should be obfuscated by the isolation techniques used by the cloud provider.

The environment used to conduct the body of the research is a public cloud environment, and to replicate a public environment, a local Kubernetes cluster was be provisioned using Docker as the container engine. This virtualization technique provides the isolation between co-located workloads in the node, this environment is shown in Figure 2. A co-located tenant is classed as a pod deployed and executing in the node, and this pod could potentially contain 1 or more Docker containers.

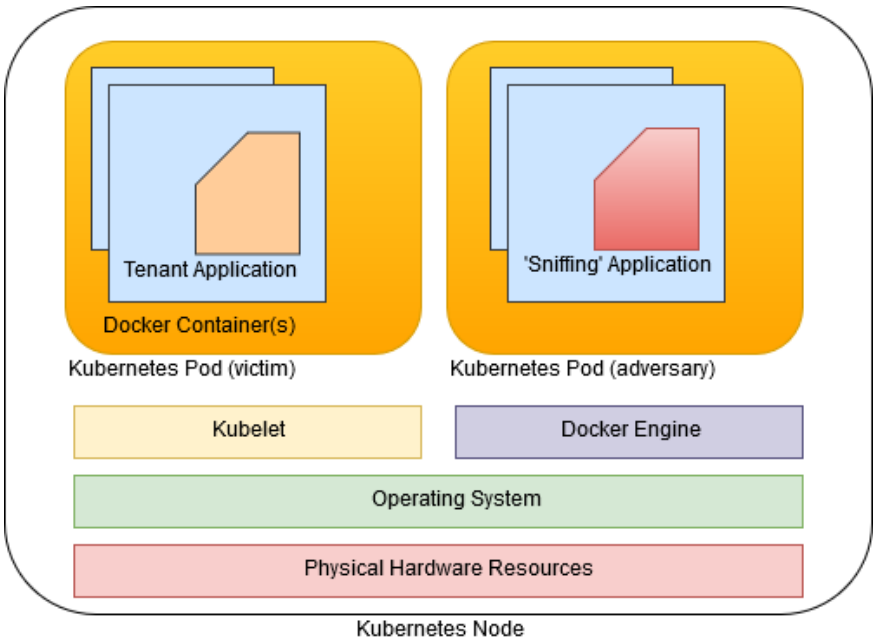


Figure 2: Target Cloud Deployment Environment

3.1.1 Architecture

The inspiration for the approach taken was sourced from both Delimitrou and Kozyrakis (2017) and Patros et al. (2016), where the plan was to create a set of micro-benchmark applications, referred to from now on as 'sniffers'. Each sniffer targets a specific resource and aims to introduce contention in the resource between itself and a co-located tenant. The assumption was that if a high level of contention can be induced, the sniffer will be able to measure the backpressure caused from the co-located tenant. Executing the set of sniffers acts as a profiling stage, where if a broad range of resources are chosen, the resulting data set should represent the current state of the system at that point in time. The data extracted from this profiling stage is key to being able to infer the workload of the co-located tenant. To be able to infer what a co-located tenant's executing tasks are, insights and characteristics in the collected data need to be identified or classified. The strategy to do so is taken from Delimitrou and Kozyrakis (2017) and this is to incorporate the use of a machine learning model. Thus, feeding the data from the profiling stage and into a trained machine learning model, where the goal

is observing prediction accuracy statistics regarding the tenant workloads predicted. This high-level architecture is illustrated in Figure 3. The benefit of this approach for identifying a co-located tenant executing task is it allows a modular application to be created that would support repetitions of this profiling method to be executed, and this enables the ability to track the state of the Kubernetes node by predicting the tenant workloads as they vary over time.

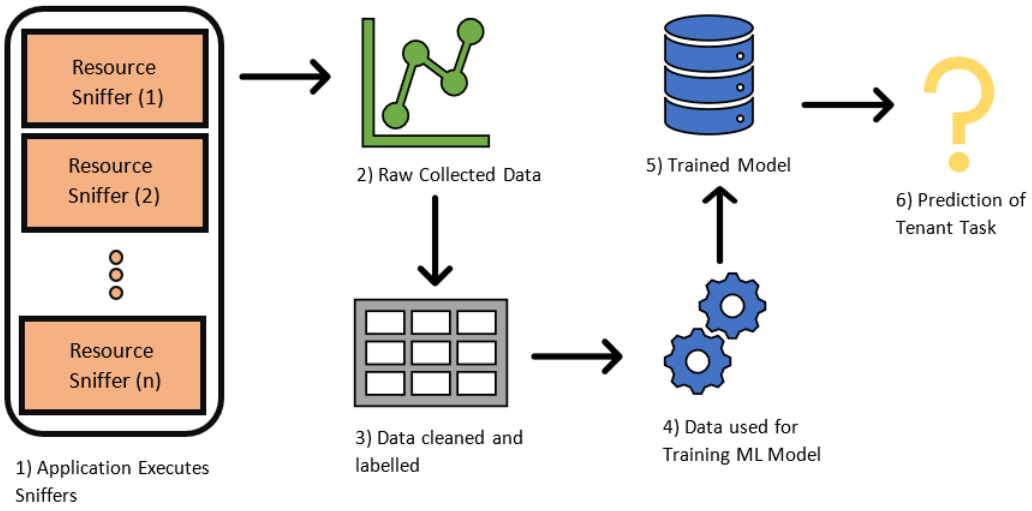


Figure 3: High-level Architecture

3.2 Methodology

The research methodology approach consisted of first identifying the key resources to target, and the plan was to select a set diverse set of resources, where each potentially would give a unique perspective on the system state and load. From the conclusions drawn in both Delimitrou and Kozyrakis (2017) and Patros et al. (2016) regarding the significance of the resources each paper focused on, the following resources were investigated in order:

1. **CPU Data Cache:** The CPU cache was the first resource investigated because monitoring the CPU data caches of a system can give an insight into the memory accesses of an executing workload. Commonly there are 3 levels of CPU data cache, and by measuring the backpressure in

each of the 3 levels, the aim is to uncover patterns or characteristics in a co-located tenant.

2. **CPU:** Monitoring the CPU intensiveness aims to capture the characteristics of a tenants utilization and the frequency instructions are executed.

3. **Memory:** Similar to the data caches above, monitoring the systems accesses into the memory aims to identify the type of data the workload is processing.

4. **Disk:** As the underlying hard disk is an off-core resource and represents the storage sub-system on a machine; monitoring this resource can give an insight into the workloads writing and reading patterns to the disk. Workloads heavily utilizing the hard disk are potentially persisting large amount of data or using data that is too large to fit into the main memory of the system.

5. **CPU Instruction Cache:** Commonly, there are only 1 or 2 instruction cache levels in a machine, and the first level is split with the level 1 data cache. The aim of profiling the instruction cache is to detect any patterns of the tenant workloads instruction execution. If a large backpressure is measured, this could indicate the co-located tenant is having to execute a high number of different instructions by fetching them from main memory.

6. **Network:** Another off-core resource to identify diverse workloads is the network traffic of the system. This could assist in determining if a tenant application utilizes or requires network communication.

For each of the above resources, techniques to cause contention in the specific resource were investigated. This involved conducting further research into how each resource operated and interacted with other resources at the systems level. In conjunction with techniques on how to induce contention in the resources, a set of metrics needed to be proposed in order to be able to measure any backpressure due to performance interference in the resource. Additionally, the method of how to collect the data was proposed. In contrast to the method Delimitrou and Kozyrakis (2017) used in *Bolt*, where a single data point was taken as the measurement of the interference intensity in each resource, i.e., each profiling run returned 4 datapoints. The approach

for this research is to extend the profiling stage to monitor and measure the resource interference over a defined time range. Deciding that the profiling stage for each resource would gather multiple points of data over a time period allows the gathered data to represent a timeseries dataset. The goal of using a timeseries dataset for each resource would allow a greater depth of information to be extracted about the co-located workload in comparison to only a single data value. As this method should allow for more complex patterns and characteristics to be identified, the complexity would be at the expense of the speed of the profiling stage (depending on the magnitude of the profiling time chosen).

Using theoretical knowledge of how to induce contention in the specified resources and the strategy to measure multiple datapoints, an independent sniffer was developed for each of the resources above. At the conclusion of each sniffer's development, an intermediate evaluation phase was carried out to ensure that the theoretical contention could be experimentally observed and if any modifications or refinements were to be made.

These evaluation phases included creating a set of scripts to automatically run the sniffer and plot the datapoints or choose to export the raw datapoints for further analysis. The executing environment for these intermediate evaluations was within an un-isolated system, i.e., the development environment. Following the development of the sniffers and the evaluation phases, the next step in the approach was to investigate the effect of interference in the cloud environment.

This was conducted by first running the complete set of sniffers in a bare metal environment on top of a Linux operating system, first in isolation (only application executing on the hardware) to get a baseline. Following gathering this baseline data, the sniffer application was containerized into a Docker container for deployment into the Kubernetes node. This Docker container was then executed as a Kubernetes job on the node, again in isolation (only application executing in the virtualized environment). The resulting data was compared with the bare metal experiment data to investigate the effect of containerizing the sniffers and observe the influence Kubernetes components had on performance interference.

Following the isolation experiments, further experiments were carried out in the Kubernetes cloud environment using a copy of the sniffers acting as a co-located tenant. The purpose of this was to observe the level of backpressure measured in each resource under perfect conditions, when two identical sniffers are causing maximum contention in a resource. If these experiments

were unable to yield any measurements of backpressure, achieving a successful result for the research question would be in doubt.

Following the proof of concept experiments for the sniffers, the approach was to apply the sniffers Kubernetes job to a set of diverse real-world applications. This environment would emulate a real-world situation where the sniffers Job represents a malicious tenant and the other tenants become victims. The aim of this step in the investigation was to produce a raw dataset that contained multiple profiling stages of the sniffers while the real-world workloads were also executing, one at a time so there was only ever one co-located tenant in the environment.

The above dataset was used as the base of investigating timeseries analysis techniques on the data. The goal was to observe what characteristics and key statistics could be extracted from a profiling run and condense the data from many raw values, to a set of relevant statistics. Additionally, the purpose of performing timeseries analysis on the raw dataset was to produce a separate labelled dataset where a row in the dataset contained both labels and attributes for each execution run of the sniffers. The attributes for a run represented the extracted time-series statistics, and the labels included what real-world workload was executing (co-located tenant) while the sniffers profiled and gathered the data. Using the extracted characteristics from the raw dataset, this labelled dataset was used to investigate the effectiveness of applying machine learning techniques to predict the labels (workloads) within the data. The approach was to use this dataset to both train and test multiple machine models using cross validation and compare the accuracy of each. As the final step in the research approach, the final accuracy values allowed for drawing conclusions regarding the initial research question of whether exploiting performance interference enables co-located tenant task information to be extracted in a containerized cloud environment.

4 Implementation

This investigation required the development of individual applications to monitor interference, handle data and train machine learning models.

4.1 Sniffers

Each sniffer is an independent application that aims to induce contention in a specific resource and measure the magnitude of the backpressure caused by another tenant over the duration of the sniffer's execution. To achieve reliable and consistent data, the C programming language was chosen to implement the sniffers. C was selected because it is an efficient compiled language and allows a finer grain of control over memory allocation. The reason a higher level of memory control and a compiled language were best applicable for this situation was to reduce the level of un-controlled entropy in the system when the sniffer is executing. The need to reduce randomness in a sniffer is to ensure that when the data is collected during execution, no other processes have an observed affect within the data. Minimising inefficiencies in a sniffer was also a requirement because to achieve the goal of inducing contention in a resource, a high frequency of instructions or memory allocations (etc.) were required to occur in order to stress that resource. In contrast to using a compiled language, if an interpreted language (e.g. Perl) was chosen for the critical section of a sniffer, it is at the discretion of the language interpreter for how memory is managed within the executing program. If an interpreter were to execute a garbage collection process during a critical time, it has the potential to influence the data collected, this is because the sniffer aims to measure the backpressure purely from a co-located tenant and this rogue backpressure could jeopardise the integrity of the data. The same concept applies to runtime languages, such as Java that utilise a VM to compile byte code.

4.1.1 Structure of sniffers

Each sniffer has the same core structure, a total of 3 sections make up each sniffer (Figure 4). The first section is the initialization phase, this phase takes the parameters passed to the program and instantiates any structures or memory requirements that will be used during the phase that collects data. Following the initialization section, the critical section is executed.

The critical section utilizes structures or memory allocations set up in the previous section to induce contention in the target resource and measures the performance interference, i.e., collect and store the data. The critical section of each sniffer is stripped of any code or inefficiencies that could potentially reduce the magnitude of performance interference caused within the section. At the conclusion of the critical section, the data has been collected and the final section frees any memory or structures used within the critical section, before exiting.

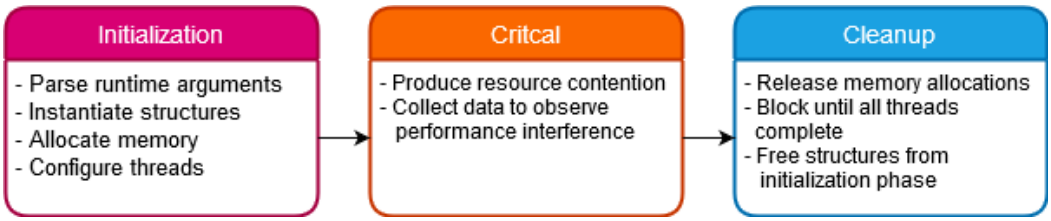


Figure 4: Sniffer Structure

The critical section in each sniffer follows the same structure for both collecting and persisting data. The purpose of each sniffer is to induce contention in a resource and measure the backpressure over a defined time range, this range is passed as an argument to the sniffer and represents the duration in seconds for which the sniffer should execute its critical section. The critical section is comprised of a loop and a set of operations that get executed for each loop iteration. The critical loop iterates until that defined time range is exceeded and data collected within each iteration includes the total elapsed time the section has been executing for and the time for the current iteration. The metric and unit of time is specific to each sniffer.

Python was the language used to develop the application responsible for executing the sniffers and handling the data generated. Python was selected to implement the application because of the wide range of libraries available to handle data and Pythons ease of use. The previous argument against using an interpreted language does not apply here, as this orchestrating application handles the data management after the sniffers have executed. Hence, this application can be passed runtime parameters and either plot the results from each sniffers execution or export the sniffer data files. The software architecture of the entire sniffer application is shown in Figure 5.

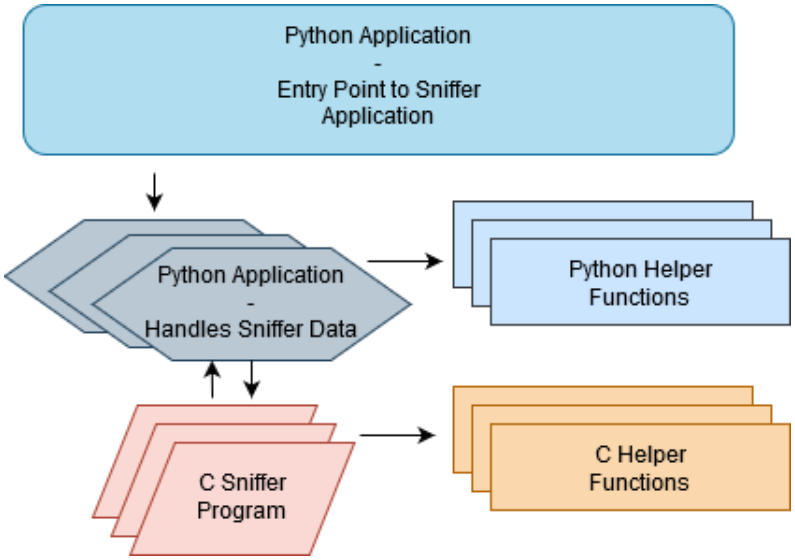


Figure 5: Sniffer Application Architecture

Docker was used to containerise the python application and the sniffers. A docker file was created to instantiate an image representing the application that can be deployed as a container, and the image layers used are shown in Figure 6. The docker command within the file automatically executes the application. The runtime parameters for the python application and each sniffer are configured to be retrieved from an external file in the system, this allowed parameters to be modified without having to repeatedly create a new Docker image.

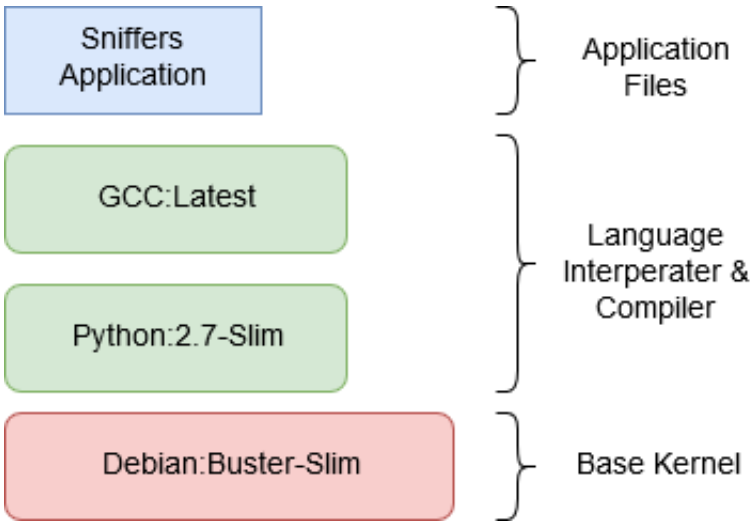


Figure 6: Sniffer Docker Image Layers

4.1.2 Processor Affinity

Processor affinity is the technique that enables a process to be bound to a defined number of CPU cores within a system. Hence, if a process is bound to only a single CPU core, then the process will only be executed on the specified core. If a system does not use process affinity then it is at the distraction of the scheduler to what CPU core a process executes on, and under certain circumstances a process can change to another core during execution.

Process affinity is an aspect that was not addressed within Delimitrou and Kozyrakis (2017) and Bolts micro-benchmarks so it is assumed they did not implement any core affinity. The reason process affinity was addressed in this research was to ensure that the data collected by the sniffer accurately represented the state or load within the system on each CPU core. Hence, a process affinity helper framework is proposed as a part of the sniffers and implemented alongside them. This process affinity framework enables each sniffer that targets an on-core resource to allocate a thread for each CPU core active in the host and bind each thread to that core only, shown in Figure 7. This technique ensures that a copy of the sniffers critical section is independently executing on each core in parallel and the results returned illustrate the interference for only that specific core. This process affinity

should improve the accuracy of the data collected by the sniffers and reduce the level of ambiguity in comparison to not utilizing affinity.

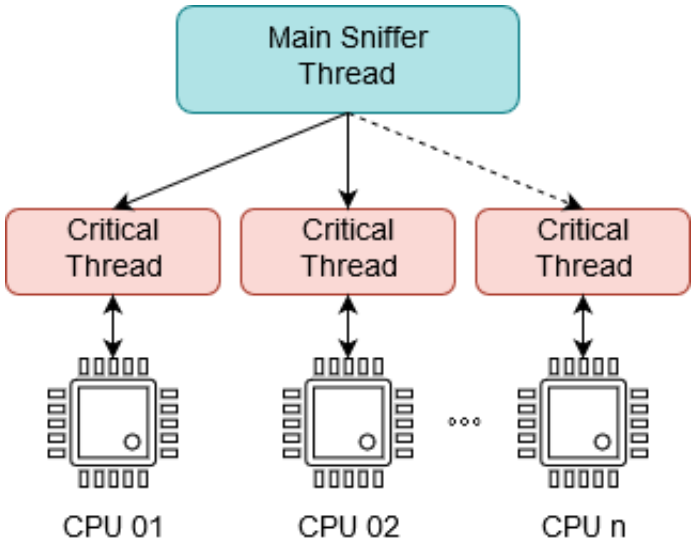


Figure 7: Sniffer Processor Affinity

4.1.3 CPU Data Cache

The CPU data cache sniffer is designed to cause performance interference in each level of every CPU data cache within the host machine. As this research focused on developing sniffers that are designed to be deployed as tenant application, it is assumed that the sniffers are unable to access hardware counters of the host. These hardware counters would have provided absolute values regarding the statistics of cache accesses. In the case of the CPU data caches, hardware counters could provide the number of cache misses for each level. However, as the hardware counters are assumed unavailable, the decided approach was to use time values to convey the magnitude of contention in the data caches, but the limiting factor using this approach is the accuracy of the systems clock.

The specific metric focused on for the data caches is cache latency. The latency of a cache level is defined as the time taken by the CPU to fetch data from the corresponding cache level for use. Hence, the level 1 cache has the

lowest latency and the last level cache has the highest latency. The goal of monitoring and collecting data regarding the cache latency of each level is to observe any change in the latency over time. If the sniffer induces contention by flooding a specific cache with data and then constantly attempts fetch the data, this stresses the cache. When the sniffer is the only application using the cache, a predictable latency should be observed, but if another application (co-located tenant) is also performing cache fetches, the latency to fetch the sniffers data should theoretically increase due to its data being evicted to the next cache level. This increase in latency is taken as the magnitude of performance interference experienced by the data cache.

Hence, the task of the data cache sniffer is to flood the specific cache level with data and then fetch the data while monitoring cache latency times. The implemented algorithm used for the sniffer is inspired from the micro-benchmark proposed by Celio and Caulin (2011). The micro-benchmark is a part of the benchmarking suit Ccbench[7], developed by the authors Celio and Caulin (2011). This micro-benchmark is designed to empirically determine the cache level sizes of a system, and it does this by allocating contiguous arrays of increasing size, then fetching the array data and monitoring the cache access latency. At the point the latencies increase, it can be assumed the array no longer fits in that level of cache and this gives away the size of the cache. Aspects of the algorithm have been taken and adapted for the purpose of monitoring cache latencies given a specific cache size as an argument. Hence, each run of the sniffer will collect data regarding one cache level, the general algorithm for this sniffer is given in Algorithm 1.

[7] https://github.com/ucb-bar/ccbench

Data:
$C \leftarrow$ Cache level,
$S \leftarrow$ Size of cache,
$I \leftarrow$ Number of iterations,
$T \leftarrow$ Time
Result: File of timeseries cache latency value
Calculate number of array elements; `/* Initialization section */`
Determine number of active CPU cores;
foreach *CPU core* **do**
    Allocate sniffer thread;
    Allocate non-shared array;
    foreach *Page size in array* **do**
        Allocate array indices based on stride size of cache;
    end
end
$t \leftarrow 0$;
while $t < T$ **do** `/* Critical section */`
    Start timer;
    for $i \leftarrow 0$ **to** I **do**
        Traverse array by using array elements as indices;
    end
    Stop timer;
    $t \leftarrow t+$ elapsed time;
    Output(L, threadID, t, (elapsed time)/I);
end
Wait for all CPU threads to finish; `/* Clean up section */`
Free arrays and pointers;

Algorithm 1: High-level CPU Data Caches Sniffer Algorithm

A key aspect that had to be accounted for when attempting to induce contention within a target cache beyond the level 1 cache was to ensure that a large number of cache misses were forced within the lower level caches. A cache miss is when the system attempts to fetch data from within a cache region and finds that the data does not exist within this layer, it then must continue to look in the next cache region. These cache misses must occur because when the target is the level 2 data cache, any data retrieved directly from the level 2 cache will be stored back within the level 1 cache. Hence, this is an issue because the data needs to be stored back into the level 2 cache

when the system next fetches it. To achieve these cache misses, the data values within the array are initialised using a technique that takes advantage of how the system fetches data from a cache. When the system is instructed to fetch data, it fetches a small contiguous region of data, and this is because the system expects the data it will need to fetch next will be near the previous data. This region of memory fetched is known as a cache line. Thus, the technique is to ensure that accesses into the sniffer array are separated by at least the size of a cache line, and this will thwart the systems ability to take advantage of data locality.

Another systems level efficiency that needed consideration was the data prefetcher. This prefetcher looks ahead of the currently executing instruction and fetches data for the following instructions, this effectively removes any data retrieval latency. This is an issue because if the prefetcher can effectively pre-empt the sniffer's data, it reduces the level of contention that can be induced in the target cache. The technique used to overcome the prefetcher was to obfuscate the accesses into the array so that only at runtime would they be revealed. This was implemented by embedding the next array index to be accessed within the current index, this meant the system would only know the next location to access when it processed the previous one.

4.1.4 CPU

The aim of the CPU sniffer was to ensure that it caused a high level of CPU utilization within its critical section. If this sniffer was able to cause a high degree of contention in each CPU of the host, it should allow for the measurement of any backpressure caused by a co-located tenant also trying to perform CPU intensive workloads. The implementation of the CPU sniffer uses a similar methodology to the data cache sniffer, whereby measuring the length of time for each repetition of a task that the sniffer instructs the system to perform and observing the change in time as the magnitude of performance interference.

An aspect that needed consideration when this consideration when designing this sniffer was to ensure that contention was minimised in any other resources in the system. The reason for minimising contention in other resources of the system is to ensure that any performance interference experienced due to co-located tenants is caused from the tenants use of the CPU and no other resources. If other resources impacted this sniffer, it would reduce the ability of this sniffer to illustrate the state of the CPU utilization in

the system. Hence, it was ensured that the CPU workload chosen minimised memory and data manipulations.

The CPU intensive workload of the sniffer has modified aspects of the CPU cloud burner from the set of Java cloud burners proposed by Patros et al. (2016). The workload induces contention in the CPU by forcing it to carry out arithmetic instructions in order to determine if a number is a prime number, the algorithm of the sniffer is given in Algorithm 2. To determine if a number is prime the critical section of the sniffer iterates through a for-loop to test if every number from 2 to the given number is a factor, and this is done by checking if the given number modulus the current number is equal to 0. The concept is to give the sniffer a large prime number as the target number, forcing the CPU core to make a large number of calculations and this maximises utilization of the CPU.

Data:
$N \leftarrow$ Prime number,
$T \leftarrow$ Time
Result: File of timeseries CPU data value
Determine number of active CPU cores; `/* Initialization section */`
foreach *CPU core* **do**
  Allocate sniffer thread;
end
$t \leftarrow 0$;
while $t < T$ **do** `/* Critical section */`
  Start timer;
  for $i \leftarrow 2$ **to** N **do**
    if $(P\%i) == 0$ **then**
      Break;
    end
  end
  Stop timer;
  $t \leftarrow t+$ elapsed time;
  Output(threadID, t, elapsed time);
end
Wait for all CPU threads to finish; `/* Clean up section */`
Free arrays and pointers;

Algorithm 2: High-level CPU Sniffer Algorithm

4.1.5 Memory Bandwidth

This sniffer is designed to induce contention in the systems memory bus, this action is the result of the CPU fetching data from main memory. Memory bandwidth is also known as the memory throughput, and a systems total bandwidth is the maximum number of megabytes transferred per second (MB/s). To observe performance interference within this resource, the aim of the sniffer was to thrash this memory bus and force the CPU to make constant data fetches from main memory. Similar to the data cache sniffer, the data has to be ensured that it is not fetched from a faster memory region, e.g., cache.

The concept is that if the sniffer can make constant data transfers from main memory, the memory throughput can be monitored. Any observed deviation in the throughput can be taken as performance interference caused by a co-located tenant that is also making requests to the main memory. The inspiration for the technique used in the memory bandwidth sniffer was derived from aspects of the STREAM memory bandwidth benchmark proposed by McCalpin (1995). The STREAM benchmark was designed as a synthetic benchmark to monitor sustained memory bandwidth, and at its core measures the performance of 4 long vector operations shown in Table 1. These operations are representative of the 'building blocks' of long vector operations and intended to eliminate data re-use in the benchmark.

Name	Kernel	Bytes
COPY	$a(i) = b(i)$	16
SCALE	$a(i) = q * b(i)$	16
SUM	$a(i) = b(i) + c(i)$	24
TRIAD	$a(i) = b(i) + q * c(i)$	24

Table 1: STREAM Vector Operations (Source:McCalpin (1995))

The sniffer creates a set of 4 arrays that are each at least 4x the size of the largest cache in the system, this ensures that each request into the array causes the data to be transferred from memory as the array is too large to fit into the caches. The critical section of the sniffer then uses those arrays and iterates over each index performing the vector operations in Table 1. As similar to the data cache sniffer, the technique to avoid the prefetcher is used by obfuscating the array indices by storing them within an index of another array. Following the vector operations on the arrays, the resulting

average bandwidth is calculated for the iteration and persisted. The high-level algorithm of the memory bandwidth sniffer is given in Algorithm 3.

```
Data:
S ← Array size,
T ← Time
Result: File of timeseries memory throughput values
Determine number of active CPU cores; /* Initialization section
 */
foreach CPU core do
    Allocate sniffer thread;
    Allocate 4 arrays of size S;
    Initialize Array values;
end
t ← 0;
while t < T do                              /* Critical section */
    Start timer;
    for i ← 0 to S do
        Perform COPY operation;
    end
    for i ← 0 to S do
        Perform SCALE operation;
    end
    for i ← 0 to S do
        Perform SUM operation;
    end
    for i ← 0 to S do
        Perform TRIAD operation;
    end
    Stop timer;
    t ← t+ elapsed time;
    b ← total number of bytes transferred;
    Output(threadID, b/(elapsed time), t) // throughput(MB/s)
end
Wait for all CPU threads to finish;         /* Clean up section */
Free arrays and pointers;
```

Algorithm 3: High-level Memory Bandwidth Sniffer Algorithm

4.1.6 Disk Bandwidth

The disk bandwidth sniffer is designed to induce contention within the underlying hard disk of the system. Like memory bandwidth, the disk bandwidth is the rate at which data can be transferred from the storage medium of the system, this can be in the form of read or write operations. Read operations fetch data from storage for use by the CPU, in contrast write operations transfer data from the CPU and persist it to a hard disk. Using the same approach as the memory bandwidth benchmark the goal is to instruct the system to make a large number of read and write operations to saturate the system's ability to read and write data, while monitoring the throughput.

This sniffer takes inspiration from the Java-based disk cloud burner proposed by Patros et al. (2016) and modifies aspects for the purpose of the sniffer. The disk bandwidth sniffer's high-level is shown in Algorithm 4. The initialization stage of this disk bandwidth sniffer involves creating a temporary file, and this file will be used to persist and read data during the critical section of the sniffer. The disk block size for the system is also determined, the block size is the smallest readable or writable unit that can be addressed in the file system. This block size is the number of bytes written and read within the critical section. An array also gets allocated and instantiated with random digits, this represents the data that will be used in the critical section. The critical section of the sniffer first writes all the data within the array to the temporary file, the sniffer then flushes the write buffer to ensure the file's in-core state is synchronised with the storage device. Following this the same data is then read back from the file, again the buffer is flushed to ensure all the data has been read. The resulting data outputted is the time taken to perform both the read and write operations.

Data:
$W \leftarrow$ Write/Read size,
$T \leftarrow$ Time,
(Optional: $P \leftarrow$ File path)
Result: File of timeseries read and write time values
Determine number of active CPU cores; `/* Initialization section */`
foreach *CPU core* **do**
    Allocate sniffer thread;
    Create temporary file;
    if $W == 0$ **then**
        $S \leftarrow$ System block size;
    else
        $S \leftarrow W$
    end
    Allocate read and write array of size S;
end
$t \leftarrow 0$;
while $t < T$ **do** `/* Critical section */`
    Instantiate write array with random digits;
    Start timer;
    while *Data written* $< W$ **do**
        Write remaining data to file;
    end
    while *Data read* $< W$ **do**
        Read remaining data from file;
    end
    Stop timer;
    $t \leftarrow t+$ elapsed time;
    $b \leftarrow$ total number of bytes transferred;
    Output(threadID, elapsed time, t);
end
Wait for all CPU threads to finish; `/* Clean up section */`
Free arrays and pointers;
Free read and write buffers;
Delete temporary files;

Algorithm 4: High-level Disk Bandwidth Sniffer Algorithm

4.1.7 CPU Instruction Cache

The CPU instruction cache sniffer is responsible for stressing each instruction cache within a CPU core. The instruction cache sniffer uses the same principles as the CPU data cache sniffer, this is to cause contention in the instruction cache by causing the system to make repeated requests and fetching a high frequency of instructions. If a co-located tenant is also executing a large number of instructions, the performance interference should be observed by the reduction of instructions able to be processed by the sniffer.

The difficulty in implementing this sniffer was creating code that was able to overcome the instruction prefetcher of the system. This mechanism is able to fetch instructions and store them within the instruction cache before they are needed to be executed on the CPU. This minimises latency, and any latency observed is most likely the latency of fetching data from memory. Equally an issue was the GCC compiler, this was the compiler chosen to compile the sniffers C code into machine code for execution of the sniffers. The compiler uses multiple optimisation techniques to ensure that the C code is compiled down to efficient assembly code and then machine instructions, and even if these optimisations are disabled the compiler condenses code features like for-loops to reduce the number of instructions. This was discovered when the first prototype of the benchmark was implemented purely in C, using techniques to fool the prefetcher. The techniques involved using misleading optimisation commands to cause the speculative execution mechanism to always fetch the wrong instructions (Drepper (2007)), but the volume of instructions produced were not sufficient.

Hence, the approach taken to overcome these issues was to manually create custom assembly code instructions. Implementing assembly code manually, removes the compilers ability to condense or optimise the code, and each assembly instruction is compiled to a corresponding machine instruction. This assembly code is then linked with the sniffer C code by the compiler.

Within the manually generated assembly code there were 12 functions created, each function is made up of the same set of one repeated instruction. The 'nop' instruction (short for 'no operation') is the repeated instruction, this has minimal effect on the CPU as it instructs the CPU to do nothing. Within the sniffer C code (high-level algorithm shown in Algorithm 5), an array is provisioned with integers that correspond to the assembly functions. The critical section then traverses the array and executes the assembly function, thrashing the CPU by fetching all the 'nop' instructions.

```
Data:
S ← Size of instruction cache,
T ← Time
Result: File of timeseries instruction cache latency values
Calculate number of array elements; /* Initialization section */
Determine number of active CPU cores;
foreach CPU core do
    Allocate sniffer thread;
    Allocate non-shared array;
    Allocate array indices with function numbers;
end
t ← 0;
while t < T do                                  /* Critical section */
    Start timer;
    for i ← 0 to S do
        switch array[i] do
            case 1 do
                Assembly function 1;
            end
            case 2 do
                Assembly function 2;
            end
            case 3 do
                Assembly function 3;
            end
        end
    end
    Stop timer;
    t ← t+ elapsed time;
    Output(threadID, t, elapsed time);
end
Wait for all CPU threads to finish;             /* Clean up section */
Free arrays and pointers;
```

Algorithm 5: High-level CPU Instruction Cache Sniffer Algorithm

4.1.8 Network Bandwidth

The final sniffer implemented was designed to observe performance interference relating to network communication caused by a co-located tenant. The network bandwidth is the rate at which the system can transfer bytes of data through the network, this could involve both sending and receiving packets of data. This sniffer was designed to flood the network with outgoing packets of data to saturate the systems network throughput, to achieve this the technique was adapted from the network burner proposed by Patros et al. (2016). This sniffer requires 2 programs to be running in order to execute.

The first program executes in the cloud environment along with the remaining sniffers, this program takes as arguments the URL or IP of a corresponding system to send data packets, the port of the system, the volume of data to send as packets in each iteration and the time the critical section should execute for. The high level algorithm is shown in Algorithm 6. This program initially establishes a connection with the system in which to send packets to. The sniffer then allocates an array with random digits, this represent the data that will be sent across the network. Within the critical section, the sniffer sends the data within the array to the target system and the resulting output is the average rate of throughput measured in Megabytes per second, and this completes one iterations of the critical section. Any deviation in throughput experienced by this sniffer can be taken as performance interference due to a co-located tenant.

Data:
$U \leftarrow$ Url/IP,
$P \leftarrow$ Port,
$S \leftarrow$ Message Size,
$T \leftarrow$ Time
Result: File of timeseries network throughput values
Allocate array of S data values; `/* Initialization section */`
Establish connection with network Sink;
$t \leftarrow 0$;
while $t < T$ **do** `/* Critical section */`
    Start timer;
    while *Data sent* $< S$ **do**
        Send remaining data to sink;
    end
    Stop timer;
    $t \leftarrow t+$ elapsed time;
    Output(S/(elapsed time), t);
end
Free arrays and pointers; `/* Clean up section */`

Algorithm 6: High-level Network Sniffer Algorithm

The second program as a part of the sniffer acts as an external network sink. The concept is that this program is executed on a system external to the cloud environment and becomes the receiver for the data sent from the main sniffer program. Hence, the program accepts a connection from the sniffer and acts as a network sink. This network sink ensures that the cloud environment has to physically send data packets across the network and cause contention.

4.2 Data Analysis

When the sniffer application has concluded a profiling run, the resulting data from each individual sniffer is stored locally within the sniffer in a results file. Python was used to develop an application that retrieves each data file and aggregate all the data, cleans, sorts and prepares the data to be applied to a machine learning model. Python was selected for this purpose because of the data manipulation libraries available.

Pandas[8] is the main python data manipulation library used in this python application. To aggregate the data from a profiling run by the sniffers, a Pandas Dataframe is created, and this Dataframe is an object used to store and manipulate all the data from each sniffer csv data file. The data gets sorted by both sniffer type, CPU thread (for on-core resources) and in chronological order (represents the timeseries). The structure of an example data frame object is shown in Table 2.

	Id	Elapsed Time	Sniffer	Value
0	isolated	0.01	caches-L1_0	1.48
1	isolated	0.02	caches-L1_0	2.95
2	isolated	0.03	caches-L1_0	2.05
...	...	...	...	...
4609	isolated	4.89	cpu-0	44.58
4610	isolated	4.93	cpu-0	50.09
4611	isolated	4.97	cpu-0	43.67
...	...	...	...	...
10005	isolated	0.79	icache-1	0.04
10006	isolated	0.83	icache-1	0.04
10007	isolated	0.87	icache-1	0.04
...	...	...	...	...

Table 2: Raw Timeseries Dataframe Object

Using the Dataframe that represents the raw timeseries values, the python library Tsfresh[9] is used to perform the timeseries analysis on the raw data. Tsfresh provides the capability to automatically extract timeseries statistics from data, the library also uses Pandas Dataframe objects to perform the analysis. Hence, the raw Dataframe object is structured so the Tsfresh library operated directly on it without needing to restructure the dataframe. The tsfresh functions are given the dataframe object and the criteria in which to sort the values by. i.e. ordered by elapsed sniffer execution time. The range of timeseries statistics extracted from the raw data are discussed in the evaluation section of the paper.

The Tsfresh library also returns the timeseries statistics as a Pandas

[8]https://pandas.pydata.org/

[9]https://tsfresh.readthedocs.io/en/latest/

Dataframe. For the purposes of automating the timeseries statistics extraction a Pandas Series object is also required to determine the significance of the extracted statistics. A Pandas series represents a mapping between the profiling runs of the sniffers and the workload that was executing while the sniffers executing. This mapping is also used to train the machine learning models with the timeseries statistics dataframe, the Pandas series enables the correct labelling of the data for supervised learning.

4.3 Machine Learning

To implement the machine learning aspect of the research and determine if the extracted timeseries statistics can be used to identify co-located tenant workloads, python was also used. Python was also chosen for this task because it was efficient to re-use the Pandas dataframe object returned by the Tsfresh library and utilise the machine learning libraries available. The Python library Scikit-learn[10] was the chosen library to implement the machine learning models for this research. The library enables on demand provisioning of machine models, the data from the extracted timeseries statistics will be used to train and test the models.

4.4 Sniffer Verification

Verification investigations were conducted following the implementation of the sniffers and prior to performing evaluation experiments. These investigations aimed to verify that the sniffers successfully stress the key resources they are targeting, and produce minimal contention is the remaining resources.

To implement the investigation, Linux system tools were used to profile the system while each individual sniffer executed, and the profiling results from each sniffer were compared. The profiling tools used in the investigation were:

Perf: Perf[11] is a performance analysing tool, it is designed to statistically profile the entire system including the kernel and userspace code. It utilizes hardware and software counters and tracepoints.

[10] https://scikit-learn.org/stable/index.html
[11] http://man7.org/linux/man-pages/man1/perf.1.html

NetHogs: NetHogs[12] Is a tool for providing real-time monitoring and statistics of the network traffic bandwidth per process usage.

Iostat: Iostat[13] is a Linux tool used for monitoring the system device input and output. The tool can be used to collect storage system input and output statistics.

The results of the verification investigation are shown below in Table 4. Each benchmark was given the appropriate arguments corresponding to the size of the physical resources in the machine (shown in Table 3) and the critical section of each was set to execute for 10 seconds. Each sniffer was executed 10 times and the resulting data in table 4 represents the average over those runs.

OS	CPU	L1i Cache	L1d Cache	L2d Cache	LL Cache
Ubuntu Linux	i7-8700 @ 3.2GHz, 12 cores	32Kb	32Kb	256Kb	12288Kb

Table 3: Verification Test Machine Specifications

From observing the results in Table 4, it is clear that each sniffer significantly stresses the resource that it targets. However, it is also noted that contention is not always avoided in other resources, an example of this is that the instruction cache sniffer has on average only 1.7% less CPU utilisation in comparison to the CPU sniffer. The explanation for this is the instruction cache sniffer forces the CPU to cycle through a high frequency of instructions, resulting in a high utilisation. Another example is comparing the number of last level cache misses between both the memory bandwidth sniffer and the data cache sniffer applied to the last cache level, the results show the memory sniffer with a larger number of cache misses. Again, this is because for the system to retrieve data from main memory, it first must search the last level cache, and this then registers as a cache miss.

[12]https://linux.die.net/man/8/nethogs
[13]https://linux.die.net/man/1/iostat

Sniffer	CPU Utilisation (%)	L1-icache Load Misses ($\times 10^6$)	L1-Data Cache Load Misses ($\times 10^6$)	Last-Level Cache Load Misses ($\times 10^6$)	Network Bandwidth (Kb/s)	Disk Bandwidth (Kb/s)
CPU	97.04	19.7	9.6	0.03	0	1.8
Icache	95.34	5098.3	366.7	0.2	0	14.6
L1 Cache	78.08	0.9	31775.3	0.03	0	1.8
L2 Cache	68.92	6.4	9895.9	0.09	0	1.5
Last Level Cache	68.34	6.3	1249.8	1070.2	0	2.1
Memory BW	42.12	7.8	4430.7	1430.9	0	3.1
Network BW	41.06	1918.7	640.6	114.1	2315.2	16.4
Disk BW	4.23	67.4	21.2	0.3	0	2884.4

Table 4: Sniffer Verification Results

In conclusion of the sniffer validation, the results above in Table 4 validate that the sniffers are effective at stressing the target resource and will be able to induce a high level of contention. Hence, this shows the algorithms and techniques implemented within the sniffers are appropriate for this purpose.

5 Evaluation

This investigation carried out three main phases of evaluation to determine if the approaches used were able to predict a co-located tenant. The first phase evaluated the execution of the sniffers in a cloud environment. Next, the sniffers were subjected to artificial performance interference to observe the data collected by the application. Finally, a dataset was created using the sniffers and used to train and test machine learning models for tenant prediction.

5.1 Kubernetes Baseline

The goal of the following experiments was to observe and compare the effect of executing the sniffer application in an unvirtualized environment and in a Kubernetes cloud environment. These results will illustrate if any of the individual sniffers behave differently within the containerized environment.

The specifications of the machine that both the unvirtualized environment and the Kubernetes node were run on are shown in Table 5. It was ensured that the specifications of the machine used in both the unvirtualized environment and the Kubernetes node were equal, this ensured minimal variation in the underlying hardware resources that each environment utilized.

CPU	L1i Cache	L1d Cache	L2d Cache	LL Cache	Memory
i7-7500 @ 2.7GHz, 2 cores	32Kb	32Kb	256Kb	4096Kb	2GB

Table 5: Unvirtualized Environment Machine Specifications

The Kubernetes cloud environment was provisioned by using the Minikube[14] tool provided by Kubernetes. Minikube provides the capability to provision a single node Kubernetes cluster, and because the scope of this research focuses on performance interference of co-located tenants, a single node is all that is required to be provisioned. This single node was instantiated as a

[14]https://kubernetes.io/docs/setup/learning-environment/minikube/

virtual machine and accessible through the Kubernetes command line interface. This interface is used to deploy the Sniffers Docker container as a Kubernetes Job. A Kubernetes Job is a workload that is designed to execute Kubernetes Pods and ensure the specified pods run to completion and terminate successfully. This workload applies to the sniffer application because each individual sniffer is only designed to execute for a short period before terminating. In these experiments, the Kubernetes Job will only run a single pod and within that pod, a single sniffer container.

5.1.1 Methodology

The strategy of these experiments was to first execute the sniffers in the unvirtualized environment, the sniffer application is the only user process executing in the environment. Hence, the application should not experience any performance interference unless caused by OS processes. The application is executed 10 times and the parameters given to each individual sniffer are shown in Table 6, the description of each parameter can be found in the sniffer algorithms given in chapter 4. Following each iteration of the application the raw timeseries data from each sniffer is collected.

Following the unvirtualized experiments, the containerized sniffer application is then deployed in the Kubernetes cloud environment as a Job workload. Consistent with the unvirtualized runs, the Kubernetes Job is the only workload deployed in the environment, and if any interference is observed it would be resulting from either OS or internal Kubernetes processes. The Kubernetes Job was executed 10 times and the raw results collected to be compared with the unvirtualized data.

The parameters chosen for each sniffer executed the critical section for approximately 10 seconds and allowed for multiple datapoints to be collected per second. An exception is the memory sniffer that executed for 20 seconds, this is due to the slow nature of this sniffer returning datapoints. The disk bandwidth sniffer is configured to use the default block size of the machine, this value was 4096 Bytes on the host machine.

Sniffer	Parameters	Sniffer	Parameters
L1-Data Cache	$C \leftarrow L1$ $S \leftarrow 32768$ $I \leftarrow 77108800$ $T \leftarrow 10$	CPU	$P \leftarrow 20001001$ $T \leftarrow 10$
		L1i Cache	$S \leftarrow 100000$ $T \leftarrow 10$
L2-Data Cache	$C \leftarrow L2$ $S \leftarrow 262144$ $I \leftarrow 7710880$ $T \leftarrow 10$	Memory BW	$S \leftarrow 5000000$ $T \leftarrow 20$
		Disk BW	$W \leftarrow 0$ $T \leftarrow 10$
LL-Data Cache	$C \leftarrow L3$ $S \leftarrow 4194304$ $I \leftarrow 771088$ $T \leftarrow 10$	Network BW	$U \leftarrow 192.X$ $P \leftarrow 55555$ $S \leftarrow 500000$ $T \leftarrow 10$

Table 6: Sniffer Application Parameters

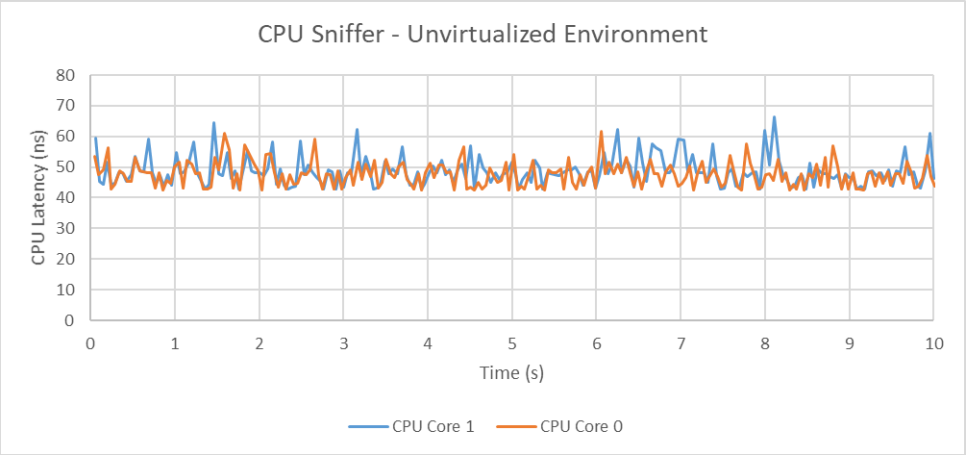
5.1.2 Results

Below, Figures 8, 9, 10, 11, 12 and 13 visually represent the data returned by the sniffer application when applied in both the Kubernetes cloud environment and the unvirtualized environment. If a sniffer utilizes process affinity, then each line in the plot represents a different CPU Core thread.

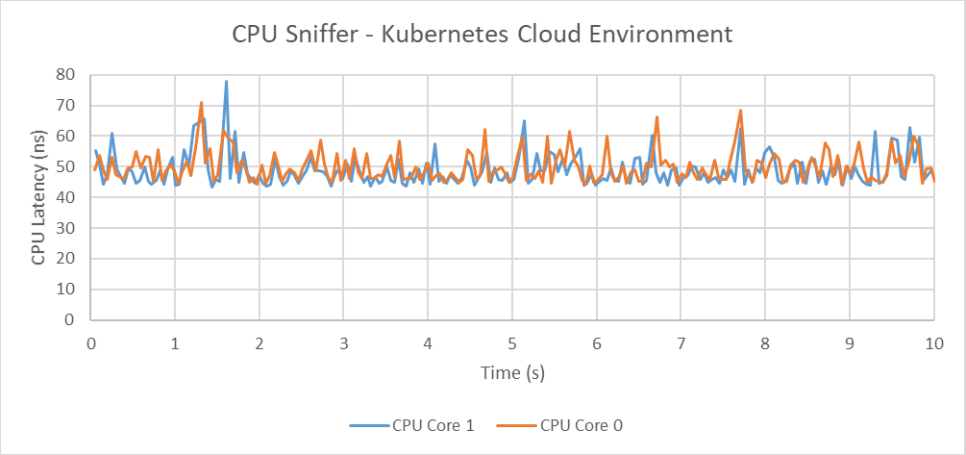
Observed in each figure is that every dataset collected contains a degree of oscillation in the sniffers iteration results. These oscillations convey the fluctuation of resource performance and likely that an aspect of the variation is due to uncertainty by using the system timer to collect the data. However, viewing the general trends of the given data in the figures, the majority of data conveys a stable average over the duration of the sniffer's execution.

Reflecting on the environment comparison for each individual sniffer and the observed metrics, the profiling runs of the CPU, instruction cache and data caches sniffer (Figures 8, 9, 10) collected data points in very close prox-

imity from both environments (each plotted in separate charts to visualise lines clearly). The resulting proximity showed that containerizing the sniffers and the internal Kubernetes processes produced minimal contention within each of CPU, instruction cache and data cache levels. However, one aspect observed in these results, was the cloud environment data contained at least one spike in latency, a reason for this could have been a Kubernetes process executing and causing interference.

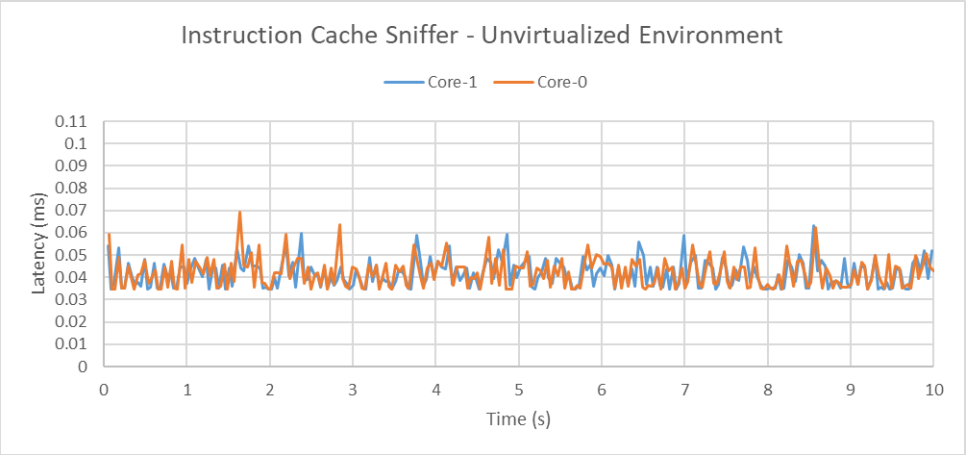


(a) CPU Unvirtualized Environment

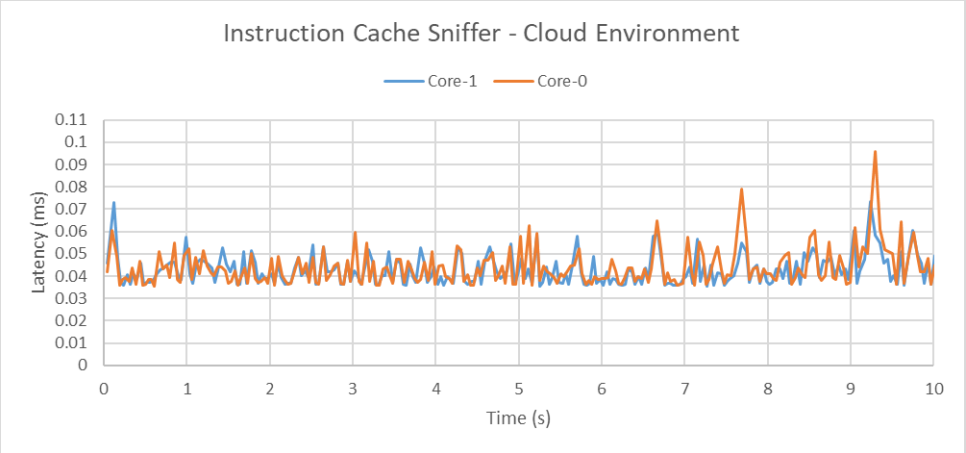


(b) CPU Cloud Environment

Figure 8: CPU Sniffer Comparison

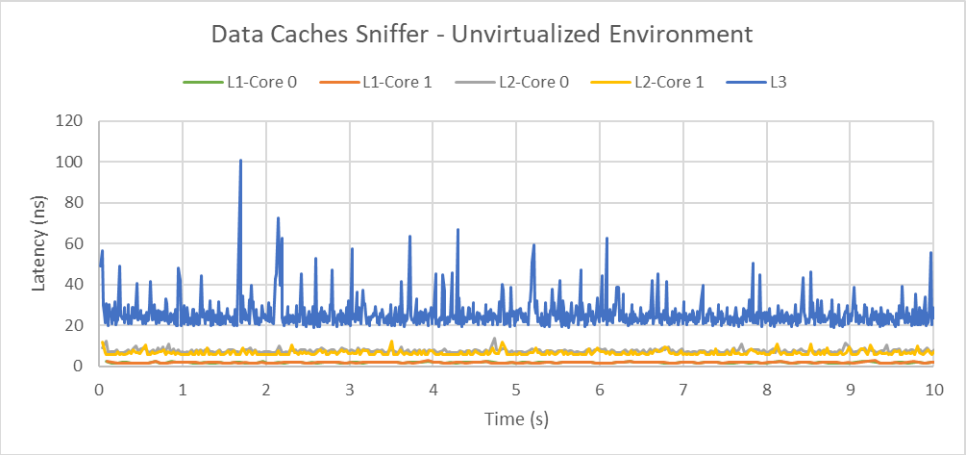


(a) Instruction Cache Unvirtualized Environment

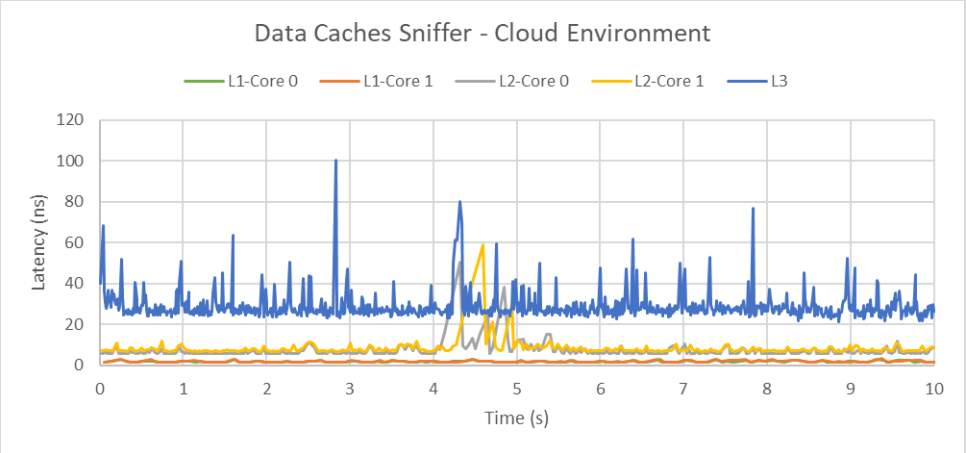


(b) Instruction Cache Cloud Environment

Figure 9: Instruction Cache Sniffer Comparison



(a) Data Caches Unvirtualized Environment



(b) Data Caches Cloud Environment

Figure 10: Data Caches Sniffer Comparison

Further, results from the memory bandwidth sniffer (Figure 11) shows a greater variation in the oscillations and a perceived difference in the collected data from each environment. However, because the oscillations cause the data lines to intersect multiple times and this sniffer executed for twice the duration of the other sniffers, it is difficult to conclude if the cloud environment caused any interference.

The network sniffer (Figure 12) showed that the cloud environment maintained an approximate 0.5 Mb/s higher throughput across the experiments compared to executing in an unvirtualized environment. This result was unexpected, as the expectation was an identical throughput because these experiments involved no network communication, an explanation for this could be varied contention from other areas in the network that the sniffer data packets encountered.

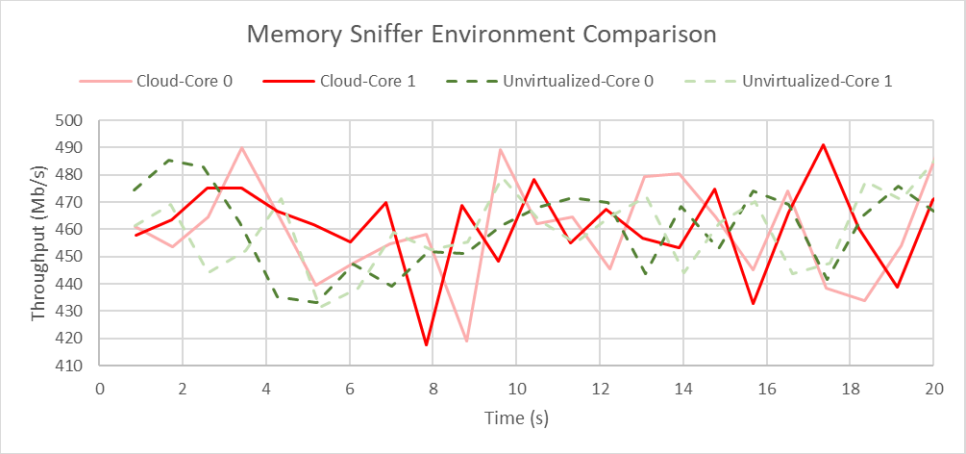


Figure 11: Memory Bandwidth Sniffer Comparison

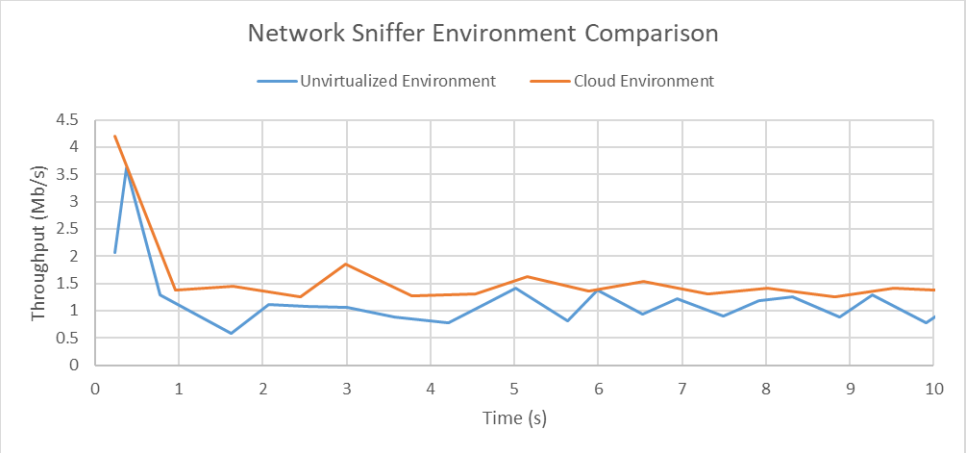


Figure 12: Network Bandwidth Sniffer Comparison

The sniffer that conveyed a large difference between the Kubernetes cloud environment and the unvirtualized environment was the disk bandwidth sniffer, shown in Figure 13. Observed in the disk sniffer chart results are 2 distinct iteration latencies for each environment. In the local unvirtualized environment, the sniffer took on average 20ms to write and read the block of data to the storage disk. In comparison, the Kubernetes environment took on average approximately 4ms to read and write the data, and these results were consistent across each experiment of the disk sniffer. This was also an unexpected result because again, no disk intensive workloads were executing within the environments during the sniffers execution. It is concluded that internal Kubernetes processes did not cause any interference to the disk I/O capacity because of the fact the latency was lower. Hence, the explanation for difference in disk latency between the environment is how Kubernetes facilitates storage. According to the Kubernetes Documentation for storage[15], the on-disk files of a container are ephemeral, and the container writable layers are stored within the nodes root partition and root directory. This method of managing storage for containers in Kubernetes is clearly efficient as illustrated by the results (Figure 13).

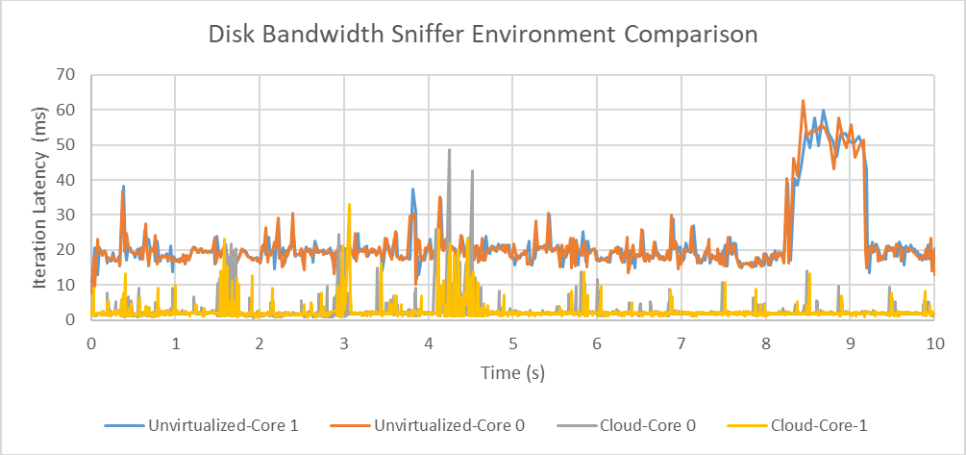


Figure 13: Disk Bandwidth Sniffer Comparison

In conclusion, the experiments show that containerizing the sniffers application and executing it within the Kubernetes environment had no significant effect on the execution characteristics for the majority of the targeted

[15]https://kubernetes.io/docs/concepts/storage/

resources. The exception was the disk bandwidth sniffer, where it was shown that the storage management of Kubernetes had a large effect on apparent read and write operations. The effect observed was a higher throughput and thus a lower latency (Figure 13), further investigation was required to determine if this effects the sniffers ability to cause contention and observe performance interference within disk I/O.

5.2 Interference Maximisation

This evaluation stage aimed to observe the effect a resource intensive co-located tenant has on the data collected during the profiling stage of each sniffer. The purpose of investigating the effect of inducing artificial load on the system was to conclude if performance interference can in fact be detected by the proposed sniffer application. Confirming that backpressure can be measured through the method of inducing contention in key resources aids in providing further evidence that the characteristics and magnitude of the collected data could be used to identify co-located workloads.

This evaluation stage used the same machine specifications (Table 5) and sniffer parameters (Table 6) as the previous baseline evaluation phase. The resulting datasets collected in the following experiments were then compared with the Cloud environment baseline datasets from the previous evaluation stage.

5.2.1 Methodology

Each of the experiments in this evaluation stage were conducted within the aforementioned Kubernetes cloud environment. The plan was to deploy and run the sniffer application while another co-tenant application was already deployed and executing. The desired outcome of the evaluation stage was to see evidence of performance interference effecting the execution of the sniffer. Hence, the methodology was to create a best-case scenario for each individual sniffer, this meant ensuring that a co-located tenant highly stressed only one key resource. To achieve this artificial load, a copy of each of the individual sniffers would be used as the co-located tenant. Using a copy of a sniffer as a tenant meant that it was ensured the maximum level of contention in the resource would occur. Hence, a sniffer is performing a profiling run against a copy of itself.

First, the co-located tenant (sniffer copy) was deployed to the Kubernetes environment, the parameters for the sniffer copy tenant were identical to the sniffer application that was performing the profiling (collecting data), except that the copy will execute for the entire time the profiling run occurs (ensures contention throughout the experiment). After deploying the co-located tenant, the actual sniffer application was deployed to conduct the profiling run. This process was repeated for each individual sniffer application acting as the only co-located tenant. Each experiment was repeated 10 times for consistency.

5.2.2 Results

Below, Figures 14, 15, 16, 17, 18 and 19 illustrate the results of executing the sniffer application within the cloud environment in both isolation, where it is the only workload running, and in the presence of a co-located tenant, where the tenant is a copy of the sniffer executing. Each dataset representing the application executing in isolation in the below charts is the dataset collected during the previous evaluation stage where the sniffers were deployed into the cloud environment.

The results in Figure 14 show that the CPU sniffer applications execution had been influenced by the co-located tenant. The resulting effect was an increase in the latency of the critical section's execution time, this increase was sustained over the duration of the profiling runs. The oscillations between iterations was still present in comparison to the isolated execution but the amplitude of the oscillations also increased. An interesting aspect is also during approximately the first 4 seconds of the profiling duration, CPU core 1 had a higher latency compared to CPU core 2, before decreasing and becoming approximately equal. This difference across CPU cores could be caused by the scheduler unevenly spreading the co-located tenant workload across the cores. This observed result confirmed that this sniffer is able to measure and observe performance interference due to a co-located tenant.

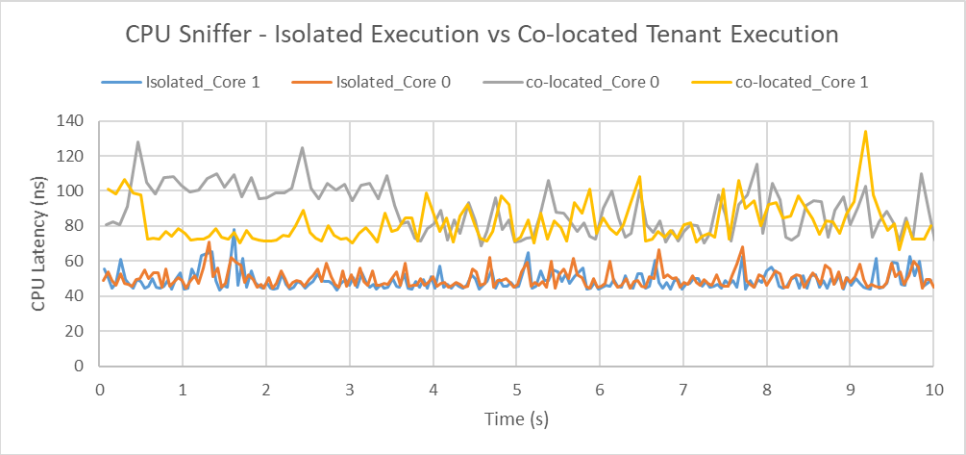


Figure 14: Isolated vs Co-located CPU Sniffer Execution

In a similar result, Figure 15 showed an increase in latency within the critical section execution time of the instruction cache sniffer. Where the isolated latency is approximately 0.04ms on average and introducing a co-located tenant caused the latency to increase by approximately a factor of 2. Also similar to the above CPU results, the instruction cache latency appeared to have a greater variation across iterations when executing in the presence of a co-located tenant. In contrast, both CPU cores had near identical execution patterns, showing that any latency variation was experience equally within each core.

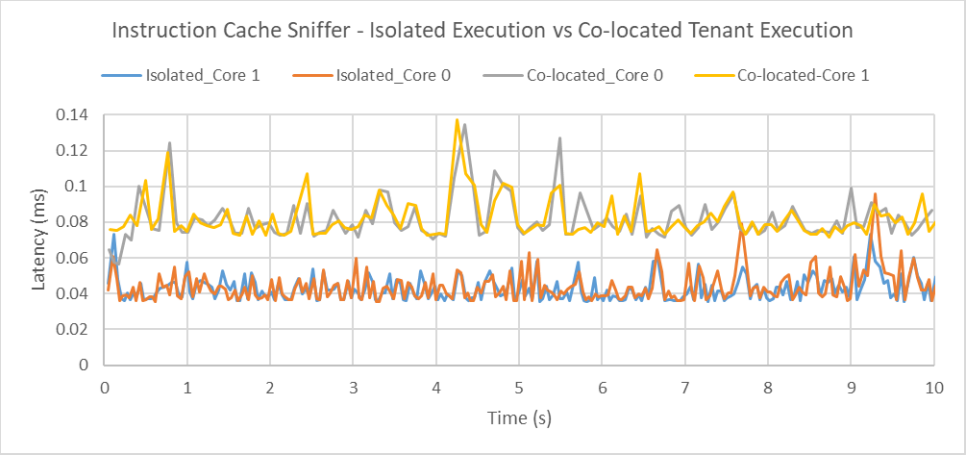


Figure 15: Isolated vs Co-located Instruction Cache Sniffer Execution

For both the network sniffer (Figure 16) and the memory sniffer (Figure 17), the throughput measurements decreased when the sniffers executed in the presence of the co-located tenant. This throughput drop showed that both sniffers were sensitive to the contention in each respective resource caused by the co-located workload. From the charts, both sniffers had shown to be able to measure performance interference from a co-located tenant. The data also is observed to have little deviation across iterations under the uniform workload introduced, if a co-located workload contains variations in its operations, this variation is expected to be reflected in the sniffers gathered data.

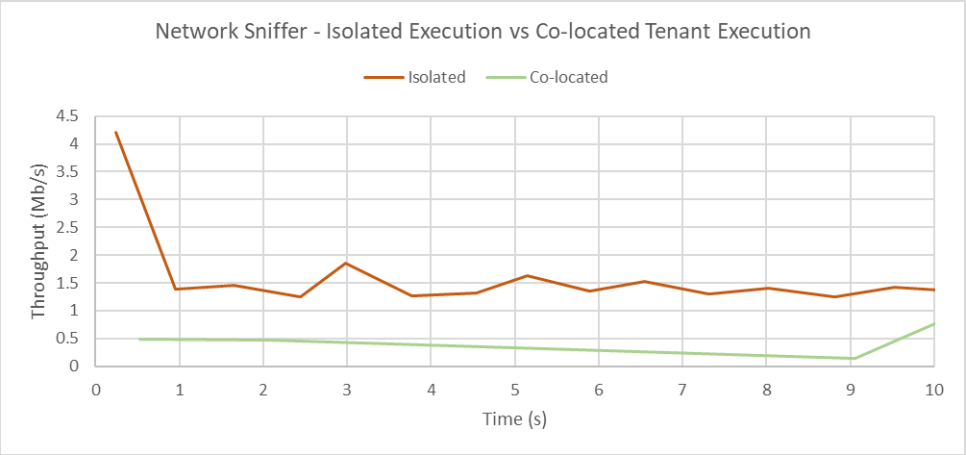


Figure 16: Isolated vs Co-located Network Sniffer Execution

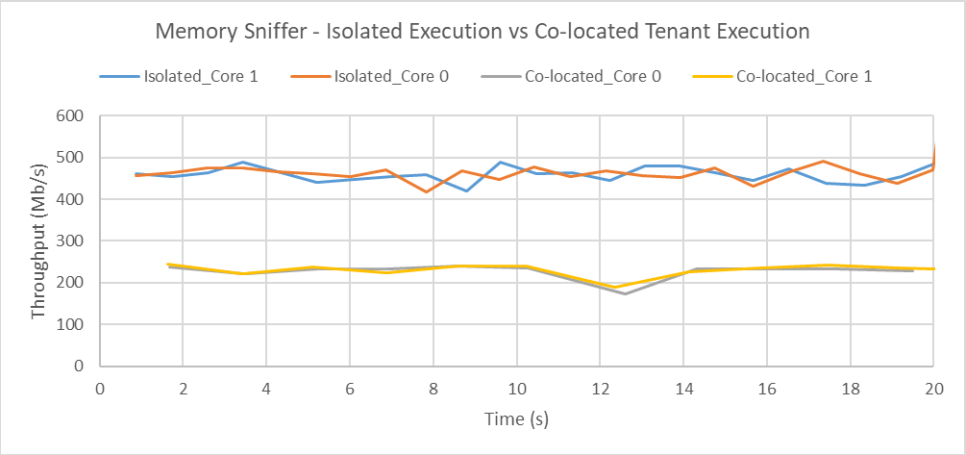
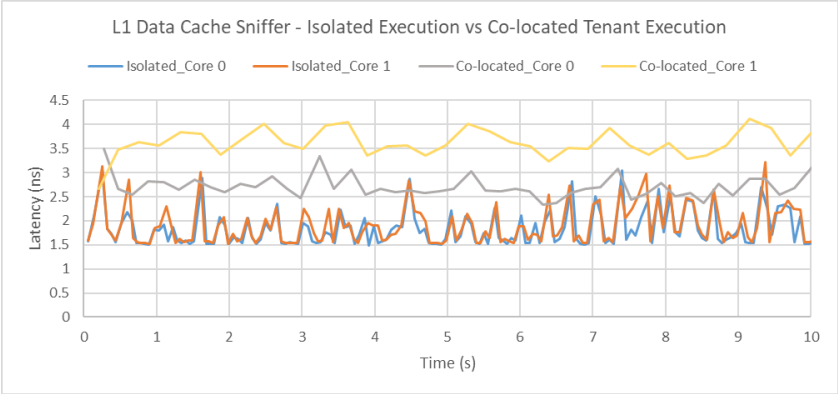


Figure 17: Isolated vs Co-located Memory Sniffer Execution

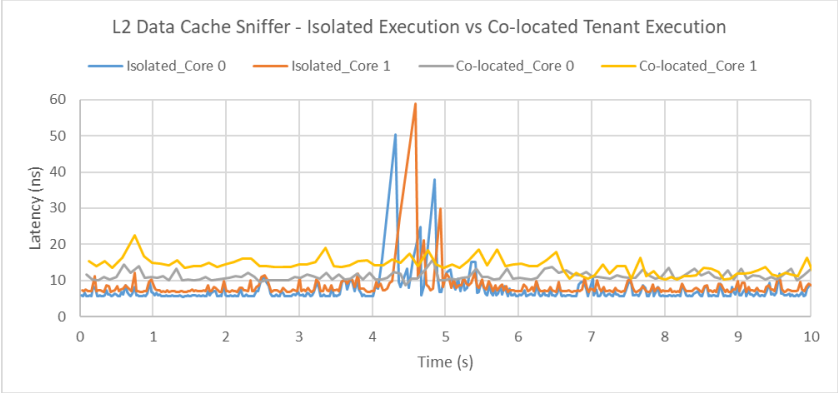
The data cache sniffer results have been split across 3 charts (Figure 18) to show the effect the co-located tenant had on each of the data cache levels. Figure 18a shows the effect on the level 1 data cache, and observed is an increase in latency due to the co-located workload. However, the increase was not equal across both CPU cores, CPU core 1 had a higher average latency than CPU core 0. This increase again showed the interference was not consistent across cores like the CPU sniffer results. In contrast, the patterns of the latency data over the duration of the critical section were similar in both cores, this showed that when a latency variation occurred, it was experienced by both cores but in different magnitudes.

Corresponding to the results of the level 2 cache (Figure 18b), the cache latency increased when a co-located tenant was introduced, similar to the level 1 cache results. With the exception of the increase in latency burst within the isolated execution results, the comparison between the two datasets illustrated the same characteristics as the level 1 data cache results.

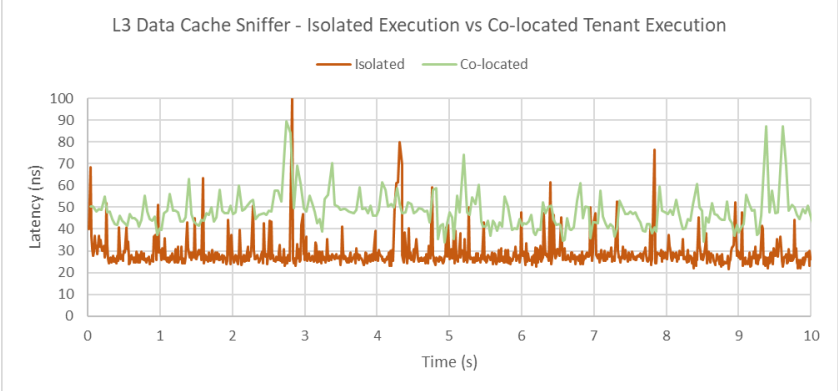
Finally, the last level data cache results shown in Figure 18c, continued the trend of the latency increasing when executed alongside a co-located tenant. However, a greater level of variation is observed across iterations of the critical section. Each of the data cache level results demonstrates that this sniffer has the ability to measure performance interference caused by a co-located workload and the effect of resource contention can be visualised at each data cache level.



(a) Level 1 Cache Execution



(b) Level 2 Cache Execution



(c) Level 3 Cache Execution

Figure 18: Isolated vs Co-located Data Caches Sniffer Execution

The disk bandwidth results are shown in Figure 19 and show that applying the sniffers in the presence of a disk intensive co-located workload resulted in no significant increase in latency of the sniffer iterations and no decrease in throughput. This unfavourable result confirmed that the Kubernetes storage management techniques makes it difficult to induce contention within the disk I/O. This casts into doubt the sniffers ability to measure backpressure from a co-located tenant, and as such, will make this sniffer potentially redundant when using it as a tool for tenant identification.

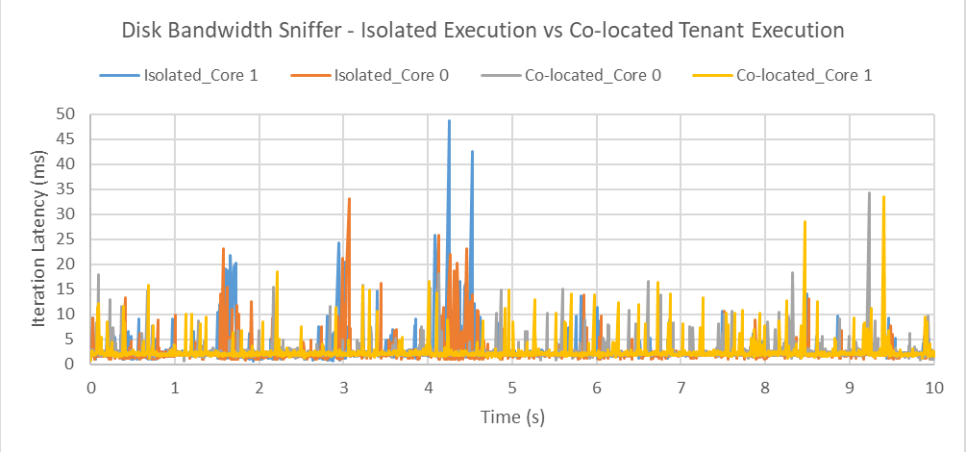


Figure 19: Isolated vs Co-located Disk Sniffer Execution

In conclusion, the above results for the CPU, instruction cache, memory bandwidth, data caches and network bandwidth sniffer prove that each is successfully able to measure a magnitude of performance interference due to a co-located workload. However, it must be noted these experiments were conducted using artificial load and maximised the potential for contention in the target resources. In contrast to artificial load, a real-world scenario or tenant workload could potentially have a larger variation and smaller magnitude of interference. In both of the evaluation stages, the disk bandwidth had produced unfavourable results. However, the disk bandwidth sniffer will continue to be incorporated in the sniffer application for further evaluation of its ability to measure performance interferencea and predict co-located tenants.

5.3 Co-located Tenant Identification

The final stage of the evaluation was determining if the proposed sniffer application can extract data and be used to identify a tenant's workload. The results yielded from this evaluation stage conclude if using the approach taken and techniques used in this investigation give proof that performance interference enables a side channel to emerge and confidential information can be extracted.

Consistent with the previous evaluation stages, the Minikube Kubernetes single node cluster was used as the cloud environment and sniffers were given identical parameters (Table 6). This environment was where multiple real-world application workloads were deployed as co-located tenants, these workloads acted as a victim application in this multi-tenant cloud environment. The sniffer application was deployed as the adversary tenant, multiple profiling runs were executed for each workload. These profiling runs were used to create a dataset consisting of the raw timeseries data labelled with the workload that was also executing.

The timeseries data was then analysed using time series analysis methods, multiple statistics were able to be generated to extract unique characteristics from the profiling runs. The extracted statistics generated a condensed, clean and labelled dataset that was able to be used to train and test machine learning models and evaluate the models' level of accuracy to draw conclusions on the research question.

5.3.1 Chosen Tenant Workloads

To begin this evaluation phase, real-world application workloads had to be sourced or developed. Because the focus of this investigation is on a containerized cloud environment, the workloads had to be run in Docker containers and then deployed to the Kubernetes cloud environment using the appropriate services. The aim was to source workloads that are diverse in the resources they utilize during execution e.g. a webserver will utilize the network compared to a data analytics application that may stress the memory bandwidth of the system.

The majority of the applications used in this evaluation phase were sourced from the Dacapo benchmarking suite[16], this suite contains are number of open-source, real world applications that execute with non-trivial workloads.

[16]http://dacapobench.org/

Each application is implemented in Java and runs on the Java VM. The details of each benchmark in the Dacapo suite are detailed in Blackburn et al. (2006), the applications used for the purpose of co-located tenants in this research were:

H2: This application executes a JDBCbench-like in-memory benchmark, it executes a number of transactions against a model of a banking application.

Batik: This application produces a number of Scalable Vector Graphics (SVG) images based on unit tests in Apache Batik (SVG library).

Lusearch: This application uses Lucene (search engine library) to perform a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.

Sunflow: This application renders a set of images using ray tracing (rendering technique).

Tradebeans: This application runs the daytrader benchmark via a Jave Beans to a GERONIMO (application server) backend with an in memory h2 (database engine similar to SQL) as the underlying database.

Xalan: This application transforms XML documents into HTML.

Each of the above Dacapo applications has a range of datasets varying in size that can be given to each as a parameter, for the purpose of this investigation each application was given the largest dataset to use. Each application was then containerized using Docker, to be deployed into the cloud environment as Kubernetes Pods.

Another application that was used in the investigation was a pre-existing NGINX container. NGINX[17] is a webserver that can be used as request load balancer, proxy and HTTP cache. This NGINX container was set up to return a helloworld style webpage to any requests made to the server. The NGINX web server was deployed to the cloud environment as a Kubernetes Service, exposing it to external requests. This allowed load to be placed on the webserver while it was running. The request load was generated by

[17]https://www.nginx.com/

setting up JMeter[18] (a load testing tool) on an external machine and sending constant requests to the NGINX server in the Kubernetes cloud environment.

Overall, 7 applications were used to create the dataset that will be used to train and test machine learning models and determine if the applications could be predicted based on the sniffer applications data. In addition to the 7 applications the sniffers were also deployed in isolation (only tenant executing) to observe if the collected data from the sniffer application could be used to identify if it was executing alone.

5.3.2 Creating the Dataset

To implement and create this dataset using the aforementioned applications, a python application was developed that used the Kubernetes API and Kubernetes-python client. Using this application, each of the example workloads could be programmatically deployed into the cloud environment followed by the sniffer application. Concluding the execution of the sniffers, the workload was deleted and replaced by another victim workload. The high-level flow of this deployment application is shown in Algorithm 7. This deployment application was executed, and the real-world victim applications were profiled by the sniffer application repetitively (because the deployment process was random, each workload was profiled a different number of times). Table 7 shows the frequency each application was profiled.

[18]https://jmeter.apache.org/

```
Load the current Kubernetes context using python client;
Configure access to the Kubernetes APIs;
Testing ← True;
while Testing do
    Randomly select the workload to deploy;
    Deploy and execute the workload on the cloud environment;
    for i ← 1 to 3 do
        Deploy and execute the sniffer application;
        Wait for the sniffers to finish;
        Get the data returned and store it locally;
    end
    Terminate and remove the workload from the cloud environment;
    if Experiments should stop then
        Testing ← False
    end
end
```

Algorithm 7: High-level Logic of Deployment Application

Workload	No. Times Profiled
H2	39
Batik	45
Lusearch	36
Sunflow	45
Tradebeans	48
Xalan	33
NGINX	57
Isolation	45
Total:	348

Table 7: Number of Sniffer Executions Per Application Workload

5.3.3 Timeseries Data Analysis

To perform the analysis on the dataset, first the data from each profiling run had to be gathered into a Pandas Dataframe object. Using this Dataframe object (Figure 2), timeseries analysis was used to extract key statistics from this raw data collected from the sniffers.

The Tsfresh library provided the capability to extract a range of statistics from the given Dataframe, for the purpose of this experiment the first set of statistics extracted were the sum of the values, median, mean, length, standard deviation, variance, maximum and minimum. These statistics were extracted for each individual sniffer thread. The resulting dataset contained 348 row labels and 71 attribute columns and will be referred to as the basic-features dataset.

A results mapping was also created, this associated the label of the row to the target co-located tenant that was to be predicted. The results of training and testing this dataset using machine learning models is detailed in the following section.

The next dataset that represented statistics extracted using timeseries analysis techniques was created using a function provided by the Tsfresh library. This function automatically extracted 61 features from the raw timeseries data, this dataset will be referred to as the comprehensive-features dataset. The list of the features extracted can be found in Appendix A and details of the features can be found at `https://tsfresh.readthedocs.iohttps://tsfresh.readthedocs.io/en/latest/text/list_of_features.html`. The results of training and testing this dataset using machine learning models are detailed in the following section.

The raw timeseries data gathered from every profiling run in the experiment, the basic-feature datatset and the comprehensive-feature dataset have all been made available to view using the following link: `https://drive.google.com/drive/folders/1gEVKWBrG5xW7s4d6KoWnJya0yvqG0eyY?usp=sharing`. The raw timseries data and comprehsive-dfeature dataset have been compressed due to the size of the files.

5.3.4 Prediction Methodology

Using the cleaned and condensed datasets created from taking the raw timeseries datasets and extracting key statistics using timeseries analysis techniques, the final aspect of this investigation was to apply the data to ma-

chine learning models and investigate the accuracy that different models can achieve when predicting a tenant application.

Supervised learning was the method used in this investigation because if this technique was to be successful in predicting tenants in a commercial cloud, a model would first have to be trained using labelled data. These labels would be an application that the sniffers were profiling. If the model was then used to predict unknown applications in a commercial cloud, the sniffers profiling data would be applied to the pre-trained model and a prediction would be made based off the data used in training.

A range of baseline classifiers were used in this investigation to observe prediction accuracy. The following models were used to test and train each of the prepared datasets:

Decision Tree: Decisions trees are a non-parametric supervised learning method, the goal of a decision tree is to create a model that predicts the value of a target variable by creating a tree like structure. The tree like structure is traversed along a path from its root, internal nodes within the tree contain a test that determines which sub tree is traversed. The path ends at a leaf node, which is the resulting prediction value (Dietterich (2000))

Support Vector Machine (SVM): An SVM is a discriminative classifier that constructs a set of hyperplanes to separate classes of data. An SVM uses a kernel to create the hyperplane boundaries, the kernel maps the original data into a higher dimensional space (Boser et al. (1992)). The type of kernels used in this investigation are linear and Gaussian.

K-Nearest Neighbours (KNN): KNN is an instance-based learning type, it does not construct a general model for the data. It stores instances of the training data and classification is computed from weighted vote of the nearest neighbours of each data point (Song et al. (2007)). KNN uses K instances to make a prediction for a given data point. The nearest neighbours are calculated using a distance metric, the Euclidean distance is used in this investigation.

RandomForest: The RandomForest classifier uses an ensemble of decision tree classifiers to make predictions. A specified number of decision trees are generated by using different subsets of the training data (generating

different trees). Each node in a decision tree is split by using a random subset of the data features. To make a prediction on data, each tree in the ensemble makes a prediction and a majority vote is the final prediction (Liaw et al. (2002)).

Adaboost: The Adaboost algorithm is also uses an ensemble of classifiers to make a final prediction. It fits a specified number of weak learners on repeatedly modified versions of the training data and uses a majority vote to produce a prediction. The data is modified by adding weights according to the classification result of the previous weak classifier in the ensemble, each incorrectly classified example receives more influence, so the subsequent classifier is forced to focus on the examples previously misclassified (Schapire (2013)).

Each classifier used in the investigation was used with its default parameters, and the accuracy of each model is evaluated using K-fold cross validation. Cross-validation involves splitting the data into K subsets and training the classifier on (k-1) of the subsets, using the left-out subset to test the classifier. Each subset is used as a test set only once, which means the classifier is trained and tested K times. For this investigation, 10-fold cross validation was used, resulting in the classifier being trained with 90% of the entire dataset.

The metrics chosen to evaluate each model were prediction accuracy percentage and the Cohen-Kappa co-efficient of the model. The prediction accuracy of the model corresponds to the ratio of correct predictions made, i.e. the predicted label matched the true label, and the total number of predictions. The Cohen-Kappa co-efficient is a statistic that is used to measure the inter-rater reliability for categorical items (McHugh (2012)), this statistic is used to further evaluate a model that is used in a multi-classification problem. The statistic returns a value between -1 and 1, this corresponds to the difference between the observed agreement and the expected agreement of the model. Essentially, Cohen-Kappa evaluates the performance of the model against that of a classifier that randomly assigns labels according to the frequency of each class. A Cohen-Kappa value closer to 1 conveys a good performing model.

5.3.5 Basic-Feature Results

First, the basic-features dataset was applied to the above machine learning classifiers. The first experiment trained the classifiers using all the attributes generated by the timeseries analysis. Each data example contained 71 attributes, the observed prediction accuracy and Cohen-Kappa results using 10-fold cross validation are shown in Table 8.

Classifier:	Decision Tree	Linear SVM	RBF SVM	KNN	Random Forest	Adaboost
Accuracy:	43%	25%	16%	37%	49%	32%
Cohen-Kappa:	0.33	0.13	0.0	0.28	0.53	0.21

Table 8: Basic-features Classification Accuracy

The results in Table 8 show that the data extracted using the basic timeseries statistics resulted in a range of moderate to poor classification accuracy for the classifiers trained with the data. The highest performing classifier was Random Forest, this classifier predicted the correct co-located tenant application approximately 50% of the time for each test iteration. The second highest accuracy was achieved by another decision tree-based model, this conveys that the data is well suited to decision tree learning. The two worst accuracy results were experienced by the SVM models, this illustrates that projecting the data onto a higher dimension was not effective in this case.

The Cohen-Kappa results equally reflected the accuracy results of the models. The highest values were achieved by Random Forest and the decision tree models. However, the Cohen-Kappa values of 0.13 and 0.0 for both SVM models indicate that these models are no better at predicting the correct label than a classifier that randomly assigns labels, this reinforces the poor accuracy values aforementioned.

To attempt to increase the classification accuracy, feature selection was used to re-train the classifiers using only the 10 most important attributes from the basic-features dataset. The 10 important attributes were determined by computing the Chi-Squared statistics for each attribute. The Chi-Squared test measures the dependence between the stochastic variables, the test result allows the removal of any attributes that are most likely to be independent of the label and as such irrelevant for classification (Greenwood and Nikulin (1996)).

The resulting 10 most important attributes calculated are shown in Table 9. The result of testing for attribute importance had returned that the most influential sniffers were both the memory and CPU sniffer. Interestingly, the disk sniffer also occurred within the top 10, after previously expecting that this sniffer would have minimal influence determining a co-located tenants disk operations.

Importance Rank	Attribute
1	memory-0 variance
2	memory-1 variance
3	memory-1 sum values
4	cpu-1 variance
5	cpu-0 variance
6	memory-0 sum values
7	disk-1 length
8	disk-0 length
9	memory-0 minimum
10	memory-1 minimum

Table 9: Attribute Significance in Basic-features Dataset

The accuracy of re-training and testing the classifiers using the attribute-reduced dataset are shown in Table 10.

Classifier:	Decision Tree	Linear SVM	RBF SVM	KNN	Random Forest	Adaboost
Accuracy:	33%	19%	16%	37%	43%	28%
Cohen-Kappa:	0.22	0.09	0.0	0.28	0.46	0.16

Table 10: Basic-features with Filtered Attributes Classification Accuracy

The results in Table 10 show that the prediction accuracy decreased for each classifier except in the case of the KNN classifier, which remained unchanged. This accuracy decrease was also reflected in the Cohen-Kappa statistics of the models. The observed decrease in both metrics convey that the models required the full set of extracted feature attributes in order to differentiate the tenant applications more accurately.

In summary of using the basic-feature dataset for tenant prediction, the accuracy level achieved by the models was considered fair, the decision tree-

based models proved to be the highest performing learning techniques. In contrast, the SVM models were shown to be near useless according to the Cohen-Kappa co-efficient. These experiments provided evidence that using basic timeseries statistics does not provide enough information to reliably differentiate between co-located tenants when using the sniffer application to measure performance interference. However, it does prove that a tenant workload can be correctly inferred, and this breaks the isolation between tenants in the cloud environment.

5.3.6 Comprehensive-Feature Results

Following the basic-feature dataset experiments, the comprehensive-feature dataset was applied to the classifiers, each classifier was trained, tested and evaluated using the same approach as the previous experiments. However, this dataset contained 5260 attributes corresponding to different timeseries statistics extracted. This dataset had its attributes filtered and only the most important 20 attributes were used to train the classifiers. The most important attributes were determined using the same statistical test applied to the basic-features dataset.

After filtering the attributes, the classifiers were trained using the new reduced dataset. The cross validation accuracy and Cohen-Kappa results are shown below in Table 11.

Classifier:	Decision Tree	Linear SVM	RBF SVM	KNN	Random Forest	Adaboost
Accuracy:	37%	22%	16%	37%	40%	31%
Cohen-Kappa:	0.29	0.09	0.0	0.27	0.42	0.20

Table 11: Basic-features with Filtered Attributes Classification Accuracy

The resulting accuracy results due to training the classifiers using the filtered comprehensive-feature dataset were worse in comparison to the results achieved when using the original basic-feature dataset. Each classifier scored both a lower accuracy and Cohen-Kappa co-efficient than its respective classifier trained using the full minimal-feature dataset. This result was unexpected and considered a negative outcome for this evaluation phase. It was expected that extracting a greater range and complexity of statistics would result in a dataset that would allow a greater depth of information to

be persisted through in the data and as a consequence, yield more accurate prediction results when applied to the classifiers. However, when comparing the comprehensive-feature results with the attribute-reduced basic-feature results, the comprehensive dataset showed a better performance across the majority of the classifier, including the decision tree learners.

5.3.7 Summary

In conclusion, each feature dataset and corresponding classification experiment achieved moderate prediction accuracy using the dataset obtained by profiling the example applications using the sniffers. The dataset used in the experiments consisted of eight target classes (applications) to predict, if a random classifer were to be used for prediction, the expected accuracy would be approximately 12.5%. The highest accuracy rate observed in the chosen trained models was 49%, using the Random Forest classifier and trained with the original minimal-feature dataset. While this result is not substantially accurate, it outperforms what would be expected by the random classifier. Equally this classifier achieved the highest Cohen-Kappa co-efficient of all classifiers, 0.53. According to Landis and Koch (1977) who proposed a scheme to characterize Cohen-Kappa values, 0.53 falls within the moderate range of agreement, drawing the conclusion that this model was the most effective of all the models for the given data and multi-class problem.

This moderate accuracy rate for co-located tenant prediction illustrates that performance interference data gathered using the proposed sniffer application may not be a robust medium in which to extract sensitive data within a containerized cloud environment. Therefore, the results from the investigation are unable to confirm that performance interference does allow the emergence of a reliable and stable side channel that leaks privileged information. However, the results do provide evidence that using the techniques in this investigation does allow for the isolation layer between tenants to be broken. The prediction accuracy achieved did indicate that the application backpressure measured by the sniffers did reveal information regarding a co-located tenant, allowing it to be predicted correctly. This is reinforced with the results observed in the baseline and interference maximisation evaluation phases that showed an influence on the sniffers execution when a co-located tenant was present.

5.4 Limitations

Within this investigation, limitations were present that could have contributed to the results achieved and impacted the final conclusions drawn.

5.4.1 Sniffers

A goal of this investigation was to take the perspective of a cloud tenant, where no knowledge of the underlying system would be required to facilitate the side channel leak. However, multiple of the sniffers require hardware information to ensure that the resources are effectively stressed when the sniffer executes. If this information was not available or able to be determine using another method, the effectiveness of the sniffer would be reduced.

5.4.2 Cloud Environment

The cloud environment used as the test environment for this investigation was provision using the Minikube tool. This meant that the Kubernetes cloud environment was hosted within a VM, this could have influenced the results because the proposed sniffers were designed to stress underlying physical hardware and the virtualization of the hardware caused by the hypervisor could have affected the results gathered by the sniffers.

Also, because the environment was within a VM, the host OS processes could have produced unintentional interference that could have affected the sniffers execution when profiling co-located tenants.

As shown in the baseline evaluation stage and interference maximisation evaluation stage, the disk bandwidth sniffer appeared unable to cause contention in the underlying storage system due to how Kubernetes manages storage. If this sniffer was modified to overcome this and cause the intended contention, the quality of sniffer data may have increased and resulted in a greater prediction accuracy for co-located tenants.

5.4.3 Dataset

The overall negative outcome for co-located prediction accuracy could have been accredited to only having 348 profiling runs of the sniffers. Potentially expanding the dataset may have resulted in an increase in prediction accuracy.

5.4.4 Prediction

Using the techniques proposed in this investigation, the machine learning models were trained using data collected from one cloud environment. To apply this technique to another cloud environment, a new set of training data would be required for the model to learn and be able to predict tenant applications. This presents an issue because in a real-world scenario, it would not be guaranteed that a host will be solely available for the purpose of generating the training data.

6 Conclusion

Overall, the main aim of this research was to investigate the feasibility of exploiting performance interference and cause a leaky side channel to emerge, allowing information to be inferred regarding a co-located tenants application within a containerized cloud environment.

The technique proposed in this investigation was to create a set of applications, referred to as sniffers, that each individually stressed a specific underlying hardware resource that would be shared between co-located tenant applications. In doing so, the application will cause contention between both the application itself and any application utilizing the resource. The idea was that this contention from another co-located application would cause a magnitude of backpressure, able to be observed and measured in the stressing application.

Each sniffer application was designed to measure the backpressure over a given period of time, the aim of this was to determine if gathering multiple datapoints resulted in a greater detail of information regarding the load on the system.

The technique used to extract key information from the sniffer's backpressure measurements was to apply timeseries analysis, this aimed to condense the data from multiple sequential datapoints into key features within the data.

To evaluate the effectiveness of the sniffers, testing was carried out to both investigate the effect of executing the sniffers in their development environment and within a containerized cloud environment using the container orchestration system, Kubernetes. The main observations from this evaluation were that containerizing the sniffer applications had no significant effect on all of the sniffers, with an exception to the sniffer that measured disk I/O bandwidth. The disk sniffer was significantly affected as executing it in the cloud environment showed the influence of how container storage is managed within a Kubernetes environment.

Following the initial evaluation of the sniffers, experiments were conducted to investigate the level of interference that could be measured in a controlled cloud environment, where artificial load was placed on each resource using known co-located tenants. The results of these experiments proved that the sniffers were affected by the interference caused by the co-located tenants and this was reflected in the data gathered by the sniffers. Again, the disk sniffer was the exception, where no significant influence was

observed due to the contention in disk I/O.

Finally, a cloud environment was provisioned, and a set of applications were deployed and profiled multiple times using the sniffers to create a labelled dataset that was used to test if the data profiled by the sniffers could be used to identify what application was executing during the profiling run.

The dataset was subject to timeseries analysis techniques and used to train and test a range of machine learning models to observe prediction accuracy.

The results yielded from the prediction experiments confirmed that co-located tenant applications could be inferred using the given performance interference data and training machine learning models on the extracted timeseries features. Using the multi-class test data, the highest accuracy rate achieved was 49% using a Random Forest model trained with basic timeseries statistics. This result was considered a successful outcome for the investigation, as it confirmed and validated the techniques used throughout each stage in the research. These techniques, consisting of developing individual sniffer applications to stress resources; collecting timeseries datapoints; applying timeseries feature extraction; utlizing machine learning models, were able to exploit performance interference as a method of extracting privileged information to predict a co-located tenant workload in a containerized cloud environment. This proved that performance interference can cause a side channel to emerge, breaking the isolation between cloud tenants.

However, the reliability of this approach was not confirmed. The final prediction results demonstrated that co-located application prediction is possible, but these results are unable to draw conclusions on the stability of the performance interference-based side channel.

6.1 Future Work

The research carried out within this investigation corresponded to concluding if it was possible to utilize performance interference as a medium to extract confidential information regarding a co-located tenant in a containerized cloud environment. Key to the research's findings were the methods used to implement the sniffers application, the strategy of extracting information from collected data and how the information was used to predict tenant operations. In each of these areas there is potential for future expansion, and investigations to be conducted using this research as a foundation.

6.1.1 Sniffers

The limitations section addressed the fact that the current implementation of the sniffers does require prior knowledge of the hardware resources that the sniffers are deployed to, this ensures the correct runtime memory allocations are made and sufficient datapoint generated. Further improvements to the sniffers would involve eliminating this prior system knowledge, this could be done by proposing different strategies to induce contention or implementing an empirical stage to the sniffers. An empirical phase could be similar to the work proposed by Celio and Caulin (2011), where the cache the sizes in a system could be programmatically determined and passed as parameters to the data cache sniffer. Removing the requirement for system knowledge would allow the sniffers to be deployed to any remote cloud system and allow for testing within a commercial public cloud.

In the evaluation section of the research, each sniffer was configured to execute its critical section for approximately 10 or 20 seconds, this corresponded to the length of time each sniffer profiled the environment and collected data. These values were arbitrarily given and produced many datapoints to be collected for the purpose of tenant identification. Further research into this area could reveal the significance of the execution duration of each sniffer on prediction accuracy. This would involve curating a dataset of profiling runs using a variation of critical section times and observing the impact on prediction accuracy.

6.1.2 Time Series Analysis

The timeseries features calculated and used to extract statistics from the raw sniffer data were provided by the Tsfresh python library. Further work in this area would be to investigate other techniques of extracting important statistics, and example would be converting the data from the time domain into the frequency domain and research the classification accuracy using frequency domain-based statistics.

6.1.3 Tenant Prediction

Within this investigation, a range of different models were used to observe the prediction accuracy that could be achieved using the sniffers transformed data. A further expansion in using machine learning would be to investigate the accuracy of training a neural network using the extracted data.

This research included developing a dataset using a set of known co-located workloads, many of these workloads were Java based. Further expansion on this would be to source a greater number and range of real-world workloads to determine if different language runtimes are more easily distinguishable using the sniffers.

This research was only scoped to predict a single tenant executing within the cloud environment. Further work would be to take into account multiple co-located tenants executing within the system simultaneously and determine if they can be distinguished.

Bibliography

AlJahdali, H., Albatli, A., Garraghan, P., Townend, P., Lau, L., and Xu, J. (2014). Multi-tenancy in cloud computing. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pages 344–351. IEEE.

Bazm, M.-M., Lacoste, M., Südholt, M., and Menaud, J.-M. (2017). Side-channels beyond the cloud edge: New isolation threats and solutions. In *2017 1st Cyber Security in Networking Conference (CSNet)*, pages 1–8. IEEE.

Bazm, M.-M., Lacoste, M., Südholt, M., and Menaud, J.-M. (2019). Isolation in cloud computing infrastructures: new security challenges. *Annals of Telecommunications*, 74(3-4):197–209.

Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA. ACM Press.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM.

Brillinger, D. R. (1981). *Time series: data analysis and theory*, volume 36. Siam.

Catteddu, D. (2009). Cloud computing: benefits, risks and recommendations for information security. In *Iberic Web Application Security Conference*, pages 17–17. Springer.

Celio, C. and Caulin, A. (2011). Characterizing memory hierarchies of multicore processors.

Delimitrou, C. and Kozyrakis, C. (2017). Bolt: I know what you did last summer... in the cloud. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 599–613. ACM.

Dietterich, T. G. (2000). Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer.

Dignan, L. (2019). Top cloud providers 2019.

Drepper, U. (2007). What every programmer should know about memory. *Red Hat, Inc*, 11:2007.

Godfrey, M. M. and Zulkernine, M. (2014). Preventing cache-based side-channel attacks in a cloud environment. *IEEE transactions on cloud computing*, 2(4):395–408.

Greenwood, P. E. and Nikulin, M. S. (1996). *A guide to chi-squared testing*, volume 280. John Wiley & Sons.

Harrington, P. (2012). *Machine learning in action*. Manning Publications Co.

Joshi, K., Raj, A., and Janakiram, D. (2017). Sherlock: Lightweight detection of performance interference in containerized cloud services. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 522–530. IEEE.

Joy, A. M. (2015). Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346. IEEE.

Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

Koh, Y., Knauerhase, R., Brett, P., Bowman, M., Wen, Z., and Pu, C. (2007). An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 200–209. IEEE.

Krebs, R., Spinner, S., Ahmed, N., and Kounev, S. (2014). Resource usage control in multi-tenant applications. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 122–131. IEEE.

Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Li, Y., Zhang, J., Jiang, C., Wan, J., and Ren, Z. (2019). Pine: Optimizing performance isolation in container environments. *IEEE Access*, 7:30410–30422.

Liaw, A., Wiener, M., et al. (2002). Classification and regression by randomforest. *R news*, 2(3):18–22.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*.

McCalpin, J. D. (1995). Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25.

McHugh, M. L. (2012). Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica*, 22(3):276–282.

Novaković, D., Vasić, N., Novaković, S., Kostić, D., and Bianchini, R. (2013). Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 219–230.

Patros, P., MacKay, S. A., Kent, K. B., and Dawson, M. (2016). Investigating resource interference and scaling on multitenant paas clouds. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, pages 166–177. IBM Corp.

Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. (2009). 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.

Rosenblum, M. and Garfinkel, T. (2005). Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47.

Sareen, P. (2013). Cloud computing: types, architecture, applications, concerns, virtualization and role of it governance in cloud. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(3).

Schapire, R. E. (2013). Explaining adaboost. In *Empirical inference*, pages 37–52. Springer.

Shue, D., Freedman, M. J., and Shaikh, A. (2012). Performance isolation and fairness for multi-tenant cloud storage. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 349–362.

Song, Y., Huang, J., Zhou, D., Zha, H., and Giles, C. L. (2007). Iknn: Informative k-nearest neighbor pattern classification. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 248–264. Springer.

Weins, K. (2017). Cloud computing trends: 2017 state of the cloud survey. `https://www.flexera.com/blog/cloud/2017/02/cloud-computing-trends-2017-state-of-the-cloud-survey/`. Accessed: 2019-09-28.

Weins, K. (2019). Cloud computing trends: 2019 state of the cloud survey. `https://www.flexera.com/blog/cloud/2019/02/cloud-computing-trends-2019-state-of-the-cloud-survey/`. Accessed: 2019-09-28.

Wu, Z., Xu, Z., and Wang, H. (2014). Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Transactions on Networking*, 23(2):603–615.

Xavier, M. G., De Oliveira, I. C., Rossi, F. D., Dos Passos, R. D., Matteussi, K. J., and De Rose, C. A. (2015). A performance isolation analysis of

disk-intensive workloads on container-based clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 253–260. IEEE.

Zissis, D. and Lekkas, D. (2012). Addressing cloud computing security issues. *Future Generation computer systems*, 28(3):583–592.

A Extracted Timeseries Statistics

```
variance_larger_than_standard_deviation
has_duplicate_max
has_duplicate_min
has_duplicate
sum_values
abs_energy
mean_abs_change
mean_change
mean_second_derivative_central
median
mean
length
standard_deviation
variance
skewness
kurtosis
absolute_sum_of_changes
longest_strike_below_mean
longest_strike_above_mean
count_above_mean
count_below_mean
last_location_of_maximum
first_location_of_maximum
last_location_of_minimum
first_location_of_minimum
ar_coefficient
change_quantiles
fft_coefficient
fft_aggregated
value_count
range_count
friedrich_coefficients
max_langevin_fixed_point
linear_trend
agg_linear_trend
augmented_dickey_fuller
number_crossing_m
energy_ratio_by_chunks
ratio_beyond_r_sigma
```

Figure 20: Timeseries Extracted Statistics