2021 International Symposium on Secure and Private Execution Environment Design (SEED) | 978-1-6654-2025-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/SEED51797.2021.00011

Cloak & Co-locate: Adversarial Railroading of Resource Sharing-based Attacks on the Cloud

Hosein Mohammadi Makrani*, Hossein Sayadi†, Najmeh Nazari*, Khaled N. Khasawneh‡, Avesta Sasan*, Setareh Rafatirad*, and Houman Homayoun*

*University of California Davis, †California State University Long Beach, ‡George Mason University

***Abstract*—The heterogeneity of resources and the diversity of applications on the cloud motivated the need for resource provisioning systems (RPSs) to meet the users' performance requirements while maximizing the resource utilization to achieve cost-efficiency. On the other hand, resource sharing-based attacks, such as side-channel, transient execution, rowhammer, and denial of service attacks, exploit shared resources to leak sensitive data or hurt the performance of a victim. Although mounting resource sharing-based attacks on the cloud is trivial once the attacker virtual machine (VM) is co-located with the victim VM, the co-location requirement with the victim limit the practicality of resource sharing-based attacks on the cloud. In this paper, we show that RPSs can be exploited to solve the co-location challenge of resource sharing-based attacks in the cloud. In particular, we propose a new attack, called *Cloak & Co-locate*, which utilize adversarial evasion attacks to force RPSs to co-locate attackers' VMs with targeted victims' VMs. Specifically, *Cloak* is a fake trace generator (FTG) that is wrapped around an adversary kernel in order to force RPSs to *Co-locate* it with a specific victim's VM, while also evading from detection and migration by the RPS.**

I. Introduction

Cloud computing is a significant paradigm shift in service for enterprise applications and has become a powerful architecture to perform large-scale computing. The advantages of cloud computing include virtualized environment, parallel processing, security, and scalable data storage. Recently, we are seeing a rapid growth of latency-critical and interactive applications, e.g., virtual reality, videoconferencing, autonomous vehicles, and online gaming. Such applications rely on near-instantaneous connections, which can't be met by the current cloud architectures that are located in isolated locations and rely on scaling out to improve performance. Therefore, to meet the increasing demands of latency-critical applications, many startups as well as cloud computing giants, such as Microsoft and Amazon, are pushing the cloud computing services closer to the edge [1], [2]. Specifically, instead of having few large data centers, they use a large number of mini-clouds, i.e., small data centers, spread all over the world to run cloud computing services closer to the application [3]. In doing so, mini-clouds help reduce the latency involved in sending computing requests to geographically far data centers by sending it to closer mini-clouds.

Regardless of the cloud architecture, in practice, different applications have different characteristics, thus, one configuration fits all does not provide the best performance for every application. As a result, clouds need to be super-heterogeneous, with diverse hardware systems, to meet the demands of different applications. Therefore, cloud service providers offer a wide range of configuration choices, such as VM instances with a variety of CPUs, memory, disk, and network configurations, and also customized VMs for analytics applications. Finding the best configuration for each application by itself is challenging, especially since applications behavior and resource requirements vary during different phases of execution. However, this is not the only challenge that cloud providers need to solve in order to generate revenue while satisfying the users' performance requirements. Specifically, scheduling applications to the best configuration needs to be cost-efficient; a non-optimal configuration results in more cost for the same performance target [4]. Therefore, picking the optimal configuration and maximizing the utilization of resources through resource sharing is the key to achieve cost-efficiency.

Aforementioned challenges can't be solved using a brute-force search since it is costive and exhaustive. Thus, motivated the need for resource provisioning systems (RPSs) in the cloud. RPSs allocate multiple applications from multiple users on the same physical hosts to increase resource utilization and cost-efficiency, in a way that applications have small impact on each other's performance. In addition, RPS facilitates various other services including security, fault tolerance, and monitoring. In order to improve the resource utilization of the cloud, researchers proposed several resource provisioning systems to achieve the performance goals while maximizing the utilization of available resources in the cloud[5], [6], [7], [8]. For example, Quasar [5] leverages machine learning and collaborative filtering to quickly determine which applications can be co-scheduled on the same machine without destructive interference. CherryPick [7] is a system that leverages Bayesian Optimization and Regression technique to build performance models for various applications to distinguish the close-to-the-best configuration. Ernest [8] uses common machine learning kernels and statistical techniques for selecting the optimal configuration on the cloud. PARIS [6] is another system that uses Random Forest for predicting performance from the application's micro-architectural behavior to find the best VM type configuration.

Although maximizing utilization, i.e., sharing of resources, is key to achieving cost-efficiency in the cloud, it also opens

978-1-6654-2025-9/21/$31.00 ©2021 IEEE
DOI 10.1109/SEED51797.2021.00011

the door for security and privacy vulnerabilities. In particular, these resources will be shared among different users, due to the multi-tenancy capability of hosts in the cloud, which facilitate a platform for performing a wide range of resource sharing-based attacks [9], [10], [11], including transient execution attacks [12], [13], [14], [15], rowhammer attacks [16], distributed side-channel attacks [17] and distributed denial of service attacks (DDoS) [18], data leakage exploitation [19], [20], [21], and attacks that pinpoint target VMs in a cloud system [22]. Mounting such attacks is trivial once the attacker is co-located with the victim. Therefore, the biggest challenge of attacks that exploit resource sharing in cloud environments is co-location [23].

In this work, we show how RPSs can become a blind spot and vulnerability that can be exploited to solve the co-location challenge of resource sharing-based attacks. In particular, we will show how adversarial attacks against machine learning models can be adopted to force RPSs to co-locate the attacker with the victim. A plethora of works on adversarial attacks exists, focusing specifically on computer vision applications [24], [25], [26], [27]. Specifically, these attacks work by adding specially crafted perturbations to the input data, i.e., an image, of machine learning models to manipulate their outcome. However, pixels of an image can be easily manipulated independently without changing the appearance of the image since images have high entropy. In contrast, adding adversarial perturbations to attack programs have different challenges since the attacker needs to ensure that the adversarial perturbations do not alter the malicious payload [28], [29].

In this paper, we propose a novel adversarial attack that we call *Cloak & Co-locate*. *Cloak & Co-locate* allows an attacker to put a *Cloak* on any malicious VM that forces the RPS to *Co-locate* the malicious VM with a specific targeted victim VM, without manipulating the malicious payload. In particular, our attack methodology is composed of two main steps, namely reverse-engineering and generating adversarial samples. Firstly, reverse-engineering is needed since we assume black-box access to the targeted RPS, i.e., we can only give input and observe the output of the RPSs. We exploit the black-box access to generate a proxy/substitute model of the targeted RPS. Assuming a black-box access allows generalization of our attack to become algorithm agnostic; target any RPS regardless of the underlying mechanism of the system. Secondly, we will utilize the proxy/substitute model of the targeted/victim RPS to generate an adversarial *Cloak* that allows an adversary to co-locate with a specific targeted victim. More importantly, the adversarial *Cloak* preserve not only the functionality of malicious payload, but also the syntax, leaving the malicious payload unchanged. This can be done by changing the overall VM trace rather than the malicious payload trace only. To achieve this, we implement the *Cloak* as a fake trace generator (FTG) that wraps around an adversary kernel in order to generate adversarial trace that matches the targeted victim trace to force co-location.

To the best of our knowledge, this is the first adversarial attack that target RPSs. We argue that this attacks not only show that current RPSs can be exploited to facilitate a wide range of attacks by solving the co-location challenge in the cloud, but also highlight a serious need for new techniques to be invented that guarantee security. Specifically, although there is a large number of defenses classes that were developed against computer vision-based adversarial attacks, these defenses are limited in defending against such attacks. In particular, these defenses assume that the attacker has a budget, i.e., the maximum amount of perturbation that can be added to an image without changing it's content. This is important in the computer vision domain, since the goal of adversaries is to perturb an image to fool a specific machine learning classifier, but can still be classified correctly by a human. However, for programs perturbations, there is no such budget/constraint, allowing the attacker to have an unlimited degree of freedom to add perturbations without risking increasing the possibility of being detected.

As a case study, we will demonstrate our attack on mini-clouds for two main reasons: (1) clouds providers are moving to mini-cloud architecture to meet the new users requirements, and (2) in academic settings we have access to small data centers compared to industry. However, the proposed attack methodology in this paper are general and can be applied regardless of the cloud architecture. Furthermore, we will target architectural trace-based RPSs since they are light weight, i.e., can be always 'on' without adding performance overhead, and demonstrated high performance in maximizing the utilization of resources in cloud environments [5].

Moreover, after co-locating the adversarial VMs with the targeted victim, they face two more challenges, namely detection and migration. Specifically, the RPS job does not stop after the instance initialization phase, i.e., initial deployment of a VM on a suitable host. It has been shown that periodic monitoring after the initial deployment can be unitized to improve both security and performance. For security, the trace information can be used to detect attacks based on computational anomaly [30]. For performance, the trace can be utilized to detect performance degradation due to resource contention or behavioral change of the running application and migrate the VM to a different host. However, we will show that our attack methodology can be extended to bypass such detection as well as avoiding VM migration.

Lastly, we argue that after co-location a wide range of attacks can be mounted due to the lack of strictly enforced resource isolation between co-scheduled instances, as shown in Figure 1. We show that even when we enforce state of the art recourse isolation technologies, such as Intel's Cache Allocation Technology (CAT), these attacks can still succeed.

II. ML-based Resource Provisioning System

Several machine learning based resource provisioning systems have been proposed for cloud systems in the literature, such as PARIS [6], ProMLB [31], Quasar [5], CherryPick [7], and Ernest [8]. RPSs attempt to meet the user performance requirements and provider cost-efficiency in terms of multiple

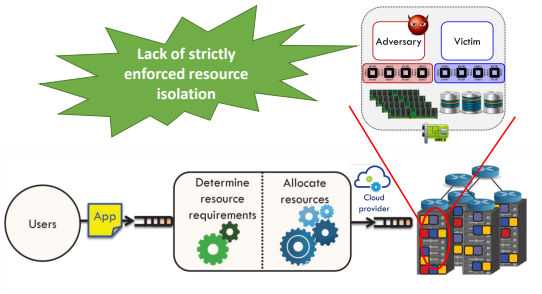

Fig. 1: Security threats in the cloud

aspects, such as load balancing among servers, minimum number of active hosts, and least response time, to avoid service-level agreement (SLA) violations in the cloud platform.

In general, a RPS, i.e., scheduler, have two main tasks [32]: 1) Instance initialization and 2) Periodic monitoring. During the instance initialization phase, when an instance is created and submitted to scheduler, the scheduler profiles the application and based on the application's behavior determines the resources required for meeting its SLA. After that, scheduler allocates the instance to a host in the infrastructure.

During the periodic monitoring phase, scheduler monitors application's behavior to guarantee the SLA all the time. In a case that application's behavior changes, the scheduler attempts to reschedule and migrate the instance to a new host to provide required resources to meet the SLA agreement.

Figure 2 shows how a general machine learning (ML) based RPS works. First, the system monitors the application and extracts its micro-architectural and system level information/behaviour. Then based on the current behavior and server configuration, it predicts the performance of application to make sure that performance of application will not be degraded. If the RPS identify a performance degradation, then by leveraging optimization techniques, it determines another suitable configuration and host for the application.

In this work, we will show how the first phase of a RPS can be exploited to facilitate resource sharing-based attacks. In addition, we will show how to evade the the detection and migration mechanisms of the second phase of a RPS.

III. Background and Threat Model

The performance unpredictability problems that stem from platform heterogeneity, resource interference, software bugs and load variation [33], are well-studied in the public cloud. However, there are other challenges that lead to security threats. Below we discuss related vulnerabilities with respect to the resource provisioning systems in the cloud. In particular, we show that RPSs hides security vulnerabilities, since they enable an adversary to extract information about an application's type and the infrastructure's characteristics.

A. Background

As different clients share hardware resources in the public cloud, isolation becomes a core security challenge for the cloud computing paradigm, which consequently enables adversaries to exploit it for their malicious desires.

A distributed attack [34] aims to retrieve sensitive information, or degrade the performance of a number of computing resources on a distributed system, where each computing resource performs processing of a part of the overall system activity. The retrieved information may, for instance, be a set of encryption keys that can be exploited to compromise the functionality of the whole distributed system. A distributed attack may also be used to retrieve information about the cloud infrastructure. In the following, we describe some characteristics and provide more details on such attacks.

Definition 1 A distributed attack over a set M_{vic} of virtualized instances running in a distributed system S, is defined as the tuple $DSCA = (S, M_{vic}, D, M_{mal}, A, CP, EP)$ where: S is a distributed system; M_{vic} are the VMs that are targeted by the attack; D is the distributed dataset to be compromised (partially or totally); M_{mal} are malicious VMs, co-located with the victim VMs; A is a set of local attack techniques (such as side channel, denial of service, or resource freeing attack); CP is a protocol to coordinate the attacker VMs in M_{mal}; EP is a protocol to exfiltrate data.

We consider $D = \{d_1, \ldots, d_n\}$ a dataset to be processed by the distributed system $S = \{s_1, \ldots, s_n\}$ implemented on a set of VMs $M_{vic} = \{m_{vic1}, \ldots, m_{vicn}\}$ on a virtualized platform. Each component s_i of S processes data d_i locally and runs on its own VM, m_{vici}. To perform the distributed attack, the adversary sets up a number of malicious VMs at least equal to the number of M_{vic}) $M_{mal} = m_{mal1}, \ldots, m_{maln}$, co-located with the victim instance M_{vic}. The adversary also masters a set $A = a_1, \ldots, a_m$ of local attack techniques, i.e., Flush+Reload.

The objective of a distributed attack is to first attack each component of the system s_i running on m_{vici} through m_{mali} running local attack technique a_j to retrieve d_i. The synchronization between attack instances and a central server may be performed using a coordination protocol CP. A protocol EP may be used to control attacking instances remotely, and to send collected information to a remote server to exfiltrate sensitive data. In the following, we briefly explain three well-known local attacks on a distributed system:

1) Side Channel Attack (SC): Lack of enforced isolation can create side-channels, due to the sharing of physical resources like processor caches, or by mechanisms implemented in the virtualization layer. A side-channel is a hidden information channel that differs from traditional "main" channels (e.g., network) in that security violations may not be prevented by placing protection mechanisms directly around data. A side-channel attack exploits a side-channel to obtain important information. SC attacks may be classified according to the type of exploited channel. Timing attacks and cache-based attacks are two main classes of SC attacks, where the processor cache memory is often exploited by adversaries to obtain information. There are attacks that attempt to extract confidential information from co-scheduled applications, such as private keys [19]. Zhang et al. [35] proposed a system that launches side-channel attacks in a virtualized environment. Wu

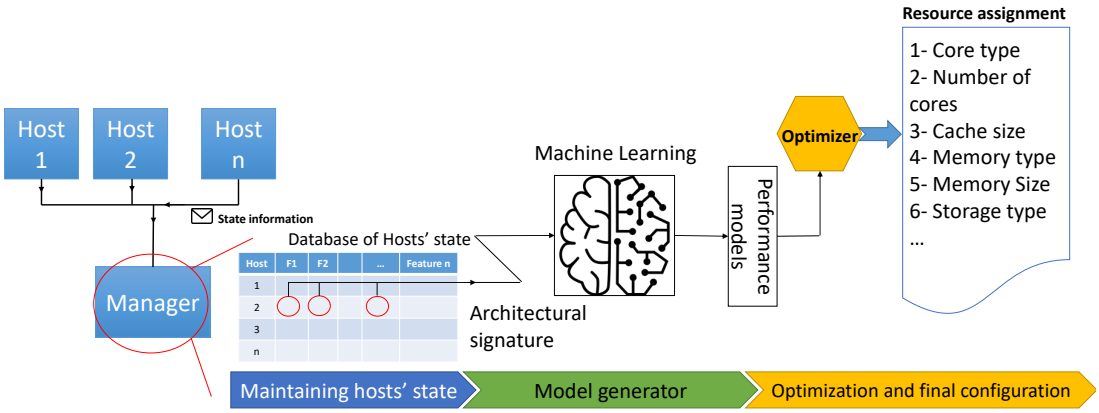

Fig. 2: ML based resource provisioning system

et al. [36] used the memory bus of an x86 processor to launch a covert channel attack and degrade the victim's performance. This type of attack requires the adversarial VM to be co-located with the victim VM.

2) Denial of Service Attack (DoS): Denial of service attacks hurt the performance of a victim service by overloading its resources. In cloud settings specifically, they can be categorized into two types; external and internal (or host-based) attacks. Internal DoS attacks take advantage of IaaS cloud multi-tenancy to launch adversarial programs on the same host as the victim and degrade its performance. For example, Ristenpart et al. [37] showed how an adversarial user could leverage the IP naming conventions of IaaS clouds to locate a victim VM and degrade its performance. Cloud providers are starting to build defenses against such attacks by increasing the instances of a service under heavy resource usage. This means that DoS attacks that overload a physical host are ineffective. However, Bolt [30] showed they can perform DoS attacks in a way to make them resilient against such defenses by avoiding resource saturation.

3) Resource Freeing Attack (RFA): Resource-freeing attacks also hurt a victim's performance, while additionally forcing it to yield its resources to the adversary [38]. While RFAs are effective, they require significant compute and network resources, and are prone to defenses, such as live VM migration. Delimitrou and Kozyrakis [30] showed that it is possible to launch host-based attacks on the same machine as the victim that take advantage of the victim's resource sensitivity, and keep resource utilization moderate, thus, evading defense mechanisms.

B. Threat Model

In this work, we target Infrastructure as a service (IaaS) providers that operate public clouds for mutually untrusting users. Multiple VMs can be co-located on the same server. Each VM has no control over where it is placed, and no priori information on other VMs on the same physical host. For now, we assume that the resource provisioning system is neutral with respect to detection by adversarial VMs, i.e., it does not assist such attacks or employ additional resource isolation techniques to hinder attacks by adversarial users. Moreover, RPS is considered ideal and treats all the VMs in a fair manner and places them according to workload characteristics rather than its place of origin or intention. Moreover, we assume black-box access to the RPSs, were we do not know the underlying model that the RPS is using for placing VM instances. Thus, we can only create VM instances and observe the placing outcome.

In this paper, we assume two types of VM as follow:

- **Adversarial VM:** An adversarial VM has the goal of getting co-located with victims and evade from detection mechanism embedded into resource provisioning system to negatively impact victims' performance or steal information.
- **Friendly VM:** This is a benign VM scheduled on a physical host that runs one or more applications. They do not employ any schemes, such as memory pattern obfuscation, to prevent detection by an adversary.

In this work, we consider the worst case for an attacker in which if the adversarial application does not change its behavior and architectural signature, the RPS will detect it. The detection mechanism can be any arbitrary technique. Therefore, adversarial applications are required to change its micro-architectural and system level profiling trace to prevent the detection. It should be noted that changing the micro-architectural and system level profiling trace does not mean hacking the performance counters or accessing the RPS's database, but it means that the adversary application changes its behavior, e.g., using CPU more, or performing dummy memory access to increase the cache misses or memory bandwidth usage. The direct result of such behavior is a change in the state of the system, without hacking the system. Additionally, the adversary must remain co-located with the victim. In the case of a change in the behavior, there is a high chance that RPS migrates the adversary VM to another host. For example, Figure 3 demonstrates how an applications' behavior changes during the runtime [39]. This is the another challenge of attack and we are going to address it in this work.

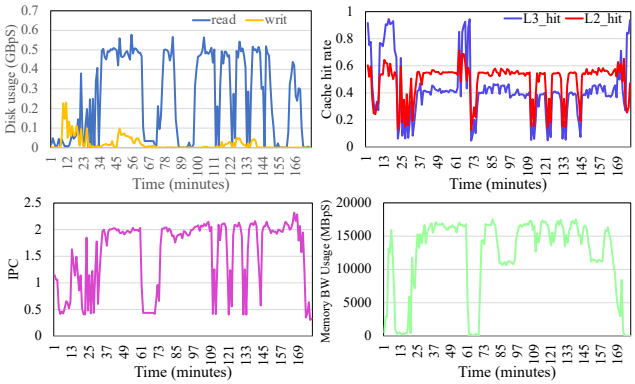

Fig. 3: Example of behavior and phase change for Hadoop: PageRank application

IV. Cloak & Co-locate

A. Contention-based attack's setting

Adversaries are rarely interested in a random service running on a public cloud. They need to pinpoint where the target resides in a practical manner to be able to perform DoS, RFA, or SC attacks. This requires a launch strategy for co-location and a mechanism for co-residency detection. The attack is practical if the target is located with high accuracy, in reasonable time and with modest resource costs. Performing any distributed attack requires a number of prerequisites steps detailed in the following:

1) Finding physical hosts running victim instances: To perform any attacks based on co-location, the attacker needs several VM launching strategies to achieve co-residency with the victim instance, which is impractical and not feasible. A pre-condition for the attack is that the malicious VMs reside on the same physical host as victim VMs, as side-channel and RFA attacks are performed locally. The first and main step is thus to find the location of physical hosts running victim VMs. Several placement variables such as datacenter region, time interval, and instance type are important to achieve co-residency. These variables may be different among IaaS clouds. However, the application type is considered as an effective factor in the placement strategy [40]. Let $P(m_{mali})$ be the probability of instance m_{mali} to be co-resident with instance victim m_{vici}. The value of P will be raised by increasing the number of launched attack instances. To make sure that both attacker and victim VMs achieve coresident placement, the adversary can perform co-residency detection techniques such as network probing [41]. The attacker can also use data mining techniques to detect the type and characteristics of a running application in the victim VM by analyzing interferences introduced in the different resources to increase the accuracy of co-residency detection [30].

2) Evasion from detection and migration : There are various techniques to detect an attack in virtualized environment. For example, as side-channel attacks are very fine-grained attacks, the detection of such attacks requires high-resolution information, mainly provided by Hardware Performance Counters (HPCs) [42]. The HPCs are a set of special-purpose registers built into modern microprocessors to capture the trace of hardware-related events such as last-level cache (LLC) load misses, branch instructions, branch misses, and executed instructions while executing an application. Those events are basically used to profile a program behavior for optimization purposes and are available for any application in the user space. These events are also used in the detection of abnormal behaviors in computer systems. We distinguish two different methods of detection: (1) signature based [43] and (2) threshold-based [44]. Signature-based approaches create the signature of the attack based on received information from HPCs, and compare the behavior of the system with the generated signature to identify any eventual malicious activity. For example, Payer et al. [43] proposed an approach to detect cache-based side-channel attacks between two processes. Their approach is based on HPCs and kernel events, e.g., number of page faults, to generate signatures for cache-based side-channel attacks. To detect a malicious process, they compare the generated signatures with the process signature.

It has been shown that resource usage correlates to the probability that an application is of a specific type. For an example, it becomes clear that cache activity is a very strong indicator of this attack type, with applications with a very high level-1 instruction cache (L1-i) and high LLC pressure corresponding to cache based side channel attack with a high probability. Disk traffic also conveys a lot of information, with zero disk usage signaling a side channel attack with very high likelihood. Correlating similarity concepts to resources shows that certain resources are more prone to leaking information about a workload, and their isolation should be prioritized.

On the other hand, threshold-based approaches utilize the HPCs trace to flag anomaly resource utilization that goes beyond a pre-specified threshold. For example, Chiappetta et al. [44] proposed to collect certain statistics through HPCs for running processes, and then use machine learning techniques (neural networks [45] and unsupervised learning) to detect a side-channel attack between two processes.

B. Proposed Approach

In any resource sharing-based attack, the instance initialization phase of RPS plays a preventive role in such attacks by avoiding a malicious instance to be co-localized with a targeted victim instance on the same physical machine. This phase of VM placement algorithm is undocumented for security reasons by cloud service providers. However, we show how an adversary can bypass the instance initialization phase and gets co-located by victims with high probability. Moreover, periodic monitoring and rescheduling is leveraged to mitigate co-residency attacks, when the attack is detected. Therefore, we will also show how it is possible to disguise the malicious behavior of adversary's VM and remain on the same host with the victim and avoid the migration.

In this work, we propose to use a *Cloak*, which is technically a fake trace generator (FTG), to wrap it around an adversary application in order to first get *co-located* with a targeted

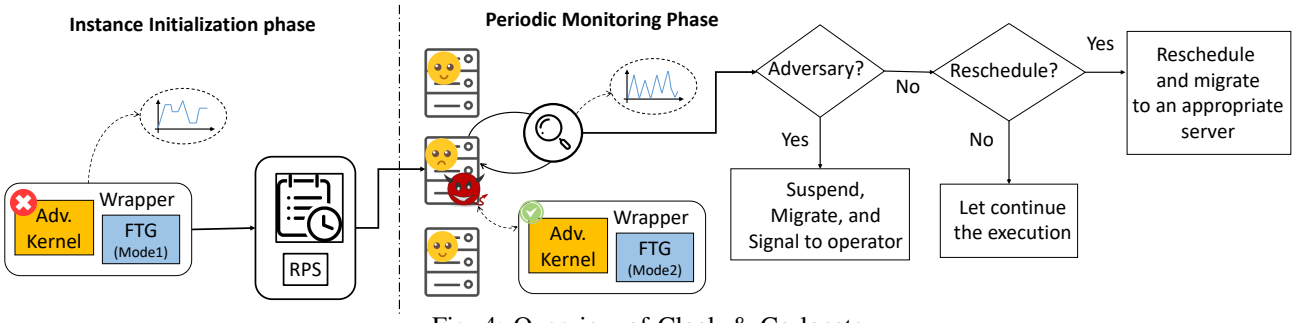

Fig. 4: Overview of Cloak & Co-locate

victim and subsequently evade detection and migration. At instance initialization phase, the *Cloak* goal is to perform a trace mimicry task that will mimic the behavior of the targeted victim application to increase the chance of co-location. If the desired server has not been assigned to the adversary VM (co-residency with the victim can be detected by network probing) then the adversary VM terminates, and a new VM must be reinstantiated as the cost of forcing RPS to migrate the adversary VM to another host that may have the victim is high. After co-residency, the *Cloak* will change its mode to constantly generate a new specialized trace that changes the behavior of the adversary application to not only evade detection but also migration.

To create the *Cloak*, i.e., FTG, we use the concept of adversarial sample in machine learning where we can add specially crafted perturbations to an input signal to fool the machine learning models. In particular, our goal is to change the model's output decision to a specific output, i.e., output that is similar to the targeted victim output. For this purpose, our attack is performed in two phases. First, we need to reverse engineer the resource provisioning system using a machine learning model to have access into to how the RPS make decisions. Second, we utilize the reversed engineering results to craft specialized *Cloak*; a FTG that will add perturbations to the adversary's application trace to force the RPS to co-locate it with a targeted victim. Figure 4 shows the overview of our proposed method. The details of this approach will be discussed in the following sections.

V. Reverse Engineering of RPS

To perform any resource sharing-based attack, such as side-channel attacks, an attacker must co-locate an adversary instance, e.g., an attacker VM with the victim VM. To find hosts that victim's VM are running on, we propose to reverse engineer the resource provisioning system in the IaaS cloud. As mentioned, RPS can be considered as a blackbox (worst-case scenario). Figure 5 shows an overview of our reverse engineering scheme. Thus, as a first step, we propose the following methodology to perform the reverse engineering. Note that we assume the RPS used in the cloud is ML-based.

The goal of reverse engineering of RPS is to create an ML model that can mimic the functionality of the original RPS. For this purpose, we train an arbitrary ML model, i.e., a proxy model, that can provision a server with the same configuration that the original RPS provisions for an incoming application to the cloud. We implement PARIS [6], a RPS proposed at UC Berkeley, to act as our original RPS, i.e., victim.

PARIS uses a machine learning technique (Random Forest) for predicting the performance from the application's fingerprint (micro-architectural signature) to find the best VM type configuration. To generate the fingerprint of an application, PARIS extracts 20 resource utilization counters spanning the following broad categories and calls it the fingerprint: CPU utilization, Network utilization, Disk utilization, Memory utilization, and System-level features. On the other hand, CPU count, core count, core frequency, cache size, RAM per core, memory bandwidth, storage space, disk speed, and network bandwidth of the server are the representation of the configuration provisioned by PARIS.

We denote the micro-architectural fingerprint and system level information of an application as $Fing$ vector. In Equation (1), a_i denotes each architectural feature.

$$Fing = \{a_1, a_2, ..., a_{20}\} \tag{1}$$

configuration parameters of the server's platform referred to configuration inputs is as follow:

$$Conf = \{c_1, c_2, ..., c_9\} \tag{2}$$

where $Conf$ is the configuration vector and c_i is the value of the ith configuration parameter (number of sockets, number of cores, core frequency, cache size, memory capacity, memory frequency, number of memory channels, storage capacity, storage speed, network bandwidth).

The RPS is responsible to provision $Conf$ based on $Fing$:

$$Conf = f(Fing) \tag{3}$$

Note that $f(Fing)$ is just a data model, which means there is no direct analytical equation to formulate it.

A. Data collection and model training

To use an arbitrary machine learning model to act as a resource provisioning system, we need a training dataset to train a proxy model. The dataset has two parts: the first part is the application's fingerprint and the second part is the corresponding configuration provisioned by RPS. Then we can

TABLE I: Detailed information of local cluster

Server (Xeon)	Freq. (GHz)	Socket	Core	Cache (MB)	Mem. (GB)	Storage	Server type	Count
E5-4669 V4	2.2	4	22	55	96	SSD PCIe	HPC	2
E5-4667 V4	2.2	4	18	45	64	SSD SATA	HPC	2
E5-4650 V4	2.2	4	14	35	32	SSD SATA	HPC	2
E5-2690 V4	2.6	2	14	35	512	SSD / HDD	Memory opt.	4
E5-2650 V4	2.2	2	12	30	256	SSD / HDD	Memory opt.	4
E5-2667 V4	3.2	2	8	25	32	SSD PCIe	I/O opt.	4
E5-2643 V4	3.4	1	6	20	32	SSD PCIe	I/O opt.	4
E5-2660 V2	2.2	2	10	25	16	HDD	General purp.	6
E5-2650 V2	2.6	2	8	20	16	HDD	General purp.	6
E5-1630 V4	3.7	1	4	10	8	HDD	Power opt.	2
E5-1680 V4	3.4	1	8	20	12	HDD	Power opt.	2
E3-1270 V6	3.8	1	4	8	8	HDD	Power opt.	2

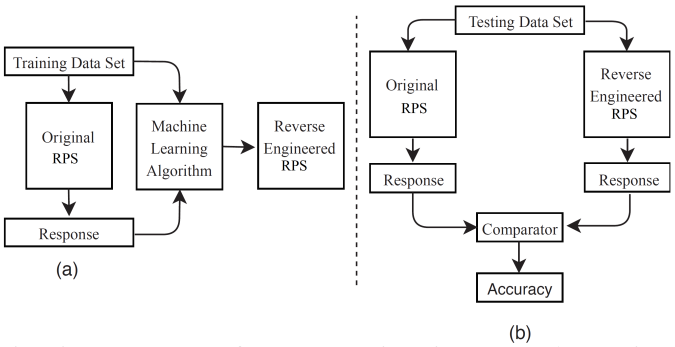

Fig. 5: (a) Process of reverse engineering RPS; (b) Testing Performance of Reverse-Engineered RPS

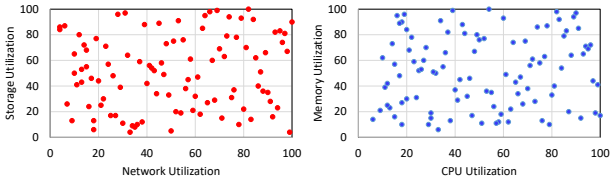

Fig. 6: Coverage of application's characteristic

use the dataset for training our proxy model to map those fingerprints to final configurations.

We perform the data collection in a controlled environment, where all applications are known. We use a medium size 40-machine cluster (presented in Table I), and schedule a total of 120 workloads, including batch analytics in Hadoop and Spark and latency-critical services, such as webservers, Memcached and Cassandra. For each application type, there are several different workloads with respect to algorithms, framework versions, datasets, and input load patterns. The training set is selected to provide sufficient coverage of the space of resource characteristics. Figure 6 shows the coverage of resource characteristics for applications in the training set. The selected workloads cover the majority of the resource usage space. This enables us to match any new application that has not been previously seen.

We submit all of these applications to the targeted RPS. In the beginning, the RPS profiles the application and extracts the fingerprint. Then, the RPS uses the Random Forest model to determine an appropriate server configuration. We collect all the fingerprints and their correspondent configurations generated by the RPS to shape our dataset.

We then use an Artificial Neural Network (ANN) to map the $Fing$ to $Conf$. We exploit one ANN per each c_i in the $Conf$. Therefore, we totally train 9 ANNs that each of them has 20 inputs and one output. ANNs have 5 fully connected layers. We found out this model achieves our desired accuracy. Therefore, we did not use a more complex model [46], such as a deep neural network. Each layer of ANN has 230 hidden neurons. The number of neurons for the hidden layer is decided through Grid Search [47] to reach the highest possible accuracy.

B. Performance of reverse engineered RPS

The testing set consists of 108 diverse applications that include web servers, various analytics algorithms and datasets, and several key-value stores and databases. Note that there is less than 30% overlap between training and testing sets in terms of algorithms, datasets, and input loads. The Original RPS is fed with all applications and the responses are recorded. These responses are utilized in order to compare the functionality of ANNs versus the original RPS. We observed that ANNs perform well with an overall accuracy of 92.7% to mimic the original RPS, the precision of 0.90, recall 0.93 and F1-score of 0.91. Now, our proxy model is ready to be utilized for generating adversarial samples.

VI. Cloak Generator

Our proposed *Cloak*, i.e., FTG, works in two modes. Mode (1) is for the instant initialization phase (before co-location) and mode (2) is for the periodic monitoring phase. When an adversary application is submitted to the RPS, the adversary kernel is deactivated. However, FTG must start working and change the behavior during instant initialization phase to generate a trace similar to the victim application in order to fool the RPS and get co-located with the victim on the same host.

After co-location, when co-residency has been detected by using any techniques mentioned in Section IV-A1, the adversary kernel can starts its attack. While starting the

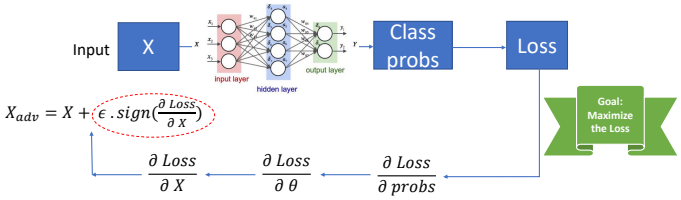

Fig. 7: Fast Gradient Sign Method

Fig. 8: Activation of FTG and increase in the similarity of fake trace and victim's fingerprint during instant initialization phase

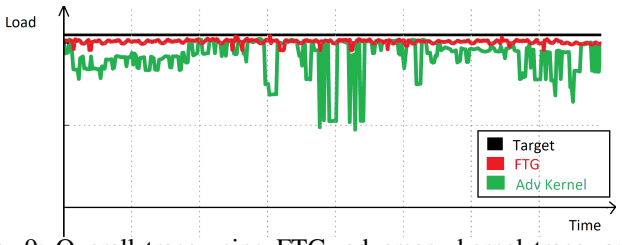

Fig. 9: Overall trace using FTG, adversary kernel trace, and the target

attack, the *Cloak*, i.e., FTG, switches to mode (2), where it is required to carefully generate a fake trace that disguises the behavior of the adversary kernel. Moreover, this trace must fool the RPS to provision the same configuration as it provisioned during the instant initialization phase. For this purpose, FTG must constantly monitor the system's state. In order to monitor the system state and extract the Hardware performance counters (HPC) information, FTG uses $Perf$ tool available under Linux. Perf provides rich generalized abstractions over hardware specific capabilities. It exploits $perf - event - open$ function call in the background which can measure multiple events simultaneously. It is non-trivial to determine the level of perturbations that need to be injected into the application's micro-architectural patterns in order to get the desired host configuration. Crafting a trace that can fool the RPS is only possible by performing a targeted adversarial attack on RPS. We discuss this targeted adversarial attack in the next subsection of the paper.

When the trace generated by FTG has been calculated (either in mode (1) or mode (2)), FTG runs a few micro benchmarks of tunable intensity from iBench [48] that each put pressure on a specific shared resource. iBench consists of a set of carefully-crafted benchmarks that generate contention on core, the cache and memory hierarchy, and the storage and networking subsystems.

A. Targeted adversarial attack to RPS

To perturb the micro-architectural patterns, we employ a low-complex gradient loss based approach, similar to Fast-Gradient Sign Method (FGSM), which is widely employed in image processing. The advantage of this approach is its low complexity and low computational overheads. In order to craft the adversarial perturbations, we consider the proxy model generated from the reverse engineering of the targeted RPS. In our experiments, the proxy model is a neural network with θ as the hyperparameters, x being the input to the model (current fingerprint of adversary kernel), and y is the output (current configuration of server) for a given input x, and $L(\theta, x, y)$ be the cost function used to train the neural network. Then the perturbation required to change the output to the target configuration is determined based on the cost function gradient of the neural network (in this case). Eventually, this perturbation is the trace that must be generated by FTG. The adversarial perturbation is calculated based on the gradient loss, as shown in Figure 7, similar to the FGSM [25] and is given by:

$$x^{adv} = x + \epsilon \ sign(\nabla_x \ L(\theta, x, y)$$

where ϵ is a scaling constant ranging between 0.0 to 1.0 and is set to be very small such that the variation in $x(\delta x)$ is undetectable. In case of FGSM, the input x is perturbed along each dimension in the direction of gradient by a perturbation magnitude of ϵ. Considering a small ϵ leads to well-disguised adversarial samples that successfully fool the ML model. In contrast to the images where the number of features is large, the number of features, i.e., micro-architectural metrics are limited; thus, the perturbations need to be crafted carefully and, more importantly, can be generated during runtime by the applications. For instance, a negative value cannot be generated by an application. Hence, we provided a lower bound on the adversary values.

Figure 8 shows a side channel attack and its fingerprint's similarity with a victim application (Spark: recommender system) during instant initialization phase. After the activation of FTG, the similarity of fingerprint increases to 78% and therefore the adversary VM can be identified as friendly VM. Figure 9 illustrates how FTG can generate the desired trace proposed by adversarial sample generator during periodic monitoring phase (when co-residency has been detected). The green line shows the trace of adversarial kernel. The target trace is the black line, which shows the overall trace must be the same to be able to evade detection and migration. The red line shows the overall trace (with the impact of FTG trace), which is the trace after adding perturbation. We can observe that the fake trace generator is successful in disguising the adversarial kernel.

VII. Evaluation

To evaluate our proposed approach, we implemented 8 distributed attacks as follow: SC1: Prime + Probe, SC2: Flush + Reload, SC3: Flush + Flush, SC4: Evict + Time, DoS1: increasing latency by saturating the network, DoS2: decreasing throughput by saturating storage, RFA1: freeing

TABLE II: Success Rate (SR) of distributed attacks based on the application's type.
(*: SR $\leq 25\%, ** : 25\% < SR \leq 75\%, *** : SR \geq 75\%$)

	SPEC	Hadoop	Spark	Memcached	Cassandra
SC1	***	*	**	*	**
SC2	***	*	*	*	**
SC3	***	*	**	**	*
SC4	***	**	**	*	**
DoS1	*	*	***	***	**
DoS2	*	***	*	*	***
RFA1	*	**	***	***	**
RFA2	***	**	***	***	*

memory resource, and RFA2: freeing CPU resource. We perform these attacks on 20 unseen victim applications from different domains (SPEC, Hadoop, Spark, Memcache, and Cassandra). Based on our evaluation, the success rate of being co-located with victims, evading the detection and migration, and getting the desired outcome from attack depends on many factors such as victim's type, the period of monitoring phase, and amount of perturbation.

A. Experimental results

Table II shows the impact of victims' type on the success rate of each type of attack. The interesting observation is that there is a meaningful relationship between the application's type and the nature of the attack by itself. For instance, we observe that side-channel attacks are more successful when the cache hit rate of the victim is low. Similarly, we observed that RFA is more successful when the resource utilization of the victim is high. One reason is that in such case, FTG can generate a better fake trace to convince the RPS to stay at the current host. In a case that the difference between the behavior of the adversary kernel and the victim is high, the FTG has to generate more perturbation and this may lead to a migration decision by RPS.

In order to provide an insight into the impact of the monitoring period on the evasion chance of attack and the effectiveness of FTG, we performed each attack under different monitoring periods. Figure 10 shows the results of this experiment. One can observe that by increasing the period, the chance of evasion increases. The reason is that the fingerprint of application changes more frequently when the period is small and FTG is not able to generate the desired trace. However, when the period increases, the perturbation that FTG generates becomes more effective.

It is important to know how much perturbation is required to evade the detection but still have a good chance for remaining on the same host and evade migration. Figure 11 shows the average migration chance of each attack and the similarity of our adversary VM's behavior with the victim's behavior. It was expected that by increasing the similarity between the adversary and victim, the migration chance reduces. However, we observed that generating fake trace to exactly follow the target trace proposed by adversarial sample is hard. Therefore, reaching to 0% chance of migration is out of reach. Moreover, Figure 12 demonstrates the impact of the perturbation amount on the migration chance. The results show that the amount of

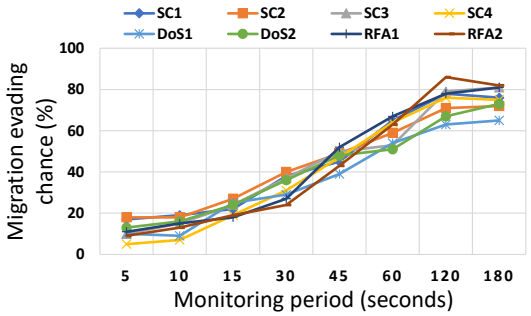

Fig. 10: Impact of monitoring period on the chance of evasion from migration

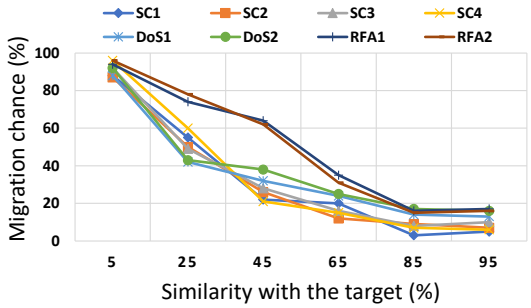

Fig. 11: Impact of similarity with the target trace on migration chance

perturbation should be carefully determined as a high or low amount of perturbation causes migration.

We evaluated the impact of perturbation on the effectiveness of the attack by itself and obtaining the desired outcome. As Figure 13 shows, side-channel attacks are very sensitive to a perturbation such that by increasing the perturbation, the attacks become useless. This is due to the fact that side channel attacks are customized and fine-grained and any noise on the system, especially on the hardware performance counters, can prevent the attack from being successful.

Last but not least, we evaluated the number of VM's per host and its impact on the success rate of Cloak & Co-locate attack. We run randomly multiple friendly applications from our application pool on each host and perform an attack on one of them. The success rate of attacks mostly depends on the server's type as the amount of resources available on the host impacts the applications' fingerprints, and configuration assignment. Figure 14 demonstrates the average success rate of attacks corresponding to the number of friendly VMs on the same host. The trend shows that increasing the number of applications on the host significantly reduces the attack's success rate from 60% to 16% on average.

The breakdown of attacks' success or failure rate on a general purpose server is presented in Figure 15. The red bar shows the percentage of failing to co-locate with victim at the instantiate phase. The gray bar shows that the adversary co-located with the victim at the instantiate phase but RPS migrated the adversary VM to another host during the monitoring phase. The blue bar shows the percentage that the adversary co-located with the victim and remained co-locate

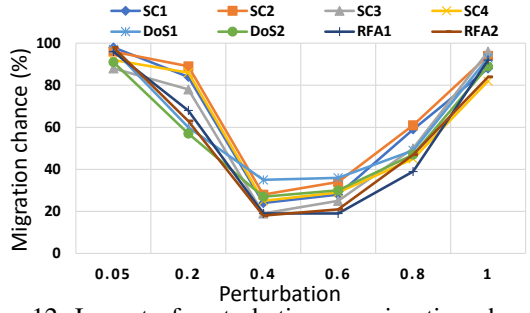

Fig. 12: Impact of perturbation on migration chance

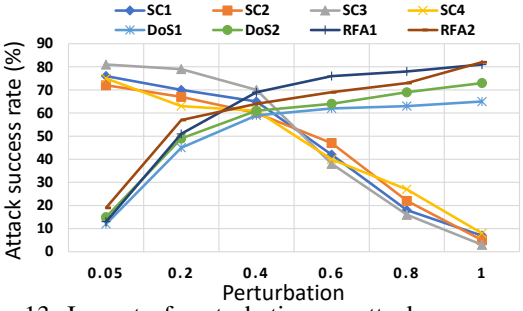

Fig. 13: Impact of perturbation on attack success rate

on the same host and was not detected or migrated. However, the adversary could not perform the attack successfully. The green bar shows that the attack was completely successful. This figure shows that by increasing the number of VMs per host, the chance of migration or not being co-locate by the victim increases up to 4X. It shows even if the adversary co-locates with the victim, the chance of success would be low as there are other applications running on the host that impacts the attack's setting.

B. Improving security using resource isolation

In order to study the effects of resource isolation, we enforce several resource partitioning and isolation techniques. We employ core isolation (thread pinning to physical cores), to constrain interference context switching. We employ the Cache Allocation Technology (CAT) available in Intel chips [49] to isolate last level cache (LLC). The size of cache partitions can be changed at runtime by reprogramming MSR registers. We also use the outbound network bandwidth partitioning capabilities of Linux's traffic control. We employ the `qdisc` [50] to enforce bandwidth limits. To perform DRAM bandwidth partitioning, RPS monitors the DRAM bandwidth usage of each application using Intel PCM to co-locate jobs on the same machine where it can accommodate their aggregate peak memory bandwidth usage.

Figure 16 shows the impact of isolation techniques on the effectiveness of attacks. As expected, when no isolation is used, we have a significantly higher success rate. As a result, introducing thread pinning benefits, since it reduces core contention. The dominant resource usage of each application determines which isolation technique benefits it the most. Thread pinning mostly benefits workloads bound by on-chip resources, such as L1/L2 caches and cores. Adding network bandwidth partitioning lowers success rate for DoS

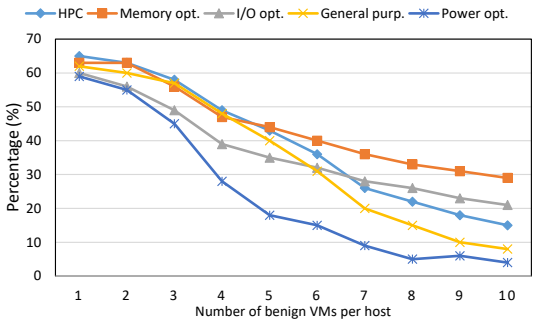

Fig. 14: Average of attacks' success rate when multiple VMs are running in the host

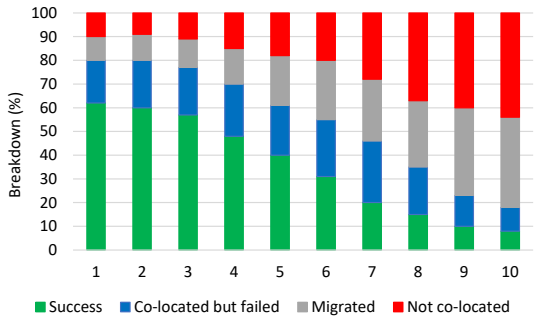

Fig. 15: Breakdown of attacks' success or failure on a general purpose server (X-axis shows the number of concurrent jobs)

attacks. It primarily benefits network-bound workloads, for which network interference conveys the most information for detection of co-residency. Cache partitioning has the most dramatic reduction in success rate of SCA, as they are LLC-bound applications. Enforcing core isolation is also sufficient to degrade the success rate of RFAs. Finally, memory bandwidth isolation further reduces success rate by 10% on average, benefiting jobs dominated by DRAM traffic. It is more effective on DoS and RFA and has less impact on SCAs.

The number of co-residents also affects the extent to which isolation helps. The more co-scheduled applications exist per machine, the more isolation techniques degrade success rate, as they make distinguishing between co-residents harder. Improving security using isolation, however, comes at a performance penalty of 32% on average in execution time, as threads of the same job are forced to contend with one another. Alternatively, users can overprovision their resource reservations to avoid performance degradation, which results in a 47% drop in utilization. This means that the cloud provider cannot leverage CPU idleness to share machines, decreasing the cost benefits of cloud computing.

Our analysis highlights a design problem with current datacenter platforms. Traditional multicores are prone to contention, which will only worsen as more cores are integrated into each server, and multi-tenancy becomes more pronounced. Existing isolation techniques are insufficient to mitigate security vulnerabilities, and techniques that provide reasonable security guarantees sacrifice performance or cost efficiency, through low utilization. This highlights the need for new

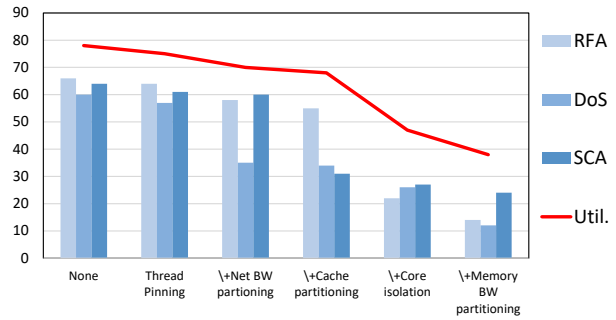

Fig. 16: Attacks' success rate and CPU utilization (Util) with isolation techniques

techniques to be invented that guarantee security at high utilization for shared resources.

C. Transferability analysis

Different RPSs use different underling mechanisms to pre-form resource provisioning. For instance, Quasar [5] leverages collaborative filtering to quickly determine which applications can be co-scheduled on the same machine without destructive interference. CherryPick [7] and Ernest [8] also uses machine learning for selecting close to optimal configuration on the cloud. Therefore, we tried to answer the question of whether an adversarial *Cloak* that is targeting a specific RPS can also force other RPS, which uses different underling mechanisms, to co-locate it with the victim. Specifically, we are testing the transferability of our attack methodology across RPS, which will showcase the robustness of the proposed adversarial sample crafting.

For this purpose, we setup an experiment where we generate *Cloaks* that target a specific RPS and see if they can also fool the other RPS. Table III shows the results of our transferability analysis for all combinations. It should be noted that the analysis is performed under these circumstances: perturbation is set to 0.4, the monitoring period is one minute, the number of VM allowed on the same host is two, and none of the isolation techniques is used. Each value of table III is attacks' average success rate (in percentage). Each row represents the original RPS and each column represents the substituted RPS. Where the column and the row are equal, it means we did not replace the RPS. For example, we used PARIS as our original RPS and then we reverse-engineered it. For transferability analysis, we replaced the original RPS (PARIS) with another RPS (CherryPick in this example). Then we used the Cloak & Co-locate approach to attack the system. Afterward, we evaluated the success rate of our attacks. In the above example, the average success rate of attacks was 56.5% as shown in Table III.

The results from Table III show that the reverse-engineered model from PARIS is more generalizable than others as the transferability of attacks is 55.1% (shown in blue color). Because of this generalization, we selected PARIS as our original RPS to study the Cloak & Co-locate attack throughout the paper. The results also show that the traces generated by FTG for PARIS can be applied to CherryPick, Ernest,

TABLE III: Attacks' average success rate (all values are percentage). Each row represent original RPS and each column is for substituted RPS

	PARIS	CherryPick	Ernest	Quasar	Average
PARIS	62.3	56.5	53.2	48.7	55.1
CherryPick	54.4	65.4	54.0	44.1	54.4
Ernest	46.8	55.7	67.4	33.5	50.8
Quasar	53.6	47.3	52.9	59.3	53.3
Average	54.3	56.2	56.9	46.4	63.6

and Quasar with a success rate of 56%, 53%, and 48%, respectively. This indicates that the ML model used to craft the adversarial samples is transferable to other systems as long as we can mimic the RPS's functionality.

On the other hand, results show that Ernest has the lowest transferability as it is the simplest RPS in our evaluation, and the average success rate of attacks transferred from Ernest to other RPSs is 50.8%. Ernest is based on a statistical technique and therefore cannot capture the complexity of the system. For the same reason, it is the most vulnerable RPS as the transferability of attacks from other RPSs to it is the highest (average success rate is around 56.9%, shown in red color). We observed that Quasar is the most resilient RPS against Cloak & Co-locate attack. The average success is only 46.4% as shown in the green color. The Quasar's collaborative filtering and its complexity are the reasons behind this immunity. Without considering the transferability (no substitution of original RPS), the average success rate of attacks is around 63% (the purple color).

VIII. Conclusions

In this work, we proposed Cloak & Co-locate – a novel approach to improve the effectiveness of distributed attacks on cloud infrastructure. For this purpose, we demonstrated that by reverse-engineering the resource provisioning system and employing the adversarial machine learning attack, adversary VM can co-locate itself with the victim and evade detection, as well as migration caused by the scheduler. For this purpose, we proposed to use a fake trace generator (Cloak) and wrap it around the adversary kernel. The fake trace generator is spawned as a separate thread, generating a pattern close to the victim VM's pattern, fooling the scheduler to co-locate it with the victim VM. After co-location, FTG continuously crafts new behavior to disguise itself and fool RPS for remaining co-located on the same host as the victim. Our results show that applying strict isolation can reduce the impact of such attacks while increases the cost of cloud (lower utilization). This research work, while deployed on a medium-size cluster, will motivate real-world public cloud providers to introduce stricter isolation solutions in their platforms and systems architects to develop robust RPSs that provide security and performance predictability at high utilization.

References

[1] M. Dano, "Edge computing mini data centers are rolling out for real. what's next?" https://www.lightreading.com/the-edge/edge-computing-mini-data-centers-are-rolling-out-for-real-whats-next/d/d-id/755493, 2019.

[2] Amazon, "Aws for the edge," https://aws.amazon.com/edge/.

[3] A. S. Khalid, "Mini-cloud system for enabling user subscription to cloud service in residential environment," Jul. 15 2014, uS Patent 8,782,637.

[4] C. Delimitrou and C. Kozyrakis, "Hcloud: Resource-efficient provisioning in shared cloud systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 473–488.

[5] ——, "Quasar: resource-efficient and qos-aware cluster management," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 127–144.

[6] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach," in *ACM SoCC*, 2017.

[7] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 469–482.

[8] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: efficient performance prediction for large-scale advanced analytics," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 363–378.

[9] H. Sayadi, H. Wang, T. Miari, H. M. Makrani, M. Aliasgari, S. Rafatirad, and H. Homayoun, "Recent advancements in microarchitectural security: Review of machine learning countermeasures," in *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2020, pp. 949–952.

[10] H. Sayadi, Y. Gao, H. Mohammadi Makrani, T. Mohsenin, A. Sasan, S. Rafatirad, J. Lin, and H. Homayoun, "Stealthminer: Specialized time series machine learning for run-time stealthy malware detection based on microarchitectural features," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, 2020, pp. 175–180.

[11] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh, "Constructing and characterizing covert channels on gpgpus," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 354–366.

[12] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[13] C. Canella, K. N. Khasawneh, and D. Gruss, "The Evolution of Transient-Execution Attacks," in *GLSVLSI*, 2020.

[14] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[15] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[16] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 19–35.

[17] F. Liu, L. Ren, and H. Bai, "Mitigating cross-vm side channel attack on multiple tenants cloud platform." *JCP*, vol. 9, no. 4, pp. 1005–1013, 2014.

[18] S. Gupta and P. Kumar, "Vm profile based optimized network attack pattern detection scheme for ddos attacks in cloud," in *International Symposium on Security in Computing and Communication*. Springer, 2013, pp. 255–261.

[19] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[20] H. Wang, H. Sayadi, T. Mohsenin, L. Zhao, A. Sasan, S. Rafatirad, and H. Homayoun, "Mitigating cache-based side-channel attacks through randomization: A comprehensive system and architecture level analysis," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1414–1419.

[21] H. Wang, H. Sayadi, S. Rafatirad, A. Sasan, and H. Homayoun, "Scarf: Detecting side-channel attacks at real-time using low-level hardware features," in *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 2020, pp. 1–6.

[22] Z. Xu, H. Wang, and Z. Wu, "A measurement study on co-residence threat inside the cloud," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 929–944.

[23] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 990–1003.

[24] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, "Evasion attacks against machine learning at test time," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2013, pp. 387–402.

[25] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[26] P. Laskov *et al.*, "Practical evasion of a learning-based classifier: A case study," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 197–211.

[27] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against deep learning systems using adversarial examples," *arXiv preprint arXiv:1602.02697*, vol. 1, no. 2, p. 3, 2016.

[28] K. N. Khasawneh, N. Abu-Ghazaleh, D. Ponomarev, and L. Yu, "Rhmd: Evasion-resilient hardware malware detectors," in *The 50th International Symposium on Microarchitecture (MICRO)*, 2017.

[29] M. S. Islam, K. N. Khasawneh, N. Abu-Ghazaleh, D. Ponomarev, and L. Yu, "Efficient hardware malware detectors that are resilient to adversarial evasion," *IEEE Transactions on Computers*, 2021.

[30] C. Delimitrou and C. Kozyrakis, "Bolt: I know what you did last summer... in the cloud," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1. ACM, 2017, pp. 599–613.

[31] H. M. Makrani, H. Sayadi, N. Nazari, S. M. P. Dinakarrao, A. Sasan, T. Mohsenin, S. Rafatirad, and H. Homayoun, "Adaptive performance modeling of data-intensive workloads for resource provisioning in virtualized environment," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 5, no. 4, pp. 1–24, 2021.

[32] K. Mills, J. Filliben, and C. Dabrowski, "Comparing vm-placement algorithms for on-demand clouds," in *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE, 2011, pp. 91–98.

[33] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[34] M.-M. Bazm, M. Lacoste, M. Südholt, and J.-M. Menaud, "Isolation in cloud computing infrastructures: new security challenges," *Annals of Telecommunications*, vol. 74, no. 3-4, pp. 197–209, 2019.

[35] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *2011 IEEE symposium on security and privacy*. IEEE, 2011, pp. 313–328.

[36] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud," *IEEE/ACM Transactions on Networking*, vol. 23, no. 2, pp. 603–615, 2014.

[37] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 199–212.

[38] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: improve your cloud performance (at your neighbor's expense)," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 281–292.

[39] H. M. Makrani, S. Rafatirad, A. Houmansadr, and H. Homayoun, "Main-memory requirements of big data applications on commodity server platform," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 653–660.

[40] W. Zhang, X. Jia, C. Wang, S. Zhang, Q. Huang, M. Wang, and P. Liu, "A comprehensive study of co-residence threat in multi-tenant public paas clouds," in *International Conference on Information and Communications Security*. Springer, 2016, pp. 361–375.

[41] M. S. Inci, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "Co-location detection on the cloud," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2016, pp. 19–34.

[42] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[43] M. Payer, "Hexpads: a platform to detect "stealth" attacks," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 138–154.
[44] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.
[45] N. Nazari and M. E.Salehi, *Hardware Architectures for Deep Learning*. Materials, Circuits and Device, 2020, ch. Binary Neural Networks, pp. 95–115.
[46] N. Nazari, M. Loni, M. E. Salehi, M. Daneshtalab, and M. Sjodin, "Totnet: An endeavor toward optimizing ternary neural networks," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2019, pp. 305–312.
[47] S. K. Smit and A. E. Eiben, "Comparing parameter tuning methods for evolutionary algorithms," in *Congress on Evolutionary Computation*, 2009.
[48] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *IISWC*. IEEE, 2013.
[49] *Intel R 64 and IA-32 Architecture Software Developer's Manual, vol3B: System Programming Guide, Part 2, September 2014.*
[50] *Martin A. Brown. Traffic control howto. http://linux-ip.net/articles/Traffic-Control-HOWTO/.*