Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Driftor: mitigating cloud-based side-channel attacks by switching and migrating multi-executor virtual machines*

Chao YANG†, Yun-fei GUO, Hong-chao HU, Ya-wen WANG, Qing TONG, Ling-shu LI

National Digital Switching System Engineering & Technological Research Center, Zhengzhou 450003, China

†E-mail: 1989600235@qq.com

Received Aug. 31, 2018; Revision accepted Nov. 26, 2018; Crosschecked May 13, 2019

Abstract: Co-residency of different tenants' virtual machines (VMs) in cloud provides a good chance for side-channel attacks, which results in information leakage. However, most of current defense suffers from the generality or compatibility problem, thus failing in immediate real-world deployment. VM migration, an inherit mechanism of cloud systems, envisions a promising countermeasure, which limits co-residency by moving VMs between servers. Therefore, we first set up a unified practical adversary model, where the attacker focuses on effective side channels. Then we propose Driftor, a new cloud system that contains VMs of a multi-executor structure where only one executor is active to provide service through a proxy, thus reducing possible information leakage. Active state is periodically switched between executors to simulate defensive effect of VM migration. To enhance the defense, real VM migration is enabled at the same time. Instead of solving the migration satisfiability problem with intractable CIRCUIT-SAT, a greedy-like heuristic algorithm is proposed to search for a viable solution by gradually expanding an initial has-to-migrate set of VMs. Experimental results show that Driftor can not only defend against practical fast side-channel attack, but also bring about reasonable impacts on real-world cloud applications.

Key words: Cloud computing; Side-channel attack; Information leakage; Multi-executor structure; Virtual machine switch; Virtual machine migration

https://doi.org/10.1631/FITEE.1800526 **CLC number:** TP393

1 Introduction

In recent years, cloud computing has experienced explosive development (Li et al., 2017, 2018; Wu et al., 2017, 2018) due to its advantages such as flexibility and cost-effectiveness, which occur as a result of resource sharing. While beneficial, shared resources are exposed to various threats, both conventional and unconventional. Side-channel attack (Yarom and Falkner, 2014; Liu et al., 2015; Gruss et al., 2016), one of the major unconventional threats, benefits from shared resources in cloud, because it provides an easy way for attackers to observe other tenants' behaviors and deduce private information (or secret) from them. Many countermeasures (Zhang and Reiter, 2013; Li et al., 2014; Liu and Lee, 2014; Pattuk et al., 2014) have been proposed at different levels of cloud-based virtualization structure; however, none of them are practical, because they either target specific side channels or require significant modifications to current cloud platforms.

Fortunately, there is a promising method that prevents an adversary from achieving co-residency with victims, which is general and immediately deployable. While the static version (Kwiat et al., 2015; Han et al., 2016, 2017; Ezhilchelvan and Mitrani, 2017) that interferes with a virtual machine (VM) allocation process seems helpless once the adversary has achieved co-residency, a dynamic approach

* Project supported by the National Natural Science Foundation of China (Nos. 61521003 and 61602509), the National Key Research and Development Program of China (Nos. 2016YFB0800100 and 2016YFB0800101), and the Key Technologies Research and Development Program of Henan Province of China (No. 172102210615)

ORCID: Chao YANG, http://orcid.org/0000-0002-4796-7011

© Zhejiang University and Springer-Verlag GmbH Germany, part of Springer Nature 2019

(Zhang YL et al., 2012; Moon et al., 2015; Wang et al., 2016) that periodically migrates VMs between servers becomes favorable. Among security-oriented VM migrations, some coarse-grained studies (Zhang YL et al., 2012; Wang et al., 2016) focused on whether a VM should be migrated. In contrast, Moon et al. (2015) first set up an information leakage model, and developed a scalable algorithm to calculate an optimal strategy which shows new destinations for each VM. However, there are two major shortages: (1) the adversary model is impractical; (2) fast side channels fall beyond the defense.

In this study, we draw two key observations: (1) the final goal of defense is mitigating valid attacks, where the adversary has stolen a certain percentage of secret which can be used to recover the whole content of that secret; (2) the number of VMs is one of the decisive factors that influence the efficiency of the defense algorithm. Therefore, we first establish a practical adversary model, and accordingly set our defense target. Then we propose Driftor, a system that realizes periodic VM migration by creating, switching, and migrating executors for each VM. An executor of a VM is a replica, which provides the same service and shares the same data with that VM. If we properly place different executors of a VM, VM migration can be simulated by periodically switching serving entity between executors, thus making the overhead of migration (actually switching) algorithm negligible. To enlarge the scope of VM migration, Driftor conducts real migration for executors between different servers. This global migration problem is initially solved by reducing it to CIRCUIT-SAT, whose unsatisfactory scalability is improved by a greedy-like algorithm. Experimental results show that Driftor can successfully defend against effective side-channel attacks, while the fast attacks described in Irazoqui et al. (2015) and Liu et al. (2015) can be mitigated with acceptable performance degradation.

2 Background and related work

2.1 Side-channel attacks in cloud

Side-channel attack has long been a research point. When Ristenpart et al. (2009) carried out a real co-residency attack in Amazon EC2, side-channel attacks became a concern in cloud security. Existing research mostly focuses on constructing side channels through shared resources, such as cache (Yarom and Falkner, 2014; Liu et al., 2015; Gruss et al., 2016) and memory (Bosman et al., 2016), and this has been a trend in cloud security research. According to the way by which tenants co-reside in cloud, cloud-based side-channel attacks can be divided into two types: cross-process side channel (Zhang et al., 2014) and cross-VM side channel (Zhang YQ et al., 2012).

The cross-process side channel is usually used in the Platform-as-a-Service (PaaS), where tasks of different tenants usually run in different Linux containers in a same VM. Computing resources and the operating system are shared by all tenants in the VM.

The cross-VM side channel is usually used in the Infrastructure-as-a-Service (IaaS), where different users share the same hardware platform.

In a cross-VM attack, the adversary usually launches VMs and tries to make the VMs co-locate with a victim VM on the same server. After verifying the target, the attacker constructs side channels and steals secrets from co-resident VMs by operations, such as Prime+Probe (Irazoqui et al., 2015; Liu et al., 2015), Flush+Reload (Yarom and Falkner, 2014), or Flush+Flush (Gruss et al., 2016). Such side channels pose a serious threat on cloud clients who are equipped with encryption keys, and the attack might last 2–3 min (Liu et al., 2015). Most public clouds provide service through VMs and act as IaaS, such as Amazon EC2 (2018), Rackspace (2018), and Microsoft Azure (2018). They may all suffer from cross-VM attacks. Thus, we focus on the defense against cross-VM side-channel attacks.

2.2 Countermeasures

According to the virtualization layer where the defense is enforced, current proposed countermeasures against side channels are divided mainly into four types. The hardware-based methods (Wang and Lee, 2007, 2008; Liu and Lee, 2014) are effective in theory yet very complex in practice, involving a lot of considerations, such as side effects, economic feasibility, extensibility, and flexibility. Thus, it may take years to put into commercial use. The second type of defense works inside the operating system (OS) of a guest, such as that proposed by Zhang and Reiter (2013) to inject noise to L_1 and L_2 caches for protected processes. The third type of defense takes effect at the application level by periodically partitioning a cryptographic key into multiple parts and distributing

them across multiple VMs (Pattuk et al., 2014).

The last type defense is called the "hypervisor-based approach." Vattikonda et al. (2011) and Almeida et al. (2016) adopted a method of hiding the program's running time. Li et al. (2014) provided defense by alerting when timing was exposed to an external observer. All these might induce a side effect on cloud routine. Some isolation-based methods can provide defense, such as statistical multiplexing of shared resources and isolating VMs from each other as much as possible (Moscibroda and Mutlu, 2007; Raj et al., 2009; Feng et al., 2011; Kim et al., 2012). However, such defense would likely reduce the use of the memory and induce a waste of computing resource, which has a counter effect on cloud's prospects. Varadarajan et al. (2014) proposed a way by modifying the Xen scheduler and limiting the frequency of central processing unit (CPU) preemption to defend against the side-channel attacks proposed by Zhang YQ et al. (2012), but for cross-core attacks which do not need CPU preemption, this defense method would fail.

Migration-based defense (Moon et al., 2015), which limits the co-residency time of VMs between different tenants, is another approach to solve the problem. It works by periodically migrating co-located VMs onto different servers. The original ILP strategy to calculate the optimal placement at each epoch is time-consuming. Therefore, a greedy-like algorithm based approximation is promoted to improve scalability in the following three aspects:

1. Incremental benefit computation

Since re-computing a benefit occupies much run time of the baseline greedy, Nomad computed the change of the current value of the objective function by updating only information leakage for the set of dependent client pairs, whose leakage amount is affected by moves.

2. Search space reduction

Aiming at reducing search space which consists of machines and moves, Nomad employed hierarchical methods which group clients into clusters. Besides, Nomad employed a pruning operation to wipe off some useless move sets.

3. Lazy evaluation

Since updating move sets generates all possible moves making the computation of benefit time-consuming, Nomad raised a lazy evaluation at the beginning of an epoch where the entire move table is populated, followed by the traversal of the move sets starting from the move that gives the largest benefit. If a move is feasible and its claimed benefit lies within 95% of the current value, then that move will be adopted. Otherwise, the move is re-inserted with an updated benefit.

Inspired by Nomad, we adopt a VM migration as a baseline defense mechanism. Driftor differs from Nomad from the following three aspects:

1. Adversary model

Nomad proposed an attack model to maximize the whole information leakage over cloud, while our model focuses on effective side channels; that is, the attacker aims at stealing enough portion of secret. Therefore, our model is definitely more practical.

2. Form of VM

Nomad migrated a normal VM, while Driftor sets up a multi-executor structure for each VM, which consists of several replicated VMs (called "executors") and a proxy. To decrease the information leakage rate, only one of the executors can be activated at any time, while others are suspended and kept synchronized through the shared database. The proxy is responsible for forwarding data between users and any of the executers.

3. Type of VM migration

Instead of practically migrating VMs between servers, Driftor is equipped with a mechanism called an "executor switch," which switches the active state between all VM's executors. In this way, a VM that can be attacked for information leakage is migrated to the server of a different executor. Nevertheless, real VM migration is adopted to enlarge the scope of switch.

2.3 Motivations

Our work is motivated by a lot of previous research. For example, StopWatch (Li et al., 2014) triplicated each VM in cloud and placed three replicas, so that they were co-resident with non-overlapping sets of other VMs. Then the timing of input/output (I/O) events at these replicas was collected to determine timings observed by each one or by an external observer, so that observable timing behaviors are similar. We set several replicas of each VM, but do not change their timing behaviors since we aim at defending against co-resident attacks, while

StopWatch focuses on defense against remote timing-based side channels.

Another motivating work is mimic defense (Wu, 2016; Hu et al., 2018), which features a mimic structure with several executors and a proxy. Executors, the same or different, provide the same service as that of the whole system. In their system, all executors were activated and used in parallel, while the proxy forwarded data between end users and executors, and conducted majority decisions on response data from all executors. Our work differs in that only one executor is used at any time, which saves a lot of cost. Besides, our proxy acts as not only a reverse proxy, but also an executor switch by forwarding data to a designated executor.

3 Adversary model

In this section, we describe a general and practical adversary model that captures the most of co-residency based side-channel attacks in cloud. First of all, we give some assumptions about tenants in cloud:

1. Potential victims

We assume that each client has some private information (called "secret," such as encryption keys or private database records), which is interesting for attackers. Therefore, any VM of this client might be the targets of side-channel attacks.

2. Potential attackers

We assume that malicious clients cannot be figured out by any client or cloud provider.

3. Information replication across VMs of a tenant

Different VMs of the same tenant share the secret of that tenant. While this condition might fail, we propose a powerful adversary model so that our defense can be widely applicable.

The goal of an adversary is to extract enough portion (more than δ) of secret (with its length denoted as L) which can be used to recover the whole content. For example, only enough leaked bits can make a brute-force attack of a long (e.g., 256 bits) encryption key possible. Some partitioned encoded information can be recovered only with enough blocks of secret (Pattuk et al., 2014); otherwise, the acquired portion of secret is meaningless. Some capabilities of the adversary are as follows:

(1) Cross-VM side-channel attacks. The majority of side channels in cloud are co-residency based cross-VM attacks, which are carried out based on various resources (e.g., CPU, memory, and network) shared between the attacker and its co-resident VMs. In fact, Driftor can be applied to defending against side channels on any movable entity in cloud with minor changes, such as container (Kämäräinen et al., 2015). These side channels can be roughly divided into two types: slow side channels (Zhang YQ et al., 2012) and fast side channels (Irazoqui et al., 2015; Liu et al., 2015). In fact, their boundary is not clearly defined, and an empirical value is half an hour.

(2) Constant leakage rate. To simplify the formulation of side channels, we assume a constant leakage rate of K bits per epoch (assuming that time is divided into epochs, denoted as ΔT) for any side channel without considering details of different attacks. We admit that different attacks may have different leakage rates or different temporal properties (e.g., K may decrease or increase with time), which means that our model can be further improved as a future work.

(3) Efficient information accumulation across time. We assume that the adversary can accumulate information from side channels across epochs under co-residency with a target VM. For example, if the adversary is co-resident with the victim at time T_1 and T_3, but not T_2, the information gathered during T_1 and T_3 can be combined. Even though a real attacker might possibly derive duplicate/useless information across epochs, we prefer a more powerful adversary model for better applicability of Driftor.

(4) Information replication across VMs of the victim. Information gathered by co-residency with all VMs of the same victim can be added up to a total amount of secret that an adversary's VM gets from the victim. Despite duplicate information derived from different VMs, we are building this adversary model strong.

(5) No collusion across adversaries. We assume that different clients do not collaborate because a new identifier (e.g., a verified credit card) is very costly, so that Sybil attacks for collusion (Douceur, 2002) are impossible.

(6) Information collation across VMs of the adversary. A strong adversary model is established

when information derived by any adversary VM can be added up to a total amount of secret that the attacker steals from the victim.

Taking Fig. 1 as an example, the numbers of leakage epochs from the victim to the attacker after three epochs' co-residency are shown in Tables 1–3.

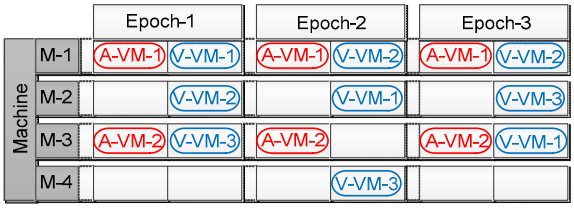

Fig. 1 Co-residency between an adversary and a victim

Table 1 Across time

	V-VM-1	V-VM-2	V-VM-3
A-VM-1	1	2	0
A-VM-2	1	0	1

Table 2 Information replication of the victim

	Victim
A-VM-1	3
A-VM-2	2

Table 3 Information sharing across attacker's VMs

	Victim
Adversary	5

For the adversary, this side-channel attack is effective when $\delta \times L < 5K$. Therefore, our defense aims at preventing leakage of more than δ percent of secret between client pairs, as shown in the next sections.

4 System overview

In this section, we present Driftor, a system which creates a multi-executor structure for each VM and periodically provides defense against side channels by switching and migrating executors. Fig. 2 shows an overall system architecture of the Driftor.

1. High-level idea

For the practical and powerful adversary model, where information leakage between tenants is characterized by information replication between victim's VMs and by collaboration between attacker's VMs in Section 3, we envision VM migration which limits co-residency between different clients' VMs as a basic defense strategy. However, migration alone fails to defend against fast side-channel attacks, because the migration decision algorithm involving all VMs in cloud is of high computational complexity. Moreover, frequent migration leads to severe performance degradation.

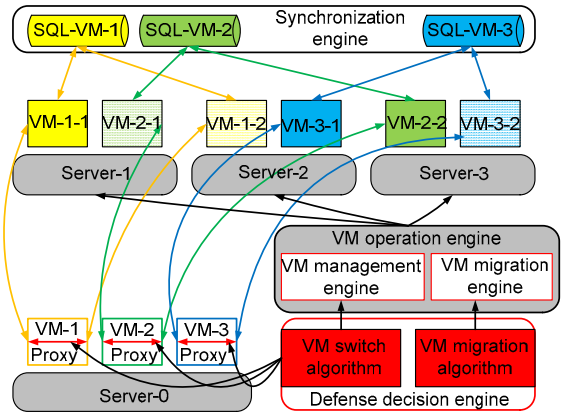

Fig. 2 System overview of the Driftor

Therefore, we realize fast migration by creating and distributing multiple replicas of a VM which provide completely the same service as a VM, and switching between them to mitigate co-residency. Since this migration is limited in a few servers hosting executors, we enlarge the scope by real migration of executors, which, however, might fail to meet scalability. Considering that the attacker's target is an effective attack aiming to leak enough portion of a secret, we try to work out a minimum migration strategy by developing a greedy-like algorithm. Thus, Driftor is able to cope with any side channel in the adversary model at a reasonable cost.

2. Multi-executor structure

A fundamental element of Driftor is the multi-executor structure of VMs in cloud, which is shown in Fig. 3. It contains the following components:

(1) Executors. Replicas of an original VM provide the same service as VM (the original VM is an executor). All of them share a common database, which stores both running states and data. To reduce co-residency, they are usually distributed onto different servers. Besides, there is only one running executor at any time providing service outwards, and

other executors are kept suspended (not closed) to facilitate quick activation.

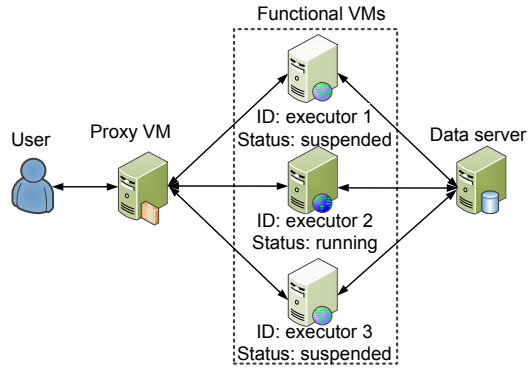

Fig. 3 Multi-executor structure of virtual machine (VM)

(2) Proxy. Another VM or a single process on a common VM is responsible for transmitting communication data between a user and an executor, and switching serves executor by transmitting data of the new session to a different executor. Since proxy does not have any secret, we can consider it as side-channel free.

(3) Synchronization engine. A single module integrated in the cloud management system is responsible for synchronizing running states and data of all executors. It is implemented according to Thompson et al. (2014), using an additional database to store running states and data of all executors. The running executor keeps updating the database. When a suspended executor is activated, this engine updates its running state to the same as that of the newest state of the last running executor.

(4) Defense decision engine. As another critical factor of Driftor, the defense decision engine consists of two parts: switching module and migration module. As shown in Fig. 4, the switching module (in the left) determines the running executor in the next epoch with the VM's set of executors and the history of co-residency between clients. The result is used to guide the switching operation by the executor management engine and to change the history of the co-residency afterwards. When calculating migration decision in the migration module, sets of executors to be migrated (that is, co-residency of different tenants' executors during the last epoch enables a new effective attack, so that co-resident VMs have to be migrated) must be first worked out with the history of co-residency. Then the migration algorithm takes the data, current VM assignments, and server workloads as inputs to obtain a minimum migration solution, which is used to instruct the executor migration operation.

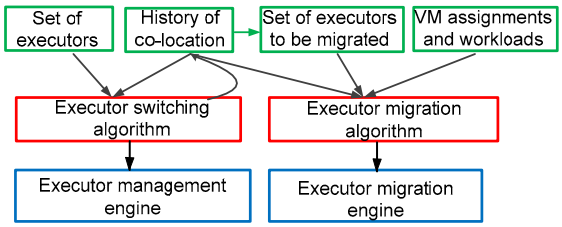

Fig. 4 Main components of the defense decision engine

3. End-to-end workflow

Take VM-1 in Fig. 2 as an example. All communication data from users of VM-1 will first be sent to VM-1 proxy, which redirects the stream to a current active executor (VM-1-1 or VM-1-2). Similarly, responses are returned to the user through the proxy. At the beginning of each defense interval, the executor management engine activates a new executor (or keeps the current active executor) for each VM according to decisions from the switching algorithm. When a new executor becomes available, the old active executor should be immediately suspended after current sessions with this executor have ended. Then all new requests would be redirected to the new active executor by the proxy. During the same defense epoch, the migration algorithm computes a minimum migration solution, which drives the executor migration engine to work as desired.

4. Security implications of Driftor

The security is achieved by the executor switch and executor migration:

(1) Executor switch simulates the effect of security-oriented migration by switching from one executor to another, and co-residency with an old active executor would no longer leak any secret. A new active executor on another server would not be co-resident with those VMs that are co-located with the old active executor in the previous defense interval; thus, co-residency is limited. In addition, fast side channels can be handled in this way, because we can reduce our defense interval to be much smaller than the attack duration of the fast side channels (2–3 min). Because of the switch decision algorithm and switch operation results in minor overhead, security-oriented frequent switch is acceptable.

(2) Executor migration improves the defense ability against side channels. Migration simulated by executor switch can achieve only static and restricted defense, because the scope of switch destinations is limited. Therefore, an effective attack will be finally fulfilled within a relatively small duration. When the switch range is enlarged by migration of executors, an effective attack can be delayed much longer time. Thus, the security level can be improved.

5 Design of a reasonable defense scheme

In this section, we will design a reasonable defense scheme for Driftor. We first present a basic idea of the defense strategy, followed by the specifications of the switch algorithm and migration algorithm. Symbols and denotations used in this section are listed in Table 4.

5.1 Basic idea

Driftor periodically carries out both executor switch and executor migration. At the beginning of each defense interval, history pair-wise information leakage will first be updated according to co-residency of clients during the last interval. Then a new executor for each VM is activated to provide service, while the old active one would be suspended. When the executor switch for all VMs has finished, we check the history pair-wise information leakage if there is any new client pair that has been co-resident for enough time to enable an effective attack. If there is no such client pair, defense engine stops until the next interval. Otherwise, it has to make a minimum migration decision, which at least separates new client pairs that cannot co-exist on the same server. To decrease the pair-wise information leakage rate under the adversary model in Section 3, we regulate the co-residency pattern between tenants. These two different clients can co-locate only their VMs to form one VM pair (executor pair), so that the leakage rate between co-resident clients is no larger than the minimum value of K.

Defense interval is designed as follows: Our defense should be able to handle target side channels, so the interval should be smaller than T_{ATT} (actually, it is $\delta \times T_{\mathrm{ATT}}$ since we focus on effective attacks). Switch decision and operation, as well as migration decision and operation, should be completed within one interval. Since we consider a paralleled switch decision, a switch operation, and a migration operation, we have

$$t_{\mathrm{DE}}^{\mathrm{SW}} + T_{\mathrm{OP}}^{\mathrm{SW}} + t_{\mathrm{DE}}^{\mathrm{MIG}} + T_{\mathrm{OP}}^{\mathrm{MIG}} < \Delta t < \delta \times T_{\mathrm{ATT}}. \quad (1)$$

For simplicity, we define the interval as

Table 4 Symbols and denotations

Symbol	Meaning	Symbol	Meaning
δ	Security threshold of information leakage attacks	$\mathrm{Loc}(t+\Delta t)=\left\|l_{i,j}^{\mathrm{tar}}\right\|$	Placement of VMs at current epoch
Δt	Defense interval	$\mathrm{NoCo}(t)=\|nc_{i,j}\|$	Client pairs that cannot co-exist at the current epoch
T_{ATT}	Time to fulfill the target side-channel attack	$\mathrm{Co}(t)=\|c_{i,j}\|$	Number of co-resident executors between each pair of clients
$t_{\mathrm{DE}}^{\mathrm{SW}}$ and $T_{\mathrm{OP}}^{\mathrm{SW}}$	Time of a single switch operation and a switch decision, respectively	$\mathrm{NoMig}(t)=\|nm_i\|$	Activated executor for each VM which cannot be migrated in the case of service interruption
$t_{\mathrm{DE}}^{\mathrm{MIG}}$ and $T_{\mathrm{OP}}^{\mathrm{MIG}}$	Time of a migration operation and a migration decision, respectively	$\mathrm{Co}_{\mathrm{up}}(t)=\left\|c_{i,j}^{\mathrm{up}}\right\|$	Number of co-resident active VMs between each pair of clients
N_{C} and N_{S}	Numbers of clients and servers, respectively	$\mathrm{Co}_{\mathrm{up}}^{\mathrm{acc}}(0,\ t)=\left\|c_{i,j}^{\text{acc-up}}\right\|$	History leakage between each pair of clients
N_{EXE}	Number of executor for each VM	$S_{\mathrm{EXE}}^{i,j}=\left\{e_1^{i,j},\ e_2^{i,j},\ \ldots,\ e_{n\text{-exe}}^{i,j}\right\}$	Executors of a VM
$N_{\mathrm{S}}^{\mathrm{Cap}}$	Capability of servers to host VMs	$S_{\mathrm{EXE}}=\left\{e_k^{i,j}\right\}$	All VMs in cloud
$N_{\mathrm{C}}^{\mathrm{VM}}=\left\|n_i^{\mathrm{VM}}\right\|,\ i\in[1,\ N_{\mathrm{C}}]$	Number of VMs for each client	$S_{\nrightarrow\mathrm{Co}}$ and $S_{\nrightarrow\mathrm{Co}}^{t}$	All client-pairs and new client pairs that cannot co-exist
$\mathrm{Loc}(t)=\|l_{i,j}\|$	Placement of VMs at the last epoch		

$$\Delta t = \frac{\delta \times T_{\mathrm{ATT}}}{k},$$
$$k = 2,\ 3,\ ...,\ \left\lfloor \frac{\delta \times T_{\mathrm{ATT}}}{t_{\mathrm{DE}}^{\mathrm{SW}} + T_{\mathrm{OP}}^{\mathrm{SW}} + t_{\mathrm{DE}}^{\mathrm{MIG}} + T_{\mathrm{OP}}^{\mathrm{MIG}}} \right\rfloor, \quad (2)$$

where $t_{\mathrm{DE}}^{\mathrm{SW}}$, $T_{\mathrm{OP}}^{\mathrm{SW}}$, $t_{\mathrm{DE}}^{\mathrm{MIG}}$, and $T_{\mathrm{OP}}^{\mathrm{MIG}}$ should be assigned with their largest values with a reasonable normal size of cloud or a cluster (Moon et al., 2015). $\left\lfloor \frac{\delta \times T_{\mathrm{ATT}}}{t_{\mathrm{DE}}^{\mathrm{SW}} + T_{\mathrm{OP}}^{\mathrm{SW}} + t_{\mathrm{DE}}^{\mathrm{MIG}} + T_{\mathrm{OP}}^{\mathrm{MIG}}} \right\rfloor$ defines the defense ability against (especially fast) side-channel attacks: the larger, the better. As important parameters, δ and T_{ATT} can either be assigned with values offered by cloud providers or vary along with the fastest side-channel attack detected in cloud. As another critical parameter controlled by the cloud provider, we empirically set the value of k to three, since it provides Driftor a better defense capability than the case of two, while keeping a relatively low defense frequency to reduce defense overhead.

It is necessary to discuss N_{EXE}, which is expressed as

$$2 \le N_{\mathrm{EXE}} < N_{\mathrm{S}}^{\mathrm{Cap}}. \quad (3)$$

The lower bound of N_{EXE} is obvious. For the upper limit, we resort to the absurdity that if $N_{\mathrm{EXE}} \ge N_{\mathrm{S}}^{\mathrm{Cap}}$, executors of a single VM would occupy at least $\left\lfloor N_{\mathrm{EXE}} / N_{\mathrm{S}}^{\mathrm{Cap}} \right\rfloor$ servers. To decrease co-residency with other clients and to save server resource, $\left\lfloor N_{\mathrm{EXE}} / N_{\mathrm{S}}^{\mathrm{Cap}} \right\rfloor \times N_{\mathrm{S}}^{\mathrm{Cap}}$ executors among them would be placed on $\left\lfloor N_{\mathrm{EXE}} / N_{\mathrm{S}}^{\mathrm{Cap}} \right\rfloor$ exclusive servers in the best strategy. Then those $\left\lfloor N_{\mathrm{EXE}} / N_{\mathrm{S}}^{\mathrm{Cap}} \right\rfloor \times N_{\mathrm{S}}^{\mathrm{Cap}}$ executors contribute nothing to defense against side channels. So, $N_{\mathrm{EXE}} < N_{\mathrm{S}}^{\mathrm{Cap}}$, and inequality (3) is proved to be correct. In this study, we assign N_{EXE} with an empirical value of three, which is the same as that in Li et al. (2014).

5.2 Switching algorithm

To provide a satisfactory defense, executor switch should be fast and unpredictable, so Driftor enforces a random SwitchDecision algorithm (Algorithm 1**)** on each VM in parallel:

Algorithm 1 SwitchDecision that selects an active executor for the j^{th} VM of the i^{th} client in the current defense interval

Input: $S_{\mathrm{EXE}}^{i,j} = \{e_1^{i,j},\ e_2^{i,j},\ ...,\ e_{n\text{-exe}}^{i,j}\}$: executors of the j^{th} VM of the i^{th} client; $\mathrm{Co}_{\mathrm{up}}^{\mathrm{acc}}(0,\ t) = \left| c_{i,j}^{\text{acc-up}} \right|$: history co-residency time of different tenants; $S_{\neg\mathrm{Co}}$: client pairs that cannot be co-resident in the current interval; $S_{\neg\mathrm{Co}}^{t}$: new client pairs that cannot co-exist; $e_{\mathrm{c}}^{i,j}$: current active executor

Output: $e_x^{i,j}$: an executor to be active in the current interval

```
1  s_c^{i,j} ← CoresidentVMs(e_c^{i,j})
2  flag←1
3  for each VM x in s_c^{i,j} do
4    if x is active then
5      p←MasterTenant(x)
6      if p==i then: continue; end if
7      c_{i,p}^{acc-up} ← c_{i,p}^{acc-up} + 1
8      if c_{i,p}^{acc-up} ≥ kΔt then
9        flag←0; S_{¬Co} ← S_{¬Co} + <i, p>
10       S_{¬Co}^{t} ← S_{¬Co}^{t} + <i, p>
11     end if
12   end if
13 end for
14 S_{cand-EXE}^{i,j} ← S_{EXE}^{i,j}
15 if flag==0 then: S_{cand-EXE}^{i,j} ← S_{cand-EXE}^{i,j} − e_c^{i,j}; end if
16 for each VM x in S_{cand-EXE}^{i,j} do
17   y←HostServer(x)
18   if isServerExclusive(i, y) then
19     S_{cand-EXE}^{i,j} ← GetVMonServer(i, j, y); return;
20   end if
21 end for
22 e_x^{i,j} ← S_{cand-EXE}^{i,j}[rand(sizeof(S_{cand-EXE}^{i,j}))]
```

1. Updating pair-wise information leakage (line 7). After one defense interval, we identify client pairs that co-locate their active executors on the same server, and increase their mutual information leakage by one.

2. Getting new client pairs that cannot co-exist (lines 8–11). According to refreshed pair-wise information leakage, we further identify new client pairs that have co-located for enough time to enable effective side-channel attacks. This is used for the migration decision algorithm.

3. Selecting active executor (lines 16–22). First of all, we collect the distribution of all executors of the VM, each server of which is judged by whether

the executor's client has exclusive occupation. If these servers are exclusive, any executor on this server can be chosen as a new active one; otherwise, a randomly selected executor will be activated. It should be noted that a current active executor cannot be a candidate if co-residency with other tenants has been increased to the upper limit time of effective side channels.

5.3 Migration algorithm

5.3.1 Modeling and solving a satisfiability problem

Since Driftor aims at defending against effective side channels, our model makes migration decision as a satisfiability problem subject to some constraints, which are expressed as

$$\forall i \in \left[1, N_{\mathrm{EXE}} \times \sum_{i=1}^{N_{\mathrm{C}}} n_i^{\mathrm{VM}}\right], j \in [1,\ N_{\mathrm{S}}],\ l_{i,j}^{\mathrm{tar}} \in \{0,1\}, \quad (4)$$

$$\forall i \in \left[1,\ N_{\mathrm{EXE}} \times \sum_{i=1}^{N_{\mathrm{C}}} n_i^{\mathrm{VM}}\right],\ \sum_{j=1}^{N_{\mathrm{S}}} l_{i,j}^{\mathrm{tar}} = 1, \quad (5)$$

$$\forall j \in [1,\ N_{\mathrm{S}}],\ \sum_{i=1}^{N_{\mathrm{EXE}} \times \sum_{i=1}^{N_{\mathrm{C}}} n_i^{\mathrm{VM}}} l_{i,j}^{\mathrm{tar}} \le N_{\mathrm{CAP}}, \quad (6)$$

$$\begin{cases} \forall i_1,\ i_2 \in [0,\ N_{\mathrm{C}} - 1],\ i_1 \ne i_2, \\ k_1,\ k_2 \in [1,\ n_{i_1}^{\mathrm{VM}} \times N_{\mathrm{EXE}}],\ k_3 \in [1,\ n_{i_2}^{\mathrm{VM}} \times N_{\mathrm{EXE}}], \\ k_1 \ne k_2,\ j \in [1,\ N_{\mathrm{S}}], \\ l^{\mathrm{tar}}_{N_{\mathrm{EXE}} \times \sum_{i=1}^{i_1} n_i^{\mathrm{VM}} + k_1, j} \times l^{\mathrm{tar}}_{N_{\mathrm{EXE}} \times \sum_{i=1}^{i_1} n_i^{\mathrm{VM}} + k_2, j} \times l^{\mathrm{tar}}_{N_{\mathrm{EXE}} \times \sum_{i=1}^{i_2} n_i^{\mathrm{VM}} + k_3, j} = 0, \end{cases} \quad (7)$$

$$\begin{cases} \forall i_1,\ i_2 \in [0,\ N_{\mathrm{C}} - 1],\ i_1 \ne i_2, \\ c_{i_1,i_2} = \sum_{j=1}^{N_{\mathrm{S}}} \sum_{k_1=1}^{n_{i_1}^{\mathrm{VM}} \times N_{\mathrm{EXE}}} \sum_{k_2=1}^{n_{i_2}^{\mathrm{VM}} \times N_{\mathrm{EXE}}} l^{\mathrm{tar}}_{i_1 \times N_{\mathrm{EXE}} + k_1, j} \times l^{\mathrm{tar}}_{i_2 \times N_{\mathrm{EXE}} + k_2, j}, \end{cases} \quad (8)$$

$$\forall i_1,\ i_2 \in [0,\ N_{\mathrm{C}} - 1],\ i_1 \ne i_2,\ c_{i_1,i_2} \le nc_{i_1,i_2}. \quad (9)$$

Here are the explanations of the above constraints: constraint (4) regulates the value range of the placement matrix; constraint (5) describes the fact that a VM should be placed on exactly one server, while constraint (6) presents another necessary condition that capacities of servers that should not be exceeded; constraint (7), along with constraints (8) and (9), poses a constraint that different clients can exactly co-locate with one VM; constraint (9) implies that co-residency should be forbidden between client pairs in the case of effective side-channel attacks with further co-residency.

We solve this satisfiability problem by reducing it to satisfiability for Boolean circuits (CIRCUIT-SAT) (Shyamasundar, 1996) with the intent of adopting an SAT solver as an oracle. However, since CIRCUIT-SAT is known to be NP-complete (Garey and Johnson, 1983), we find that it is unsatisfying in large clouds, because inequality (1) has to be met to defend against fast side channels (Irazoqui et al., 2015; Liu et al., 2015). Generally, we need to make sure that $t_{\mathrm{DE}}^{\mathrm{MIG}} \ll \delta \times T_{\mathrm{ATT}}$, which refers to values of seconds or tens of seconds. This motivates the need for heuristic approximations, which we will describe in the next subsection.

5.3.2 Greedy-like heuristic

The main bottleneck of CIRCUIT-SAT is the large number of migration candidates, which has been expanded to N_{EXE} times the original scale, exponentially increasing possible placement strategies. Besides, sequential search in the solution space results in many useless efforts. Therefore, we propose a GreedyLikeHeuristic algorithm (Algorithm 2**)** that searches for potential minimum solutions.

Algorithm 2 GreedyLikeHeuristic that tries to find minimum migration solutions in huge search space

Input: Loc(t)=$|l_{i,j}|$: placement strategy in the last defense interval; $S_{\mathrm{EXE}} = \{e_k^{i,j}\}$: all executors in cloud; $S_{\neg\mathrm{Co}}$: VM pairs that cannot be co-resident in the current interval; $S_{\neg\mathrm{Co}}^{t}$: new client pairs that cannot co-exist; $e_{\mathrm{c}}^{i,j}$: current active executor

Output: $\mathrm{Loc}(t+\Delta t) = \left|l_{i,j}^{\mathrm{tar}}\right|$: new placement for the current defense interval; Error: no satisfactory solution is found

1 $S_{\neg Co}^{\mathrm{det}} = \left\{< e_{k1}^{i1,j1},\ e_{k2}^{i2,j2},\ s >\right\} \leftarrow$ getUncolPairs$\left(S_{\neg\mathrm{Co}}^{t},\ \mathrm{Loc}(t)\right)$

2 $S_{\neg\mathrm{Co}}^{\mathrm{det}-S} = \left\{< s,\ \left\{< e_{k1}^{i1,j1},\ e_{k2}^{i2,j2} >\right\} >\right\} \leftarrow$ groupByServer $\left(S_{\neg\mathrm{Co}}^{\mathrm{det}}\right)$

3 $S_{\mathrm{Mig}}^{\mathrm{min}-S} = \left\{< s, \{\{e_k^{i,j}\}\} >\right\} \leftarrow \mathrm{calMinMigSets}\left(S_{\neg\mathrm{Co}}^{\mathrm{det}-S}\right)$

4 $N_{\mathrm{Mig}}^{\mathrm{Init}} \leftarrow \mathrm{numPossibleMigSets}\left(S_{\mathrm{Mig}}^{\mathrm{min}-S}\right)$

5 **for** $p \leftarrow 1$ **to** $N_{\mathrm{Mig}}^{\mathrm{Init}}$ **do**

6 $\quad S_{\mathrm{Mig}}^{\mathrm{Init}} = \left\{e_k^{i,j}\right\} \leftarrow \mathrm{pickInitialMigSet}\left(S_{\mathrm{Mig}}^{\mathrm{min}-S},\ p\right)$

7 $\quad \left\{N_{\mathrm{Mig}}^{\mathrm{Client}},\ S_{\mathrm{Mig}}^{\mathrm{Init\text{-}C}} = \{< i, \{e_k^{i,j}\} >\}\right\} \leftarrow$ groupByClient $\left(S_{\mathrm{Mig}}^{\mathrm{Init}}\right)$

8 $S_{\text{Mig}}^{\text{Init-}C\text{-}S}=\left\{<i,\{e_k^{i,j}\}>\right\}\leftarrow \text{sortByClient}\left(S_{\text{Mig}}^{\text{Init-}C},S_{\neg\text{Co}}\right)$
9 $n_{\text{C}}\leftarrow 0$
10 **while** $n_{\text{C}}<N_{\text{Mig}}^{\text{Client}}$ **do**
11 clientId $\leftarrow\left(S_{\text{Mig}}^{\text{Init-}C\text{-}S}[n_{\text{C}}].i\right)$
12 $S_{\text{VM}}\leftarrow$getClientVMs(clientId);
13 $S_{\text{EX-}S}\leftarrow$getExclusiveServers(clientId);
14 **if** ¬host(S_{VM}, $S_{\text{EX-}S}$) **then**: **break**; **end if**
15 $n_{\text{C}}\leftarrow n_{\text{C}}+1$
16 **end while**
17 **if** $n_{\text{C}}\geq N_{\text{Mig}}^{\text{Client}}$ **then**: **break**; **end if**
18 $S_{\text{EM-}S}\leftarrow$getEmptyServers()
19 **while** $n_{\text{C}}<N_{\text{Mig}}^{\text{Client}}$ **do**
20 clientId $\leftarrow\left(S_{\text{Mig}}^{\text{Init-}C\text{-}S}[n_{\text{C}}].i\right)$
21 $S_{\text{VM}}\leftarrow$getClientVMs(clientId)
22 **if** ¬host(S_{VM}, $S_{\text{EX-}S}$) **then**: **break**; **end if**
23 $n_{\text{C}}\leftarrow n_{\text{C}}+1$
24 **end while**
25 **if** $n_{\text{C}}\geq N_{\text{Mig}}^{\text{Client}}$ **then**: **break**; **end if**
26 $S_{\text{RE-}S}\leftarrow$getRemainingServers()
27 $S_{\text{FO-}S}\leftarrow$getFullyOcuppiedServers($S_{\text{RE-}S}$)
28 **while** $n_{\text{C}}<N_{\text{Mig}}^{\text{Client}}$ **do**
29 clientId $\leftarrow\left(S_{\text{Mig}}^{\text{Init-}C\text{-}S}[n_{\text{C}}].i\right)$
30 $S_{\text{VM}}\leftarrow$getClientVMs(clientId)
31 $S_{\text{OCE-S}}\leftarrow$getOtherClientExclusiveServers($S_{\text{RE-S}}$)
32 $S_{\text{CCR-S}}\leftarrow$getCurrentClientResidentServers($S_{\text{RE-S}}$)
33 $S_{\text{RE-S-C}}\leftarrow S_{\text{RE-S}}-(S_{\text{FO-S}}\cup S_{\text{OCE-S}}\cup S_{\text{CCR-S}})$
34 $n_{\text{VM}}\leftarrow 0$
35 **while** $n_{\text{VM}}<$sizeof(S_{VM}) **do**
36 isHosted$\leftarrow$0
37 **for each** server **in** $S_{\text{RE-S-C}}$ **do**
38 $S_{\text{C}}\leftarrow$getResidentClients(server)
39 **if** canCoResident(clientId, S_{C}) **then**
40 host(VM, server); isHosted$\leftarrow$1
41 **end if**
42 **end for**
43 **if** isHosted==0 **then**: **break**; **end if**
44 $n_{\text{VM}}\leftarrow n_{\text{VM}}+1$
45 **end while**
46 **if** $n_{\text{VM}}<$sizeof(S_{VM}) **then**: **break**; **end if**
47 update($S_{\text{RE-S}}$); update($S_{\text{FO-S}}$)
48 $n_{\text{C}}\leftarrow n_{\text{C}}+1$
49 **end while**
50 **if** $n_{\text{C}}<N_{\text{Mig}}^{\text{Client}}$ **then**: **print** Error; **return**
51 **Output** $\text{Loc}(t+\Delta t)=\left|l_{i,j}^{\text{far}}\right|$; **return**
52 **End for**

One important input to this algorithm is $S_{\neg\text{Co}}^{t}$, which consists of new client pairs that cannot co-exist in the coming defense interval. First of all, we will find co-resident active executors and their hosting server in the last placement for each client pair (line 1). Then all these clients with their last active executors will be grouped by servers (line 2), followed by computing different minimum sets of executors for each server (line 3). Such sets of a server consist of executors to be migrated, so that the remaining executors can still co-exist on this server at least for the next defense interval.

The initial global migration set is picked from one of such sets on those servers. So, we compute the number of possible initial global migration sets (line 4), and traverse all of them to find if there is any satisfactory solution (line 5). The logic of the judgement is that for each initial global migration set (line 6), we regroup its elements by clients (line 7) and sort them by the number of the client's non-coexistent tenants (line 8). Then we assign the destinations of executors from the first client to the last one by the following logic: for executors of each clientId, we first place as many of them as possible on client's exclusive servers (lines 10–16); if there is any executor left (line 17), we resort to empty servers (lines 18–24).

If some executors remain (line 25), we consider other servers if they are not fully occupied (lines 27 and 33), nor exclusive for other clients (lines 31 and 33). Besides, it should not host any executor of clientId (lines 32–33), because we regulate that each client pair can co-locate at most one executor of each to control the information leakage rate between tenants. Then for each VM of the current client (lines 34 and 35), we test whether those remaining servers (lines 37 and 38) can host it by deciding if this VM's owner client can co-resident with owners of all other VMs on servers (line 39). If any VM of the rest of clients cannot be hosted by any server (lines 43 and 46), we jump out the loop, and the "Error" is returned (line 50). Otherwise, we have successfully derived the satisfactory new placement (line 51). This algorithm achieves a much better scalability than CIRCUIT-SAT.

6 System implementation

In this section, we introduce the implementation of Driftor on OpenStack (2018), which is an open-source cloud computing platform to deploy the IaaS solutions. It supports three types of VM migration, of which we choose non-live migration (OpenStack,

2018) for Driftor because it is migrating suspended VMs. Our system is built according to the architecture shown in Fig. 2, each of which is:

1. Defense decision engine

Decisions for switch and migration are periodically made in this module, which is implemented in Nova-Scheduler at the controller node with roughly 800 lines of Python code. This is a natural implementation choice, since this engine needs a global view of all machines and VMs. We modify and expand the code to implement this engine as a service at the controller.

The defense decision engine works as follows: At the beginning of each defense interval, this engine queries OpenStack's database to obtain all necessary data, including mappings from the VM to host, and VMs belong to the same multi-executor structure. The latter information is initially stored by a synchronization engine, which creates a multi-executor structure for each newly coming VM. Both switch decision and migration decision are delivered to the defense operation engine.

2. Defense operation engine

This engine is responsible for enforcing defense decision from the defense decision engine. For switch operation, it first resumes the target VM through Nova-Compute. When it is fully operational, it sends a controlling message (consisting of switching order and destination) for the proxy process of current VM to switch the interaction with executors. Then it suspends an old active executor through Nova-Compute. For migration operation, it migrates a VM as desired. This module is implemented with 104 lines of Python code, and is placed at the controller node.

3. Proxy

Proxy is a single process responsible for forwarding communication data between users and serving executors. It is implemented by modifying Nginx (2018), which is stable and efficient. The modification consists of roughly 72 lines of C code. The additional logic attached to Nginx is mainly the communication with the defense operation engine. Upon receiving switch order from that engine, it uses an address included in the control message to modify the target to redirect users' requests, and responds to inform whether the switch is successful or not.

Since proxy would not process any privacy information, we consider it as side-channel free. So, we use common VMs to host the proxy code, and each such VM will launch a new process as proxy when a new VM is coming into the cloud. Each such process is bound with a public address, which enables the process to receive data originally sent to its corresponding VM. Similarly, when a VM leaves, its proxy process will be terminated.

4. Synchronization engine

This engine is responsible for synchronizing states of different executors of the same VM. For simplicity, current implementation places a shared database which stores as much data of the executor as possible. We adopt MariaDB (2018), which is the default database used in OpenStack. We demand that a service on the running executor will proactively and frequently store and update new data into the database. Similarly, when a new executor is activated, it will first proactively update its own data inside a VM according to the content of the database. We admit that current implementation might affect the type of service that a VM can provide, and leave expanding generality of Driftor as future work.

Another function of this engine is to create a multi-executor structure for a newly coming VM. This engine replicates this VM to create two new VMs that are totally the same as the coming one before launching it. Of course, a new proxy process is created, as stated above. The client ID, VM ID, and executor ID are all put into OpenStack's database for future use. Then a random executor is selected to be first activated. This module is implemented at a controller node with 283 lines of Python code.

7 Evaluation

We will answer the following questions here:

1. Is the switch decision algorithm scalable to clouds of large deployment? How about the migration decision algorithm?

2. How much does a single switch operation cost? How about migration of a single VM? What is the impact of these two operations on real-world cloud applications?

3. How resilient is Driftor's defense strategy against information leakage attacks?

4. To what extent is Driftor effective in defending against practical side channels?

7.1 Performance analysis

7.1.1 Scalability tests

For scalability test, we set up workloads as follows: All clients' VMs were assigned with the same resource consumption for convenience. Each client had two VMs, and each server, capable of hosting four VMs, was filled with occupancy of 50%. The number of clients was the same as that of servers, and the number was increasing to simulate different sizes of the problem. It should be noted that since a VM was changed to N_{EXE} executors in Driftor, the problem size was enlarged to N_{EXE} times, that is, N_{EXE} timed the number of clients and N_{EXE} timed the number of servers.

Fig. 5 shows the results of scalability tests for the switch algorithm with different values of N_{EXE}. To reduce the influence of randomness, each line is composed of average values of 100 samples. Fig. 5 shows that the cost of the switch algorithm with the same N_{EXE} almost remains unchanged with different problem sizes. It is reasonable, because this algorithm focuses on switching between executors, instead of servers or clients. Besides, by comparing cases with different N_{EXE}, we find that there is no difference about the costs; that is to say, N_{EXE} has little or even no impact on switch overhead. Nevertheless, even the highest cost in Fig. 5 is less than 90 ns, which is negligible in our defense. Therefore, the switch algorithm is quite scalable to large deployments.

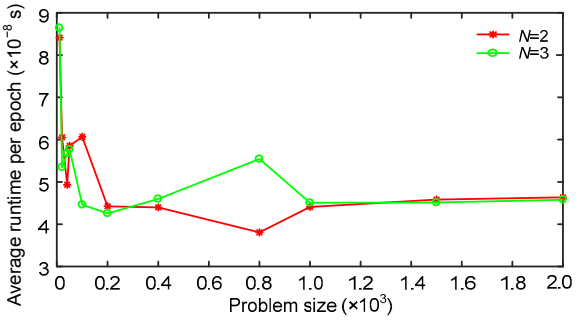

Fig. 5 Scalability test of the switch decision algorithm

To test the scalability of the migration algorithm, we need to compare greedy-like heuristic, CIRCUIT-SAT, and Nomad in <R, C> mode, which is the same as Driftor. Our SAT solver is Lingeling (2018), which does not have native binaries for Windows. Therefore, we use Cygwin (Rackspace, 2018). Since Nomad's scalability is possibly influenced by migration budget, we set it to 15%, which is the general setting of Nomad. Fig. 6 shows the results of the scalability tests for the migration algorithm. Similarly, each test has been conducted five times, of which the average value is adopted.

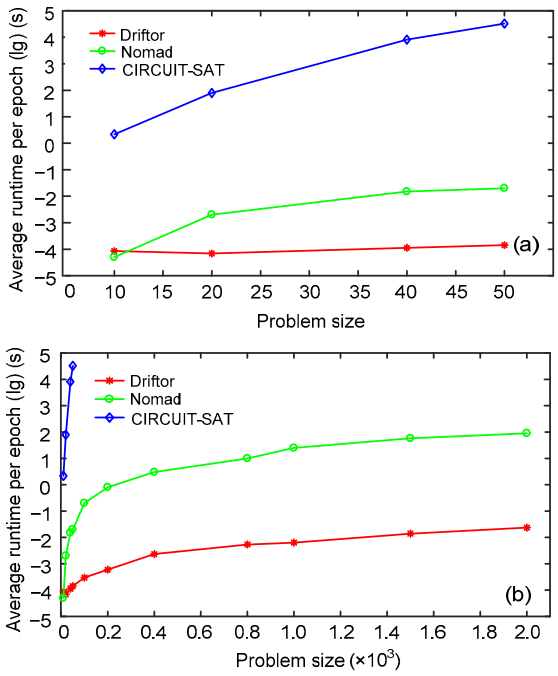

Fig. 6 Scalability test of the migration decision algorithm: (a) problem size ranging from 10 to 50; (b) problem size ranging from 10 to 2000

Fig. 6 shows that the proposed greedy-like heuristic quite outperforms Nomad, which is much faster than CIRCUIT-SAT. Since CIRCUIT-SAT is proved to be NP-complete, adopting it as the migration decision algorithm is obviously intractable. The perfect scalability of greedy-like heuristic lies in the fact that its performance is influenced mainly by the number of servers, but not the number of clients; therefore, its size is only linear to the problem size. Besides, the greedy-like algorithm gradually expands minimum migration sets until a satisfactory migration strategy is found. Therefore, since even the largest duration (0.0235 s in Fig. 6) is negligible, Driftor's migration algorithm has been proved to be quite scalable to large deployments.

7.1.2 Real defense overhead

To test the system in a real-world environment, Driftor was deployed in a local cloud which was organized with OpenStack Pike. There were 30 servers for Driftor tests, while 10 servers were enough for other tests, each of which was equipped with 2.00-GHz 64-bit Intel® Xeon® CPU E5-2683 v3 processor with 56 cores, 32-GB RAM, 1-TB disks, and four network interfaces with 1-Gb/s network speed. KVM was selected as a unified hypervisor which runs on Ubuntu 16.04 (Linux kernel v4.4.0).

1. Overhead of switch operation

Major overhead of switch operation lies in the time of activating a suspended VM. Therefore, we measure the duration of activation for three different instances, and present the results in the second column of Table 5, where each data is the average of 100 tests. Experimental results show that the overhead of switch operation is almost negligible. It should be noted that the switch operation will not induce any service downtime, because a new executor will be activated to provide service before the old executor has been suspended. New requests will be seamlessly redirected to the new executor.

Table 5 Time overhead for activation and migration of different types of instances

Type of instance	$T_{\mathrm{OP}}^{\mathrm{SW}}$ (s)	$T_{\mathrm{OP}}^{\mathrm{MIG}}$ (s)
Ubuntu-cloud (512-MB RAM, 1.5 GB)	1.04	2.12
Cirros (512-MB RAM, 132 MB)	0.77	0.41
Ubuntu (2048-MB RAM, 7 GB)	1.26	8.09

2. Overhead of migration operation

A suspended VM should be migrated by cold migration, so the underlying network turns out to be a bottleneck. We configure 1-Gb/s cable with 40-Gb/s optical fiber connecting transmitting nodes (in our case, between switches), and migrate suspended VMs between servers. The third column in Table 5 shows that the overhead of migration operation is sustainable.

So far, we have practically verified Driftor's defense ability against information leakage, especially fast side-channel attacks (Liu et al., 2015) by proving $t_{\mathrm{DE}}^{\mathrm{SW}}+T_{\mathrm{OP}}^{\mathrm{SW}}+t_{\mathrm{DE}}^{\mathrm{MIG}}+T_{\mathrm{OP}}^{\mathrm{MIG}}\le\delta\times T_{\mathrm{ATT}}$, which, however, cannot be satisfied in Moon et al. (2015).

3. Impacts on cloud-based applications

To evaluate the influence of deploying Driftor on real-world cloud workloads, we selected two representative cloud applications: web-server and MapReduce workloads.

(1) Web-server evaluation. Web service was simulated with WikiBench (2018), a web hosting benchmark that facilitated stress testing of systems used to host web applications. Different from conventional benchmarks which usually create a toy application to deal with synthetic workloads, WikiBench chose a real popular web application (MediaWiki) with real data that was actual database dumps of the Wikipedia website. In addition, it created real web traffic by replaying recorded history traffic to Wikipedia. We have downloaded the trace file since September 2007, and modified it to make an access rate of approximately 10 to 15 HTTP requests per second.

In this experiment, the capability of each server was fixed to four VMs, which was the same as that in the scalability tests. The difference was that there were four benign clients who had three server VMs running the same as the WikiBench backend, one proxy VM balancing the traffic between three servers, and one user VM sending HTTP GET requests following the pattern of WikiBench trace file. We run this experiment for 20 min (5 min per epoch) with three types of security configurations: (a) normal (without protection); (b) Nomad; (c) Driftor. Fig. 7 shows the distribution of these four clients' throughput (i.e., the number of completed requests per 10 s) over the entire run, which is in the form of box-and-whiskers plots with the 25$^{\mathrm{th}}$, 50$^{\mathrm{th}}$, and 75$^{\mathrm{th}}$ percentiles, as well as the minimum and 98$^{\mathrm{th}}$ percentile values.

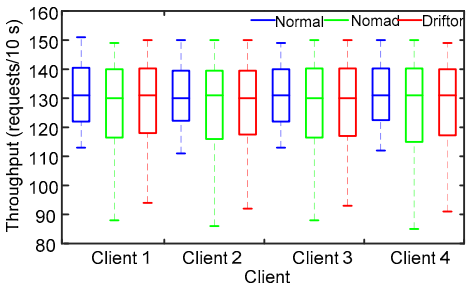

Fig. 7 Distribution of throughput for WikiBench workloads

Fig. 7 shows that the distribution is almost identical in all four cases, indicating that both Driftor

and Nomad have limited impact on real cloud applications. Another observation is that Driftor performs worse than in the case without any security enforcement, because the lower end of its throughput distribution is obviously smaller, so does Nomad. It can also be observed that Driftor outperforms Nomad a bit by comparing the lower end of their distributions. At last, it can be found that all clients suffer from similar performance degradation due to the security defense operations (mainly switch and migration), indicating the fact that Driftor is fair across different clients.

(2) MapReduce evaluation. To enforce the above conclusions, we set a different workload as sorting 800-MB data using Hadoop Terasort. The experimental setup was the same as that of webserver evaluation except that the epoch duration was reduced to 1 min and that each benign client was equipped with five VMs, of which one was master and four were slaves. This time we focused on the job completion time, which is collected and displayed in Fig. 8.

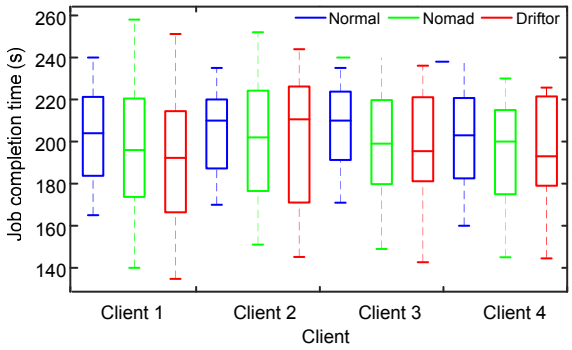

Fig. 8 Distribution of job completion time for Hadoop workloads with three different defense strategies

By comparing completion time with different security setups in Fig. 8, we can derive almost the same observations as that in webserver evaluation: (1) Normal setup outperforms Driftor, which achieves better results than Nomad; (2) Two security configurations bring about only a little influence on the cloud-based web application.

To sum up, both tests on real-world applications have demonstrated that Driftor induces little impact on cloud-based applications. This is an advantage due to the multi-executor structure of the VM in Driftor, because switch operation happens in a way that the service provider will be seamlessly switched to a newly activated executor when it is fully activated before the old executor has been suspended. Therefore, whether Driftor can be deployed in cloud depends on its security improvements.

7.2 Security analysis

7.2.1 Information leakage

To evaluate our defense against information leakage, we adopted the same setting as that of scalability tests in Section 7.1.1. Additional settings included the number of clients (and servers) being 20 and the migration budget being 15%. We compared the per-client leakages of Driftor, CIRCUIT-SAT, and Nomad in <R, C> mode. Fig. 9 presents the cumulative distribution function (CDF) of inter-client information leakage measured over five epochs with a randomly generated initial placement with three different defense strategies. Fig. 9 shows that Nomad slightly outperforms both Driftor and CIRCUIT-SAT, while the latter two achieve almost the same resilience against information leakage. This is because Driftor focuses on defense against fast side-channel attacks, which requires a very small value of $t_{\mathrm{DE}}^{\mathrm{MIG}}$. Therefore, optimization for overall information leakage is more or less sacrificed. Luckily, this disadvantage is limited, as shown in Fig. 9. With regard to the problem of fairness across different client pairs, we figure out that all defense strategies achieve similar satisfactory fairness since even the 95$^{\mathrm{th}}$ percentile of the distribution is relatively low (Fig. 9).

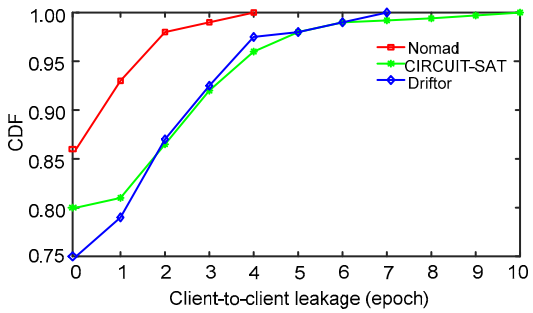

Fig. 9 Cumulative distribution function of client-pair information leakage under different defense strategies

7.2.2 Practical side-channel attacks

To verify Driftor's capability of defense against real-world side channels in cloud, we adopted the

same experimental settings as that in Section 7.1.2. Attackers were simulated by randomly selected clients by conducting LLC-based Prime+Probe (Liu et al., 2015) and a practical fast side channel which had a wide scope of application. Corresponding victims were decrypting a short file encrypted with a 3072-bit ElGamal public key, of which a 403-bit secret was the real target. GnuPG 1.4.18 was used in the decryption. The sliding-window exponentiation algorithm will be executed during decryption. This attack took about 12 min of online LLC-based Prime+Probe.

To reduce the impacts of multiple types of noise, each selected client can act as only one of roles, and all of his/her executors in their live cycles were carrying out attacks or performing vulnerable operations according to the roles. We randomly selected attackers and victims (Table 6). Numbers in the first row refer to attacker's client ID, while the numbers in the first column refer to victim's client ID. This experiment lasted about 60 min, which was divided into 30 epochs or defense intervals as 2 min per epoch; thus, Driftor allowed two clients to co-locate for at most five epochs. This time was selected to provide enough time for possible side channels. Each pair of attacker-victim corresponded to an entry consisting of two values. The left number is the (active) co-residency time of these two clients, and the right number is an indicator showing whether the corresponding attack is successful (that is, an effective side-channel attack has been completed).

Table 6 Results of Driftor's defense against real-world side-channel attacks

Victim's client ID	Attacker's client ID							
	1		10		14		17	
3	1	0	4	0	5	0	0	0
8	0	0	0	0	2	0	3	0
16	3	0	0	0	0	0	0	0
18	2	0	4	0	0	0	1	0

From Table 6, we can observe that there is no successful attack between any pair of clients in 30 epochs, because even the longest co-residency time that occurs between clients 3 and 10 is only five epochs, which equals 10 min. It is definitely less than the necessary time for a complete side channel. Another observation is that Table 6 resembles a sparse matrix with a lot of zeros, indicating a large margin of defense space. It shows that a lot of client pairs can be migrated to the same server to achieve co-residency even without single tenancy after 30 epochs. This benefit, that is, co-residency is slowly growing, is another advantage introduced by the multi-executor structure of VM and the regulation. To sum up, Driftor is capable of well defending against practical side channels.

7.2.3 Pure switch vs. Driftor

To demonstrate the necessity of executor migration in Driftor, we compared a strategy of pure switch with Driftor. We used totally the same experimental settings, including a same initial placement to make these two results comparable.

Table 7 shows the experimental results of tenant oriented pair-wise co-residency epochs with a pure switch strategy. Although most of the co-residency time is zero, there are two special client pairs, of which client pairs <3, 14> co-exists for seven epochs, and <10, 18> co-exists for six epochs. They are both larger than five epochs, which is the threshold time for an effective attack. Besides, other non-zero pairs are relatively large compared with the most of the entries in Table 6, indicating that migration can help distribute the co-residency time over different client pairs. Therefore, migration is quite necessary to complement the defense ability of the executor switch.

Table 7 Results of pure switch defense against real-world side-channel attacks

Victim's client ID	Attacker's client ID							
	1		10		14		17	
3	0	0	4	0	7	0	0	0
8	0	0	0	0	0	0	4	0
16	3	0	0	0	0	0	0	0
18	3	0	6	0	0	0	0	0

8 Conclusions and future work

In this study, we have proposed Driftor, a side-channel resistant cloud system which mitigates information leakage attacks by creating a multi-executor structure for each VM and switching and migrating those executors. First of all, we have set up a practical unified adversary model, which focuses on the concept of effective side-channel attacks. To

defend against fast side channels that are beyond the capability of current migration-based approaches, we have proposed a new cloud system where we set up a multi-executor structure for each VM and switch the only active state among executors on different servers to simulate the defensive effect which is similar to that of migration. Besides, real migration of executors was enabled to enlarge the scope of switch, thus increasing the resilience against side channels. Instead of solving this satisfiability problem of migration with an intractable CIRCUIT-SAT, we have proposed a greedy-like algorithm which searches viable solutions by gradually expanding the size of solutions from limited has-to-migrate executors. Experimental results showed that Driftor can not only defend against practical fast side-channel attacks, but also introduce reasonable impact on real-world cloud applications.

Driftor has the following two limitations: (1) The number of necessary servers is proportional to that of executors per VM, thus making defense cost multiplied; (2) Synchronization between executors becomes difficult if there are too many running states generated during a defense interval. For the first limitation, we have observed that a suspended VM consumes only limited disk resource, which is the least influential factor when defining a server's capability to host VMs. In other words, a suspended VM may not be counted as a resource-consuming entity which costs the server's slot. Therefore, we can place many more VMs on a server than its defined capability; thus, the defense cost might be effectively reduced. The second limitation is quite tough since it exists even in synchronizing states between two running instances, let alone the VM being suspended in our case. Maybe we can resort to a new computing entity whose running states can be sliced to fine-grained granularity, so that each such slice can be used for an immediate state recovery, or the suspended VM can be periodically activated to keep its states up-to-date, thus reducing the time to keep synchronization.

References

Almeida JB, Barbosa M, Barthe G, et al., 2016. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. 23rd Int Conf on Fast Software Encryption, p.163-184. https://doi.org/10.1007/978-3-662-52993-5_9

Amazon EC2, 2018. Amazon EC2. https://amazonaws-china.com/cn/events/ec2/?sc_channel=ps&sc_campaign=inbounddg&sc_publisher=baidu&sc_detail={ec2%20amazon}&sc_country=cn&sc_geo=chna&sc_category=ec2&sc_segment={AWS%20EC2|brand}&sc_outcome=field&trkCampaign=inbounddg_ec2&trk=Baidu|AWS%20EC2|brand|ec2%20amazon&audience=205636 [Accessed on Aug. 4, 2018].

Bosman E, Razavi K, Bos H, et al., 2016. Dedup est Machina: memory deduplication as an advanced exploitation vector. IEEE Symp on Security and Privacy, p.987-1004. https://doi.org/10.1109/SP.2016.63

Douceur JR, 2002. The Sybil attack. 1st Int Workshop on Peer-to-Peer Systems, p.251-260. https://doi.org/10.1007/3-540-45748-8_24

Ezhilchelvan PD, Mitrani I, 2017. Evaluating the probability of malicious co-residency in public clouds. *IEEE Trans Cloud Comput*, 5(3):420-427. https://doi.org/10.1109/TCC.2015.2451633

Feng DG, Zhang M, Zhang Y, et al., 2011. Study on cloud computing security. *J Softw*, 22(1):71-83 (in Chinese). https://doi.org/10.3724/SP.J.1001.2011.03958

Garey MR, Johnson DS, 1979. Computers and intractability: a guide to the theory of NP-completeness. W.H. Freeman & Co., New York, NY, USA, p.498-500. https://doi.org/10.2307/2273574

Gruss D, Maurice C, Wagner K, et al., 2016. Flush+Flush: a fast and stealthy cache attack. Int Conf on Detection of Intrusions and Malware, and Vulnerability Assessment, p.279-299. https://doi.org/10.1007/978-3-319-40667-1_14

Han Y, Alpcan T, Chan J, et al., 2016. A game theoretical approach to defend against co-resident attacks in cloud computing: preventing co-residence using semi-supervised learning. *IEEE Trans Inform Forens Secur*, 11(3):556-570. https://doi.org/10.1109/TIFS.2015.2505680

Han Y, Chan J, Alpcan T, et al., 2017. Using virtual machine allocation policies to defend against co-resident attacks in cloud computing. *IEEE Trans Depend Secur Comput*, 14(1):95-108. https://doi.org/10.1109/TDSC.2015.2429132

Hu HC, Wu JX, Wang ZP, et al., 2018. Mimic defense: a designed-in cybersecurity defense framework. *IET Inform Secur*, 12(3):226-237. https://doi.org/10.1049/iet-ifs.2017.0086

Irazoqui G, Eisenbarth T, Sunar B, 2015. S$A: a shared cache attack that works across cores and defies VM sandboxing -- and its application to AES. IEEE Symp on Security and Privacy, p.591-604. https://doi.org/10.1109/SP.2015.42

Kämäräinen T, Shan YQ, Siekkinen M, et al., 2015. Virtual machines vs. containers in cloud gaming systems. Int Workshop on Network and Systems Support for Games,

p.1-6. https://doi.org/10.1109/NetGames.2015.7382987

Kim T, Peinado M, Mainar-Ruiz G, 2012. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. 21st USENIX Conf on Security Symp, p.1-11.

Kwiat L, Kamhoua CA, Kwiat KA, et al., 2015. Security-aware virtual machine allocation in the cloud: a game theoretic approach. Proc IEEE 8th Int Conf on Cloud Computing, p.556-563. https://doi.org/10.1109/CLOUD.2015.80

Li H, Ota K, Dong MX, et al., 2017. Multimedia processing pricing strategy in GPU-accelerated cloud computing. *IEEE Trans Cloud Comput*, p.1. https://doi.org/10.1109/TCC.2017.2672554

Li H, Ota K, Dong MX, 2018. Virtual network recognition and optimization in SDN-enabled cloud environment. *IEEE Trans Cloud Comput*, p.1. https://doi.org/10.1109/TCC.2018.2871118

Li P, Gao DB, Reiter MK, 2014. StopWatch: a cloud architecture for timing channel mitigation. *ACM Trans Inform Syst Secur*, 17(2):28. https://doi.org/10.1145/2670940

Lingeling, 2018. Lingeling, Plingeling and Treengeling. http://fmv.jku.at/lingeling/ [Accessed on Aug. 4, 2018].

Liu FF, Lee RB, 2014. Random fill cache architecture. 47th Annual IEEE/ACM Int Symp on Microarchitecture, p.203-215. https://doi.org/10.1109/MICRO.2014.28

Liu FF, Yarom Y, Ge Q, et al., 2015. Last-level cache side-channel attacks are practical. IEEE Symp on Security and Privacy, p.605-622. https://doi.org/10.1109/SP.2015.43

MariaDB, 2018. The MariaDB Foundation–Supporting Continuity and Open Collaboration in the MariaDB Ecosystem. https://mariadb.org [Accessed on Aug. 4, 2018].

Microsoft Azure, 2018. Microsoft Azure. https://azure.microsoft.com/zh-cn/ [Accessed on Aug. 4, 2018].

Migrate Instances, 2018. Migrate Instances. https://docs.openstack.org/nova/rocky/admin/migration.html [Accessed on Aug. 4, 2018].

Moon SJ, Sekar V, Reiter MK, 2015. Nomad: mitigating arbitrary cloud side channels via provider-assisted migration. 22nd ACM SIGSAC Conf on Computer and Communications Security, p.1595-1606. https://doi.org/10.1145/2810103.2813706

Moscibroda T, Mutlu O, 2007. Memory performance attacks: denial of memory service in multi-core systems. Proc 16th USENIX Security Symp, Article 18.

Nginx, 2018. Nginx News. http://nginx.org/ [Accessed on Aug. 4, 2018].

OpenStack, 2018. The Open Infrastructure Summit CFP is Now Open! https://www.openstack.org/ [Accessed on Aug. 4, 2018].

Pattuk E, Kantarcioglu M, Lin ZQ, et al., 2014. Preventing cryptographic key leakage in cloud virtual machines. Proc 23rd USENIX Conf on Security Symp, p.703-718.

Rackspace, 2018. Transform the Way You Do Business. https://www.rackspace.com/ [Accessed on Aug. 4, 2018].

Raj H, Nathuji R, Singh A, et al., 2009. Resource management for isolation enhanced cloud services. Proc ACM Workshop on Cloud Computing Security, p.77-84. https://doi.org/10.1145/1655008.1655019

Ristenpart T, Tromer E, Shacham H, et al., 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. Proc 16th ACM Conf on Computer and Communications Security, p.199-212. https://doi.org/10.1145/1653662.1653687

Shyamasundar RK, 1996. Introduction to algorithms. *Resonance*, 1(9):14-24. https://doi.org/10.1007/BF02837777

Thompson M, Evans N, Kisekka V, 2014. Multiple OS rotational environment an implemented moving target defense. 7th Int Symp on Resilient Control Systems, p.1-6. https://doi.org/10.1109/ISRCS.2014.6900086

Varadarajan V, Ristenpart T, Swift M, 2014. Scheduler-based defenses against cross-VM side-channels. Proc 23rd USENIX Conf on Security Symp, p.687-702.

Vattikonda BC, Das S, Shacham H, 2011. Eliminating fine grained timers in Xen. 3rd ACM Workshop on Cloud Computing Security Workshop, p.41-46. https://doi.org/10.1145/2046660.2046671

Wang HX, Li F, Chen SQ, 2016. Towards cost-effective moving target defense against DDoS and covert channel attacks. Proc ACM Workshop on Moving Target Defense, p.15-25. https://doi.org/10.1145/2995272.2995281

Wang ZH, Lee RB, 2007. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Comput Arch News*, 35(2):494-505. https://doi.org/10.1145/1273440.1250723

Wang ZH, Lee RB, 2008. A novel cache architecture with enhanced performance and security. 41st IEEE/ACM Int Symp on Microarchitecture, p.83-93. https://doi.org/10.1109/MICRO.2008.4771781

WikiBench, 2018. WikiBench. http://www.wikibench.eu/ [Accessed on Aug. 4, 2018].

Wu J, Dong MX, Ota K, et al., 2017. FCSS: fog computing based content-aware filtering for security services in information centric social networks. *IEEE Trans Emerg Top Comput*, p.1. https://doi.org/10.1109/TETC.2017.2747158

Wu J, Dong MX, Ota K, et al., 2018. Big data analysis-based secure cluster management for optimized control plane in software-defined networks. *IEEE Trans Netw Serv Manag*, 15(1):27-38. https://doi.org/10.1109/TNSM.2018.2799000

Wu JX, 2016. Research on cyber mimic defense. *J Cyber Secur*, 1(4):1-10 (in Chinese). https://doi.org/10.19363/

j.cnki.cn10-1380/tn.2016.04.001

Yarom Y, Falkner K, 2014. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. Proc 23rd USENIX Conf on Security Symp, p.719-732.

Zhang YL, Li M, Bai K, et al., 2012. Incentive compatible moving target defense against VM-colocation attacks in clouds. In: Gritzalis D, Furnell S, Theoharidou M (Eds.), Information Security and Privacy Research. Springer Berlin Heidelberg, Germany, p.388-399. https://doi.org/10.1007/978-3-642-30436-1_32

Zhang YQ, Reiter MK, 2013. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. Proc ACM SIGSAC Conf on Computer & Communications Security, p.827-838. https://doi.org/10.1145/2508859.2516741

Zhang YQ, Juels A, Reiter MK, et al., 2012. Cross-VM side channels and their use to extract private keys. Proc ACM Conf on Computer and Communications Security, p.305-316. https://doi.org/10.1145/2382196.2382230

Zhang YQ, Juels A, Reiter MK, et al., 2014. Cross-tenant side-channel attacks in PaaS clouds. Proc ACM SIGSAC Conf on Computer and Communications Security, p.990-1003. https://doi.org/10.1145/2660267.2660356