A Stealth Migration Approach to Moving Target Defense in Cloud Computing

Saikat Das(✉), Ahmed M. Mahfouz, and Sajjan Shiva

Department of Computer Science, The University of Memphis, Memphis, TN 38152, USA
{sdas1, amahfouz, sshiva}@memphis.edu

Abstract. A stealth migration protocol is proposed in this paper that obfuscates the virtual machine (VM) migration from intruders and enhances the security of the MTD process. Starting by encrypting the VM data and generating a secret key that is split along with the encrypted data into small chunks. Then the fragments are transmitted through intermediate VMs on the way to the destination VM. As a result, the chances of an intruder detecting the VM migration is reduced. The migration traffic is maintained close to normal traffic by adjusting the chunk size, thereby avoiding the attention of the intruder. Finally, the normal and migration traffic patterns are analyzed with the proposed protocol.

Keywords: Stealth migration · Cloud computing · Moving Target Defense · Secure MTD · Live migration

1 Introduction

Traditional cyber networks tend to be static, giving an attacker plenty of time to study them. During this time, the attacker can generate footprints, determine vulnerabilities, and decide when, how, and where to attack the network. Besides, once the attacker gains a privilege, he can maintain it for an extended period without being discovered by the system administrators. Various detection-based security approaches have emerged, but they struggle to detect attacks with accuracy and precision [1]. A new promising security technique, Moving Target Defense (MTD), has been adopted as a solution to overcome the challenge of network stat`ic nature. The idea came from the tactic of the battlefield where occasionally fighters change their positions and resources so that their enemies get confused and have difficulty attacking them. Likewise, in computer systems, moving the target resources could increase the complexity and uncertainty for attackers. In cloud computing, the virtualization techniques solely rely on distributing virtual machines (VMs) across different host machines around the world. Live Migration of VM instances from one physical device to another can be treated as MTD since the targeted host offloads its resident VM instances to a different host [12]. However, the process of VM live migration itself is not secure enough and is vulnerable to various active and passive attacks [3]. But, if the live migration is performed in a trusted and secure environment [8], it could be considered as a guaranteed MTD strategy against several types of attacks. In this paper, a stealth migration protocol is proposed that obfuscates the VM migration from the intruder and enhances the security of MTD.

The rest of the paper is organized as follows: In Sect. 2, an overview of MTD along with Live Migration is provided. Section 3 summarizes the related work. In Sect. 4, the threat model is discussed. The Stealth Migration Protocol is proposed in Sect. 5 to secure the Live Migration process and evaluate it by analyzing the normal vs. migration traffic patterns in Sect. 6. Finally, Sect. 7 concludes the paper and provides direction for future research.

2 Moving Target Defense

2.1 What is MTD?

MTD is the concept of controlling change across multiple system dimensions to increase uncertainty and apparent complexity for attackers, reduce their window of opportunity, and increase the costs of their probing and attack efforts [2]. MTD mainly aims to rebalance cyber-security by integrating dynamism, randomization, and diversification techniques in computer networks to hide the properties that an attacker can exploit to compromise the system. By changing the status of the system to be more dynamic and less deterministic, MTD boosts the system immunity to different attacks and increases the workload for the adversary.

MTD utilizes a wide range of systems security techniques and strategies [10, 11] that include Software Diversification, Runtime Diversification, Communication Diversification, and Dynamic Platform Techniques [12]. All these techniques intend to mutate the target system, making it unfamiliar to the attacker and force him to learn about the target repeatedly and newly, thus decreasing the probability of discovery and making the attacks costlier or unachievable.

2.2 Why is MTD Insecure?

MTD normally doesn’t happen across fully secure networks. It is probable that the MTD traffic movement paths span across multiple networks and significant geographic distances [4] that allow attackers to identify the traceroute and discover the network footprint. Moreover, a compromised cloud system employing MTD can facilitate untrusted access to the moving VMs [9]. The ability to view or modify data associated with MTD or influence the movement services on the source and destination hosts raises several important security questions [5].

2.3 Live Migration

Live Migration of VM in cloud computing is the process of moving the VM from one physical host to another without affecting the service availability to users. It requires the transfer of the complete state of a VM from a host to another that comprises all the resources the VM uses in the source device. The resources include volatile storage, permanent storage, the internal state of the virtual CPUs, and connected devices (e.g., LAN Cards) [12]. Since the network-attached storage provides permanent room in the data center, it is not required to move the permanent storage during the VM migration process. The internal states of the virtual CPUs are usually only a few kilobytes of data and does not take considerable amount of time to be transferred. More extended periods are required to move the volatile memory contents which affect the performance of the live migration process and hence more attention is given to improve the transfer of volatile memory from the source to the destination [12].

3 Related Work

Several solutions have been proposed and implemented to secure the MTD approach over different live migration protocols. Some of these solutions are listed below.

Isolating Migration Network: This approach separates the virtual LAN consisting of the source and the destination hosts from the migration traffic of other networks, thus reducing the risk of exposure of migration to the whole network.

Network Security Engine Hypervisor (NSE-H): Xianqin et al. [13] proposed a secure VM migration framework that is based on hypervisors included with NSE. The framework provides an extension to the hypervisor by allowing the functionality of firewall and IDS/IPS to secure the migration from external attacks. It can also check the network for any intrusion and generate an alarm in case of any intrusion detection. The drawback of this framework is that the migration data may not remain unmodified during the transmission process because the data is not hashed or encrypted.

Secure VM-vTPM Migration Protocol: Berger et al. [14] classify the requirements and propose a new design of a virtual Trusted Platform Module (vTPM). The module consists of various steps starting from authentication, attestation, and data transfer stage. It also checks the integrity of the source and the destination. Only after verifying the integrity, the source VM starts the transfer to the destination VM. The file sent by the source VM is encrypted at vTPM and transferred to the destination VM. After completion of the transfer, the data at the vTPM is deleted. In the improved version, the source VM and target VM first authenticate each other to establish the trusted channel and then verify the integrity. Both the source and the destination negotiate keys with each other using DH key exchange algorithm. After the channel is established, the transfer begins. A specific mechanism for detecting and reporting suspicious activities is missing in this research.

SSH Tunnel: SSH tunnel is established between the proxies for secure movement. The proxy server at the source and the destination cloud communicate with each other and hide the details of both source and destination VMs [6]. In this research, an attacker can still examine the payloads in the flow by applying algorithms that are based on statistical characteristics to perform traffic analysis.

IPSec Tunnel: IPSec tunnel protects the data flow at server-to-server levels or from the edge router to edge router [15]. If MTD were done through IPSec tunnel, IP packets would be encrypted making it challenging to sniff data and trace. However, this approach slows down the migration process resulting in increased live migration downtime.

The above approaches have significant impact on securing the migration process. However, their overhead and differing attacking intention of intruders do not converge to a comprehensive solution. A protocol is proposed in this paper whose primary goal is to secure the migration process. Besides, the aim of this paper is to minimize the overhead and migration downtime.

4 Threat Model

In the threat model, an attacker is considered who is using Internet route hijacking to perform a man-in-the-middle attack on routes over the network to capture appropriate data from the traffic. Such an attack is emerging and represents a real security threat [16, 17]. The pre-condition to perform this attack is that the attacker can recognize the exact movement of the VM on the cloud network. Even with applying security measures like using encryption, secure tunnels or onion routing, an attacker can still use traffic characteristics to perform traffic analysis and detect the VM movement [7]. The attacker can view critical information such as the traffic speed, size, duration, and the involved endpoint hosts [18, 19] and launch his attack on the movement flow. Once the attacker detects the VM movement and the destination address, he can continue to attack the VM. This attack type can be reduced by utilizing secret sharing and moving VMs using multiple chunks.

5 Proposed Solution

Here, the new stealth migration technique is proposed that secures the live migration process at both the application and the live migration levels. First, a brief introduction on secret sharing encryption technology is given that has been used in the protocol, followed by the description of the protocol design.

5.1 Secret Sharing

Secret sharing is the method of dividing a secret among a group of participants where each participant allocates one secret share. The secret can only be reconstructed only when a certain number of shares are combined. Individual shares are of no use by themselves.

Shamir's secret sharing [10] is a popular way of splitting the secret. It divides the secret S (for example, the combination to a secure lockable box) into n pieces of data: S_1, S_2, …. S_n in such a way that the knowledge of any k-1 or fewer S_i pieces leaves S completely undetermined, in the sense that the possible values for S seem as likely as with knowledge of 0 pieces.

(i) The knowledge of any k or more secret pieces (S_i) makes the secret (S) computable. That is, the construction of complete secret S can be done from any combination of k pieces of data [10].

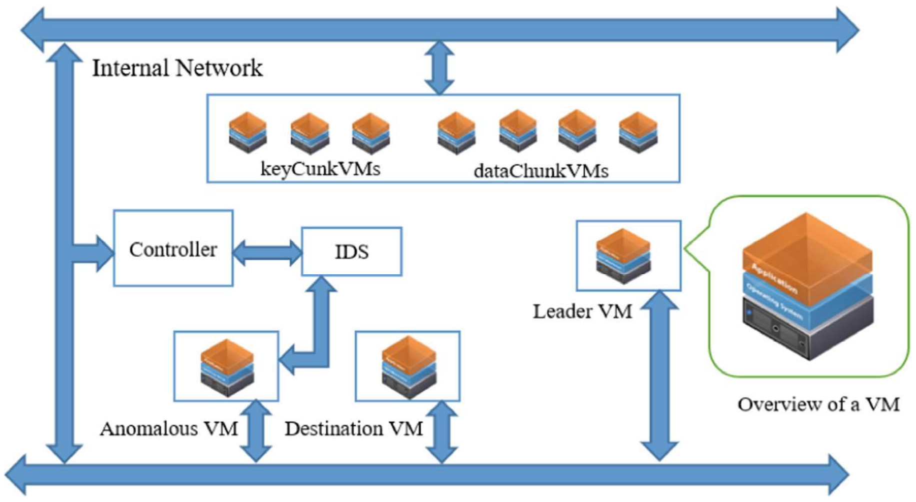


Fig. 1. A top-level view of stealth migration protocol

(ii) The knowledge of any k-1 or fewer Si pieces leaves S completely undetermined, in the sense that the possible values for S seem as likely as with knowledge of 0 pieces. Said another way, the secret S cannot be reconstructed with fewer than k pieces [10].

This scheme is called (k, n) threshold scheme. When $[k = n]$, then every piece of the original secret S is required to reconstruct the secret.

Secret sharing encoding scheme is used in this stealth migration approach to split and hide the key information over the network. The key will be distributed by *k/n* pieces where *k* is the minimum number of shares to construct the secret.

5.2 Stealth Migration Protocol

Our stealth migration protocol is now discussed along with different algorithms used to migrate the VM using our stealth migration protocol.

A top-level view of the stealth migration protocol is shown in Fig. 1. The blue lines indicate the internal network by which all the five components communicate with each other. A virtual machine is also shown on the popup in detail. The stealth migration protocol is designed with five major components:

(i) Controller: Controller of the virtual machine manager (VMM) that controls each component.
(ii) $VM_{anomalous}$: The virtual machine that is identified as affected/anomalous.
(iii) VM_{leader}: Controller marks a VM as leader who leads all intermediate VMs.
(iv) $VM_{available}$: The virtual machines that accept the data chunks and send to VM_{leader}.
(v) $VM_{destination}$: The virtual machine where the affected VM's data will be transferred finally.

In earlier research [20, 21], collaborative runtime monitors as IDS were created. They were attached to each running VM and continuously monitor VMs activity. As soon as the VM acts abnormally, IDS raises an alarm and sends the necessary VM information to the controller for further analysis. Figure 2 shows the workflow of the stealth migration protocol.

Each of the components and their tasks are briefly discussed next.

The Controller Component: The controller is the heart of the VMM, which has the full control over any component of this stealth migration protocol. Typically, it receives notifications (signals) from VMs, intrusion detection systems (IDS) and starts VM migration and manages all components to finish the migration. After successful completion of a migration, it maintains the connectivity of all components on the network. The controller works according to the following algorithm:

Algorithm (1): Maintain the connectivity among VMs and control over all VMs to operate the VM migration.

Input: Notifications (From IDS, VMs)

Output: Controlling signal, assigning task

```
while(true):
    notification = push_notification_from_IDS()
    if notification = anomalous
         normalTraffic = normal_traffic_size()
         VMsize = check_size($VM_{anomalous}$)
         VMstate = state_of_VM($VM_{anolamous}$)
         chunkSize = calculate_chunk_size()
         availableVMs = check_available_VMs()
         VMs = choose random 9VM from availableVMs
         $VM_{destination}$ = VMs[random(1-9)]
         dataChunkVMs, keyChunkVMs = split_VMs()
         $VM_{leader}$ = random(from dataChunkVMs)
         send_instruction_to_$VM_{destination}$()
         secretKey = cerate_secret_key(256 bit)
         send_instruction_to_$VM_{anomalous}$()
         send_instruction_to_$VM_{leader}$()
         for each VM in VMs
    send_instruction_for_all_VMs()
         if notification (found from VM)
    make_available(VM)
         if notification (found from $VM_{destination}$)
    make_$VM_{anomalous}$_honeypot()
    disconnect_$VM_{anomalous}$_from_network()
    connect_$VM_{destination}$_to_network()
    notify_IDS()
end while
```

The utility and usage of some of the methods used in the above algorithm are discussed below.

normal_traffic_size(): By using this method, the controller gets the normal traffic size from the IDS. The IDS provides the normal traffic size (packet transfer per second) to the controller by analyzing its normal network traffic activity.

check_size($VM_{anomalous}$): Uses the IDS to determine the actual size of the affected virtual machine including OS, applications, dirty pages, etc.

state_of_VM($VM_{anolamous}$): IDS provides the current state of the anomalous VM to the controller. State can be 'STARTING', 'RUNNING', 'STOPPED', 'SUSPENDED', etc.

calculate_chunk_size(VMsize, 5n, normalTraffic): Calculates the chunk size by using the normal Traffic and VM_{size}. Since five intermediate VMs are used to migrate the data, the method determines and adjusts the chunk size by multiples of five to maintain the migration traffic similar to the normal traffic. For an example, the normal traffic is 6 MB/s and the VMsize is 500 MB. If the 100 MB data are transferred over the network at a single instance to five different VMs, an intruder can easily detect it as a migration traffic. So, if the chunk size can be maintained at approximately 6 MB, it can easily be transferred to intermediate VMs that eventually retains the normal traffic pattern. The purpose of maintaining the chunk size is to obfuscate the migration from intruder.

check_available_VMs(): Returns all VMs that are currently available.

split_VMs(): Here, nine VMs are used to operate our whole migration process. This method splits two types of VM lists from the available VM list except for the one that has already been marked as the destination VM where the affected VM's data will be moved eventually. The two lists are dataChunkVMs (where data segment will be transferred) and keyChunkVMs (where key segment will be transferred).

send_instruction_to_$VM_{destination}$(): Sends an instruction to destination VM address. The instruction instructs the VM to open a port and listen to the data coming from the VM_{leader} address.

send_instruction_to_$VM_{anomalous}$(VMs address, $VM_{destination}$ address, chunkSize, secretKey): This method sends chunkSize, secretKey, $VM_{destination}$ address, dataChunkVM lists (where to send data chunk), keyChunkVM lists (where to send the key chunk) as parameters to $VM_{anomalous}$ address. It also sends other instructions that will be discussed later in $VM_{anomalous}$ component section.

send_instruction_to_VM_{leader}($VM_{anomalous}$ address, dataChunkVMs, keyChunkVMs address): Sends the $VM_{anomalous}$ address, dataChunkVMs list, keyChunkVMs list as parameter to VM_{leader} address.

send_instruction_for_all_VMs($VM_{anomalous}$ address, VM_{leader} address): Sends the $VM_{anomalous}$ address, VM_{leader} address as a parameter to all VMs. The instruction commands to transmit the data that has already been received from the anomalous VM to the leader VM address.

make_available(VM): This method marks the VM as available for further use.

disconnect_$VM_{anomalous}$_from_network(): Disconnects the anomalous VM from the network.

make_VM$_{anomalous}$_honeypot(): Marks the anomalous VM as a honeypot to play further games with the intruder.

connect_VM$_{destination}$_to_network(): Connects the destination VM to the real network.

Notify_IDS(): Finally, the controller notifies the IDS (from where it raised an anomalous activity alarm). The notification signal consists of monitoring the same VM or monitoring a new assigned VM.

The VM$_{\mathbf{anomalous}}$ Component: In this component, an anomalous VM receives the instruction from the controller and performs the tasks that have been assigned to it by the controller. The tasks are conducted in accordance with the following algorithm:

Algorithm (2): Data wrap up, encryption, split data and key, send.

Input: VM$_{destination}$, VM8Addresses[], chunkSize, secretKey.

Output: Send data chunks to 8 VMs, notify controller.

```
while(true)
       if instruction (found from the controller)
       wrapData = wrap_data(VMData, VM_destination address)
       encryptedData = encrypt_data(wrapData, secretKey)
       dataChunks[dataSegment]=split_data_chunks()
       dataChunks[keySegment]=split_key_chunks()
       for VMAddress in VM8Addresses
      send_to_each_VMAddress(dataChunks[i])
   notify_controller(sending successful)
end while
```

The goal of this component is to send the data chunks and key chunks to the intermediate VMs in a normal traffic fashion. To do so, the VM wraps all of its VM data and merges it with the destination VMaddress. VM wraps data into the host machine by executing secure instructions provided by the controller. The merged data is then encrypted by using the secret key that is provided by the controller. After the encryption, the encrypted file is split according to the chunk size that is given by the controller. Finally, data chunks are sent one by one to the intermediate VMs listed on dataChunk VM list. The secret key is then split by *2/3* using the secret sharing scheme where 2 is the minimum share to construct the key and 3 is the total number of shares. Some of the methods used in the VM$_{anomalous}$ component are discussed below:

wrap_data(VMData, VM$_{destination}$ address): The affected VM uses this method to wrap up its whole content and adds the destination VM address with it to make a new dataset.

encrypt_data(wrapData, secretKey): Using this method, affected VM encrypts the whole dataset with the secret key given by controller.

split_data_chunks(chunkSize, encryptedData): After the encryption from the encrypt_data() method, affected VM splits the encrypted data into small chunks using the chunkSize. Since five VMs are used to migrate the affected VM intermediately, this method splits the whole file in such a way that it can be equally distributed to five VMs without affecting the normal traffic flow.

split_key_chunks(secretSharing scheme 2/3): This method splits the secret key into 2/3 size, where 2 is the minimum share to construct the complete secret and 3 is total share size.

send_to_each_VMAdress(dataChunks[i]): Affected VM sends each data chunk and key chunk to the specified VM addresses using this method. The method has the list of datachunkVMs, keychunkVMs and transfers the data chunks and key chunks to different VM lists.

notify_controller(sending successfully): Finally, using this method, affected VM sends the successful task completion signal to the controller.

The VMdatachunks and VMkeychunks Component: VMdataChunks and VMkeyChunks components are the intermediate state of this stealth migration protocol. The affected VM transfers the chunks (dataChunks, keyChunks) to the VMs that are listed to the dataChunk or keyChunkVM list as an intermediate storage. The basic difference between those two components is marking the chunked segment either by data or key. All VMs, receive chunks (dataChunk or keyChunk), mark them as data segment or key segment and send them to the VM_{leader} address. The following algorithm does the tasks:

Algorithm (3): Receive datachunk, mark it either as data segment or key segment and send it to the VM_{leader} address.

Input: $_{VManomalous}$ adress, dataChunks, VM_{leader} address

Output: Send data chunks to VM_{leader} address.

```
while(true)
        if instruction (found from the controller)
              open_port()
              listen(from VM_anomalous address)
        if recv()
             // mark segment for VMdatachunks component
              mark_as_data_segment(dataChunks)
             // mark segment for VMkeychunks component
              mark_as_key_segment(dataChunks)
              send_to_VM_leader()
              wipe(whole memory)
              make_available(make its available)
                 notify_controller(availability)
end while
```

The methods used in the above algorithm are described below:

mark_as_data_segment(dataChunks): In this method, the VMs mark their data chunks as a data segment which is necessary to do for the further computation on VM_{leader} section. It helps the leader VM to identify the data chunks and the key chunks to construct the data and the key.
mark_as_key_segment(dataChunks): Here, the VMs mark their data chunks as a key segment, then transfer the data to the leader VM and after successfully transmitting the data, the VM wipes itself.
send_to_VM_{leader}(dataChunks, VM_{leader} address): By using this method, VMs send their received data chunks to the leader VM.
wipe(full memory): The VMs wipe their whole memory along with the data chunks in preparation for the future migration operation.
make_available(makeavailable)&notify_controller(availability): Each VM marks itself available and notifies the controller about its availability by using these two methods.

The VM_{leader} Component: In this component, leader VM receives all data and key chunks and constructs the secret key from the key chunks by using secret sharing encoding scheme. After successfully retrieving the secret key, it merges all data chunks and decrypts the complete data. After decoding the data, this component reveals the destination address for the data. Those tasks utilize the following algorithm:

Algorithm (4): Construct secret key, reveal destination VM address after decrypting the data and send data to the destination VM.

Input: dataChunks[], VMAddresses[]

Output: send data to $VM_{destination}$ address

```
while(true)
      if instruction (found from the controller)
             open_ports (for 7 VM address)
             listen_from_ports(VMAddresses[])
             if rccv()
      dataSegments[],keySegments[]=separate_data_or_ke
      y_chunks(dataChunks)
      secretKey = construct_key(keySegment[])
      data = construct_data(dataSegments[])
      VMdestinationAddress,VmData = decrypt_data()
      send_data(VMData, VMdestinationAddress)
      wipe(whole memory)
      make_available(make its available)
           notify_controller(availability)
end while
```

The details of the methods used in the above algorithm are discussed below:

dataSegments[], keySegments[] = separate_data_or_key_chunks(dataChunks): In this method, the leader VM separates the data segments and key segments from chunks that is received from the open port. This method differentiates the data chunks and key chunks as the previous component. Every chunk was marked either data or key when it was sent from intermediate VMs.

construct_key(keySegment[]): Secret sharing encoding scheme is used in this method to construct the key from the key segments. Since 2/3 encoding scheme is used, only 2 or more shares can reveal the key. Less than two shares never reveal the key according to the secret sharing encoding scheme. Shamir's secret sharing [10] scheme is used to construct the key.

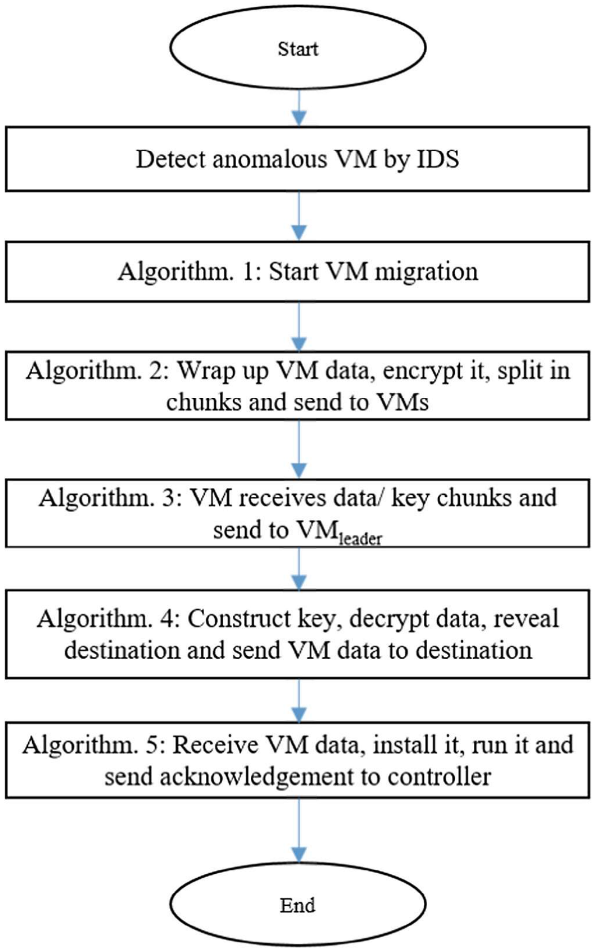

Start
Detect anomalous VM by IDS
Algorithm. 1: Start VM migration
Algorithm. 2: Wrap up VM data, encrypt it, split in chunks and send to VMs
Algorithm. 3: VM receives data/ key chunks and send to VM_{leader}
Algorithm. 4: Construct key, decrypt data, reveal destination and send VM data to destination
Algorithm. 5: Receive VM data, install it, run it and send acknowledgement to controller
End

Fig. 2. Workflow of the stealth migration protocol

construct_data(dataSegments[]): Here, leader VM merges the data from the data segments.
decrypt_data(data, secretKey): In this method, complete data is decrypted through a decryption process. From the data, VM leader separates the destination VM address and the VM data content to be transferred.
send_data(VMData, $VM_{destination}$Address): This method is used to send the data to destination VM address.

Methods *wipe(),* make *available(),* and *notify_controller()* have been described earlier.

The $\mathbf{VM_{destination}}$ Component: In this component, the destination VM receives the data from leader VM, installs it, and then makes it runnable before notifying the controller. The following algorithm performs the tasks:

Algorithm (5): Receives data, install it, notify controller after successful running.

Input: VMData, VM_{leader} Address

Output: Notify controller

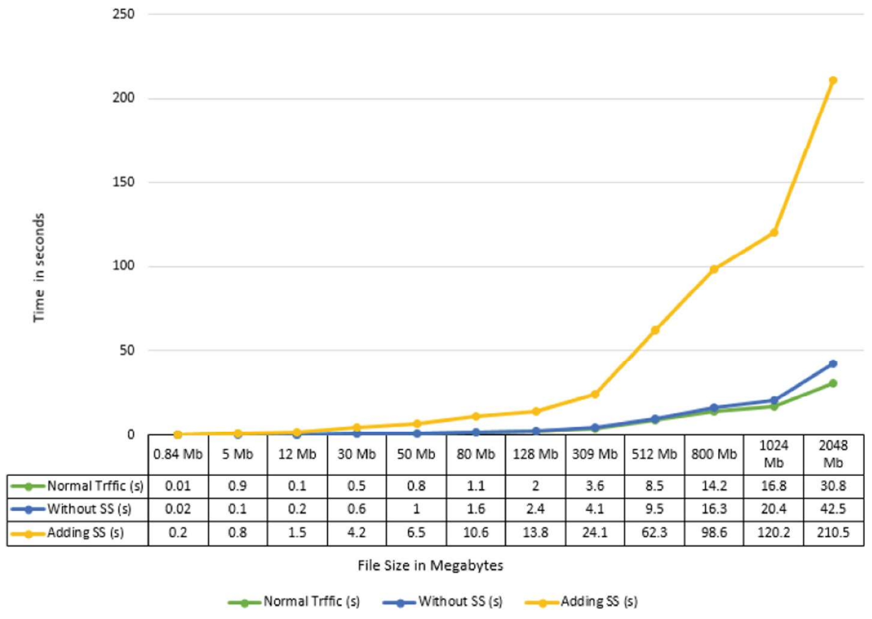
```
while(true)
       if instruction (found from the controller)
           open_port()
           listen (from VMleader Address)
           if recv()
      installation(VMData)
      checking_feasibility()
      test_all_use_cases()
      if running () = successful
            send_ack(to controller)
end while
```

Methods used in the above algorithm are described below:

installation(VMData): In this method, destination VM installs the VM data for further processing.
checking_feasibility() and test_all_use_cases(): After installing the VMData, destination VM does the feasibility checking and checks all the test cases to make the VM workable. If VM works perfectly, it sends an acknowledgment to the controller with its current VM state.

6 Experimental Results

A cloud environment with OpenStack has been setup for this experiment. The testbed has one controller node, and four compute nodes. All the nodes are Dell Optiplex 960 machine with 16 GB RAM, 500 GB Hard Disk, and 3 GHz processor. Each of the nodes has two gigabit-Ethernet cards. One of the Ethernet network interfaces is connected to all other nodes via a switch. The other Ethernet interface is connected to the internet with public IP address. All the nodes are running on the Ubuntu 12.4 LTS server. The VM live migration is implemented with OpenStack for Moving Target defense as directed by the guidelines in [1]. Network-attached Storage was implemented among the three nodes via the local area network. In this current OpenStack implementation, the software does not support automated live migration. Only the user with admin privilege can issue the command for live migration either from the command prompt or from the horizon dashboard webpage. So, to automate the live migration, a python script is written which runs in controller node and any time it gets a notification from the IDS that is attached to the possible anomalous VM, it automatically instructs the affected VM to live migrate through the stealth migration protocol.



	0.84 Mb	5 Mb	12 Mb	30 Mb	50 Mb	80 Mb	128 Mb	309 Mb	512 Mb	800 Mb	1024 Mb	2048 Mb
Normal Trffic (s)	0.01	0.9	0.1	0.5	0.8	1.1	2	3.6	8.5	14.2	16.8	30.8
Without SS (s)	0.02	0.1	0.2	0.6	1	1.6	2.4	4.1	9.5	16.3	20.4	42.5
Adding SS (s)	0.2	0.8	1.5	4.2	6.5	10.6	13.8	24.1	62.3	98.6	120.2	210.5

Fig. 3. Time taken to transfer different size of VMs in different ways

In the experimentation, VMs of different sizes ranging from 840 KB to 2048 MB were migrated and measured the time taken to transfer the VM over the network. The time taken for file transfer was measured in three different ways: normal traffic, traditional migration traffic, and migration traffic using our stealth migration protocol. In typical traffic scenario, a similar size file (not a VM) is transferred through the application and recorded the time duration to transfer it completely. In traditional migration traffic scenario, a VM is migrated through the traditional OpenStack live migration process and recorded the time duration to move a VM from one physical address to another physical address. Finally, similar size VMs were migrated from one location to another through our stealth migration protocol. Figure 3 shows the time taken to transfer different size VMs for three different scenarios. Figure 3 shows that as long as the VM size stays between 840 KB to 512 MB, there are no major significant time changes. When the size of the VM increases from 512 MB, a spike is shown on the graph, which means that the cost of time needs to be compromised for the sake of the system security.

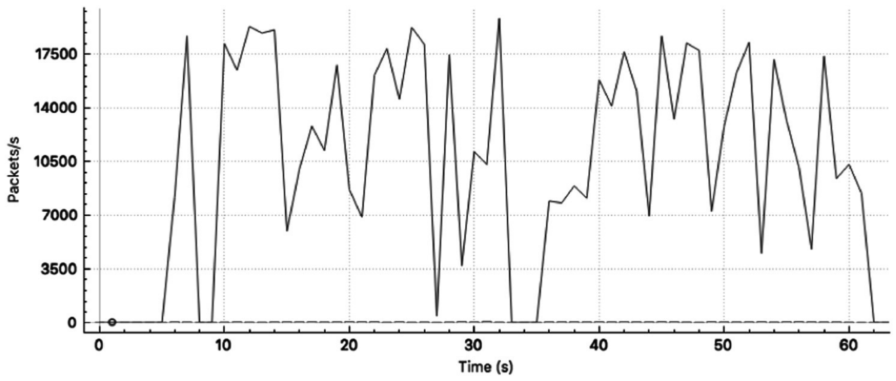


Fig. 4. Network analysis of a normal traffic

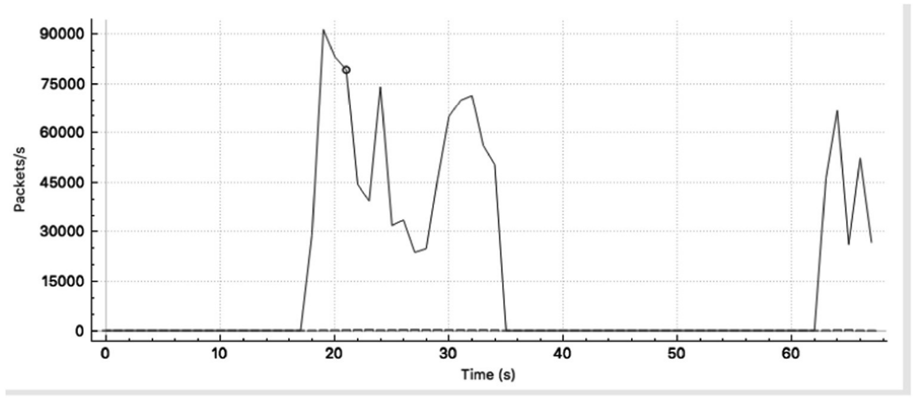


Fig. 5. Network analysis of a migration traffic

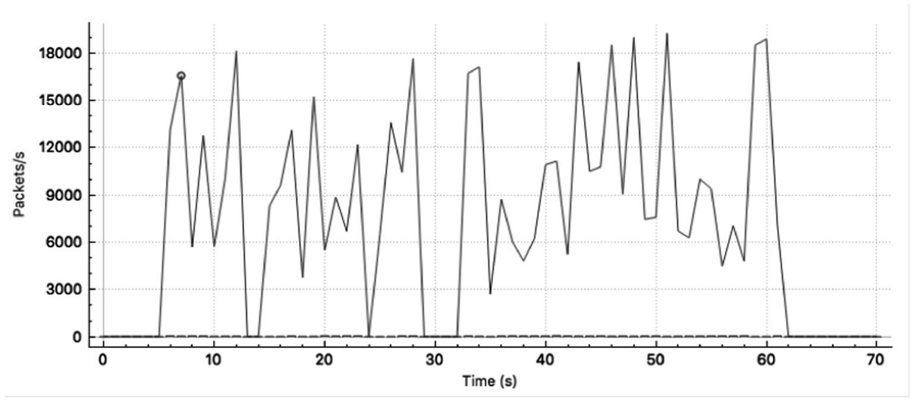


Fig. 6. Network analysis of migration traffic using Secret Sharing scheme and stealth migration protocol

Figures 4, 5 and 6 show respectively the normal traffic, traditional migration traffic, and migration traffic pattern that used stealth migration while it was transferring 900 MB of data or a VM. It is obvious that in Fig. 5, an attacker can easily distinguish migration traffic while he was analyzing the VM traffic pattern. The proposed protocol converts the distinguishable migration traffic pattern into a typical traffic pattern, which eventually obfuscates it from the attackers. Figures 4 and 6 are pretty much similar in terms of packets transferred per second which indicates that the attackers would not be able to distinguish the migration traffic from the normal traffic when the migration goes through the stealth migration protocol.

7 Conclusion and Future Work

The security concerns of VM live migration and deduced state of the art to secure MTD are discussed here. In this research, a stealth migration protocol is designed to minimize the risk and reduce the chance of being identified by an intruder to enhance the security of MTD. The protocol was implemented on the development testbed using "OpenStack" and the migration traffic was made indistinguishable from the normal traffic to help hide the migration information from the intruder. In experimentation, different sizes VMs are used for live migration purpose, and it can be concluded that if the size of VM is increased beyond 500 MB, the cost of time becomes prominent. Since the primary goal of this proposed protocol is to migrate the VMs securely while obfuscating it from the intruder, the cost of security turns into migration downtime. Investigation of those issues to minimize the overhead regardless of VM size and plans to adjust the migration traffic chunk size at a variable rate that could be more accurate to hide the migration traffic from the intruder are continuing. Further, machine-learning techniques to detect the migration traffic anomalies and to reduce the false positive alarms during the migration process are also being investigated.

References

1. Yackoski, J., et al.: Mission-oriented moving target defense based on cryptographically strong network dynamics. In: Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop. ACM (2013)
2. https://www.dhs.gov/science-and-technology/csd-mtd
3. Ahmad, R.W., Gani, A., Hamid, S.H.A., Shiraz, M., Xia, F., Madani, S.A.: Virtual machine migration in cloud data centers: a review, taxonomy, and open research issues. J. Supercomput. **71**(7), 2473–2515 (2015)
4. Oberheide, J., Cooke, E., Jahanian, F.: Empirical exploitation of live virtual machine migration. In: Proceedings of BlackHat DC Convention (2008)
5. Kozuch, M., Satyanarayanan, M.: Internet suspend/resume. In: Proceedings of Fourth IEEE Workshop on Mobile Computing Systems and Applications. IEEE (2002)
6. Duncan, A., et al.: Cloud computing: Insider attacks on virtual machines during migration. In: 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE (2013)
7. Achleitner, S., et al.: Stealth migration: hiding virtual machines on the network. In: INFOCOM 2017-IEEE Conference on Computer Communications (2017)
8. Suetake, M., Kizu, H., Kourai, K.: Split migration of large memory virtual machines. In: Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems. ACM (2016)
9. Deshpande, U., et al.: Fast server deprovisioning through scatter-gather live migration of virtual machines. In: 2014 IEEE 7th International Conference on Cloud Computing (CLOUD). IEEE (2014)
10. https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing
11. Clerk Maxwell, J.: A Treatise on Electricity and Magnetism, 3rd edn, vol. 2, pp. 68–73. Clarendon, Oxford (1892)
12. Polash, F., Shiva, S.: Automated live migration in openstack: a moving target defense solution. J. Comput. Sci. Appl. Inform. Technol. **2**(3), 1–5 (2017). https://doi.org/10.15226/2474-9257/2/3/00119
13. Xianqin, C., Xiaopeng, G., Han, W., Sumei, W., Xiang, L.: Application-transparent live migration for virtual machine on network security enhanced hypervisor. China Commun. **8**, 32–42 (2011). Research paper
14. Berger, S., Caceres, R., Goldman, K.A., Perez, R., Sailer, R., Doorn, L.: Virtualizing the trusted platform module. In: USENIX Security, pp. 305–320 (2006)
15. Tamrakar, A.: Security in live migration of virtual machine with automated load balancing. Int. J. Eng. Res. Technol. (IJERT) **3**(12) (2014)
16. Bgp hijacking. https://www.blackhat.com/docs/us-15/materials/us-15-Gavrichenkov-Breaking-HTTPS-With-BGP-Hijacking-wp.pdf
17. Hierarchy token bucket theory. http://research.dyn.com/2013/11/mitm-internet-hijacking/
18. Fu, X., et al.: On effectiveness of link padding for statistical traffic analysis attacks. In: Proceedings of 23rd International Conference on Distributed Computing Systems. IEEE (2003)
19. Houmansadr, A., Brubaker, C., Shmatikov, V.: The parrot is dead: observing unobservable network communications. In: 2013 IEEE Symposium on Security and Privacy (SP). IEEE (2013)

20. Dharam, R., Shiva, S.G.: Runtime monitors for tautology based SQL injection attacks. In: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec). IEEE (2012)
21. Shiva, S., Das, S.: CoRuM: collaborative runtime monitor framework for application security. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE (2018)