Bio-Inspired Formal Model for Space/Time Virtual Machine Randomization and Diversification

Noor Ahmed and Bharat Bhargava, *Fellow, IEEE*

Abstract—Studies on resiliency against system attacks have contributed well established defensive techniques, sound protocols and paradigms in distributed systems' literature. One of this contribution is credited to redundancy and replication techniques which is proven to be a double–edged–sword, by increasing the number of nodes inherently increases the system's attack-vector – the set of ways an attacker can compromise a system. To remedy this issue, system randomization and diversification has been considered as an effective defensive strategy, referred to as a Moving Target Defense (MTD). In this article, we introduce a bio-inspired formal model for space/time system randomization/diversification and a quantification scheme for virtual machines (VMs) in a cloud computing environment. We show the practicality of the model with a MTD framework *(Mayflies)* integrated into the cloud management software stack *(OpenStack)* and illustrate with realistic VM attacks and proactive defense use cases.

Index Terms—Cloud computing security, moving target defense, formal model, hidden Markov model, dynamic bayesian networks, byzantine fault tolerant, software defined networks, virtual Machines, OpenStack

1 Introduction

The traditional defensive security strategy commonly employs well established defensive techniques such as perimeter-based fire walls, redundancy and replications, and encryption. Given sufficient time and resources all of these methods can be defeated, especially, with sophisticated attacks that target zero-day exploits. This is due to the fact that the traditional security motto is to stay one-step ahead of the attackers at all times in which is proven to be ineffective defensive strategy. With the ever increasing adaptation on cloud computing due to its virtualized computing model built on commodity off–the–shelf hardware and software components, and programmable networking powered by *Software Defined Networks (SDN)* – the core building blocks of the cloud networking, attacks on these platforms and its networking fabric has risen in recent years.

Moving Target Defense (MTD) [1] is considered as a game changer in dealing with sophisticated attacks than the traditional defensive security strategies. *MTD* is a defensive strategy that aims to reduce the need to stay one-step ahead of the attackers by disrupting their gain-loss balance of the system. The core of this defensive strategy is to continuously shift the system's attack surface [2] – the set of ways/entries an adversary can exploit/penetrate the systems, with the goal of increasing the cost of an attack and the perceived benefit of compromising it by randomizing/diversifying system components (i.e., OS, Memory, CPU, and networking).

For decades, randomization/diversification techniques have been the ultimate defensive strategy to safeguard against attacks on memory structures, CPU registers, VMs, and applications. For example, Instruction Set Randomization [3], Address Space Randomization [4], randomizing runtime [5], and system calls [6], have been employed to effectively combat against memory and CPU exploits *(i.e., return-oriented/code injection)*. Interestingly, these techniques are considered mature and tightly integrated into most modern operating systems. Similarly, techniques such as N-Version Programming [7] for running variable binary forms of the same program, and N-Variant Systems [8] for running multiple variants of the same system in synchrony with a given input then monitoring for their convergence, are considered to deal with application-level attacks. However, these techniques are ineffective when attacks originate beyond the application boundaries (e.g., OS kernel, networks, side channel).

For this, randomization/diversification frameworks for VMs or containers with the applications running on them (i.e., [10], [11], [21], [24]) was later introduced to combat against attacks on the VMs (i.e., VM Hopping [15]). This followed by network IP randomization techniques (i.e., [9], [27]), also referred to as *IP-Hopping*, to deal with attacks against the networking fabric (i.e., network poisoning [27]). In general, the overarching goal of randomization and diversification defensive techniques is to disrupt attackers' gain-loss system balance, however, a formal model to reason such defensive solution scheme has not yet been sufficiently explored, thus, the focus of this paper.

Inspired by the the principles of the species' population dynamics of *preys* (VMs) and *predators* (attackers) [16], we

- *N. Ahmed is with AFRL/RI, Rome, NY 13411 USA. E-mail: ahmed24@purdue.edu.*
- *B. Bhargava is with the Purdue University, W. Lafayette, IN 47906 USA. E-mail: bbshail@purdue.edu.*

Manuscript received 24 Nov. 2018; revised 11 Apr. 2019; accepted 20 Apr. 2019. Date of publication 30 Jan. 2020; date of current version 7 June 2022. (Corresponding author: Noor Ahmed.) Recommended for acceptance by V. Piuri. Digital Object Identifier no. 10.1109/TCC.2020.2969353

formally model VM randomization/diversification with a combination of *Probabilisitic Finite State Automata (PFSA)* [26], *Hidden Markov Model (HMM)* [28], and *Dynamic Bayesian Networks (DBN)* [31]. To illustrate the practicality of the model, we use *Mayflies* MTD framework for the VM randomization/diversification with realistic attacks and proactive defense use cases introduced in our previous papers [21] and [22]. In this work, we make the following three contributions:

1) We propose a formal model for Time-Interval application Run-time Execution, dubbed *TIRE*.
2) We propose a sound theoretical model to mathematically formulate cloud-based MTD defensive strategy and a quantification method using well established modeling tools and techniques.
3) We introduce an integrated algorithms for a MTD framework with proactive live VM monitoring without changes to the cloud software stack to show the practicality of the model.

We have organized this paper as follows; we first give a brief background in Section 2 and the threat model in Section 3, then, formulate the problem in Sections 4. In Section 5, we discuss the practical implementation of *Mayflies* MTD framework, and then present the proposed formal model in Sections 6 and its quantification scheme in Section 7. Finally, we discuss the related work in Section 8 followed by the conclusion and future work in Section 9.

2 Background

One of the key success factors to cloud computing is attributed to elastic computing paradigm powered by Live VM Migration (LVMM) [13]. This is to move/diversify VMs across distinct host platforms within a data center or across geographically distributed data centers for load balancing, system maintenance, and SLA compliance for example. Although LVMM enables space/time VM randomization and diversification, adopting it as a defensive strategy is ineffective in its default formulation. This is due to the fact that migrating a compromised/infected live VM (i.e., OS) controlled by an attacker on a distinct host platform does not typically eliminate such control.

For space/time VM randomization and diversification as a defensive starategy, the VMs are terminated and a fresh instance is created/pre-prepared for the applications to run on a heterogeneous OS's on variable underlying computing platforms (i.e., hardware and hypervisors). This creates mechanically generated system instance (s) which is considered as good defense as type-checking [14], commonly referred to as Moving Target Defense (MTD). Formally modeling and quantifying the efficacy of such defensive strategy in a practical cloud setting is critical.

Although there are many LVMM inspired formal models such as [34], [35], and [36], their focus is purely on performance. Further, MTD-specific models such as [37] and [38] are based on game-theoretic approaches in which is quantified in a simulated setting (discussed in Section 8). To the best of our knowledge, this is the first bio-inspired MTD formal model in a practical cloud setting.

Inspired by the *predator-prey* interactions theory first introduced by Lotka-Voltera [16] which lay the foundation

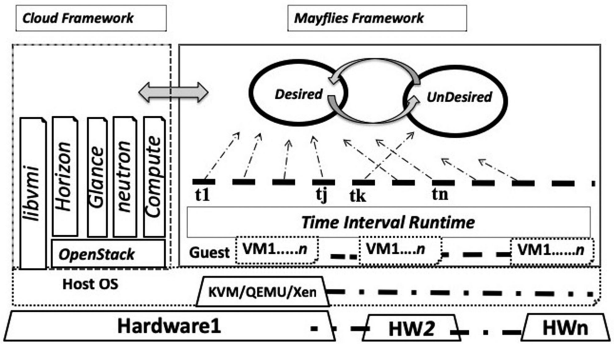

Fig. 1. High-level Mayflies architecture–the cloud infrastructure hardware (bottom), hybervisors/hosts (KVM/Xen) on each hardware (second layer) and guest VMs on the hosts (third layer). OpenStack components left box and Mayflies framework top right.

to mathematically formalize and quantify the principles of the species' population dynamics. There has been variations of *predator-prey* models studied in many species including `Mayflies` [17], aquatic insects with shortest lifespan (i.e., 5 minutes) in the species eco-system. At the core of most *predator-prey* models is measuring the *preys'* population by the proportionality of their survival/reproductive rate versus their eaten/infected rate by *predators*.

With this, we adopt the *Predator Satiation* [18] concept that describes the interaction between *preys/predators* in which the quantity of a particular *prey* at a given point in time far exceeds the potential number that can be taken by *predators* for any given time interval. Furthermore, we consider the quantification setup scheme introduced in *prey/predator Moving Target Defense* framework [19], an inducible defense scheme for preys (i.e., plants) against the predators (i.e., herbivores), to characterize our quantification scheme (discussed in Section 7).

Analogous to the cloud computing environment, we consider the VMs as the *prey* population and the attackers as the *predators*. The principle cornerstone of *prey-predator* model is to effectively control the *preys/VMs* survival/reproductive rate in order to assure a `desirable` *prey/VM* population at all times. With this, we consider two high-level system states, *preys/VMs* in a `desired` state and those in a `undesired` (compromised) state, then we reason the system behavior in terms of the transitions between these two states. Hence, quantify in terms of the overall proportion of time visited each state overtime. As such, we employ a *Mayflies* MTD framework to control the VMs reproductive/survival (by replacing) rate, and a proactive live VM monitoring scheme using *Library for Virtual Machine Introspection (LibVMI)* [40] to manage their eaten/compromised rate introduced in our previous papers [21] and [22].

We recently showed [23] the efficacy of *Mayflies* framework with a *Byzantine Fault-tolerant System (BFT-Smart)* [32] deployed on a private cloud platform *(OpenStack)*. In this paper, we integrate *Mayflies* with *LibVMI* and introduce two abstraction layers as illustrated in Fig. 1 below; two high-level system states *(Desired and UnDesired)* depicted as ovals in the top right quadrant, and *Time Interval Runtime Execution (TIRE)* abstraction depicted as dotted lines below the two states. This is to model each abstraction independently and logically compose for the desired proposed formal model, discussed in Section 4.

With *TIRE* abstraction, we manage the consensus of the VM population in time-intervals, as a result, accurately reason the transition between the *Desired* and the *Undesired* states overtime. We model the high-level system states with *Binary Random Walk* between the set of the two states as a *PFSA*. We construct the model using *HMM* structured as *Hierarchical HMM (HHMM)* [29] and represent as *DBN*. *HHMM* is an extension of *HMM* designed to model domains with hierarchical structure (i.e., natural language) where *DBN* generalizes *HMMs* state space to be represented in factored time-linear form, discussed in Section 6.

3 Threat Model

Typically, attackers take control of the system by gaining the systems' high privileges, thereby, altering the critical aspects of the systems' reliability and integrity. In this case, The attackers' advantage is the unbounded time of keeping the system under their control till exposed. MTD defensive strategy shifts this gain/loss balance in the system defenders' favor.

As such, we assume the attacker takes a minimum time t to compromise a VM *(VM_i)*, and having seen or attempted to compromise the VM with a given tactic devised for a given exploit will not reduce the time to compromise a new VM *(VM_j)* where $j > i$. This is because the new VM_j will require a new tactic and new exploit to compromise it given the fact that it starts with a different characteristics such as different OS, on different hardware and platform/hypervisor. Furthermore, we consider the adversary can employ arbitrary attacks on the VMs and assume the cloud software stack, the hypervisor and the networks (SDN) are secure.

4 Problem Formulation

Typically, we deploy systems in a *Desired* state (fresh/pristine VMs) with all its protective security measures in place. Then, there is a possibility that some of the VMs transition into an *UnDesired* state (i.e., exploited/compromised) without the system defenders' knowledge, a valid assumption in cyberspace. The overarching goal of the proposed model is to formally reason the transition between these two states in order to keep the VMs in *Desired* state at all times. Inspired by the *preys-predator* model, we aim to achieve this by controlling the VMs survival/reproductive rate with *Mayflies* MTD framework (Section 5) and for controlling their eaten/infected rate with VM attack detection library with *LibVMI* (Section 5.3). As illustrated in Fig. 1 (top right box), we introduce *Mayflies* with two abstraction layers; a pair of high-level hidden system states S, dubbed $S_{Desired}$ and $S_{UnDesired}$, and a Time-Interval application Runtime Execution *(TIRE)* as the driving engine for the two states.

4.1 Time-Interval Runtime Execution (TIRE)

Formally, *TIRE* is simply the break points of the infinite sequences of states in the traditional application runtime execution model, denoted by Q^{ω}. In each time-interval T_i where $i = 1, 2, 3 ..., n$, at least a VM v_i is replaced with v_i', thus, the execution sequences for v_i will be those $\{q_0 \dots q_{i-1}\} \in Q^i$ generated within T_0 to T_{i-1} time intervals, then the execution sequences for v_i' will be those $\{q_i \dots q_j\} \in Q^j$ of T_i to T_j where $i < j$, and so on. Thus, the runtime sequences of v_i, v_i', v_i'', ... are isolated in the form of $\{Q_v^i,\ Q_v^j,\ Q_v^k,\ ....\} \in Q^{\omega}$, thereby, allowing us to safeguard the individual VMs in time intervals rather than its entire runtime as the traditional runtime model that tends to be ineffective.

4.2 Reasoning the Hidden System State Transitions

With *TIRE* breaking the application runtime execution into time-intervals T_0 ...T_n. in each time-interval T_i, we assess the systems' current state $S \in [S_{Desired}, S_{UnDesired}]$ by proactively scanning all the VMs for attacks (infected VMs) using *LibVMI* and simultaneously replacing a VM in that time-interval. It's intuitive to see that these proactive observations are probabilistic in nature (either attack detected or not detected). As a result, we formulate the problem as a *Binary Random Walk* on the set of these two states moving randomly one move per time-interval T_i (i.e., as low as a minute) according to the following scheme.

We start with $S_{Desired}$ in the first time-interval (T_1) since the system is initially deployed in the *Desired* state before any attack takes place. Then, in each time-interval, we observe a random outcome of the system status as a coin flip, for example, in which we can either move to $S_{UnDesired}$ state or stay in $S_{Desired}$ state according to the outcome of the observation of a time-interval (T_i) (Section 5.2). Similarly, the next time interval T_j, T_k, and so on. However, for a typical system, the *UnDesired* state might consist of a set of internal states such as *compromised*, *failed*, *crashed*. Then, the observations can be viewed of as rolling a fair dice, for example, in which we either move to $S_{Compromised}$ if the die comes up 1 or 2, stay at S_{Failed} if the dice comes up 3 or 4, and move to $S_{Crashed}$ in the case of a 5 or 6. As a result, we quantify the model in terms of the number of visits made in each state overtime (Section 7).

5 VM Randomization and Diversification Framework

To lay the context and the practicality of the proposed formal model, in this section, we first discuss the design and implementation of *Mayflies* MTD framework for VM randomization and diversification, then, a proactive VM monitoring scheme to prioritize VM replacements and detect attacks using *LibVMI* (Section 5.2), and finally discuss the design challenges in Section 5.3.

5.1 MTD Framework Design and Implementation

Mayflies is a MTD framework integrated into OpenStack cloud software stack [41] introduced in our previous paper [21]. *OpenStack* is a widely adopted open source cloud management software stack that consists of a range of interconnected components such as *nova compute*, *horizon*, and *neutron*, to simplify cloud computing infrastructure management at scale with less user (admin) interactions. Fig. 1 above illustrates the high-level architecture of *Mayflies* framework (top right) and OpenStack cloud framework components (bottom and left quadrant).

Mayflies adopts a cross-vertical design that operates on three different logical layers of *OpenStack*; the *nova compute* at the application layer (GuestOS layer), the *VMI* at the hypervisor layer (HostOS layer), and the *neutron* at the networking

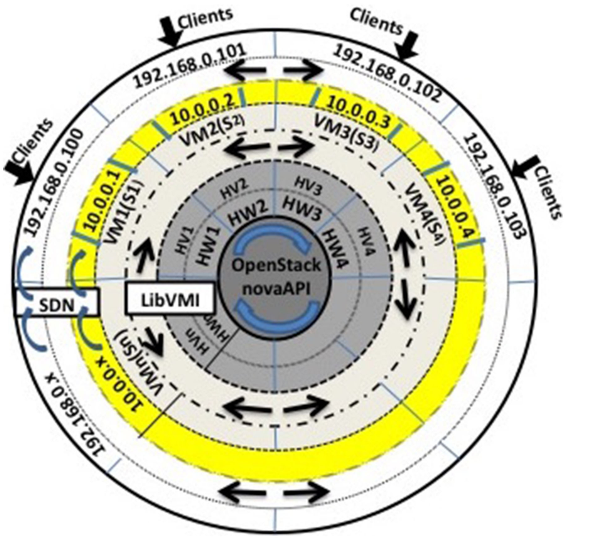

Fig. 2. Cross section view of cloud infrastructure. At the core inner circle is *Openstack*, the second ring depicts the hardware (HW1...HWn) and the hypervisors/*host OS* (HV1..HVn) on the third ring, and one or more *guest* VMs (VM1..VMn) on each *host* show on the fourth ring. The outer two rings depict the internal IPs (10.x.x.x) known as *Fix* IPs and the externally visible IPs (192.x.x.x) as *Floating* IPs, both are referred to as *port* in *SDN*.

layer. In addition, we integrated *Mayflies* with *LibVMI* [40], a library for virtual machine introspection to proactively monitor the VM's below the hypervisor (depicted next to OpenStack components). This is to detect attacks in real-time in order to prioritize VM replacements as well as to avoid diversifying vulnerable OS or a combination of certain system configurations (Section 5.2).

As the cloud software stack (OpenStack) abstracts the VM compute nodes from the applications' architectural style (i.e., SOA) or its communication model (i.e., synchronous versus asynchronous) with a unified deployment models (i.e., IaaS, AaaS, SaaS), *Mayflies* extends *OpenStack* to further abstract the applications' runtime from the VMs in order to break the runtime into observable time-intervals regardless of the applications running on them. In each time-interval (as low as a minute) we destroy a VM and replace it with a fresh copy, discussed next. In this paper, we introduced two abstraction layers; a pair of high-level hidden system states: *Desired* and *Undesired*, and *TIRE*, depicted as ovals and the dotted lines in the top right quadrant in Fig. 1, discussed in the previous section. This is to formally model each abstraction layer independently and accurately reason the transition between the hidden system states (Section 6).

5.1.1 VM Replacement

Fig. 2 illustrates the conceptual cross-section view of a cloud computing building blocks. At the core is *OpenStack*, the cloud software stack with a set of hardware (HW1...Hn) and hypervisors on each hardware (HV1...Hn) on rings (1 and 2) with *LibVMI* (rectangle box) operating in this layer. *Mayflies'* continuously substitutes guest VMs (third ring) and simultaneously reprogram network interfaces powered by SDN (outer 2 rings), discussed next. The idea is to rotate the outer three rings in sync without the users knowledge. However, the fundamental problem is dealing with the terminating VMs' application state for the newly instantiated VM (discussed in Section 5.3.2).

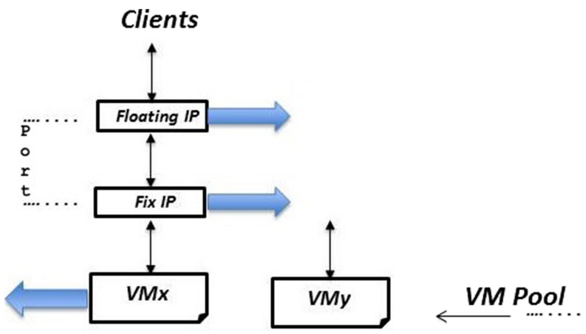

Fig. 3. Illustration of VM compute and Network interface swaps. VM_x seamlessly replaces VM_y from a pool of VMs.

Inspired by the clouds' VM resource (i.e., CPU) scheduling scheme, referred to as live VM migration, the VMs are paused/stopped then migrated across platforms without the users knowledge for load balancing. *Mayflies* dynamically replaces VMs by simply detaching the network interface of the active target VM, then, destroying/terminating the VM (round robin or random) using the cloud software stacks' command line interface (CLI) *nova-create VM*, *nova-destroy VM* and dynamically reprogram the network interface. Note that the CLIs are designed for provisioning and de-provisioning VMs and we used it as a defensive mechanism without any changes to the software stack.

Algorithm 1. VM Replacement

Input: *VMid*
1: **procedure** Replace()
2: $targetVMconfig \leftarrow CopyConfig(VMid)$
3: $DestroyVM(VMid)$
4: $newVM \leftarrow$ *GetNewVM()*
5: $SWITCHINTERFACES()$ ▷ algorithm 2.
6: $newVMconfig \leftarrow targetVMconfig$
7: **end procedure**

Algorithm 1. shows the VM replacement process. In Algorithm 1, we first save the target VM application configuration files and other related runtime state information including network interfaces IDs in line 2, then, destroy the target VM in line 3. We get a fresh/new VM from the standby VM pool in line 4 and swap the network interfaces in line 5 (as illustrated in Fig. 3 and also shown in algorithm 2). Then, copy back the configuration files in line 6.

Note that the fresh/new VMs can be from a pool of *prepared* VM with the applications installed without network interfaces or created *on-demand* then installed the applications and configured. The major difference of these two strategies is the start time. Depending on the systems' workload, the preprepared VMs start time is less then 20 seconds and over a minute for on-demand. Consequently, this is the inherent overhead for cloud based MTD solution. The details of the pros and cons of VM prepration and selection strategy is discussed in our previous paper [23].

5.1.2 Network Interface Replacement

Effectively terminating a VM and replacing it with a fresh new VM in a timely manner is simplified by *Software Defined Networking (SDN)*, a programmable networking fabric that decouples the control plane (virtual routers and switches) from the data plane. In SDN environment, active VMs are attached to a virtual network interface, referred to as *ports*, with a *fixed* IP address for internal access (among the servers) and a *floating* IP address for external access that can be assigned/bind to it at anytime. This is the virtualized version of the traditional network settings of *Local Area Network (LAN)* and *Wide area Network (WAN)* respectively. Note that both *Fix and Floating* IP addresses are bound to the *port* even after it's separated from the VM, thereby, transferable to another VM. As illustrated in Fig. 3, we detach the *port* from the target VM *(VMx)*, then get *VMy* from the prepared pool of VMs with the application and all its configuration files pre-installed and attach the *port*. Once the network *port* is attached to the new VMy, we *ssh* it to inject the necessary application runtime state information from the terminated target VMx and start the application.

Algorithm 2. Network Interface Switch

Require: VM_x, VM_y
1: **procedure** SwitchInterfaces()
2: **if** $VM_y Interface == NULL$ **then**
3: $neutron\ port-create\ <options>$
4: $neutron\ port-attach\ <options>$
5: $nova\ interface-associate\ <FloatingIP, VM_x>$
6: **else**
7: $port_{ID} \leftarrow GetPortID(VM_x(ID))$
8: $nova\ interface-dis-associate\ <VM_x, FloatingIP>$
9: $nova\ interface-detach\ <VM_x, VM_{x_{port_{ID}}}>$
10: $nova\ interface-attach\ <VM_y VM_{x_{port_{ID}}}>$
11: $nova\ interface-associate\ <FloatingIP, VM_y>$
12: **end if**
13: **end procedure**

Algorithm 2 shows the network interface swap procedure. In algorithm 2, we first check if the new VM_y from the pool was created with network interface in line 2 and create one for it if needed in lines 3 and 4, then, associate the known external IP address *(Floating IP)* of the terminating VM_x to the new VM_y in line 5. Note that the <options> for *port-create/attach* includes creating the interface with a specific IP address. We dis-associate the *Floating IP* if the VM_y has network interface in line 8, then swap the interfaces in lines 9 and 10. We finally associate the known IP from VM_x to the VM_y in line 11. This allows the servers/replicas to continue using the known IP and the clients reconnect to this replica through its *floating* IP (*i.e., 192.x.x.x*) as the old server/replica had dropped off of the network and came back. Typically, the new VM *(VMy)* has different characteristics such as Windows OS or variable Linux-based OSs (ubuntu/Fedora) than *VMx*.

Note that, depending on the OS image (i.e., Ubuntu or Windows) used for *VMy* in some cases, a reboot is required after the `nova interface-attach <options>` call. Furthermore, since `nova`, `neutron`, and all of the cloud software stack components communicate through asynchronous messaging bus, network-swap time varies depending on the *SDN* load (discussed in details in Section 5.3.1). With this, we consider the network swap overhead in part of the VM replacement time—time logged when *(VMy)* first contacted to the other VMs/servers.

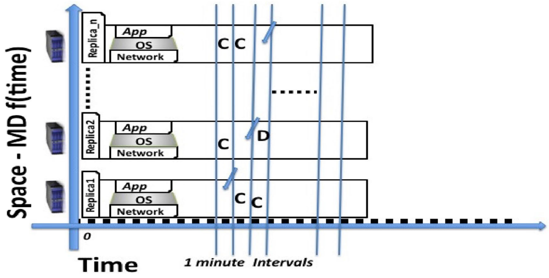

Fig. 4. An illustration of space/time randomization. Infected VMs are marked *D* for Dirty and *C* for *Clean* otherwise.

5.2 Proactive Monitoring With LibVMI

The concept of live VM migration is to blindly move VMs across distinct host platforms for load balancing the cloud infrastructure. This is ineffective for resiliency against attacks due to the possibility of migrating an infected VM. In contrast, MTD-based VM randomization terminates the VM and recreates a fresh VM with different characteristics (i.e., OS) on distinct host platforms. Since the combination (entropy) of the common OSs used in these VMs are limited (i.e., Linux, Windows), we integrate *Mayflies* MTD framework with *Library for Virtual Machine Introspection (LibVMI)* to detect vulnerable OS/applications and hardware platforms in order to avoid recreating it.

LibVMI is an open source library for live memory analysis (i.e., malware) [40]. It captures a running VM's memory content at the hypervisor-level with neglegible performance impact on the application [20]. This content information includes: the process ID's, the process/application names, and the their start and end address block offsets of all the processes running in the VM. In this work, we simply use *LibVMI* to detect the structural changes of this content commonly caused by certain attacks (i.e., code injection).

The hypervisor allocates memory address space with specific start/end address offsets upon spawning a new VM. This structure gets altered whenever any foreign code (attack) to be executed is inserted into the VM during runtime. The idea is to profile the VM before deploying it by taking a snapshot of its address offset structure (block start and end), then, comparing it with the subsequent snapshots. For book keeping, we simply mark *C* for *Clean* if the address structure is intact, otherwise mark *D* for *Dirty*, as illustrated in Fig. 4 above. This enables us to prioritize the replacements as well as to avoid vulnerable known OSs/platforms in the subsequent time-intervals.

Algorithm 3 shows the VM memory Introspection procedure. In Algorithm 3, for a new VM/node, we first save the initial VM memory structure (start and end-address offsets) in line 5 and mark it *Clean* since this is a fresh VM that is currently being profiled. Then, for subsequent snapshots, mark *Dirty* if the VM's address offsets differ/altered from the initially recorded offsets in lines 8, 9, and 10 respectively. This allows us to replace the dirty VM before the scheduled VM in the next time-interval.

To illustrate, we performed two attack scenarios using simple applications (*attack1* and *attack2*) that print a number every couple of seconds. In Fig. 5, the top box shows an attack to mimic when the application *attack1* with process ID [1767] (circled) is stopped and a malicious one is executed. We detected this change by the mismatch of the processes ID

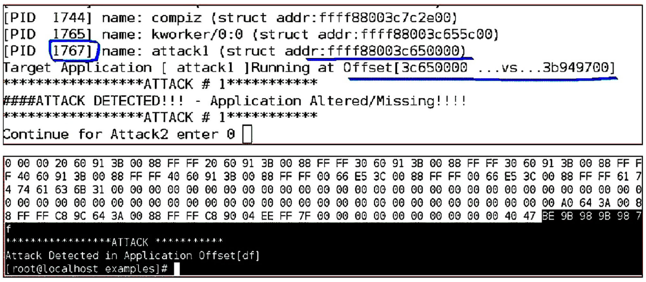

Fig. 5. An output of live VM attack detection with *LibVMI*. Process IDs (PID), process name and its address block offsets (shown in the top box), and the internal structure of one process address block (bottom box).

and the address offset from the previous offset (underlined). The bottom box shows *attack2* scenario where the application is hijacked without stopping it using a code injection attack. The address offsets within the applications memory block changed (highlighted in black). This is due to the fact that the small code injected in order to take control of the application cause the block to shift for the injected code to execute. A detailed description of our scheme can be found in our previous paper [22].

Algorithm 3. Virtual Introspect

Input: $node$
Output: $Clean$ or $Dirty$
1: **procedure** Introspect($node$)
2: **if** $node == new$ **then**
3: $initialProc \leftarrow GetProcessMemory(node)$
4: **nodeStatus** $\leftarrow Clean$
5: **else**
6: $currentProc \leftarrow GetProcessMemory(node)$
7: **if** $initialProc_i(key, val) \neq currentProc_i(key, val)$ **then**
8: **nodeStatus** $\leftarrow Dirty$
9: **end if**
10: **end if**
11: **end procedure**

To gain a holistic view of the high-level system state, in some time interval (i.e., one hour), we determine whether the system is in a *desired* state or *undesired* state by calculating the proportion of the VMs that are found with *Dirty* memory structures and how fast these VMs are being replaced within that time interval. This allows us to have full control over the *preys/VMs* population within pre-specified time intervals.

5.3 Design and Implementation Challenges

In this section, we discuss the inherent challenges of randomizing and diversifying VMs in virtualized cloud platforms, and dealing with the application state when replacing VMs.

5.3.1 Cloud Software Stack Limitation

The process of replacing a node/VM in *Mayflies* is greatly simplified by the combination of *nova* for provisioning/deprovisioning VMs and *neutron* for dynamically programming the network interfaces. However, these two components are asynchronous (functions have no return values to determine whether the next call can be safely performed). For example, detaching the network interface off of the VM with the `nova interface-detach <options>` to free it's *fix and floating* IPs in order to attach it to the new *VM* instance using the `interface-attach<options>` throws an error "IP is still in use". The reason is that all OpenStack component *(i.e., nova, neutron, horizon, glance, cinder, etc.)* are done through RESTful messaging (i.e., AMQP) for efficiency and interoperability.

A typical workaround is to insert *sleep(x)* to hold the process for an x amount of time before proceeding to the next call, however, this x will vary depending on the load of the controller which is difficult to predict, thereby, increasing the refresh time if x is large or disrupting the system (crashes) if x is too small. We synchronized the *nova* calls by making other `nova` reporting function calls `(i.e., nova show -minimal` and `nova interface-list)` in a while loop as illustrated in the following code snippet.

```
#/bin/bash
...
nova interface-detach <options>
while [ 1 ]
 do
  isactive=$(nova interface-list replicaID
   | awk '/\ACTIVE\y/ {print $2}');
  if [ -z ''$isactive'' ]
  then
break;
  fi
sleep 1
 done
nova interface-attach <options>
...
```

Basically, the loop holds the execution of the next function call by repeatedly calling `nova interface-list replicaID` function that reports the status of the given *replica ID* every second. We parse the value *ACTIVE* in `isactive` variable from the result returned by the `nova interface-list` command using `awk`, then, break once the value is *null* with the *-z* condition. This is to prevent on blindly waiting for asynchronous message based function calls. For this, we consider the network interface swap time in part of the VM start time as described in Section 5.1.2.

5.3.2 Application State

Mayflies framework partitions the traditional runtime execution of the system by terminating/destroying a VM and replacing it with another freshly spawned VM. The inherent challenges of this runtime partitioning are *1) dealing with the application state transfers, and 2) the performance impact on the application*. Generally, application state is an abstract notion of a continuous memory region of the application at runtime. Destroying/terminating VMs with a predefined time-interval (as low as a minute), breaks the continuity of the application state, thus, requires the state of the terminating VMs to be transferred to the freshly activated VMs, however, the implementation of such abstraction is dictated by the applications' communication model (i.e., *synchronous* versus *asynchronous*) among the applications/services or the client and the servers, therefore, MTD-based VM randomization is application dependent.

In [23], we deployed an implementation of a *Byzantine Fault-tolerant System (BFT-Smart)* [32] in *Mayflies* on a private cloud setting built with *OpenStack*. *BFT-Smart* is a replicated quorum-based *synchronous* system model where the replicas continue to guarantee reliability even a fraction of the nodes/VMs are malfunctioning (compromised/malicious). In this system, the state for the application includes; the systems' current transaction number and known leader, number of the participating replicas in the quorum, to aid the recovering VM/replica upon crash or failure. Replacing a VM in this system model only requires injecting the updated configuration files without any state information because the recovering/replaced VM connects to the rest of the replicas to synchronize. For this, we exploited the built-in reliability properties of the replicated systems to enable effective VM randomization and diversification with negligible performance impact.

Furthermore, VM randmization and diversification defensive scheme can be effective for applications like RESTful web services, a *asynchronous* stateless client/server service model, for example, the client requests are processed and responded by the servers without any system state is preserved. In this service model, the communication protocol that is bound to the client/server or between services attempt to reconnect when the VM is terminated and a new/fresh instance is activated in a timely manner. This is because the communication protocol (i.e., http/https) retries the connection without user intervention. In contrast, for stateful services, referred to as SOAP-based services, for instance, the services are bound to not only communication protocols but also security sessions (i.e., WS-*, WS-Secure Conversation) that cannot be disrupted or terminated and re-initiated, however, one can develop a work around for this limitation which we consider in the future work.

6 Formal Model

In this section, we first describe the proposed model, then, the construction and the formulation schemes, and discuss our quantification scheme in the next section.

6.1 Model Description

Finite State Automata (FSA) is widely adopted mathematical machinery for specifying systems with both Deterministic Finite Automata (DFA) and Non-Deterministic (NFA) properties. Buchi automaton [25], a type of ω-automaton which is NFA is the most popular kind of automaton used in modeling distributed systems. It is extremely challenging to develop an effective proven methods for high-level system state transitioning under the non-deterministic nature of the cyber space, therefore, we model the system with *Probabilistic FSA (PFSA)* [26].

PFSA is simply a NFSA (with no ϵ transition) with probabilities for all transitions of the FSA. By definition, PFSA is a generative model, where as the FSA (non-probabilistic) finite automaton, are accepting devices for strings generated by grammars in formal languages. We don't specify any alphabet a input string $\sum$ for our automaton, however, we use the output alphabet donated by Λ where $a \in \Lambda$.

We consider the *Time Interval Runtime Execution (TIRE)* observation outcomes generated by *LibVMI* (discussed in 5.2) to represent the output alphabet $a \in \Lambda$ that drives the high-level system state *(Desired/UnDesired)* transitions, discussed in Section 6.3. The outcome of these observation can be either *true* or *false* in which *true* is the *accepting* transition to another state and *false* is staying in the same state. The expressiveness of the *Accept* lies the power of the Buchi automaton to model *TIRE* and the correctness property violations can be specified in terms of the *accept* condition.

A property is specified as a Buchi automata A and then the characteristics of the structure of this automata are used to classify its properties. Modeling *Mayflies* framework with PSFA allows *TIRE* probability observations to be modelled as *Hidden Markov Model (HMM)* [28]. We achieve such structured characteristics by constructing the *HMM* with *Hierarchical Hidden Markov Model (HHMM)* [29] and representing it with *Dynamic Bayesian Networks (DBN)* [31], a time-linear representation of *HMM*.

FSA enables modeling complex systems by decomposing into multiple automaton and then chaining one automaton output to a second automatons' input, thereby, reasoning about the system behavior separately while composing them to achieve the desired results. Thus, the proposed model enbales the composition of other formal automata models such as application interface automata [33] and attack surface [2]. As such, the proposed model fills the gap to formally model an end-to-end system spectrum of the cloud ecosystem.

6.2 Model Construction

As we formulated our model in Section 4, we typically deploy a system in a *desired* state and at some point in time we end up in *undesired* state (i.e., compromised/infected) without the system defenders' knowledge (in most cases). This is mostly credited to the successful stealthy attacks that create *turbulence* state infinitely many times until the system is *compromised*, ex-filtrated data or less usable (*fail or crash*). These high-level uncertainties are driven by what's happening at the application's runtime level, for instance, if a node/server is *compromised* and is still running, then, the system is in a *compromised* state, in contrast to when a node crashes in which the system enters into a *failed* state. One way to formalize this behaviour is through *HHMM*.

As the name implies, *HHMM* forms a hierarchy of *HMMs* where each state itself is an *HHMM* with sub level of *HMMs* as its abstract/internal states. The top-level states in the hierarchy are called the hidden states and the low-level is the production state that emit observations. We adopt the *HHMM* automatic construction concept used to detect semantic patterns in motion video introduced in [30]. An *HHMM* is defined as a 3-tuple $H = < \lambda, \xi, \Sigma >$ where $\lambda \supseteq (A, \Pi, B)$ which represents the set of the transitions for the horizontal matrix, the vertical vector and the probability distributions respectively. The ξ is the topological structure which specifies the levels and parent-child relationships of all the states, and Σ is the observation alphabet.

We construct an *HHMM* in which the hidden states S are *Desired*, *UnDesired* and *Time Interval Runtime Execution (TIRE)* as the omitting/observable state. As illustrated in Fig. 6, we define the topology of the *HHMM* hierarchy as follows: The *Desired* state (D) as the root state in level I (i.e., initial state), the *UnDesired* set of states *Compromised* (C) and *Failed* (F) in level II (can be represented as many states and levels), and *TIRE* as the leaf state in level III.

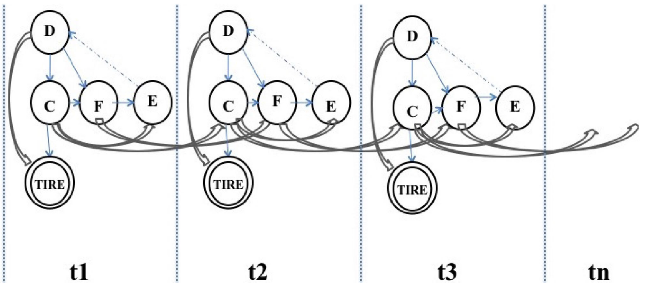

Fig. 6. Mayflies DBN model – system states are Desired, UnDesired ($Compromised, Failed$) labeled as *D*, *C*, *F*, followed by the Exit state *E*. The dotted lines on *E* depicts for the control returning to the parent node *D* bases on the observations. TIRE is the observing state in double circles.

With this *HHMM* construction, we represent the model with *Dynamic Bayesian Network* [31] as depicted in Fig. 6. DBN represents *HHMM* with time-linear transition partitions *(t1, t2, ..., tn)* to drive a much simpler and faster algorithms for inference, classifications, prediction and learning which we consider in our future work. In this work, the representation and the encoding of the observation sequences and the transitions between the hidden states *(desired/undesired)* of the model is sufficient to illustrate the objective of the proposed model. Since we are not interested in contracting the model and learning by its probability distributions, and the hidden state themselves are not internal *HHMMs* states with abstract sub-levels of the *HMMs*, we treat our *HHMM* as a flat *HMM* to reason the transition probabilities of the hidden states. In fact, the hidden state are visible to us as we anticipate of the observation outcomes from *LibVMI*, thereby, enabling us to bounce the system back to a *desired* state at any given time.

Thus, we map the VM status observations captured in time-intervals (i.e., one minute) by the *LibVMI* at the hypervisor level to the *TIRE* state *S* (S_{TIRE}) emissions. We consider the following three observations:

- A VM is *clean* which is typically the initial state when the system is first deployed.
- A VM is *failed* which can be either not-reachable due to network drop or hardware/software failures.
- A VM is *dirty* when the memory integrity violations is detected.

We define the guiding principle of state *transitions* as following:

- The systems starts in a *Desired* (S_D) state and transitions to either *Failed* (S_F) state if S_{TIRE} emit *in-active*, or to a *Compromised* (S_C) state if S_{TIRE} emit *dirty*. Otherwise, stays in (S_D).

To illustrate how we map the VMI observations to the high-level *DBN* state machine automata model, consider at time $t = 1$ in Fig. 6, the system starts in a *desired* (S_D) state and consider S_{TIRE} emits *dirty* after the first observation, then the system transitions to a *compromised* (S_C) state in $t = 2$. We cannot change the state till (S_C) transitions to S_E signaling for its exit according to *DBN* representations. At this point, we refresh the compromised VM and asses the system so the next time in $t = 2$ we anticipate S_{TIRE} emit *active* and the system transitions to S_D at $t = 3$. Modeling *Mayflies* with HHMM and encoding it in this manner, we can reason the system behavior by the transitions between the *DBN* states (discussed next), therefore, we quantify it in terms of the overall proportion of the time $\{t_i, t_i, t_k, ...\}$ the system visited in the each of the hidden states (Section 7).

6.3 State Transition Probabilities

As illustrated in Fig. 6, we constructed three hidden states S_D, S_C, S_F and S_{TIRE} as the driving engine of the model. In this section, we discuss the transitioning probabilities of *TIRE* and the high-level hidden states *Desired/Undesired*. Note that the S_C and S_F are considered as *Undesired* state where S_E is for the *DBN* exit state of each level of the *HHMM* hierarchy.

6.3.1 TIRE Transitions Probabilities

As described in Section 4.1, the *Time-Interval Runtime Execution (TIRE)* is an abstraction that breaks the traditional runtime execution model denoted by Q^{ω} into infinite sequences of states$\{q_0 \ldots q_n\} \in Q^{\omega}$. We consider each sequence of q_i as a time unit/interval T (as low as a minute) for a VM to exist (i.e., *VM lifespan)*. In each time-interval T_i, where $i = 1, 2, 3 \ldots n$, we simultaneously scan all of the VM's for attacks using *LibVMI* (described in Section 5.2). If an attack is detected in any one of the rest of the VMs, then, we replace the comprised VM(s) before it reaches its predefined *lifespan*. This is to keep the VM population in the *Desired* state in the next time interval T_j, then T_k, and so on.

One way to formalize and model such observations *O* (i.e., whether a VM status has changed) is through *Burnolli Trails* in *Hidden Markov Model (HMM)*. A Markov chain/process is a sequence of events or states $Q = \{q_1, q_2, \ldots q_n\}$, and HMM represent stochastic sequences as Markov chains where the states are associated with a probability density function (*pdf*). The *pdfs* in each state q_i are characterized by the probabilities of the emission $p(x|q_i)$ and the transition $q_{i,j}$ where the transition to a next state is independent of the past states. An elaborate introduction of the theory of HMM and its applications can be found in [28].

Formally, let $\{O_j$, j = 1, 2,...$\}$ be observations of all of the VMs collected by the *LibVMI*. We model these observation as a Bernoulli processes where $O_j \in \{0, 1\}$, where $O_j = 0$ indicates a VM v_i is *clean* and $O_j = 1$ indicates *dirty*. Each v_i is defined as a tuple: $v_i = \langle v_{start}, v_{\rho} \rangle$ where

- $v_{start} \in \mathbb{R}^+$, represent the real time the VM starts.
- $v_{\rho} \in [v_{start}, < \rho | O_i^t >]$, represent the *lifespan* of the VM which includes the VM start time v_{start} and the end time where the end time can be either its *lifespan* ρ or terminated prematurely based on the observation *i* result at time-interval *t* O_i^t due to attacks.
- v'_{start}, $v'_{\rho'}$, represent the real time a VM v_i is replaced with v'_i, call it v_j, with its new lifespan ρ', thus, the tuple for $v_j = \langle v_{start}, v_{\rho} \rangle$.

Therefore, *TIRE* transition function is simply a real number – time assigned to the structure which breaks the system runtime into manageable intervals (i.e., one minute intervals). Thus, we define the transitioning function as:

$$\alpha T_{i,j} \rightarrow \mathbb{R}^+.$$

Using $\alpha T_{i,j}$, we simply observe node(s) status between αT_i and αT_j. At the transition point αT_j, we generate a sequences of observations $O = o_1, o_2, o_3, \ldots$ of *inactive* and/or *dirty* VM. *TIRE* transitions $T = t_0 \ldots t_n$ and observations $O = o_0 \ldots o_n$ lie the probability distributions to easily reason about the high-level system state transitions (discussed next). Thus, for each state *S* in *Mayflies*, we associate that state with random variable taking values in Λ according to certain (state-dependent) probabilities.

For this, an HMM observation *o* is the logical predicate over *Mayflies'* high-level states. Each T_i is considered a state predicate evaluates to *true* or *false*. We say that state transitions at each T_i satisfies a state predicate if the predicate evaluates to *true* and vice-verse. Hence, by definition of the first-order HMM, transition t_i to t_j is dependent only upon the current state at t_i. Therefore, the probabilistic nature of that transition can be defined as:

$$\alpha T_{i,j} = Pr\Big[T_{i+1} = j \Big| T_t = i\Big].$$

We make a first-order HMM assumption regarding the transition probabilities.

$$Pr\Big[T_i, \Big| T_{i-1}, T_{i-2}, \ldots, T_0\Big] = P\Big[T_i \Big| T_{i-1}\Big], i \in 0,1,2,3\ldots.$$

Similarly, we assume the emission probabilities of the model on how the observed event from S_{TIRE}) results system state transition:

$$Pr\Big[o_i, \Big| T_i, \ldots, T_0, o_{i-1}, \ldots, o_0\Big] = P\Big[o_i \Big| T_i\Big], o \in O.$$

Modeling TIRE as an observable HMM and formulating it in this manner enables us to anticipate the high-level hidden state transitions where the probability of system transitioning to any of the two state in T_i can go either way *(i.e., desired/undesired)*. We anticipate this outcome if it results against our favor to bounce the system back to our *desired* state in the next time interval (T_{i+1}). Thus, each *TIRE time interval* (T_i) is represented as the transition state, and the transition between the states are the *invariant* that must be preserved. We assert that the continuity of the underlying runtime execution is preserved if these invariant hold.

Note that the fundamental problem of time-interval based observations is choosing the perfect observation intervals, for example, if the observation time is too long, we will have the case where the observation o_{i-1} results that we are in a *Desired* state, then at o_i end up in a *Compromised* state before we get the observation o_{i+1}, a valid assumption in cyber space. In contrast, if the observation time is too short, then we will introduce unnecessary performance burden on the applications.

6.3.2 High-Level Hidden State Transition Probabilities

Typically, at the deployment time, the system starts in a *Desired* state, call it $S_{Desired}$. *TIRE* observation generates transition probabilities of either to a $S_{Compromised}$ or S_{Failed} state. The probability that a transition can happen before observation is collected is:

$$\alpha_{T_{ij}} Pr[T_0 = 0].$$

Therefore, assuming the system starts in $S_{Desired}$ state and further assuming in that state till the first observation collected. Certainly, this is the base case.

For the 1st observation or $\forall T_i$ where $i > 0$, the probability of seeing the observed events $o_1, o_2, o_3, \ldots$ of a sequence up to o_{i-1} observations and reaching in state T_{i-1} time interval, then transitioning to state $S_{Compromised}$ at the next step is:

$$\begin{aligned}&P(T_0, T_1, T_2, \ldots, T_{i-1}, o_{i-1} = S_{Desired}, o_i = S_{Compromised})\\&= \alpha_{T_{ij}}(S_{Desired}) Pr\Big(o_i = S_{Compromised} \Big| o_{i-1} = S_{Desired}\Big).\end{aligned}$$

Similarly, for the 1st observation or $\forall T_i$ where $i > 0$, the probability of seeing the observed events $o_1, o_2, o_3, \ldots$ of a sequence up to o_{i-1} observations and reaching in state T_{i-1} time interval, then transitioning to state S_{Failed} at the next step is:

$$\begin{aligned}&P(T_0, T_1, T_2, \ldots, T_{i-1}, o_{i-1} = S_{Desired}, o_i = S_{Failed})\\&= \alpha_{T_{ij}}(S_{Desired}) Pr\Big(o_i = S_{Failed} \Big| o_{i-1} = S_{Desired}\Big).\end{aligned}$$

In general, the probability that we are starting in $S_{Desired}$ at T_{i-1} time-interval given the observed events up to o_{i-1}, and given that we will be in state other than *Desired* state at time-interval T_i observation o_i, the transitioning probabilities are equally likely, thus, preserving for all cases.

Although we used *Mayflies* MTD framework with *LibVMI* to illustrate the practicality of the model (keeping the *preys/VMs* population in balance within the *Desired* state at all times), one can use any MTD framework that randomizes/refreshes VMs and any real time intrusion detection system with this model.

7 Model Quantification

In *predator-pray* MTD model discussed in Section 2, the *preys* population is measured by the proportionality of their survival/reproductive rate versus their eaten/infected rate by *predators*. To effectively control the survival/reproductive rate of the VMs, we consider two competing time; the system defenders' and the attackers' costs as a function of time:

- The defensive cost is the *VM Replacement Cost RC(T)* – time to replace a node/VM including the network interface replacement plus the *Observation Cost OC(T)* which is the time it takes for *LibVMI* scan all the VMs.
- The *Attack Cost AC(T)* – time it takes for any attack to be carried (i.e., OS finger printing, code injection time) and succeed.

Although it's extremely difficult to accurately calculate the attackers' cost for compromising a system, we anticipate to calculate this cost relative to the rate of the VMs observed with *dirty* memory structures in each time-interval T_i. With this, we quantify the long-run distribution (i.e., one hour) by calculating the proportion of the time T_n that the system visits the hidden states *(Desired/UnDesired)* over that time frame.

Formally, let ν be the expected overhead time of *replacing* a VM and μ be the expected overhead time of system *observations* in one time interval T_i, where $i \in 1,2,3\ldots n$, then:

$$RC(T_n) = \sum_{i=1}^{n} \nu_i,$$

and

$$OT(T_n) = \sum_{i=1}^{n} \mu_i,$$

where *RC*(T_n) and *OC*(T_n) is the total cost of the MTD defensive strategy of some time interval T_n (i.e., one hour).

Let $p_{q_{ij}}$ denote the probability of going from state q_i to q_j in one step, which is characterized by the rate proportional to the the rate of the *prays/VM* being eaten/compromised γe_i over the number of the states *S* [16].

$$q_{ij} = \frac{\gamma e_i}{S},$$

Let λ_i represent the matrix *P* whose entries are the p_{ij}.

For each state S_i, we define:

$$\lambda_i = \frac{\sum_{i=1}^{n} S_i}{T_n},$$

where $\sum_{i=1}^{n} S_i$ is the total number *n* of visits the process makes to each state S_i over the time-intervals $T_i \in T_i, T_j, T_k. \ldots T_n$. Intuitively, the existence of λ_i translates to changes in system states in which in turn is not in a single state *(i.e., undesired)* as long as our *observations* and node *replacements* is being performed within the acceptable time of T_n's. Note that the Markov process model is an exponential distribution, in that, the decisions are dependent only in the current state. As such, if we are at *Desired* state now, the probability to any other state will be *1/3rd* (with the 3 states) no matter where we were *(Failed or Compromised)* in the past.

Let λ denote the row vector of the elements of the λ_i, given the underlying HMM state transition for each state S_i, then we have a matrix in the form of $\lambda = \lambda P$ subject to $\sum_i \lambda_i = 1$. Calculating λ in each transition results a solution set of $< X_D, X_C, X_F >$ time units for the three states, which means that in the long run we spent *X* amount of the time at the *Desired* state, *X* amount of time in *Compromised* state, and *X* amount of our time at *Failed* state. Thus, we can easily reason the status of the high-level system state in any time interval, for instance, if we run the system for 1 hour, then, we get time intervals like; for 55 minutes we operated under normal conditions in a *Desired* state, 3 minutes in a *Compromised* state, and 2 minutes in *Failed* state.

8 Related Work

There have been several studies on modeling VM randomization/diversification fueled by live migration on VMs. For example, iAware [34], a lightweight interference-aware live VM migration and co-location model that aims to minimize the performance interference during and after the migration of the VMs, and a model for predicting the perfromace impact inherent on the hardware variablity on tenant applications [35]. A comprehensive survey on models and performance mangement on VMs is discussed in [36]. While our proposed model is purely focused on defensive security, all of these models are complementary to our work.

Several other VM migration models focused on defensive security include: a game theoretic MTD-based formal model and a quantification approach [37], and a theory of MTD systems and an attacker theory that defines how elements of the MTD systems and attacker theories interact in which the effectiveness of the model is quantified in terms of the success likelihood of intrusion [38]. Unlike our bio-inspired formal model and the quantification scheme with a practical cloud design illustration and implementations of algorithms, most of the previous MTD-based formal models are theoretical and analyzed in a simulated settings.

9 Conclusion

We introduced a simple but effective bio-inspired formal model for reasoning and quantifying Virtual Machine (VM) space/time diversification and randomization across cloud computing platforms with a combination of *HMM*, *HHMM and DBN*. To illustrate the practicality of the model, we described *Mayflies*, a MTD framework, and discussed the implementation details of VM and network interface replacements in *Openstack* private cloud software stack. For future work, we consider a through experiments on the inference, classifications, prediction and learning from the model to predict attacks using machine learning techniques.

Acknowledgments

We credit the idea of the probability transition function setup in sec. 6.3 and the matrix calculation format in sec. 7 from an informal notes that we could not locate its url.

References

[1] S. Jajodia *et al.*, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, vol. 54, Berlin, Germany: Springer, 2011.
[2] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 371–386, May/Jun. 2011.
[3] G. S. Kc, A. D. Keromytis, V. Prevelakis, "Countering code injection attacks with instruction-set randomization," in *Proc. 10th ACM Conf. Comput. Comm. Security*, 2003, pp. 272–280.
[4] S. Bhatkar, D. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proc. 12th USENIX Sec. Symp.*, 2003, pp. 105–120.
[5] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proc. 22nd Annu. Comput. Secur. Appl. Conf.*, 2006, pp. 339–348.
[6] S. Rauti, S. Laurén, S. Hosseinzadeh, J. M. Mäkelä, S. Hyrynsalmi, V. Leppänen, "Diversification of system calls in linux binaries," in *Proc. Int. Conf. Trusted Syst.*, 2015, pp. 15–35.
[7] L. Chen and A. Avizienis, "N-version programming: A fault tolerance approach to reliability of software operation," in*Proc. Digest 8th Int. Symp. Fault-Tolerant Comput*, 1978, pp. 3–9.
[8] B. Cox, D. Evans, A. Fillipi, J. Rowanhill, and, W. Hu, "N-variant systems: A secretless framework for security through diversity," *Proc. 15th Conf. USENIX Secur. Symp.*, 2006, Art. no. 9.
[9] J. Jafarian, E. Al-Shaer, and G. Duan, "Open flow random host mutation: Transparent moving target defense using software defined networking," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw.*, 2012, pp. 127–132.
[10] M. Carvalho *et al.*, "A human-agent teamwork command and control framework for moving target defense," in *Proc. 8th Annu. Cyber Secur. Inf. Intell. Res. Workshop*, 2013, Art. no. 38.
[11] H. Okhravi, A. Comella, E. Robinson, and J. Haines,"Creating a cyber moving target for critical infrastructure applications using platform diversity," *Int. J. Crit. Infrastructure Protection*, vol. 5, no. 1, pp. 30–39, Mar. 2012

[12] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proc. 6th Workshop Hot Topics Operating Syst.*, 1997, pp. 67–72.

[13] C. Clark *et al.*, "Live migration of virtual machines," in *Proc. 2nd Conf. Symp. Networked Syst. Des. Implementation*, 2005, Art. no. 286.

[14] F. Schneider, 2010, "From fault–tolerance to attack tolerance. [Online]. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a548748.pdf

[15] T. Ormandy, "An empirical study into the Security exposure to hosts of hostile virtualized environments," in *Proc. CanSecWest Appl. Secur. Conf.*, 2007, pp. 1–18.

[16] A. J. Lotka, *Elements of Physical Biology*. WIlliams and Witkins, Baltimore MD, USA: Franklin Classics Trade Press, 1925.

[17] B. Sweeney, *Mayflies and Stoneflies: Life Histories and Biology*. Norwell, MA, USA: Kluwer Academic Publisher, 1987.

[18] B. Sweeney and R. Vannote, "Population synchrony in mayflies: A predator satiation hypothesis," *Evolution*, vol. 36, pp. 810–821, 1982.

[19] F. Adler and R. Karban, "Defended fortresses or moving targets? Another model of inducible defenses inspired by military metaphors," *Amer. Naturalist*, vol. 144, pp. 813–832, 1994.

[20] T. Garfinkel and M. Rosenblum, "A virtual machine-based architecture for intrusion detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2003, pp. 191–206.

[21] N. Ahmed and B. Bhargava, "Mayflies: A moving target defense framework for distributed systems," in *Proc. ACM Workshop Moving Target Defense*, 2016, pp. 59–64.

[22] N. Ahmed and B. Bhargava, "Towards targeted intrusion detection deployments in cloud computing," *Int. J. Next-Gener. Comput.*, vol. 6, no 2, pp. 129–139, 2015.

[23] N. Ahmed and B. Bhargava, "From byzantine fault-tolerant to fault-avoidance: An architectural transformation to attack and failure resiliency," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2018.2814989.

[24] M. Villarreal-Vasquez, B. Bhargava, P. Angin, N. Ahmed, D. Goodwin, K. Brin and J. Kobes, "An MTD-based self-adaptive resilience approach for cloud systems," in *Proc. IEEE 10th Int. Conf. Cloud Comput.*, 2017, pp. 723–726.

[25] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI-Quarterly*, vol. 2, no 3, 1989. [Online]. Available: https://groups.csail.mit.edu/tds/papers/Lynch/CWI89.pdf

[26] M. Robin, "Probabilistic automata," in *Proc. Inf. Control*, vol. 6, pp. 230–245, 1963.

[27] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and counter measures," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 8–11.

[28] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proc. IEEE.*, vol. 77, no. 2 pp. 257–286, Feb. 1989.

[29] S. Fine, Y. Singer, and N. Tishby, "The hierarchical hidden Markov model: Analysis and applications," *Mach. Learn.*, vol. 32, pp. 41–62, 1998.

[30] O. Samko, A. D. Marshall, and P. L. Rosin, "Automatic construction of hierarchical hidden Markov model structure for discovering semantic patterns in motion sata," in the *Proc. 5th In. Conf. Comput. Vis. Theory Appl.*, vol. 1, 2010, pp. 275–280.

[31] K. P. Murphy, "Dynamic Bayesian Networks: Representation, Inference, and learning," PhD Dissertation, Dept. Comput. Sci., Univ. California at Barkley, California, CA, USA 2002.

[32] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMaRT," in *Proc. 44th Annu. IEEE/IFIP Int. Depend. Syst. Netw.*, 2014, pp. 355–362

[33] L. Alfaro and T. Henzinger, "Interface automata," in *Proc. 8th Eur. Soft. Eng. Conf. Held Jointly 9th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2001, pp 109–120

[34] F. Xu, F. Liu, L. Liu, H. Jin, B. Li, and B. Li, "iAware: Making live migration of virtual machines interference-aware in the cloud," *IEEE Trans. Comput.*, vol. 63, no. 12, pp. 3012–3025, Dec. 2014.

[35] F. Xu, F. Liu and H. Jin, "Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud," *IEEE Trans. Comput.*, vol. 65, no. 8, pp. 2470–2483, Aug. 1, 2016.

[36] F. Xu, F. Liu, H. Jin and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," in *Proc. IEEE*, vol. 102, no. 1, pp. 11–31, 2014.

[37] Z. Yulong, B. LiKun, and Z. YuWanyu, "Incentive compatible moving target defense against VM-colocation attacks in clouds," in *Proc. Int. Inf. Secur. Conf.*, 2012. pp 388–399.

[38] R. Zhuang, PhD Dissertation, Dept. Comput. Inf. Sci., Kansas State Univ., Manhattan. 2015. [Online]. Available: http://hdl.handle.net/2097/20525

[39] L. Cheng, Z. Hong-Qi, T. Jing-Lei, Z. Yu-Chen, and L. Xiao-Hu, "Moving target defense techniques: A survey," in *Proc. Secur. Commun. Netw.*, vol. 2018, [Online]. Available: https://doi.org/10.1155/2018/3759626

[40] Library for Virtual Machine Introspection, 2018. [Online]. Available: LibVMI. http://libvmi.com

[41] OpenStack, 2014. [Online]. Available: https://www.openstack.org/

Noor Ahmed received the BSc degree from Utica College, in 2002, the MSc degree from Syracuse University, in 2006, and the PhD degree from Purdue University, in 2016, all in computer science. He is currently a computer scientist with AFRL/RIS since 2003. His research interests include blockchain applications, security/privacy and quality of services issues in service oriented architectures, reliability and resiliency on cloud computing platforms with emphasis on moving target defense.

Bharat Bhargava (Fellow, IEEE) is currently a professor of computer science with Purdue University. He is the founder of the IEEE Symposium on Reliable and Distributed Systems, IEEE conference on Digital Library, and the ACM Conference on Information and Knowledge Management. His research interests include the security and privacy issues in service oriented architectures and cloud computing, and internet scale routing, and mobile networks.