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

Noor Ahmed and Bharat Bhargava, *Fellow, IEEE*

Abstract—Advances on resiliency to arbitrary faults and system failures have contributed well established, sound protocols and paradigms in distributed systems literature. The corner stone of this contribution lie redundancy/replication techniques in which is a double–edged–sword, by increasing the number of nodes inherently increases the system's attack-vector – the set of ways an attacker can compromised 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 paper, we introduce a bio-inspired formal model for space/time system randomization and diversification. and a quantification scheme for virtual machines in cloud computing environments. We show the practicality of the model with a MTD framework *(Mayflies)* integrated into cloud management software stack *(OpenStack)*.

Index Terms—Cloud Computing, Security, Byzantine Fault Tolerant, Software Defined Networks, OpenStack, Moving Target Defense.

1 Introduction

The traditional defensive security strategy for distributed systems is to safeguard applications against malicious activities or prevent attackers from gaining control of the system using well established defensive techniques such as; perimeter-based fire walls, redundancy and replications, and encryption. Although these techniques have been widely adopted, given sufficient time and resources, especially, sophisticated threats that target zero-day exploits, all of these methods can be defeated. While defensive security strategies against arbitrary faults and system failures for distributed systems have been studied for decades, defending against sophisticated adversaries still remains challenging. This is due to the fact the security motto is based on staying one-step ahead of the attackers.

With the ever increasing adoption on cloud computing, due to its simplified service-based management model built on commodity off–the–shelf hardware and software components, cyber threats have risen in recent years. Moving Target Defense (MTD) [1], is a defensive strategy that aims to reduce the need to stay one-step ahead against cyber threats by disrupting attackers gain-loss balance of the system. The core of this 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.

- *N. Ahmed is a Computer Scientist at AFRL/RI, Rome NY 13411. E-mail: ahmed24@purdue.edu*
- *B. Bhargava is with Purdue University, W. Lafayette, IN, 47906. E-mail: see http://cs.purdue.edu/homes/bbshail*

Manuscript received xx xx. 2018; revised xx xx. 2018; xxxxx. Date of publication 0 . 0000; date of current version 0 . 0000. (Corresponding author: Noor O. Ahmed.) Recommended for acceptance by J. Cao. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. xx.xxxx/TCC.2018.xxxxxxx

For decades, randomization and diversification techniques have been applied to all aspects of the system to combat against specialized threats that target memory structures, CPU registers, applications and networks. These include; Instruction Set Randomization [3], Address Space Randomization [4], randomizing runtime [5], and system calls [6] that have been used to effectively combat against system-level exploits *(i.e., return-oriented/code injection)*. These randomization techniques are considered mature and tightly integrated into most modern operating systems. Diversification techniques such as N-Version programming [7] aims to diversify variable binary forms of the same program, and N-Variant Systems [8] execute multiple variants of the same system in synchrony with a given input and monitoring for divergence to combat against application-level threats.

Recent advances in *Software Defined Networks (SDN)*, the core building blocks of the cloud networking, have further amplified atacks on the systems in cloud platforms *Virtual Machines (VMs)*. *SDN* separates the data plane and the control plane to allow the network functionality to be dynamically programmed in order for the VMs to be managed independently from the network interfaces, thus, increased the attack surface of the cloud infrastructures (i.e., network poisoning) [22]. To remedy this issue, a network-level randomization techniques, referred to as *IP-Hopping*, an MTD solution scheme to combat against such exploits has been proposed in recent years [10].

Furthermore, to prevent attackers from gaining full system control, VM-level randomization and diversification across cloud platforms has been introduced in early framworks such as TALLENT [11] and MARCO [14]. Recently, *Mayflies* [18], an MTD framework integrated into the cloud software stack (OpenStack) developed by the same authors is introduced that allows VMs withstand against attacks in short time-intervals in the hope of limiting their window of exposure by continuously substituting VMs with different characteristic (i.e., OS) across platforms. The overarching goal of VM randomization and diversification frameworks is to disrupt adversaries' gain/loss balance of the system by continuously shifting the attack surface, however, a formal model to reason about the system behavior have not yet been sufficiently explored, thus, the focus of this paper.

In this paper, we propose a bio-inspired formal model (consensus of the species' populations model [15]) for space/time system VM randomization and diversification using *Dynamic Bayesian Network* [**?**]. To illustrate the efficacy of the proposed model, we first discuss the practical implementation of VM randomization and diversification framework (*Mayflies*) introduced in our previous short papaer [18], then present the proposed model and its quantification scheme.

We make two contributions in this work:

1) A simple but yet effective space/time randomization algorithms on virtualized cloud platforms.
2) A sound theoretical foundation to mathematically reason about MTD system randomization behavior and a quantification scheme using well established tools and techniques.

We have organized this paper as follows, we first give a quick background of the topic in section 2, then discus an VM randomization and diversification MTD framework to illustrate the practicality of the proposed model in section 3. We present the proposed formal model and its quantification scheme in section 4. Finally, the conclusion and future work is discussed in section 6.

2 Background

Advances on resiliency to arbitrary faults and system failures have contributed well established sound protocols and paradigms in distributed systems, however, resiliency against sophisticated attacks still pose a challenging task. This is due to the fact that replication/redundancy is the corner stone of building reliability guaranteed fault-resilient systems, however, this solution approach is double-edged-sword in which increasing reliability through replication increases the system's attack-vector (more nodes to protect).

The criticality of diversity as a defensive strategy in addition to replication/redundancy was first proposed in [12]. Diversity and randomization allow the system defender to deceive adversaries by continuously shifting the system's attack surface – the set of ways/entries an adversary can exploit/penetrate the systems [2].

In general, space/time randomization and diversification techniques is simply transforming the traditional services that are designed to be protected their entire runtime to services that deal with attacks in *time intervals* through restarting/refreshing or migrating across platforms. Such transformation is simply achieved by allowing the applications run in heterogeneous OS's on variable underlying computing platforms (i.e., hardware and hypervisors), thereby, creating a mechanically generated system instance (s) that are diversified in time and space which is considered as good defense as type-checking [13]. To the best of our knowledge, a formal model for system randomization and diversification has yet been explored, thus, the focus of this work.

Inspired by the consensus of the species' populations model first introduced in [15] and further studied in insects in [16], the *preys'* population is measured by the proportionality of their survival/reproductive rate vs. their eaten rate by *predators*. Analogous to Virtual Machines (VMs) on cloud computing environment, the *preys* population imply the *systems/VMs* and the *predators* imply the *attackers*. Thus, we can quantify the species' population (VMs) in terms of their survival from exploits/attacks *(eaten)* vs. reproductive/replacement rate at any given time.

The principle cornerstone of this model is to effectively control the VMs survival/reproductive rate in order to guarantee desirable *prey/VM* population in a desired state at all times using a *Moving Target Defense (MTD)* solution scheme. MTD is a defensive strategy to refresh VMs by randomizing and diversifying across platforms in time intervals for the hope of keeping them away from exploits. As such, the proposed model allows formalizing MTD system behavior and to effectively control VM population by keeping the VM rate of changes, the refresh/reproductive time vs. attack success time (i.e., OS finger printing, code injection), in balance at all times for the system defenders' favor.

To illustrate the afficacy of the proposed model in virtualized cloud environment, we use *Mayflies* [18], an MTD framework built as an extension to Openstack cloud software stack [32], discussed in section 3.1. The main idea of *Mayflies* is to randomize/diversify VMs across heterogeneous cloud platforms/space in *time* intervals. We use *Library for Virtual Machine Introspection (LibVMI)* [17], an open source library for proactive monitoring VMs below the hypervisor, to detect in progress attacks by examining live memory structures, and also avoid randomizing VMs on vulnerable platforms (discussed in section 3.5). This is the driving engine for our quantification scheme, the survival/reproductive (VMs) rates and the eaten/compromised rates at any given time interval.

With this model, we consider systems are initially deployed in a *desired* state (known pristine VMs), then, it's possible that some of the systems transition into an *undesired* state (i.e., exploited/compromised), a valid assumption in cyber space. The overarching goal of the model in conjunction with any VM randomization and diversification framework is to formally reason the behavior of the systems on virtualized cloud environment, and quantify in terms of proportionality of the surviving vs. the compromised VMs between the *Desired* and *Undesired* states constructed with *Hierarchical Hidden Markov Model (HHMM)* and reasoned with *Dynamic Bayesian Networks (DBN)*.

3 VM Randomization and Diversification Framework

In this section, we give a brief overview of *Mayfies*, a VM randomization and diversification MTD framework introduced in our previous paper [18], then, discuss the practical implementation of VM replacement and network interface swapping (section 3.4) to lay the context of the proposed model (section 4). For those interested in the details of the framework design are suggested to reffer to our previous paper [18], and the proactive cloud monitoring scheme with *VMI* in [19].

3.1 Mayflies Overview

Mayflies [18] is a MTD framework built on top of OpenStack cloud software stack [32]. *OpenStack* is a widely adopted open source cloud management software stack that consists of a wide array of components such as; *nova compute, horizon*, and *neutron*, to simply cloud computing infrastructure management at scale with less user (admin) interactions. *Mayflies* adopts a cross-vertical design that operate 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 layer.

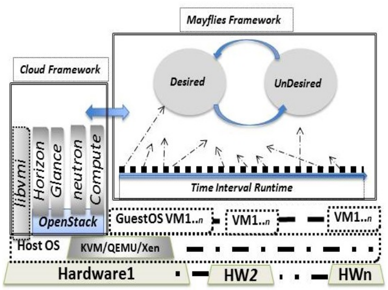

Fig. 1: High-level Mayflies architecture

Figure 1 illustrates the high-level architecture of *Mayflies* framework (top right) and OpenStack cloud framework components (bottom and left quadrant). In the cloud framework, starting from the infrastructure at the bottom layer lie the hardware. Typically, in each platform there is a *host* OS, a hypervisor (KVM/Xen) to virtualize the hardware for the *guest* VMs, and the cloud software stack (i.e., OpenStack) depicted as the vertical bars on the left quadrant. The core components we leveraged in this work include; *nova, neutron, horizon, and glance*. In addition, a *libvmi* [31], a library for virtual machine introspection to proactively monitor the VM's below the hypervisor by taking snapshot of the VMs memory at runtime. This is to detect attacks in realtime and guide the VM replacement decisions across platforms, discussed in section 3.5.

As the cloud software stack (OpenStack) abstracts the VM compute nodes from the application's architectural style (i.e., SOA) or it's communication model (i.e., synchronous vs. 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 application type. In each time-interval (as low as a minute) we destroy a VM and replace it with a fresh copy, discussed in section 3.4. The fundamental problem of VM replacement is the application state where the terminating VMs' state must be transferred to the freshly instantiated VM, discussed in section 3.3.

In *Mayflies*, we introduce two abstraction layers; a high-level *System State: Desired and UnDesired*, and *Application Runtime-level* abstraction, dubbed *Time Interval Runtime (TIRE)* as illustrated in Fig. 1 (top box) depicted in dotted line. This abstraction allows us to model both the high-level system states *desired/undesired* and the applications' time-interval runtime independently, thus, accurately reason the transition between the *Desired* and the *Undesired* states. The driving engine of the two high-level states transitions is observations from the *TIRE* abstraction layer, discussed in section 3.5.

3.2 Problem Formulation

As illustrated in Figure 1, given two high-level hidden states S $\{S_{Desired}, S_{UnDesired}\}$, we formulate the problem as a *Binary Random Walk* on the set of the 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 since the system is initially deployed in the *Desired state* before any attack tookes place, then in each time-interval (as low as a minute), we observe a random outcome of the system status as a coin flip, for example, we can be at 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)*. Similarly, the next time interval T_j, and so on.

However, for a typical system, the *UnDesired* state could 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, we 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. It's intuitive to see that these observations are probabilistic in nature.

Thus, we map the *Random Walk* probabilistic observations to the *Library for Virtual Machine Introspection (LibVMI)* intrusion detection observations discussed in section 3.5. to guide the VM randomization priorities, and reason the high-level system state transitions. Although we used *Mayflies* MTD framework with *LibVMI* to illustrate the efficacy 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 in this model.

3.3 Application State

As any MTD framework, *Mayflies* partitions the traditional runtime execution of the system by terminating/destroying the VM and replacing it with another freshly spawned VM. The inherent challenges of this runtime partitioning are *a) dealing with the application state transfers, and b) 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* vs. *asynchronous*) among the applications/services or the client and the servers, therefore, *Mayflies'* VM randomization is only suitable for certain applications.

In *Mayflies*, we exploit the built-in reliability properties of the applications, especially, replicated systems. In [20], we deployed an implementation of a *Byzantine Fault-tolerant*

System (BFT-Smart) [26] in *Mayflies* on a private cloud setting built on *OpenStack*. *BFT-Smart* is a 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 replica upon crash or failure. Replacing a VM in this system setting only requires injecting the updated configuration files, and the recovering/replaced VM connects to the rest of the replicas to synchronize before even the clients connect to it.

Applications like RESTful web services, a *asynchronous* service model for example, a stateless web service (client/server) model where the client requests are processed and responded by the servers without any system state is preserved. In this applications, the communication protocol bound to the client/server or between services attempts to reconnect when the VM is terminated and a new/fresh instance is activated in a timely manner. 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 of these limitations. In general, transferring VM application state (i.e., TCP connections, security sessions, etc.) in a generic fashion is not feasible, thus, *Mayflies* VM randomization is an application dependent.

3.4 VM Replacement

Inspired by the cloud software stacks' VM replacement scheme *VMentry and VMexit* employed by the hypervisors' scheduler for mapping the virtual computing resources to the physical resources (i.e., CPU, memory), the VMs are paused/stopped without the applications knowledge or even migrated to different platforms to load balance the infrastructure. *Mayflies'* VM replacement scheme is simply a) detaching the network interface of an active target VM, b) destroying/terminating the VM using the cloud software stacks' command line interface (CLI) *nova-create VM, nova-destroy* which is designed for provisioning and deprovisioning VMs, then, c) attaching the network interface to a fresh/new VM. Note that such VM replacement strategy is only suitable for certain applications as discussed in previous section.

Figure 2 illustrates the conceptual cross-section view of a cloud infrastructure building blocks where *OpenStack* is at the inner core of the cloud ecosystem. and *Mayflies'* continuously substitutes guest VMs (fourth ring) while proactively monitoring the VMs below the hypervisor (ring 2) and simultaneously reprogramming network interfaces with *Software Defined Network (SDN)* (outer rings). It's intuitive to see how the VMs are destroyed and activated fresher copies across hardware platforms while dynamically swapping their network interfaces, thus, reason the transitions between the high-level system state *(Desired* and *UnDesired)*.

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 in line 2, then, destroy the target VM in line 3. Swap the network interfaces in line 5 (described in algorithm 2 below), then copy back the configuration files in line 6.

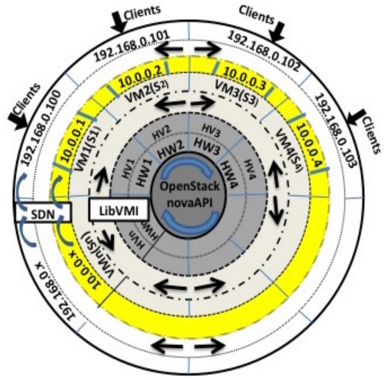

Fig. 2: Cross Section View of Cloud Infrastructure. At the core inner circle is *Openstack*, the second ring depicts the hardware and the hypervisors/*host OS* on the third ring, and one or more *guest* VMs on each *host* show on the fourth ring. The outer two rings depict the internal IPs (10.x.x.x), referred to *Fix* IPs and the externally visible IPs (192.x.x.x), referred to as *Floating* IPs

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**

3.4.1 Network Interface Replacement

Effectively terminating a VM and replacing it with a fresh new VM in a timely manner is simplified by the *Software Defined Networking (SDN)*, a programmable networking fabric that decouples the control plane (i.e., virtual routers and switches) from the data plane. In SDN environment, the active VM is attached to a virtual network interface that is referred to as *ports* with a *fix* IP for internal access (among the servers), and a *floating* IP for external access that can be later associated to the *ports*. 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* off of the target VM *(VMx)*, then get *VMy* from the prepared pool of VMs with all the application and it's configuration files installed and attach the *port*. Once the network port is attached to the new VM, then we inject all the necessary application runtime state info of the terminated target VM.

Algorithm 2 shows the network interface swap procedure. In algorithm 2, we first check if the new VM from the

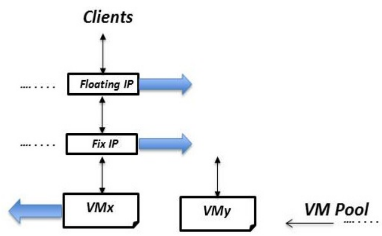

Fig. 3: Illustration of VM compute and SDN interface interchanges. VM_x seamlessly replaces VM_y from a pool of VMs.

VM 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 *Floating IP* of the terminating VM to it in line 5. Note that the *<options>* for *port-create/attach* includes creating the interface with specific IP address. We dis-associate the *Floating IP* if the VM has network interface in line 8, then swap the interfaces in lines 9 and 10. We finally associate the known IP to it in line 11. This allows the servers/replicas to continue using the known IP and the clients re-connect to this replica through it's floating IP (*192.x.x.x*) as the old server/replica had dropped off of the network and came back.

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**

Typically, the new VM *(VMy)* has different characteristics (i.e., Windows OS or variable Linux-based OSs (ubuntu/Feodra)) than *VMx*, the target VM that is getting destroyed. Note that the substituting VM can be from a pool of prepared VMs without network interfaces or created on demand. The pros and cons of the VM selection strategy is discussed in our previous paper [19]. Furthermore, depending on the OS image of the replica, a VM reboot is required after the `nova interface-attach <options>`.

3.4.2 Network Interface Replacement Challenges

The process of replacing a node in *Mayflies* is greatly simplified by the combination of *nova* for provisioning/deprovisioning VMs, and *neutron* to the dynamically program 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 replica 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 means that the interface does not exist and can proceed to the next function call, thus, prevent us to blindly wait function result in such environment.

3.5 Space/Time Replacement and Observations

As illustrated in Fig. 4. below, at least one node/VM is terminated and activated/replaced with a new one with different characteristics (i.e., OS) on a different platform/host *(y-axis)* in each time-interval *(x-axis)*. This time unit can be as low as a minute (system time unit) or upon completing certain number of *n* transactions/service responses in which translates to the time it takes to complete *n* transactions(i.e., minutes). Depending on the threat model, an effective VM replacement strategy is to randomize or in round robin fashion, however, in order to prevent from blindly replacing VMs on vulnerabile platforms/configurations (i.e., OS), we use *Library for Virtual Machine Introspection (LibVMI)* [31], an open source library for live memory introspection.

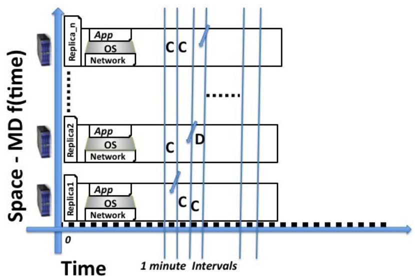

Fig. 4: An illustration of space/time Randomization. We mark *C* for clean if the VMs' memory integrity is intact as its deployed, and *D* for dirty if the memory structure changed.

Since we are interested in realtime proactive attack detection, we leveraged *vmi* to detect memory structural changes that is caused by certain attacks (i.e., code injection). The implementation details and the afficacy of the live memeory snashot capturing technique is described in our previous paper [20]. The idea is to run the VM in a sandox for profiling its memory structure (start and end offset addresses) at runtime, and then compare the initial memory references to the consequence snapshots as shown in Algorithm 3. Capturing the VMs live memory has neglegible performance impact [17], furthermore, it's intuitive to see that the memory start and end memory address offsets is also has a negligible performance impact, since it's just a basic string comparison.

Algorithm 3 shows the process of memory introspection process. 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, mark accordingly if the VM's address offsets differ/altered from the initially recorded offsets in lines 8, 9, 10 and 12. The comparison is simply checking the start/end address offsets shift. The VMs that their memory structure altered are given priority regardless of VM replacement strategy (random or round robin) adopted.

Algorithm 3 Virtual Introspect

1: **Input:** $node$
2: **Output:** $Clean$ or $Dirty$
3: **procedure** INTROSPECT($node$)
4: **if** $node == new$ **then**
5: $initialProc \leftarrow GetProcessMemory(node)$
6: **nodeStatus** $\leftarrow Clean$
7: **else**
8: $currentProc \leftarrow GetProcessMemory(node)$
9: **if** $initialProc_i(key, val) \neq currentProc_i(key, val)$ **then**
10: **nodeStatus** $\leftarrow Dirty$
11: **else**
12: **nodeStatus** $\leftarrow Clean$
13: **end if**
14: **end if**
15: **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 over a given time period. This allows us to have full control over the *preys/VMs* population within pre-specified time frame.

4 Formal Model

In this section we first describe the proposed model and the rationale behind our choice. We then discuss the *Time Interval Runtime Execution* scheme and introduce the proposed *DBN* model construction and the formulation of the high-level system state transitions. We discuss the model quantification in section 5.

4.1 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 [21], 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 adopt the probabilistic FSA (PFSA) model.

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 input string $\sum$ for our automaton, however, we use the output alphabet donated by Λ where $a \in \Lambda$ and is generated by simply observing the system's active nodes in time intervals.

Thus, we consider the *Time Interval Runtime Execution (TIRE)* observations to represent the output alphabet $a \in \Lambda$ that drives the high-level system state *Desired/UnDesired* transitions, discussed in section 4.3. These probability observation outcome 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 the time-interval runtime execution 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 characteristics of the structure of this automata are used to classify its properties. We achieve such structured characteristics by modeling the framework with PSFA, specifically, a *Hierarchical Hidden Markov Model (HHMM)* [24] represented with *Dynamic Bayesian Networks (DBN)* [25], a time-linear representation of *Hidden Markov Model (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 anbales to extend to other formal automata models such as; interface automata [27], virtual machines [28], cloud framework [29], and attack surface [30]. As such, the proposed model fills the gap for formally modeling an end-to-end system spectrum in the cloud ecosystem.

4.2 Time-Interval Runtime Execution (TIRE)

The *Time-Interval Runtime Execution (TIRE)* is an abstraction layer to break the runtime execution into intervals where each interval the system is assessed for its current high-level state, *desired* state or *compromised/failed*). Formally.

Definition 1. Runtime Execution of distributed systems is typically defined as a set of infinite sequences of states in Q, donated by Q^ω.

We define *time-interval* as follows:

Definition 2. Time-Interval in Mayflies is defined as a time unit. We use T_i to donate each time interval where *i=1,2,3...* are minutes/hours which is the prespecified *lifespan* of the VM.

TIRE is simply the break points of the infinite sequences of states in Q^ω. In each time-intervals T_i where *i=1,2,3* $\ldots$, at least a node n_i is replaced to n'_i, thus, the execution sequences for n_i will be those $\{q_0 \ldots q_{i-1}\} \in Q^i$ generated within T_0 to T_{i-1} time interval, then the execution sequences for n'_i will be those $\{q_i \ldots q_j\} \in Q^j$ of T_i to T_j where $i < j$, and so on. Thus, the runtime sequences of n_i, n'_i, n''_i, $\ldots$ are isolated in the form of $\{Q^i_n, Q^j_n, Q^k_n, \ldots\}$ $\in Q^\omega$, thereby, allowing us to safeguard the individual VM in time intervals rather than it's entire runtime in which is proven to be defeated eventually.

While we pro-actively monitoring the system at the hypervisor-level (below the OS) for runtime integrity violations, at any time interval of $T_i, T_j, \ldots T_n$ we determine whether or not we observed a violation, if a violation is detected (i.e., altered the applications internal memory structure/offset using VMI), then we replace the comprised VM(s) before they reach their predefined *lifespan* so we will be in our *desired* state in the next time interval. One way to formalize and model this probabilistic observations O (discussed next) of whether a VM status has changed or not is through a *Hidden Markov Model (HMM)*.

A Markov chain or 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 [23].

4.2.1 TIRE Observations

Formally, let $\{O_j$, j=1,2,$\ldots\}$ be observations of the VM status $n \in N$, where N is the set of nodes. We model these observation as a Bernoulli processes where $O_j \in \{0,1\}$ in which $O_j = 1$ indicates an observed VM is *clean* and $O_j = 0$ indicates the VM is *dirty*. The *dirty* VM can be either missing (i.e., network drop) or it's *compromised* (i.e., VMs address space altered).

Formally, let *n* be a node in *Mayflies* and is defined by a tuple: $n_i = \langle n_{start}, n_\rho \rangle$ where

- $n_{start} \in \mathbb{R}^+$, represent the real time the node starts.
- $n_\rho \in [n_{start}, < \rho | O^t_i >]$, represent the *lifespan* of the VM from the start to the end. Either naturally reaching it's *lifespan* ρ (no attacks) or terminated prematurely based on the observation result at time O^t_i time-interval t due to attacks. Observations $O_i \in [0,1]$ represent the VM is found to be *inactive=0 or active=1* (i.e., Dirty or Clean), thereby, is terminated accordingly.
- n'_{start}, $n'_{\rho'}$, represent the real time node *n* replaced to n' with a new predefined life expectancy ρ', thus, it's n_j tuple; $n_j = \langle n_{start}, n_\rho \rangle$

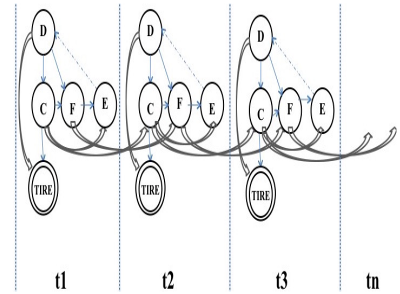

Fig. 5: Mayflies DBN System Model – system states are Desired, UnDesired($Compromised, Failed$) labeled as D, C, and F, followed by the Exit state E. The dotted lines on E depict for the control returning to the parent node. TIRE is the observing state in double circles.

4.3 Model Construction

Typically, we deploy a system in a *desired* state and at some point in time we end up in *undesired* state (i.e., compromised or failed) without our 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 Hierarchical Hidden Markov Model (HHMM) [24].

As the name implies, a Hierarchical Hidden Markov Model (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. 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.

As depicted in Figure 5, we construct an HHMM in which the hidden states S are *Desired*, *UnDesired* and *Time Interval Runtime Execution (TIRE)* as the omitting/observable state (discussed next). We define the topology of the *HHMM* hierarchy as follows: The *Desired* state (D) as the root state (i.e., initial state), the *UnDesired* set of states *Compromised* (C) and *Failed* (F) in level II, and *TIRE* as the leaf state in level III. Note that the *Undesired* state can have as many states as needed at all levels.

With this HHMM construction, we model the system with *Dynamic Bayesian Network* [**?**]. As depicted in Fig 5, DBN represents HHMM with time-linear transition partitions 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 of the model is sufficient to illustrate *Mayflies'* MTD objective.

We define S_{TIRE} *emissions* as VM status observation captured by the proactive monitoring library at the hypervisor level (i.e., VMI) in time intervals, say every minute. We consider the following three observations:

- A node is *active* which is typically the initial state when the system is deployed.
- A node is *inactive* which can be either not-reachable due to network drop or hardware/software failures.
- A node is *dirty* due to runtime integrity violations, (i.e., detected anomaly in the applications memory).

For simplicity, we treat both *in-active* and *dirty* as *Dirty* and *active* as *Clean* as described earlier. 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 *inactive*, or to a *Compromised (S_C)* state if S_{TIRE} emit *dirty*. Otherwise, stays in *(S_D)*, i.e. *VM (s) is active*.

To illustrate how we map the VMI observations to the high-level system states, consider at time *t=1* in Figure 5, 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. At this point, we refresh the compromised VM and asses the sytem so the next time in *t=2*, the S_{TIRE} emits *active* and the system transitions to S_D at *t=3*. Thus, 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), and quantify it in terms of the overall proportion of the time *{t_2, t_i, t_k, ...}* the system was in compromised state (discussed in section 5).

4.4 State Transition Probabilities

As illustrated in Figure 5, we defined three hidden states S_D, S_C and S_F and an observable state S_{TIRE} that omits observation probabilities. 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 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 being in our *desired* state at all time.

4.4.1 TIRE Transitions Propabilities

Time Interval Runtime Execution (TIRE) transition function is simply a real number, time assigned to the structure which breaks the system runtime into manageable intervals (i.e., one minutes intervals). Thus, we define the transitioning function as:

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

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, ...}* 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.

Property 1. An HMM observation *o* is a logical predicate over *Mayflies*. 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-versa.

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:

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

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

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

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

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

Modeling TIRE as an observable HMM and formulating it in this manner enable us to anticipate the high-level hidden state transitions in which the probability of transitioning to an *undesired* state in T_i can go either way *(i.e., desired/undesired)*. We anticipate this outcome if it results against our favour 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 underlying runtime execution is preserved if these invariants hold.

4.4.2 High-level 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:

$$P(T_0, T_1, T_2, \ldots, T_{i-1}, o_{i-1} = S_{Desired}, o_i = S_{Compromised})$$

$$= \alpha_{T_{ij}}(S_{Desired}) Pr\left(o_i = S_{Compromised} | o_{i-1} = S_{Desired}\right)$$

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:

$$P(T_0, T_1, T_2, \ldots, T_{i-1}, o_{i-1} = S_{Desired}, o_i = S_{Failed})$$

$$= \alpha_{T_{ij}}(S_{Desired}) Pr\left(o_i = S_{Failed} | o_{i-1} = S_{Desired}\right)$$

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.

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.

5 Model Quantification

In the species' populations model [15], the *preys* population is measured by the proportionality of their survival/reproductive rate vs. their eaten rate by *predators*. The principle cornerstone of this model is to observe the *preys'* reproductive rate in order to project their extinction. It's intuitive to see our quantification scheme follows the same principle, effectively controlling the systems' health status in realtime with the MTDs' VM replacement/reproductive defensive strategy and *LibVMIs* attack detection scheme, we aim to project VM (population) extinction/compromised at all times.

We assume VMs in *Mayflies* start with pristine status where they perform computations within a predefined *lifespan* and being replaced by the end of that lifespan (as low as a minute). The key objective is to ensure VM populations with shorter attack window of exposure exists in *Desired* state as often as possible. With this controlled VM population environment, the MTD framework allows to reason the behavior of state transitions to any of the *Undesired* states. Hence, we are interested in the long-run distribution of the process/runtime execution (i.e., one hour), for example, the long-run proportion of the time *T* that we are in the *Desired* state overtime to quantify the preys/VMs desired population.

Formally, let S_{T_i} represent the *state* of the system at time *T* where *i=1,2,3…hours*, and the time-intervals of the VM replacement t_j where *j = 0,1,2,…N* is within *T* where *N* VMs/nodes are replaced and observed within that time interval T_i. Clearly, the inherent cost of *N* VM replacement on cloud platforms *(OpenStack)* is the upper bounds of the time requires to randomize VMs on the cloud in contrast to the attackers cost of crafting an attack. Thus, the two competing time to fully control VM population in *Desired* state is the following:

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

Intuitively, for any MTD defensive strategy, in order to guarantee for the system stay in a *Desired* state as often as possible (desired VM population), the defensive cost has be less than the offensive/attackers' cost.

$$RC(T) + OC(T) < AC(T)$$

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, then:

$$RC(T_i) = \sum_{j=1}^{n} \nu_j$$

and

$$OT(T_i) = \sum_{j=1}^{n} \mu_j$$

where *RC(t)* and *OC(t)* is the total cost of the MTD defensive strategy of one time interval *T* (i.e., one hour) being replaced *N* VMs and observed. Let $p_{q_{ij}}$ denote the probability of going from state q_i to q_j in one step, and λ_i represent the matrix *P* whose entries are the p_{ij}. For each state S_i, we define:

$$\lambda_i = \frac{\sum_{j=1}^{n} S_i}{T_i}$$

where $\sum_{j=1}^{j} S_i$ is the total number *n* of visits the process makes to each state S_i over the time-intervals $T_i \in T_0, T_1, T_2 \ldots T_i$. 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 frames. 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 about the high-level system states in any time intervals, 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.

6 Conclusion

We introduced a formal model for Virtual Machine (VM) space/time diversification and randomization across cloud computing platforms. We presented *Mayflies*, a bio-inspired MTD framework, to illustrate the practicality of the model and a quantification scheme in *Openstack* private cloud software stack. We described the implementation details of VM replacement using *nova* API desinged for provisioning/deprovisioning VMs, and *neutron* for dynamic network interface swapping. For future work, we consider modeling a realistic experiments for applications deployed in a private cloud setting.

Acknowledgments

Authors would like to sincerely thank Jim Hanna for his support on the frameowrk. Special thanks to Dr. Mark Linderman, Steven Farr and Lt. Col. Scott Cunningham at AFRL for their continuous support and guidance.

References

[1] Jajodia, S., et. al., (2011). Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats. volume 54, Springer 2011.
[2] Manadhata, P.K., and, Wing, J.M., (2011). An Attack Surface Metric In the *IEEE Trans. on Software Engineering, 37, 371-386, 2011.*
[3] G. S. Kc, A. D. Keromytis, V. Prevelakis. Countering CodeInjection Attacks with Instruction-Set Randomization. In the *Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03), 2003-Oct.*
[4] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In the *Proc. 12th USENIX Sec. Symp., pages 105-20. USENIX, Aug. 2003.*
[5] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In the *Proceedings of Computer Security Applications Conference 2006. ACSAC '06. 22nd Annual, pp. 339-348, 2006.*
[6] Rauti S., LaurÃl'n S., Hosseinzadeh S., MÃd'kelÃd' JM., Hyrynsalmi S., LeppÃd'nen V. (2015) Diversification of System Calls in Linux Binaries. In: Yung M., Zhu L., Yang Y. (eds) Trusted Systems. INTRUST 2014. Lecture Notes in Computer Science, vol 9473. Springer, Cham
[7] Chen, L., and, Avizienis, A., (1978). N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation. In the *Digest of the 8th International Symposium. on Fault-Tolerant Computing, 1978, pp 3-9.*
[8] Cox, B., Evans, D., Fillipi, A., Rowanhill, J., and, Hu, W., (2006). N-variant Systems: a Secretless Framework for Security Through Diversity. Defense Technical Information Center, 2006.
[9] Hong, S., Xu, L., Wang, H., and Gu G., (2015). Poisoning Network Visibility in Software-Defined Networks: New Attacks and Counter Measures. In The *Proceedings of Network and Distributed System Security Symposium (2015), pp 8-11..*
[10] Jafarian, J., Al-Shaer, E., and Duan, G., (2012) Open Flow Random Host Mutation: Transparent Moving Target Defense Using Software Defined Networking. In The *Proceedings of The 1st Workshop on Hot Topics in Software De*
ned Networks, pages 127-132. ACM, 2012.
[11] H Okhravi, A Comella, E Robinson, J Haines Creating a cyber moving target for critical infrastructure applications using platform diversity International Journal of critical infrastructure protection, 2012 - Elsevier. Volume 5, Issue 1, March 2012, Pages 30-39
[12] Forrest, S., Somayaji, A., and Ackley, D., (1997). Building Diverse Computer Systems. In *The 6th Workshop on Hot Topics in Operating Systems, pages 67-72. IEEE, 1997.*
[13] Schneider, F., (2010). From Fault–tolerance to Attack Tolerance. http://www.dtic.mil/dtic/tr/fulltext/u2/a548748.pdf Dec 2005 – November 2010.
[14] Carvalho, M., Eskridge, T., Bunch, J., Bradshaw, M., Dalton, A., Feltovich, P., Lott, J., and Kidwell, D., (2013). A Human-agent Teamwork Command and Control Framework for Moving Target Defense (MTC2). In *Proceedings of The 8th Annual Cyber Security and Information Intelligence Research Workshop, page 38. ACM, 2013.*
[15] Sweeney B., (1987). Mayflies and Stoneflies: Life Histories and Biology. In *Kluwer Academic Publisher, 1987.*
[16] Sweeney B. and Vannote R. Population Synchrony in Mayflies: A Predator Satiation Hypothesis. In *Evolution, 36:810-821, 1982.*
[17] Garfinkel T., and Rosenblum M., (2003) A virtual Machine-based Architecture for Intrusion Detection. In *Network and Distributed System Security Symposium, Sandiego, California USA, 2003, pp. 191–206.*
[18] Ahmed, N., and Bhargava B., (2016). Mayflies: A Moving Target Defense Framework for Distributed Systems. In the *Proceedings of the 2016 ACM Workshop on Moving Target Defense. ACM, 2016.*
[19] Ahmed, N., and, Bhargava, B., (2015). Towards Targeted Intrusion Detection Deployments in Cloud Computing. In the *International Journal of Next-Generation Computing Vol. 6, No 2: IJNGC - JULY 2015.*
[20] N. Ahmed and B. Bhargava. From Byzantine Fault-tolerant to Fault-avoidance: An Architectural Transformation to Attack and Failure Resiliency. In *IEEE Transactions on Cloud Computing. March 2018.*
[21] Lynch, N., and, Tuttle M., (1988). An Introduction to Input/Output Automata. In *Publisher PN, 1988.* MIT Technical Memo MIT/LCS/TM-373. http://groups.csail.mit.edu/tds/papers/Lynch/CWI89.pdf
[22] Hong, S., Xu, L., Wang, H., and Gu G., (2015). Poisoning Network Visibility in Software-Defined Networks: New Attacks and Counter Measures. In The *Proceedings of Network and Distributed System Security Symposium (2015), pp 8-11..*
[23] Lawrence R. Rabiner (February 1989). A tutorial on Hidden Markov Models and selected applications in speech recognition. In *Proceedings of the IEEE. 77 (2): 257âĂŞ286. doi:10.1109/5.18626*
[24] S. Fine, Y. Singer and N. Tishby. The Hierarchical Hidden Markov Model: Analysis and Applications" In *Machine Learning, vol. 32, p. 41âĂŞ62, 1998.*
[25] K. P. Murphy. Dynamic Bayesian Networks: Representation, Inference, and learning. PhD Dissertation. University of California at Barkley, 2002.
[26] Bessani, A., Sousa, J., and, Alchieri, E., (2014). State Machine Replication for the Masses with BFT-SMaRT. In the *Proceedings of the 44th Annual IEEE/IFIP International Dependable Systems and Networks (DSN), 2014.*
[27]
[28]
[29]
[30]
[31] Library for Virtual Machine Introspection (2018) LibVMI.http://libvmi.com
[32] OpenStack, (2014). OpenStack.https://www.openstack.org/

Dr. Noor O. Ahmed is a Computer Scientist at AFRL/RIS since 2003. He holds a BSc (2002) from Utica College, MSc (2006) from Syracuse University, and PhD (2016) from Purdue University, all in Computer Science. His research interests include: Security in Cloud Computing, QoS and Security in Service Oriented Architectures, Semantic Computing, and Reliability and Resiliency in Distributed Systems with spacial emphasis on Moving Target Defense (MTD). Dr. Ahmed serves as a program committee and session chairs for IEEE and ACM conferences/workshops in these research areas.

Dr. Bharat Bhargava is a professor of computer science at Purdue University and an IEEE fellow. His research work deals with the security and privacy issues in Service Oriented Architectures and Cloud Computing, and secure Internet-scale routing and mobile networks. Professor Bhargava is the editor-in-chief of four journals and serves on over ten editorial boards of international journals. Professor Bhargava 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.