January 2016

Design, Implementation and Experiments for Moving Target Defense Framework

Norman Ahmed
Purdue University

Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Norman Omar Ahmed

Entitled

DESIGN, IMPLEMENTATION AND EXPERIMENTS FOR MOVING TARGET DEFENSE

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Bharat Bhargava Chair	Leszek T. Lilien
Chris Clifton	
Xiangyu Zhang	
Vernon Rego	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Dr. Bharat Bhargava

Approved by: Sunil Prabhakar/William J. Gorman 5/4/2016

Head of the Departmental Graduate Program Date

DESIGN, IMPLEMENTATION AND EXPERIMENTS FOR MOVING TARGET DEFENSE

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Noor Ahmed

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2016

Purdue University

West Lafayette, Indiana

To my sons Ayden and Ardi Ahmed.

ACKNOWLEDGMENTS

Special thanks to my Ph.D. advisor, Bharat Bhargava and professors; Chris Clifton, Xiangyu Zhang, Vernon Rego and Leszek Lilien for serving on my Ph.D. committee. I also thank to my lab partners (Drs. Rohit, Pelin, Mehdi, and Nwokedi) for their collaborations and insightful discussions.

I would like to sincerely thank to Dr. Mark Linderman for being my mentor, and special thanks to Richard Metzger for suggesting me I pursue my PhD at Purdue University and supporting me along the way. I would also like to thank Steven Farr, Jim Hanna, Lt. Col. Mike Halick, and Julie Brichacek for their continuous support and being the driving force motivating me to accomplish this work with excellence.

I am ever so grateful to all my family, especially to my fiance Jess Etienne for being supportive during my doctoral studies, and my uncle Nuureyni Abooiye for his unconditional love and support during my early years of schooling.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

VMI	Virtual Machine Introspection
IDS	Intrusion Detection Systems
BFT	Byzantine Fault-Tolerant Systems
pub/sub	Publish and Subscribe
TIRE	Time Interval Run-time Execution
HMM	Hidden Markov Model
HHMM	Hierarchical Hidden Markov Model
DBN	Dynamic Bayesian Networks

ABSTRACT

Ahmed, Noor PhD, Purdue University, May 2016. Design, Implementation and Experiments for Moving Target Defense. Major Professor: Bharat Bhargava.

The traditional defensive security strategy for distributed systems is to safeguard against malicious activities and prevent attackers from gaining control of the system. The strategy employs well-established defensive techniques such as perimeter-based firewalls, redundancy and replications, and encryption. However, given sufficient time and resources, all these methods can be defeated by advanced adversaries.

To address this issue, this dissertation proposes an attack-resilient framework that employs a novel defensive security strategy to reduce or eliminate the need to keep one step ahead of sophisticated attacks. The core of our defensive strategy is to transform systems to narrow the window of their vulnerability from hours/days to minutes/seconds. This is achieved by controlling the system runtime execution in time and space through diversification and randomization as a means of shifting the perception of the attackers' gain-loss balance. The goal of this defensive strategy, commonly referred to as *Moving Target Defense (MTD)*, is to increase the cost of an attack on a system and to lower the likelihood of success and the perceived benefit of compromising it.

The proposed defensive security paradigm is covered in five chapters: Chapter 1 introduces the framework and its core building blocks, then highlights the key contributions of the dissertation. Chapter 2 presents a proactive monitoring scheme to safeguard application runtime below the OS. Chapter 3 presents the proposed framework, referred to as *Mayflies*, a bio-inspired MTD framework for distributed systems, and discusses the formal model, design, implementation and algorithms. In Chapters 4 and 5, we show the effectiveness of the proposed framework with two

classes of widely adopted replicated systems: quorum-based Byzantine Fault-Tolerant and Event-based Publish and Subscribe, deployed on a private cloud platform with special emphasis on their resiliency to attacks and performance impact.

1. INTRODUCTION

While defensive security strategies against arbitrary faults and system failures for distributed systems have been studied for decades, defending against threats from resourceful and advanced adversaries still remains challenging. 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, have amplified these threats.

The key challenge is that the traditional defensive solutions (i.e., instruction set randomization) are ad-hoc and designed to combat against specific threat, thus, limited in scope when attacks originate outside their intended defensive parameters (i.e., memory corruption or network layer attacks). The complexity of the building blocks of the cloud software stack and the inherent security risks of sharing hardware resources (i.e., multi-tenancy) have contributed more ad-hoc security solutions that are specific to a single threat (i.e., side channel attack/defenses). Thus, the conceptual view of such defensive strategy is to always attempt to stay one step ahead of the attackers.

To remedy these issues, it is critical to design a vertical solution from the application layer down to physical infrastructure in which the protection against attacks is deeply integrated across all the layers of the system (i.e., application, runtime, network) at all times, not as add on to a specific layer of the system for a given threat. This vertical solution approach enables defensive strategies to be applied simultaneously across the board, thereby, reducing or eliminating to continuously address a single exploit for one layer of the system while another exploit is in progress (i.e., cat and mouse game).

In this dissertation, we propose a generic attack-resilient framework for distributed systems that emphasizes a novel defensive security strategy that avoids threats in time

intervals rather than defending the systems' entire runtime. The framework's fundamental defensive strategy is to disrupt the attackers gain/loss balance by increasing the cost to attack the system and the perceived benefit of compromising it, commonly referred to as *Moving Target Defense (MTD)*. We achieve this by controlling the node's exposure window of an attack through 1) partitioning its runtime execution in time intervals, 2) allowing nodes to run only with a predefined lifespan (as low as a minute) on heterogeneous platforms (i.e., different OSs), while 3) pro-actively monitoring their runtime below the OS.

The overarching goal of this dissertation is to propose abstractions and paradigms for designing a generic attack-resilient framework for distributed systems on virtualized cloud platforms. The key contribution lies in how the building blocks and the theoretical underpinnings of the framework enable us realize our desired goal, *the ability to dynamically and safely transform nodes across platforms to disrupt adversaries gain/loss balance of the system control*.

In this chapter, we give a brief overview of Distributed Systems and Cloud Ecosystem, to lay the context of the design and the model of the proposed MTD framework. We then discuss; the threat model and assumptions we considered in this work, the three logical building blocks of the framework, and the attack-resiliency model and the quantification schemes. Finally, we present the key contributions of the thesis, followed by the thesis organization.

1.1 Distributed Systems

Distributed systems is a broad subject written in many books. In order to give a brief overview relevant to our framework, we group them into their respective high-level classes to discuss their architectural styles, resiliency attributes, and system models.

1.1.1 Architectural Styles

The two common widely adopted architectural styles are Service Oriented Architectures (SOA) and Component-based. A comprehensive comparison of their difference is given in [1]. For example, RESTful and SOAP-based services (i.e., web-services) are implementations of SOA-based system models that are widely deployed in large scale enterprise systems such as facebook and twitter. Component-based Architectures focus on building exchangeable software units (i.e., DCOM, JavaBeans, Enterprise JavaBeans, and CORBA), typically managed in containers (i.e., application servers). Architecturally supporting the dynamics of MTD schemes for defensive measures is system dependent and is only considered in post system design.

In general, the static and dynamic software component relationships dictate the flexibility to adopt changes at runtime. The use of dynamic architectures to support runtime modification is first proposed in [2]. Architectural-based solutions of runtime structural flexibility to handle faults, for example, self-adoptive [3] and self-repair [4] has also been considered.

Given the complexity of the underlying computing fabric of the cloud (i.e multiple independently developed and deployed components) where the applications are deployed, it is intuitive to see that the architectural-level defensive security solutions are ill-suited, for instance, in multi-tenancy, an escaped malicious VM from a different tenant can gain control of the system, in which the architectural-level defensive solutions does not able to protect.

1.1.2 Architectural Attributes

Detailed attributes of distributed systems architecture vary from system to system, but, we can identify them with two classes of system resiliency attributes; dependability and security. Attributes of dependable systems include; reliability (the ability for a system to recover from random errors/faults or continuity of correct service [2]), availability, maintainability, and safety. Typically, reliability is achieved with some

type of replication or redundancy and quantified with the Mean Time To Recover. Security on the other hand, are the protection schemes employed to safeguard the system from natural and man-made faults while achieving its intended objective for its legitimate users (i.e., access control).

A sound system architecture is typically described using standard language like the Architecture Description Language (ADL). Standards like ADL use formal notation to unambiguously formulate and represent the structure of a system. These structures comprise the software components, their externally visible properties and their relationships, referred to the software architecture [5]. The architectural style employed, the individual building block component relationship characteristics (i.e., static or dynamic), and the variable systems' quality attributes such as: reliability, safety/security, integrity, maintainability, availability, agility, functionality, etc. are the foundation for designing a resilient architecture.

The cloud management software stack, commonly referred to as *cloud framework*, abstracts the architectural attributes of the applications into the virtual machine computing instance that these application are deployed. Hence, our proposed frameworks' core design principle is to preserve these built-in resiliency attributes threatened by the modern sophisticated attacks while the VMs are on the move across heterogeneous platforms.

1.1.3 System Models

There are three classes of distributed systems; *Synchronous*, *Asynchronous*, and *Probabilistic*. The former two are those common in distributed systems deployed in cloud environment. For *Synchronous* systems, as the name implies, the interaction/communication protocols between the nodes (i.e., SOAP-based clients/servers model) are synchronized, where as the *Asynchronous* class communication protocols (i.e., request/response, push/pull data models) are not synchronized (i.e., the request is independent of the response in time/space).

Besides the standard-based lightweight services (i.e., web-services) that are widely adopted in the commercial sector and on social sites, the event-based *Publish and Subscribe (pub/sub)* and the *Quorum-based Byzantine Fault Tolerant (BFT)* systems are the two widely deployed protocols in the cloud environments and studied in the literature. The key design difference is that in pub/sub, typically, a broker(s) mediates the exchange of topic/content-based messages between the producers (publishers) and consumers (subscribers) of the information (i.e., stock trading apps, cloud internals), thus, it is an *Asynchronous* system. In contrast to the BFT systems where a number of replica need to process client requests in an ordered and *Synchronous* fashion.

These systems are designed and modelled with different replication models (i.e., chain, quorum and others) and failure models (i.e., Byzantine Faults). The proposed framework introduces a unified and generic system agility under the constraints of the replication and faulty model differences.

1.2 Cloud Eco-System

The key promise of cloud adoption is the cost benefits of the computing resources that can scale up/down on demand, referred to as "elastic computing". Infrastructure (IaaS), Platform (PaaS), and Software (SaaS) –as–a–Service are the three service deployment models for cloud environments in which each service model has different pricing, referred to as "pay–as–you–go". Each deployment model offer different protection schemes for the applications. Since our work involves in hypervisor-level interactions, *(i.e., virtual introspection)*, we developed our solutions on IaaS model.

While on-demand elastic computing and simplified service deployment models of the cloud are undeniably useful, it is increasingly challenging to guarantee provable reliability due to the sophisticated cyber attacks. (i.e., side channel, XML wrapping attacks and just–in–time return oriented programming) in recent years. The transient hardware and software design faults inherent in commercial–off–the–shelf hardware built on such infrastructures have amplified these threats.

On the other hand, by design, such infrastructures also enable unprecedented security capabilities through the structured underlying computing architecture and virtualization. The two key capabilities are the *isolation* and *introspection* which enables to safeguard systems below the OS (i.e., hypervisor layer). The application-level isolation enabled by the *container* technologies (i.e., Docker) allows multiple independent applications to run in isolation on a single OS as if there are on a separate OSs. OS-level *isolation* allows running multiple OS virtual instances on the same physical hardware, referred as *multi-tenancy* (i.e., Xen or Quick Emulator (KVM/QEMU).

At the heart of the Virtualization world is the Virtual Machine Monitor (VMM), referred to as a *hypervisor*. During the mediation of the resource virtualization by the hypervisor, mapping virtual requests from the guest OS to the available physical resources, one can peek into the low-level live memory and CPU activities for proactive monitoring, referred as *Virtual Machine Introspection* [6]. Thus, *isolation* coupled with *VMI*-based defensive security schemes offers unprecedented defensive capabilities that were not practical in non-virtualized domains, hence, the core building blocks of our framework.

Recent computing advances in cloud technologies enables hardware assisted security. These include; Intel's SGX processor that divides the CPU into many secure enclaves/sandboxes to protect applications from each other or even from a compromised OS; and the ARM TrustZone processor divides CPU into two halves, the insecure and secure worlds that communicate via a Secure Monitor Call instruction. In this dissertation, we develop the framework with publicly available cloud software stack, referred to as cloud framework, and commodity off–the–shelf hardware architectures without customization.

There are a number of cloud frameworks that simplify the management of the cloud platforms. These include; Eucalyptus, OpenNapula, OpenStack, Cloudstack and Nimbus. We built our framework on top of an OpenStack framework [7]. OpenStack was developed by NASA and RackSpace [8], a cloud provider who serves a well established businesses like Netflix. OpenStack is currently used by many organiza-

tions such as AT&T, HP, IBM for their private cloud offerings and their own internal systems.

1.3 Threat Model and Assumptions

Our attack model considers an adversary taking control of a node/VM by bypassing the traditional defensive mechanisms, a valid assumption in the face of novel attacks. The adversary gains systems' high privileges and is able to alter all aspects of the applications. Traditionally, the adversaries' advantage, in this case, is the unbounded time and space across the replicas to compromise and disrupt the reliability of the entire system. The commonly studied disruptive behavior for reliable distributed systems known as a *Byzantine Failure Model*, in which several compromised nodes deviate from the specified system protocol.

We consider a replicated systems model where the adversary can exploit many replicas in order to collude. Specifically, we consider adversaries that exploit systems with rootkits to compromise the OS. Because of our solution approach allows the replicas to exist for a short time in which that lifespan can be hard-wired in the application, we consider the adversary to attack the system any time, right from start of the replica.

We further assume the attacker takes a minimum time t to compromise a node n, and having seen or attempted to compromise n with a given tactic devised for a given exploit will not reduce the time to compromise a new node n'. This is because the new node n' will require new tactic and new exploit to compromise it given the fact that it starts with new characteristics such as different OS, on different hardware and hypervisor. Furthermore, the adversary can employ arbitrary attacks on the nodes in the replica group only. We do not consider attacks that targets the networking fabric of the compute instances *(i.e., SDN)* that can disrupt the routing table [9].

The fundamental premise of our framework is to eliminate the adversaries' advantage of time and space and create the agility to avoid attacks that can defeat system

objective by extending the cloud management framework. We assume the cloud management software stack, specifically, the *nova compute*, *galance*, and *neutron* for the software defined networking are secure. We further assume the virtual introspection libraries are secure and capable of capturing accurate live application memory.

1.4 Motivation

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 (i.e., need to protect more nodes). Figure 1.1(a) depicts the traditional replicated system view of the space (i.e., across platforms) (y-axis) and runtime (x-axis) (i.e., elapsed time).

The criticality of diversity as a defensive strategy in addition to replication/redundancy was first proposed in [10]. 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 [11]. In general, the idea of proactive diversity is to periodically randomize replicas in the hope of reducing windows of vulnerability. Notable examples are N-Version programming [12] that proposed techniques to produce computationally variable binary forms of the same program, and N-Variant Systems [13] for running multiple variants of the same system in synchrony with a given input and monitoring for divergence. However, these techniques are often defeated when attacks target the runtime-level *(i.e., return-oriented/code injection attacks).*

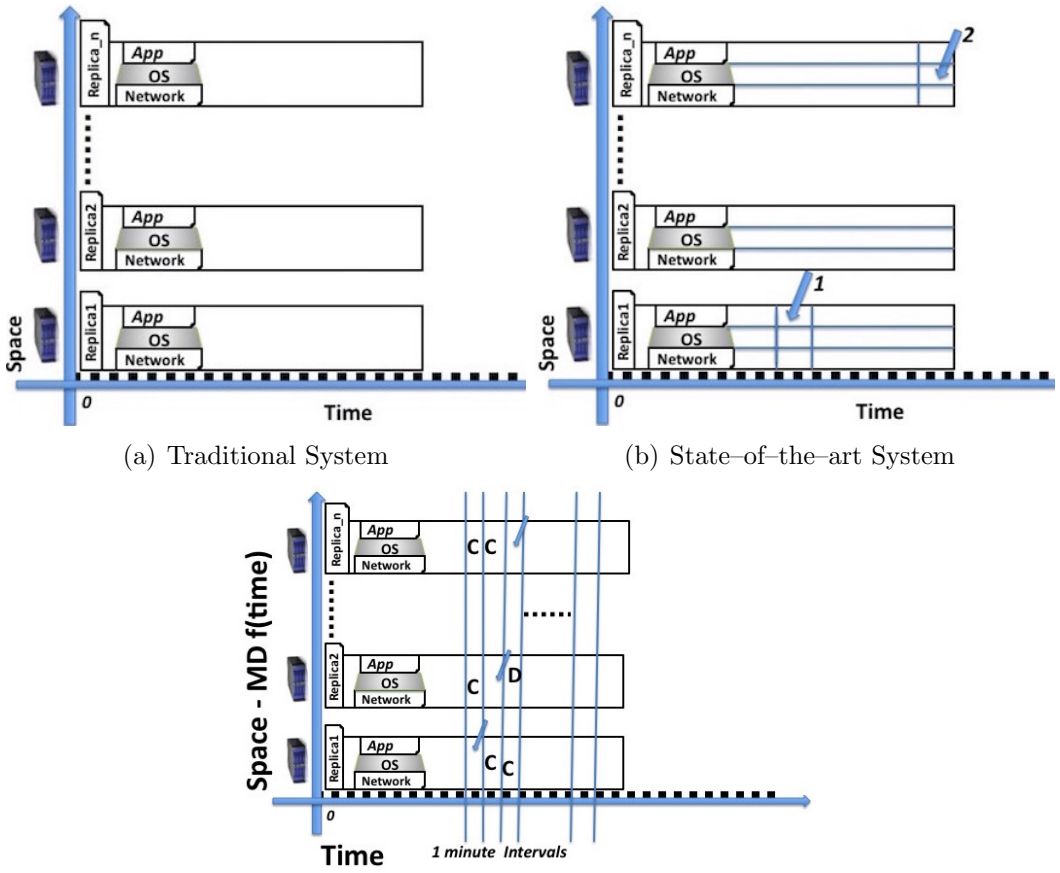
Code injection attacks target runtime execution with exploits such as buffer and heap overflows and control flow of the application (i.e., JIT-ROP). Techniques such as:

Instruction Set Randomization [14], Address Space Randomization [15], randomizing runtime [16], and system calls [17] have been used to effectively combat against these attacks. These runtime-level diversification and randomization defensive strategies are considered mature and tightly integrated into some operating systems.

Furthermore, with the increased adoption of *Software Defined Networks* as the core networking building block of the cloud, network level randomization techniques, known as *IP-Hopping*, to defend against network exploits (i.e., network poisoning) have been proposed recently [18]. These defensive security mechanisms, commonly referred to as a Moving Target Defense (MTD) [19], are independently designed to combat against specific threats in separate layers of the system; for high-level (i.e., application/replica), system-level (i.e., runtime), and network-level (i.e., IP-Hopping).

The fundamental problem of the the state–of–the–art MTD solutions are a one dimensional space-based system view. As illustrated in Figure 1.1(b), an MTD capability is typically applied on a single layer as depicted on the arrows on each cell, at the application layer (i.e., N-Version), in the space in any given time, where time is a continuous function. Applying most of these defensive MTD schemes (i.e., ASR or ISR-enabled OS) to systems in a given layer requires the system to be taken down in which typically takes minutes/hours. This allows the attacker with its unbounded time to exploit only one cell in the space to take control of the system (i.e., next zero day attack), until the vulnerability is discovered (which takes from days/months or years in some cases) and a new state–of–the–art solution is introduced to mitigate that threat. The race to stay one step ahead of the attacks continues.

We introduce a unified generic MTD framework designed to simultaneously move in space (i.e., across platforms) and in time (i.e., time-intervals as low as a minute) as shown in Figure 1.1.(c). Unlike the–state–of–the–art singular MTD solution approach, we view the system space as a multidimensional space where we apply MTD on all layers of the space (app, OS, network) in short time intervals while we are proactively monitoring (discussed in section 1.4.2) all the replicas in the system.



(a) Traditional System

(b) State–of–the–art System

(c) Proposed System

Fig. 1.1.: MTD solution space

As illustrated in Figure 1.1(c), our solution approach views the system as a Multi-Dimensional (MD) space as a function of time *f(time)*. In each time-interval, we terminate a node and activate a new one (depicted in the arrows) while we are observing the runtime of the other nodes and marking accordingly based on the proactive monitoring component results. We mark the node on either *Clean (C)* for a node whose internal runtime is intact or *Dirty (D)* for a compromised or a missing node, depicted C and D entries of the time-interval cells. To illustrate this concept, in Figure 1.1(c), in the 3rd termination round for $replica_n$ depicted on the arrow on the 3rd cell, for instance, we detect $replica_1$ to be clean and $replica_2$ as dirty as shown in it's time-interval entry *D*. In the next time-interval, we terminate $replica_2$ prior to any other

replica scheduled for termination. In general, the nodes whose entry show D takes priority over the scheduled node in each time-interval, thus, preventing the nodes to blindly move across platforms.

This defensive tactics makes the attacker's job difficult to compromise a node, for instance, by the time the reconnaissance of the node (i.e., OS fingerprinting), exploiting vulnerabilities (understanding app/memory layout), and crafting the attack (i.e., code injection attack) process completes, there is a high chance of the node changing in space upon carrying the attack. In the case where the attack was crafted earlier and succeeds in a short time, then, the node is under the control of the attacker for a short time since it gets terminated eventually in the subsequent time-intervals. If detected, then, we terminate that node instead the scheduled on, and learn to avoid that specific configuration for the next time interval.

The framework is built on randomization and diversification techniques, referred to as *Reincarnation*, discussed next, where the nodes are pre-packaged with the traditional MTD capabilities (i.e., Address Space Randomization) and moved across platforms (i.e., space) in time intervals. To prevent moving blindly in space, the framework is integrated with a proactive monitoring scheme below the OS using *Virtual Machine Introspection* (section 1.4.2). This allows to effectively move across platforms for defensive measures and avoid configuration combinations (i.e., OS, hypervisor) and platforms (i.e., hardware) that are susceptible to attacks.

With these two capabilities, coupled with the formal model of the framework, one can easily observe the high-level system behavior in each time-interval on whether we are in *desired* (i.e., initially deployed) state, or *undesired* state (i.e., under attack or compromised) (section 1.4.3). We highlight these three logical building blocks of the framework next. In this dissertation, we show how the aforementioned three logical building blocks are sufficient to address the desired security objective.

1.4.1 Node Reincarnation

Reincarnation is a technique for enhancing the resiliency of a system by terminating a running node and starting a fresh new node with different characteristics (i.e., hypervisor, OS) for its place. This new node will continues to perform the computing task as its predecessor without disrupting the computations (i.e., application runtime). All the nodes in the proposed MTD framework have a predefined short lifespan, as low as a minute, and an observation status that dictates whether the node reaches its *lifespan* or reincarnated prematurely due to attacks. For instance, some replication models (i.e., quorum-based) have 2/3rd of the nodes running in sync at all times, as a result, some nodes are exposed to attacks longer than others, thereby, prioritizing node reincarnation is critical. The technical challenges inherent in this runtime transformation is discussed in details in Chapter 3.

1.4.2 Proactive Monitoring

Hypervisors enable many isolated OS's to share hardware resources. The Virtual Machine Monitor (VMM) is the management interface between the hosting OS and the Guest OSs. The transparent mapping of the virtualized OS resources (i.e., virtual memory, vCPU) into the physical memory enabled by the VMM allows the interception of events to detect anticipated changes of the applications in the guest OS, thus, enables to safeguard the system below the OS, a process known as *Virtual Machine Introspection (VMI)* [6]. Such detection schemes are difficult to subvert by attacks originated inside the VMs of the applications. The implementation details or our proactive monitoring with VMI is discussed in Chapter 2.

1.4.3 Desired vs. Undesired System States

Typically, we deploy a system in a *desired* state and at some point in time we end up in an *undesired* state (i.e., compromised or failed) without our knowledge (in most

cases). This is mostly credited to a successful stealthy attacks that creates *turbulence* (i.e., under attack) state, infinitely many times until the system is *compromised*, data is ex-filtrated or less usable (*fail or crash*). These high-level system state uncertainties are driven by what is 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; or when a node crashes in which the system enters into a *failed* state. One way to formalize this behaviour in order to mathematically reason the correctness of the framework is through Hierarchical Hidden Markov Model (HHMM) [20], discussed in details in Chapter 3.

1.5 Thesis Statement

The thesis statement of this dissertation is:

With the increasing-level of sophistication of attacks in recent years, it is possible for the systems to resist these attacks with a combination of mobility *capabilities in order to move across platforms and* guidance *on where to move, while serving their legitimate users.*

1.6 Thesis Contribution

The key contribution of this dissertation is the extensible design of the generic framework built on a sound theoretical foundation to mathematically reason about system behavior using well established tools and techniques. Using this design principle as the cornerstone, this dissertation advances the state–of–the–art by extending the widely adopted cloud management framework with novel defensive strategies to combat against the ever evolving attacks on systems deployed on these complex platforms.

Towards realizing attack-resiliency for distributed systems on cloud platforms, this dissertation makes several fundamental contributions. These include:

- Offering a defensive strategy to reduce or eliminate the need to continuously fight threats (i.e., zero day) that is difficult to win due to the subtle and poorly understood attack-vectors of the cloud and the level of the sophistications employed in modern attacks.
- Introduces a formal model and algorithms for Moving Target Defense, a defensive strategy that is considered a game changer [21].
- Introduced a new generic and unified MTD framework for distributed systems.
- Enables operation continuity in highly-targeted systems (i.e., military, finance, health).

Our technical contributions are in the simplification of the cloud platform taxonomy, real-time application runtime integrity violation detection techniques, system randomization and diversification algorithms to anticipate threats before they become a problem. Furthermore, transforming the cloud management framework APIs from asynchronous VM management to synchronous model to support the dynamics of VM diversification without changes to the applications. These capabilities are critical to ensure operation continuity in commercial and military systems on untrusted and highly-targeted cloud computing infrastructures. We now highlight our key contributions and their impact in the following subsections.

1.6.1 Framework Design

The proposed framework adopts a cross-vertical design that operate on three different logical layers of the cloud framework; the *nova compute* at the application layer (GuestOS layer), the *VMI* at the hypervisor layer (HostOS layer), and the *neutron* at the networking layer (SDN). We know that these three logical layers of the cloud abstracts the applications deployed in these platforms regardless of their architectural styles or system models into a unified virtual computing environments (VMs). With this design, we further extend the abstraction of the applications' runtime in these

VMs without changes to the applications deployed in them as depicted in Figure 1.2. This allows us to easily reason about the high-level system status from the low-level application runtime in time-intervals.

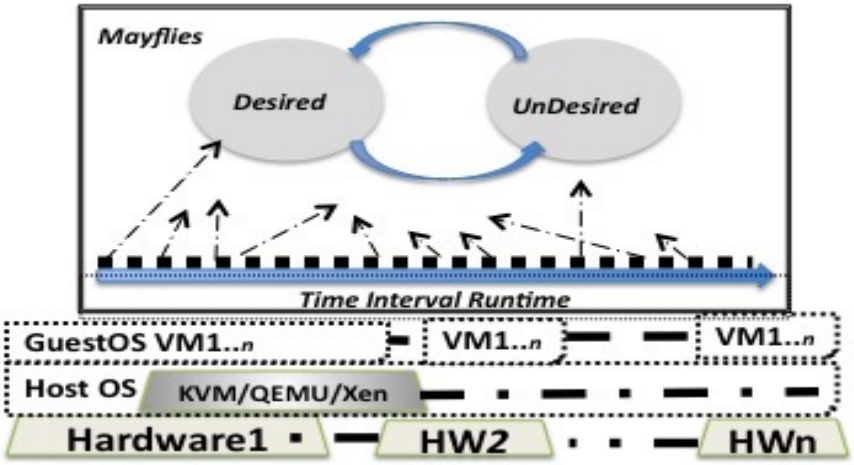


Fig. 1.2.: Mayflies MTD framework.

1.6.2 Abstractions and Paradigms

In a cloud platform built with OpenStack, the *nova compute* abstracts the virtual machines from the applications in order to isolate (i.e., multi-tenancy) each other while sharing the same physical hardware in pursuit of cost efficiency, and ease of integration and deployment. Technically, this isolation is achieved by *provisioning* and *de-provisioning* VM instances on available platforms (hardware), and the programmable *Software Defined Networking (SDN)*. The process for sharing the resources which is mediated by the hypervisor is achieved by stopping a VM from execution and resuming another one without any consideration of the actual running application architecture or system model, referred to as *VMEntry* and *VMExit*. This is similar to context switching of the CPU in application domains.

As illustrated in Figure 1.2, we introduce two abstraction layers on top of the traditional application runtime that is already abstracted within a VM by the cloud framework as eluded above. The first abstraction is the *Time-interval Runtime Execution (TIRE)*. TIRE is simply to partition the runtime into time-intervals, depicted on the dots on the arrow time line, in order to evaluate the system state (i.e., desired and undesired) within these time intervals. We developed algorithms to *pro-actively* terminate a VM and start a new one on a heterogeneous platforms (hypersors, OS's) at runtime by extending the asynchronous model of the VM *provisioning* and *de-provisioning* of *nova compute* API implementation, and dynamically swapping the network interfaces with the *neutron* API implementation of the SDN, discussed in details Chapter 3.

The second abstraction is the two high-level system states; *desired* and *undesired* states, to formally reason about the system behaviour. The driving engine of these states are; a) the *pro-active monitoring* scheme used to detect system runtime integrity violations below the OS using *virtual introspection*, and b) the *pro-active node reincarnations* in time-intervals. Based on the observation in a) depicted on the dotted arrows on the *TIRE*, we can easily reason the system state in each time-interval on whether the system is still in its *desired* state (i.e., initially deployed state) or it is in *undesired* state (i.e., compromised), if so, *reactively* anticipate state changes to our favour in the subsequent time-intervals.

These abstraction layers allows randomization and diversifications on all types of distributed systems in any cloud platform (i.e., OpenNapula, Eucalyptus).

1.6.3 Formal Model

Finite State Automata (FSA) is widely adopted mathematical machinery for specifying system with both deterministic DFA and non deterministic NFA properties, especially, fault tolerant distributed systems. Buchi automaton [22], a type of ω-automaton which is NFA are 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 in-deterministic nature of the cyber space, therefore, we model the framework using probabilistic FSA (PFSA). PFSA is simply a non-deterministic FSA (with no ϵ transition) with probabilities for all transitions of the FSA.

We formulate our model in terms of *Finite automata* which enables us to formally describe a probabilistic transitioning behavior correctly and define the resiliency of the system formally. 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. For example, if any node's runtime integrity violated, the system is considered compromised (i.e., undesired) and we need to resist such behaviour by reincarnating the unscheduled node to offset that loss.

The high-level system transition is dictated by the underlying observations collected during system runtime in a pre-specified time intervals at the *Time-Interval Runtime Execution (TIRE)* abstraction layer, therefore, it is intuitive to decompose the framework model into two automaton, separating the high-level system behavior and its internal runtime execution *(i.e., TIRE)*, to easily reason the correct behavior of each model independently.

1.6.4 Automata Decomposition

Finite State Automata allows modelling 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. Therefore, we model the two abstraction layers of the framework with independent automata; time-interval runtime execution and high-

level system states, in which the observations Λ of each time-interval that results in either *true or false* be the output of the automaton and then *accept* as input of the other.

The expressiveness of the *Accept* lies the power of the Buchi automaton to model the time-interval runtime execution, and the correctness property (i.e., *safety and liveness)* 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 modelling the framework with PSFA, specifically, a Hierarchical Hidden Markov Model (HHMM) [20] represented with Dynamic Bayesian Networks (DBN) [23], a time-linear representation of HMM.

Modelling the framework with two automatas allow us to reason individually, and to further compose with other formal automata models such as; interface automata [24], virtual machines [25], cloud framework [26], and attack surface [11]. Therefore, the framework fills the gap for reasoning attack-resiliency in all aspects of the systems deployed on these complex cloud platforms.

1.7 Thesis Organization

We addressed the proposed framework in four chapters, described below:

Chapter two, Proactive Application Run-time Monitoring with Virtual Introspection is based on the published journal article titled; Towards Targeted Intrusion Detection Deployments in Cloud Computing [27]. This chapter introduces a simplified cloud taxonomy in order to adopt a light-weight virtual introspection intrusion detection scheme as a proactive monitoring capability for the framework.

Chapter three, *Mayflies*: A Moving Target Defense Framework for Distributed Systems. This chapter introduces bio-inspired attack-resilient MTD framework for distributed systems which we call it *Mayflies*, and discusses the theoretical underpinnings, design, model and implementation of the prototype.

Chapter four, Byzantine Fault-Avoidance (BFA). This chapter deploys a Byzatine Fault-Tolerant (BFT) system prototype (i.e., synchronous system model) on *Mayflies* framework and illustrates how the framework transforms BFT systems to BFA, a distributed system that avoids threats and withstands attacks (i.e., Byzantine under attack [28]) than tolerating faults to a certain degree. This chapter is based on our published work titled; From Byzantine Fault-Tolerant to Fault-Avoidance: An Architectural Transformation to Attack and Failure Resiliency [29].

Chapter five, Disruption-Resilient Publish and Subscribe. This chapter illustrates *Mayflies* framework with an event-based systems (i.e., asynchronous system model). This work is based on the published work entitled; Disruption Resilient Publish and Subscribe [30] which shows how chain replicated system model can be diversified and randomized to combat against a wide array of attacks that are common to these systems.

The remainder of the dissertation proceeds as follows: Chapter 2 and 3 present each logical building blocks of the proposed framework described above. To illustrate the framework with widely adopted systems in a realistic cloud settings, Chapter 4 and 5 discusses the effectiveness of the proposed framework on two different classes of systems and present experimental results in terms of their performance and attack resiliency. Finally, we summarize the thesis and discuss our future work in Chapter 6.

2. PROACTIVE APPLICATION RUN-TIME MONITORING WITH VIRTUAL INTROSPECTION

Protecting application runtime in dynamic cloud platforms with the traditional security tools and techniques is increasingly challenging. One common widely protection scheme for such platforms is to continuously monitor anomalous system behavior for intrusion in order to prevent security violation incidents or reduce its impact when such violations occur using an Intrusion Detection System (IDS). Existing IDS techniques are ad-hoc and cumbersome to properly tweak for different layers of the virtualized platforms. One of the core issue is mapping the IDS taxonomy classes to the complexity of the multiple independent interconnected components (i.e., compute, network) of the underlying cloud computing fabric.

In this chapter, we propose a simplified IDS taxonomy model and a deployment decision support techniques for cloud platforms. We present a light-weight IDS using *Virtual Machine Introspection (VMI)* to safeguard application runtime for the proposed Moving Target Defense (MTD) framework. We illustrate the taxonomy mappings with two different attack scenarios and detection capabilities of the VMI. This chapter is based on a published work "Towards Targeted Intrusion Detection Deployments in Cloud Computing" [27].

2.1 Introduction

Cloud computing is a cost-efficient computing paradigm that can scale up or scale down on demand with limited system management/maintenance requirements due to its structured building blocks and the hierarchical deployment models. The three widely accepted cloud platform categories are 1) a private cloud, built in within organizational boundaries, 2) a public cloud, typically rented from commercial companies

such as Amazon, RackSpace, HP, Google and Microsoft, and 3) a hybrid cloud that provides a composition of private and public solutions. Within each of these platform categories lie three service deployment models similar to the utility services, referred to as "pay–as–you–go" service fee model: Software–as–a–Service (SaaS), Platform–as–a–Service (PaaS) and Infrastructure–as–a–Service (IaaS).

The main difference between the three service deployment models is the level of administrative responsibility between the cloud user and the cloud provider for managing and maintaining the cloud systems. For example, for SaaS, the software is offered by the cloud provider as a fee-based (i.e., salesforce.com). For PaaS in public clouds, for example, some applications (e.g., web-servers) can be deployed with little or no cloud conformance or modification requirements where the management/-maintenance responsibilities are left to the cloud provider. Additional management responsibilities are transferred to the user for the IaaS model. Thus, the specific service deployment model dictates the type of IDS scheme suitable in that specific layer.

There has been a wide array of IDS work in the past few years. Special treatment of the current techniques and open issues is given in [31]. IDS techniques vary from system to system, for instance, one IDS technique [32] employed by the cloud providers independently collect monitoring information at different levels of granularity to address the relationship behavior between the VMs rather than anomalies. Deployment strategies for selected IDS techniques are presented in this survey [33].

One common theme among all cloud-based state–of–the–art IDS solutions is that they are designed for one layer of abstraction of the cloud; thus, they require tweaks to accommodate other threats and layers across the infrastructure. Furthermore, traditional IDS taxonomies are engineered for a given layer of abstraction or for the system as whole (i.e., from the application logs down to the network traffic). Therefore, it is prudent to develop a simple taxonomy that abstracts the low-level (i.e., system and network) details for the application layer in order to adapt to a specific target for monitoring, for instance, in our case, application runtime monitoring. This is to

enable a proactive monitoring capabilities for the applications at run-time. Unlike the traditional IDS monitoring schemes, runtime monitoring requires faster decision of the VM behavior, thus, we leverage a *Virtual Machine Introspection* (VMI) [6] as an IDS solution and adopt the proposed taxonomy mapping techniques to illustrate its effectiveness.

2.2 Simplified IDS Taxonomy Model

Targeted IDS deployments in cloud environments require mapping IDS solutions to specific layers of abstraction of the cloud, however, such mapping is not a one–to–one mapping. For example, IDS solutions in the literature are classified as network or host-based. The host can be the guest VMs or their host OS (i.e., hypervisor). In cloud computing environments, the network layer is not accessible to SaaS and PaaS service models, therefore, such classifications is harder to clearly map IDS solutions to its proper cloud service deployment model. An overview of a comprehensive IDS taxonomy is given in [34]. In this section, we classify existing IDS solutions in relation to cloud service deployment models to simplify such mappings.

The simplified methodology enables effective data collection in a variety of system areas which is the core foundation of an effective IDS solutions. For network-level IDS schemes which is only available at the IaaS cloud service model, for example, the timely collection of network traffic and determining the correct anomalous behavior within that data is critical. Yet, these decisions are not 100% accurate; some cases drop to 80% [35] which is unacceptable in some applications.

Furthermore, the accuracy of detecting a threat is hindered as the systems grow. Typically, most applications are built with components authored by different developers who choose what to log [36] which further contributes to the volume of collected data that needs to be automatically analysed in a timely manner with high accuracy; a problem further worsens if the data is unstructured. Labelling and associating such data to derive an accurate anomalous behavior typically poses a greater challenge.

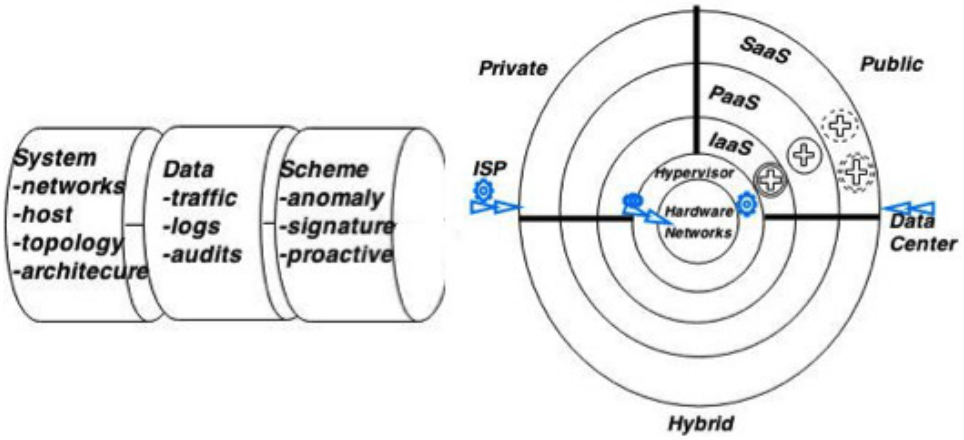


Fig. 2.1.: Simplified IDS taxonomy classes (left) and cloud service deployment models depicted as rings (right)

Therefore, it is critical to develop a simple yet effective taxonomy model that can easily map IDS solutions to the desired cloud service model for collecting real-time data. We introduce a simplified IDS taxonomy; *System, Data, and Implementation Scheme*, depicted as three rollers in Figure 2.1 (left) and, cloud deployment models are depicted as layered rings (right).

2.2.1 Cloud Service Deployment Models

Figure 2.1 (right) show the common cloud service deployment models depicted as rings, a *private* and *public* cloud flavours are depicted in the left and right quadrants of the rings, and *hybrid* on the bottom half. The inner ring represents the hardware and networking which is virtualized by the hypervisor, depicted on the second ring. On top of the hypervisor, are the three service delivery models; IaaS, PaaS, and SaaS respectively. The short double arrows represent IDS positions, e.g., by the data center (cloud providers) or at the internet service provider (ISP)/internal for corporate private clouds.

IaaS, PaaS, and SaaS service models offer different protection schemes for the applications deployed (depicted as a plus sign). From the cloud users prospective,

applications deployed IaaS have user provided (i.e., introspection) monitoring data depicted as double circles around the plus sign. By moving further away from IaaS, applications have some form of monitoring information provided by cloud provider (i.e., application server or database logs), depicted as a single circle in PaaS or dotted circles in SaaS. The arrows and gears represent positioning additional protection/-monitoring schemes like firewalls at the ISP found in private clouds or the data centers in public clouds. It is intuitive to see that these layering schemes are easier to map a simplified IDS capabilities on such environments. For example, the *network* attribute in the systems taxonomy class can only be mapped to IaaS service model.

2.2.2 Taxonomy Classes

In this section we describe the three taxonomy classes, then we give a realistic attack and *VMI-based* detection use case scenario to show its effectiveness.

System Class

In Figure 2.1 (left), we consider the traditional network and host taxonomy under the *Systems* class (depicted in the 1st roller). The main rationale behind our grouping is due to the cloud deployment models where the line between the host and network is merged into the guest and the host. For example, due to the multi-tenancy model of the cloud, SaaS and PaaS deployment models have no access to the network traffic and the host machine, therefore, the Systems class covers everything including the network, host, guest, the topology, and the architecture.

Data and Scheme Classes

Data collected and the scheme used to detect intrusion is critical to the success of the IDS. This is because different abstraction layers of the cloud offer different data sources, and the IDS scheme permissible in deploying in that abstraction layer

as depicted in the second and third rollers in Figure 2.1 (left). Therefore, we classify *Data and Scheme* independently, where the *Data* can be the network traffic data, logs, or system audit trails, and the *Scheme* is simply the scheme used to implements the IDS system. For example, anomalous-based, behavior-based, signature-based, and proactive monitoring (continuous monitoring), etc.

Anomalous-based intrusion capabilities available for the applications are in direct relation to their deployed service ring. For example, it is intuitive to see network traffic data cannot be collected from an application deployed in SaaS unless customized tenant traffic is directed to the users VM by the cloud provider. Additional data sources in this ring include: audit trails logs, applications or database logs, the cloud provider's logs (for billing purposes), and application signatures (HMAC hashes).

Additional logs at the platform level (e.g., detailed OS logs, application servers, etc.) are available in PaaS. The traditional anomalous behavior found in these applications and system logs include function calls made/logged, temporary files written somewhere, failed connections to a system or database, or too many unsuccessful login attempts. Using these as the key data sources for the IDS, one can detect specific threats. However, if the guest OS/VM is compromised and undetected, the integrity of these logs is difficult to verify, thereby, receiving logs/signatures the attacker prepared for the IDS.

Each service deployment model offers different data sources and enumerating unknown threats to craft for specific IDS is cumbersome and ineffective. Therefore, applications need to be deployed based on their critically of detection needs into the appropriate service models (rings) and map the traditional IDS solutions suitable for that ring or craft if needed. To simplify the mapping process, we introduce a decision support matrix.

2.2.3 Deployment Decision Support Matrix

Determining the target spot for IDS deployment for applications in cloud platforms is highly dependent on two factors; the threat model and the available data source to derive the anomalous behavior from. In other words, the fundamental question is what we consider anomalous behavior under a given threat and what are the data sources that we need to collect and derive the undesired behavior from? Thereby, the data sources are the key determining factor of the success or failure (miss or high false alarm rates) of the IDS solution deployed. Since this work is focused on the taxonomy simplification for targeted deployment, we don't consider covering the proposed IDS alarm rate accuracy.

Table 2.1.: IDS Decision Matrix

Model	System	Data	Scheme
SaaS	N/A SaaS	App logs, Audit logs	Signature, Behavior
PaaS	N/A SaaS	OS logs, App logs, Audit logs	Host, Signature, Behavior
IaaS	VMI	Live Memory snapshots	Proactive and Reactive

Table 2.1 show the decision matrix; the cloud provider model in column one, the IDS system in column 2, and few examples of Data and Scheme used for a given threat in columns 3 and 4. We use a motivating example use case scenario (discussed next) with the system model reference in Figure 2.1 to show how the entries are filled in for a given IDS solution.

2.3 Proactive Monitoring

Proactive is typically defined as the continuous monitoring or system observation while reactive is the reaction or the response upon detecting abnormal or intrusion behavior. To illustrate the benefits of the proposed simplified mapping scheme using the decision support matrix in Table 2.1, we show how the targeted solution adopts to

the threats of a given intrusion use case. Enumerating possible threats to proactively monitor is infeasible, we devise a specific use case for application runtime anomaly, thus, our IDS operate below the OS (VMs) at the hypervisor-level in a private cloud setting.

The hypervisor loads the next application to execute into the CPU, specifically, at CR3 register. One can detect these events and alarm anomaly if the application is not authorized. Another methods is capturing the application's live memory snapshots to detect unauthorized application/code snippet (i.e., code injection) loaded in the memory with *Virtual Machine Introspection (VMI)* [6]. We are interested in the later case given that we can further analyze the memory if needed in contrast to an application name found in the registry. Since we are interested in monitoring the application run-time we can simply peek the application's live memory structure and detect if altered, not the system as whole (OS logs, network traffic, etc.), hence, we refer to as light-weight IDS.

2.4 Proactive Monitoring with VMI

A typical cloud computing node consist of a host OS using a hypervisor (Xen or KVM) to virtualize the computing hardware resources, and a guest OS to deploy applications. Our lightweight `VMI-based` solution `pro-actively` gathers `live memory data` in intervals and `reacts` if detected anomalous behavior. With this information, we now fill the decision support matrix table an entry of the System column with *VMI* and *Data* column with memory data structure snapshots, and Scheme column, as the name implies, is filled with our proactive and reactive. The monitoring process is as follows:

1. *Start the application to be protected in the guest VM.*

2. *Take memory snapshots in time intervals at the host OS.*

3. *Analyse life memory data structure in real time and alarm any runtime integrity violation of the applications runtime as an intrusion.*

2.4.1 Use Case Scenario

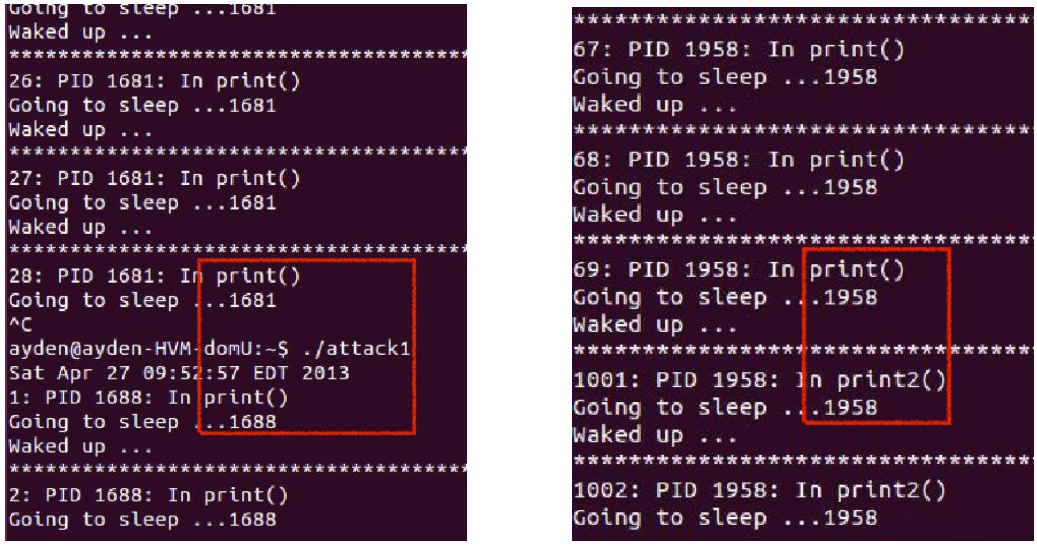
Consider an adversary compromising a VM node using a kernel root kit. For effective detection strategy, we are interested in solutions that are outside the VM that we are trying to protect. Given the threat and the solution approach, we fill the matrix in Table 2.1 entries in SaaS and PaaS System columns with "N/A" which means that system-level or architectural solutions are not available in these service deployment models. *Data* and *Schemes* available in that service model, as described earlier, are also filled in their appropriate columns.

To illustrate our intrusion scenario, we adopted two stealthy attacks that are difficult to detect from logs collected in the VM/OS or network traffic:

1. ***Attack 1:*** *We mimic a node compromise by logging into the VM, stopping the victim process, and starting a malicious one with the same name but different functionality; presumably stealing data. The process is a small c program that simply prints a counter value, its process ID, and a function name print() and sleeps for few seconds in a continues loop.*

2. ***Attack 2:*** *In a realistic setting, some applications have many runtime dependencies and may be orchestrated with other applications. Restarting the process causes to break the dependencies or the chain, thereby triggering an alarm. To circumvent, we hijack the application by loading/injecting a shared library and diverting to make additional function calls without stopping the process, print2() in this case, that is implemented in the injected shared library.*

Since the focus of this chapter is to illustrate our light-weight IDS deployment for proactive monitoring for the proposed MTD framework, we will not discuss the details of the attack implementation. For those interested, the program and the application hijacking attack process used in this work are downloaded from the public domain.



(a) Application Restart (b) Application Hijacking

Fig. 2.2.: Application runtime integrity attacks

2.4.2 VMI-based Runtime Anomaly Detection

We installed Library for Virtual Machine Introspection *LibVMI* [37], an open source VMI library, in the host OS and continuously capture live application memory snapshots in time-intervals which has a negligible performance impact [38] on the application. For the first use case, we started our victim application *attack1* in a virtual machine and killed it with *control c* as shown in Figure 2.2(a). For the second use case *(attack2)*, we injected a shared library and manipulated the victim's *print()* return function in order to execute our *print2()* function in the injected shared library and corrupt the counter by starting at 1001 as shown in Figure 2.2(b).

Attack 1 Detection

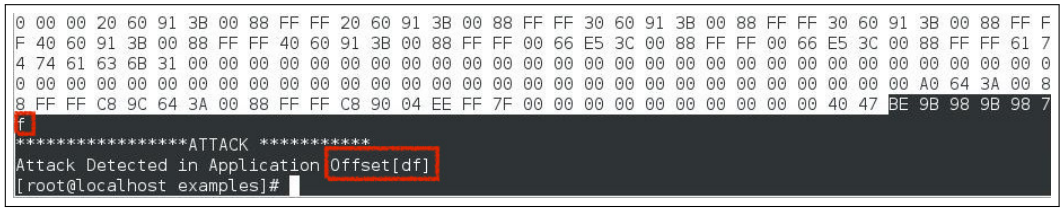
In order to effectively detect this attack shown in Figure 2.2(a), we take an initial memory snapshot, then for every snapshot (i.e., as low as a minute), we perform two comparisons, the process ID and the address offset. This is because, in some cases, we found out that the hypervisor re-assigns the address offset as soon as a process

is terminated as shown in Figure 2.3(a). Strangely, the same offsets *3b949700* and *3b949700* but different process ID, 1688 vs. 1681. Typically, as shown in Figure 2.3(b), restarting a process gets assigned with a new process ID 1767 and a new process address offset *[3c650000vs...3b949700]*.

A kernel rootkit can easily circumvent this attack by hiding the process ID from the process list, making it difficult to detect such intrusion within inside the VM/OS. We detected the attack at the host OS in two ways; a process ID or memory address offset mismatch, or both is some cases. Next, we show how this monitoring scheme can be circumvented in *Attack2*.

(a) Mismatch process ID but same address offset

(b) Mismatch address offset



(c) Corrupted internal address block

Fig. 2.3.: Integrity violation detection

Attack 2 Detection

We hijack the application using code injection attacks to avoid detection for attack 1. As shown in Figure 2.2(a), we call a function to print *print2()* implemented in the injected code while still printing it's process ID 1958. To detect such attack, we need to analyze the memory snapshots deeper in order to find any discrepancies caused by the malicious code that implements the *print2()*. Note that deep memory analysis is typically performed for Malware detection. Since the memory data structure content of a single snapshot of an application is large, deeply analysing this content is not suitable for application runtime monitoring because it is associated with a high performance cost, however, we were able to detect this attack with few string comparisons of the address block offsets as shown in Figure 2.3(c).

The memory data structure contains the continuous address block of the running applications' data and the execution parts. We simply store the first two hex characters of each block's starting address offset in order to prevent performance overhead inherent in deep live memory analysis. We then compare the incoming scan results of these values of our target application in time-intervals. We detected the block's starting address mismatch when we hijacked the application. This shift in address space at position df as shown in Fig. 2.3(c) was due to the injected library's physical address space allocated by the hypervisor.

2.5 Conclusion

In this chapter, we showed a simplified IDS taxonomy with a decision support matrix as a guide for developing and deploying a targeted IDS solution on a given cloud service model. With this taxonomy model, a light-weight IDS for real-time application runtime integrity violation detection is deployed and illustrated with a realistic attack and detection use case scenarios.

3. MAYFLIES: A MOVING TARGET DEFENSE FRAMEWORK FOR DISTRIBUTED SYSTEMS

Advances on resiliency to arbitrary faults and system failures have contributed well established, sound protocols and paradigms in distributed systems literature, however, resiliency against sophisticated attacks on these systems is poorly explored. The corner stone of this contribution lie redundancy/replication techniques in which turns out to be a double–edged–sword when facing sophisticated attacks. This is due to the fact that increasing the number of nodes for reliability purposes, increases the system's attack-vector – the set of ways an attacker can penetrate a system (i.e., more nodes to protect). To remedy this issue, in this chapter, we present an attack-resilient framework for distributed systems, dubbed *Mayflies*.

3.1 Introduction

The traditional defensive security strategy for distributed systems is to safeguard applications against malicious activities and prevent attackers from gaining control of the system. The strategy employs well established defensive techniques, i.e, perimeter-based fire walls, redundancy and replications, and encryption. However, given sufficient time and resources, all these methods can be defeated, especially, when dealing with sophisticated attacks from advanced (technically) adversaries that leverage zero-day exploits.

Moving Target Defense (MTD) [19], is a defensive strategy that aims to reduce the need to stay one step ahead against attacks by disrupting attackers gain-loss balance. The core of this strategy is to continuously shift the system's attack surface [39] – 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. This is achieved by randomization and diversification of the nodes across platforms.

There are well established MTD solutions designed specifically to combat against specific threats, but limited when exploiting beyond their boundaries, for instance, application-level redundancy and replication schemes prevents exploits that target the application code base, but fail against code injection attacks that targets runtime execution such as; buffer and heap overflows, and control flow of the application.

Instruction Set Randomization [14], Address Space Randomization [15], randomizing runtime [16], and system calls [17] have been used to effectively combat against system-level *(i.e., return-oriented/code injection)* attacks. System-level diversification and randomization is considered mature and tightly integrated into some operating systems. Most of these defensive security mechanisms (i.e., instruction/memory address randomization) are effective for their targets, however, modern sophisticated attacks require defensive solution approaches to be deeply integrated into the architecture, from the application-level down to the infrastructure simultaneously and at all-times.

We propose *Mayflies*, a bio inspired generic MTD framework for distributed systems that extends the cloud management software stack in order to defend systems against sophisticated attacks. Mayflies [40] is a short-lived (minutes to hours) winged insects known in the literature as *ephemeroptera*. Depending on the type of *Mayfly* species, females of the *Dolania Americana Mayflies*, adult females live less than five minutes where they find a mate, copulate, and lay their eggs during this short window of time [41].

The basic idea of our framework is for a node (i.e., servers or replicas) to exist in a predefined lifespan (i.e., short time intervals), participate in the overall computation, then vanish and appear (with pre-packaged with the existing MTD capabilities) on different platform with different characteristic (i.e., different OS), dubbed *Node Reincarnation*. This strategy allow us to deal attacks in short time intervals rather

than the entire system run time, thereby, avoiding in-progress attacks or the spread of a successful (undetected) attack to prevent cascading effects (colluding).

We designed our *Mayflies* MTD framework on top of a cloud framework with spacial emphasis on time (as lows a minute) and space diversification and randomization across heterogeneous cloud platforms (i.e., OS, Hypervisors) while proactively monitoring the nodes. We abstract the system runtime from the virtual machine (VM) instance to formally reason its correct behaviour using *Dynamic Bayesian Network* [23]. This abstraction allows the framework to enable MTD capabilities to all types of systems regardless of its architecture or communication model *(i.e., Asynchronous and Synchronous)* on all kinds of cloud platforms (i.e., OpenStack and OpenNapula).

3.1.1 Related Work

To the best of our knowledge, *Mayflies* is the first unified generic MTD framework for all types of distributed systems on virtualized cloud platforms. However, the ideas employed, specially, the model and the proactive monitoring with virtual introspection schemes, are not unique to the framework. To name a few, for example, Salfner and Malek [42] adopted a Semi-Markov Model and proactive monitoring techniques for online failure prediction. A game theocratic approach is introduced in Flipit [43] as a game between attackers and defenders to assess system compromise time vs. reclaiming time in a time-line window.

Other notable frameworks include: TALENT [44], an MTD framework that use OS-level virtualization to sandbox mission critical applications across heterogeneous platforms and migrate the environment in real-time. A human in-the loop framework for controlling multi MTD capabilities is proposed in [45]. Along the same lines of our cloud-based framework, a network focused MTD architecture is introduced in [18] that transparently mutates IP addresses with high unpredictability, thus compliment to our work.

The idea of Virtual Machine Introspection (VMI) techniques have been extensively studied for security designs. LiveWire [46] was the first VMI-based architecture for intrusion detection for the integrity of user programs in Linux, i.e., sshd, syslogd, etc. In our previous work [27], we discussed a simplified cloud platform taxonomy and targeted VMI deployments for detecting application runtime anomalies. In this work, we focus on the tight integration of VMI with Mayflies. Early works include protecting Unix programs such as; ls, ps, etc. Modcheker [47], proposed a scheme for detecting integrity violations for Windows OS kernel using VMI. Monitoring system calls between the VMs and its host kernel using HMM to classify VM behavior is proposed in [48]. In recent years, VMI-based solutions have shown an increased interest in the commercial domain to offer a layered set of security services to the cloud users for specific products like IBM Tivoli products or HP system Insights. We leveraged VMI capabilities for detecting application runtime integrity violations in real-time.

3.1.2 Attack-Resiliency

Resiliency to attacks in distributed systems is typically viewed as the system's ability of resisting to disorder or capable to bounce back in the event of an attack. Clearly, this is a reactive defensive model, thereby, the reliability solution approaches (i.e., redundancy and replication) becomes a double–edged–sword due to the increase in the reactions or responses needed for the multitude of disorders inherent in increasing the number of nodes in the system. In the proposed framework, having the nodes exist with a predefined short lifespan while proactively being monitored at the infrastructure layer (i.e., virtual introspection), we are able to deal with attacks before they make an impact. As a result, attack-resiliency definition, as viewed, does not semantically align in the context of this framework.

Due to luck of consensus in resiliency definition in distributed systems, we view resiliency as not an end product/system to resit attacks as commonly viewed, but

as an ever evolving conceptual system attributes in concert with the disturbances or attacks of its environment, similar to those adopted in organizational resiliency [49] and systems engineering [50]. Thus, in this context, we define resiliency to attacks as follows:

Definition 3.1.1 *Resiliency to attacks is the ability of a system to sustain operations for its intended users regardless of the disturbances (i.e., attacks) inherent in its environment.*

In order to sustain operations, the system needs to be able to continuously anticipate the environmental changes prior and during disturbances – during attacks. Therefore, resiliency must be conceptualized both *proactive*, to change before the environment changes, and *reactive*, effectively anticipate the changes while experiencing disturbances. Note that, in our conceptual resiliency model, we are only interested during the attack, not afterwards. This is due to the fact that post attack, it means that our attack-resiliency scheme has failed (i.e., no forces to resist), therefore, we need to adjust our defensive tactics to anticipate future attack. At post attack crisis stage, widely studied faulty-resiliency and recovery schemes built-in the applications effectively handle the situation.

Formulating resiliency in this manner, we can quantify it in terms of the rate of changes between the system defender (subject to node reincarnation and virtual introspection costs for monitoring), and the environment (subject to the cost of a successful/attempted attacks). Inspired by the consensus of the species' populations model [51], for example, the *preys* population is measured by the proportionality of their survival/reproductive rate vs. their eaten rate by *predators*. We consider the attackers cost relative to our defensive strategy cost, for instance, if we wish to thrive in any environment, the cost of our defensive strategies should be *much lower than or equal to* the inherent cost of attacks/disorders we face. This principle is the cornerstone of our framework design and our quantification of its effectiveness.

3.2 Framework Design

We built *Mayflies* framework on top of OpenStack cloud framework [7], a widely adopted open source cloud management software stack that consists of many independent components such as *nova compute, horizon, neutron.* We selected OpenStack due to its popularity in commercial clouds, for instance, RackSpace [8], a public cloud platform built with OpenStack used by many well-established businesses like Netflix. Further, OpenStack provides a modularized components that simplify cloud management.

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

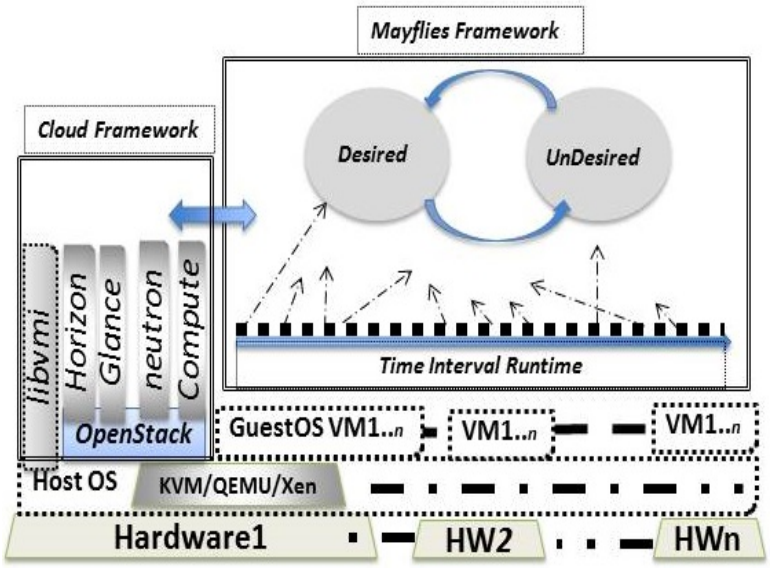


Fig. 3.1.: High-level Mayflies architecture

3.2.1 Mayflies Architecture

Figure 3.1 show the high-level architecture of *Mayflies* framework (top right) and OpenStack cloud framework (left). In the cloud framework, starting from the infrastructure at the bottom layer lie the hardware. In each hardware has a host OS, a hypervisor (KVM/Xen) to virtualize the hardware for the guest VMs on top of it, and the cloud software stack framework, OpenStack in our case. The vertical bars are some of the OpenStack framework implementation components we leveraged in this work. These include: *nova, neutron, horizon, and glance.* In addition, we installed *libvmi* [37], a library for virtual introspection to peek live memory activities at the hypervisor-level. The implementation details and the defensive capabilities of this component is discussed in Chapter 2.

In the *Mayflies* framework, we introduce two abstraction layers; a high-level *System State* (top) and the *Application Runtime* (bottom), dubbed *time-interval runtime.* To illustrate, for the system state, we consider *Desired* as the desired system state at all times, and *Undesired* as the state we like to avoid (i.e., turbulence, compromised or failed system state). The driving engine of these two high-level states are the observations from the application runtime by the proactive monitoring enabled by the *libvmi* depicted as dotted arrows (discussed next).

3.2.2 Logical Building Blocks

The *System State and the Application Runtime* are two abstraction layers that operate in synchrony. At the application runtime layer, we pro-actively refresh VMs depicted in GuestOS *($VM_1 \ldots VM_n$)* on different platforms as depicted on ($Hardware_1 \ldots HW_n$) in pre-specified time intervals, referred as *time-interval runtime.* To gain a holistic view of the high-level system state, we re-evaluate the system state at the end of each interval to determine whether the system is in a *desired* state or *undesired* state.

System State

The key objective of *Mayflies* is to start the system in a *Desired* state and stay in that state as often as possible. In the event when the system transitions into *Undesired* state, a valid assumption in cyber space, we should bounce back seamlessly into the *Desired* state. These transitions are dictated by the application runtime status, for instance, if the application fails or crashes, which we typically know, then we consider the system in *undesired* state. The challenge is when we enter into one of the unknown *undesired* state; a *turbulence* state, which is when the system is under attack and still usable, or in a *compromised* state when the attacks succeed and a node is compromised.

As the cloud frameworks (i.e., OpenStack) abstract the compute nodes from the deployed systems regardless of their 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* attempts to abstract the system's application runtime (discussed next) from the VMs that are deployed in order to break the runtime into observable time-intervals regardless of the application type. This allows us to model both the system state and the runtime independently; therefore, we can formally reason the transitions between the *Desired* and the *Undesired* states (discussed in section 3.3).

Application Runtime

Mayflies transforms the traditional services designed to be protected their entire runtime (as shown on the guest VMs on the cloud framework) to services that deal with attacks in *time intervals*. Such transformation is simply achieved by allowing the applications run on a heterogeneous OS's and 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 [52]. Formally. we define *time-interval* as follows:

Definition 3.2.1 *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.*

The goal is for each node in the system operate only with a predefined *lifespan*, as low as a minute. This time unit can be a system time unit or upon completing certain number of n transactions/service responses in which translates to the time it takes to complete n (i.e., seconds/minutes). Upon reaching this *lifespan*, the node is terminated and instantiated on a different platform, we call it *Node Reincarnation*, discussed in section 3.2.3. This is to reduce the exposure attack window time of the node and subvert in progress attacks while continuously re-assessing the system state based on the observations of the nodes that are not being *reincarnated*. Thus, it is intuitive to see that defending systems in T_0 for the run time on the entire replicas (traditional deployment) is extremely challenging in comparison when defending it in T_i, where $i > 0$, and T_j is within minutes.

Therefore, it is critical to abstract the traditional application runtime model with *Time-Interval Runtime Execution Model*, which we will discuss in details in section 3.3. This abstraction transforms the system run time into observable (with respect to security) system states. However, the key design challenge inherent in such run time execution model is dealing with the *application state* between the terminating and the new instance/node without disrupting the computation.

Application State

Generally, application state is an abstract notion of a continuous memory region of the application at runtime. Breaking this runtime into intervals (chunks) across nodes, will break the continuity of that region, however, the implementation of such abstraction is dictated by how the application constructs and preserves it at runtime. Thus, the challenge of transferring application state between a terminating node and a new node lies the communication model (i.e., *synchronous* vs. *asynchronous*) between the interconnected applications/services or between the client and the servers.

For example, the state information of Byzantine Fault-Tolerant Replicated systems *(i.e., synchronous system model)* [53], manages a static and a dynamic part of the system state. The dynamic part is typically written in a file to assist the recovering replica, thus, transferring that file implies state transfer. Another example of a *asynchronous* system is the event based systems where the state is the registered subscriptions and the events entering in the system. Terminating the node with the registration information require transferring the information to the new node.

In most applications, the static part of the application state is called the system configuration files which is typically saved in a file (`system.config` or `hosts`). The static information in these files typically contains the application parameters like the number of participating replicas and their IP addresses, the database connection strings, security keys/certificates, etc. These parameters are not updated at runtime unless the application implement protocols to handle this update, for instance, replicated systems that allow replicas to join or leave the systems.

Yet another widely adopted example is in the web services domain, for example, RESTful web services, a stateless web service (client/server) model where the client requests are processed and responded as they enter the system, thereby, no state is preserved. In contrast, for stateful services where the services are bound not only their communication protocols like WS-Secure Conversation, referred to as SOAP-based services, for instance, but also their access control token during that session.

Managing the dynamic part of the application state in a generic fashion is not feasible, since it's application dependent. In *Mayflies*, we exploit the built-in reliability properties of the application where applications retry to connect to the service/replica for few times before it gives up. Our reincarnation process completes within these tries. Thus, we don't consider transferring the dynamic part of the application state (i.e., TCP connections, security tokens). These states are typically exchanged between the running replicas and the recovering one, where in our case, is the reincarnating node. We show two widely adopted from different classes of systems *(synchronous and asynchronous)* as use cases in Chapters 4 and 5.

3.2.3 Framework Components

Figure 3.2. show the cross section view of the *Mayflies* cloud platform. At the core, is the OpenStack cloud management framework where the nodes/VMs are provisioned and deprovisioned on the hardware (HW1. . . HWn) mediated by the hypervisors (HV1. . . HVn), depicted on the second and the third rings. The arrows represent the node randomization and diversification techniques of *Mayflies* across these hardwares. The *LibVMI* and *SDN*, depicted on the rectangles, are for the proactive monitoring component and the network programming respective layers. Note that the clients access through the externally feasible IP addresses (192.x.x.x) and the VMs are interconnected with the internal IP addresses (10.x.x.x).

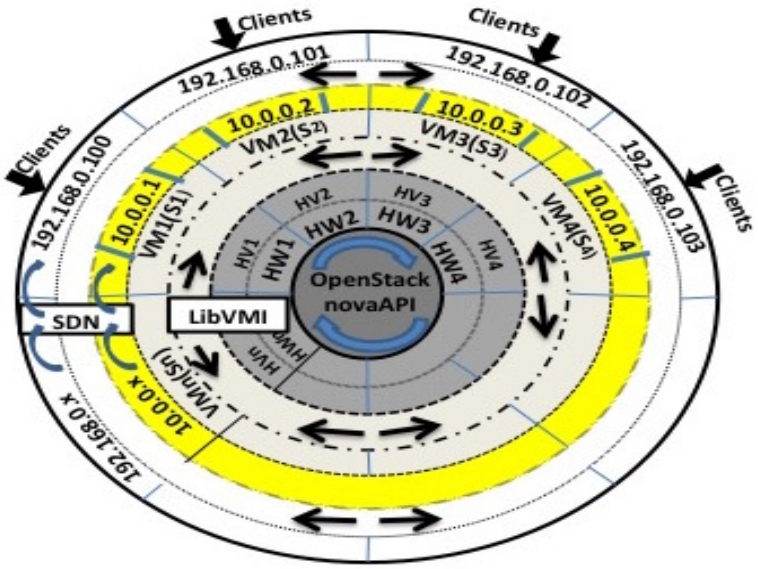


Fig. 3.2.: Cross section view of Mayflies cloud platform

Mayflies implements MTD algorithms that combines the cloud framework (i.e., Openstack) components; *nova compute*, *neutron*, and *Virtual Machine Introspection (VMI)* for detecting runtime integrity violations in real-time. The *nova compute* is designed for provisioning/de-provisioning VM instances on the cloud platforms. We continuously provisioning/de-provisioning nodes in time intervals at run time, dubbed

Node Reincarnation. We use *neutron* to dynamically reconfigure the network during the *reincarnations*. We leveraged *libvmi* [37], a library for virtual machine introspection (VMI) for *pro-active node monitoring* or application runtime. We describe each one of these components in details in the following three sections.

Proactive Node Monitoring

Pro-actively monitoring the nodes during their short *lifespan* is critical. The key idea is to prioritize node reincarnations with respect to the overall system state to prevent reincarnating nodes on a compromised cluster or reincarnating a node due to it's lifespan while another compromised node is in the system, thereby, effective monitoring prevents us from blindly moving nodes across platforms. The easiest method to get the node status is by *pinging* the node, however, *Mayflies* key objective is defend systems against advanced attacks, depending on the existence of the node status does not say anything about attacks. We define node status as follows:

Definition 3.2.2 *Node status in Mayflies is defined as the node to be clean if the observation from the internal representation of the node's runtime (i.e., memory, CPU) integrity is intact and dirty if the integrity is violated.*

We are interested in monitoring schemes at the infrastructure-level. In cloud platforms, there are numerous ways of achieving this capability. The hypervisor is the core machinery that mediates between the virtual resources of the *VM* and the physical resources such as memory and CPU. The transparent mapping of the virtualized OS memory into the physical memory enabled by the hypervisor opens the opportunity to safeguard systems below the OS which is difficult to subvert by attacks originated inside the OS.

Thus, we leveraged VMI [37] for this case. In VMI, for instance, when the application is hijacked, the address offsets show new entries for the injected code. Another instance is when the application is terminated and a new malicious one is started in which possibly ends up with a new process ID and/or different memory address offset

in its virtual memory address space. Note that VMI is a powerful memory inspection tool used for Malware analysis and other intrusion detection methods. Since we are interested memory analysis at runtime, we have adopted in its simplest fashion which has a negligible performance overhead [38]. The details of the implementation and defensive capabilities are discussed in Chapter 2. Algorithm 1 illustrates the introspection procedure, INTROSPECT().

Algorithm 1 Virtual Introspect

```
1:  Input: node
2:  Output: true or false
3:  procedure INTROSPECT(node)
4:      if node == new then
5:          initialProc ← GetProcessMemory(node)
6:          return false
7:      else
8:          currentProc ← GetProcessMemory(node)
9:          if initialProc_i(key, val) ≠ currentProc_i(key, val) then
10:             return true
11:         else
12:             return false
13:         end if
14:     end if
15: end procedure
```

In algorithm 1, the procedure saves the initial memory information of the node in line 5 and returns false for a clean new node. Then, returns accordingly when the running node's information is different/altered from the initial stored information in lines 8 and 9. The result can be either *true* if anomaly is detected in the memory structure, otherwise *false*. Note that we can check any key/value pairs in the memory data structure such as the start/end address offsets of a given process.

Formally, let $\{O_j, \text{j=1,2,}\dots\}$ be observations of the node 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 node is *clean* and $O_j = 0$ indicates the node is *dirty*. The *dirty* node can be either missing (i.e., network drop) or it's *compromised* (i.e., VMs address space altered).

Network Dynamics

In order to break the application's runtime into manageable time-intervals, we need to separate the network interface known to the users from the VM in order to attach it to the substituting node without the user's knowledge. This node can be from a pool of prepared nodes or a newly created VM. VMs are typically interconnected with a *fixed* IP addresses, similar to a LAN setting in a corporate network, and are reached by the clients through a *floating* IP addresses through a virtual router. The prepared nodes can be created on the network with fixed IPs (i.e., LAN IP assigned by DHCP but not externally feasible) or off the network (i.e., no network card).

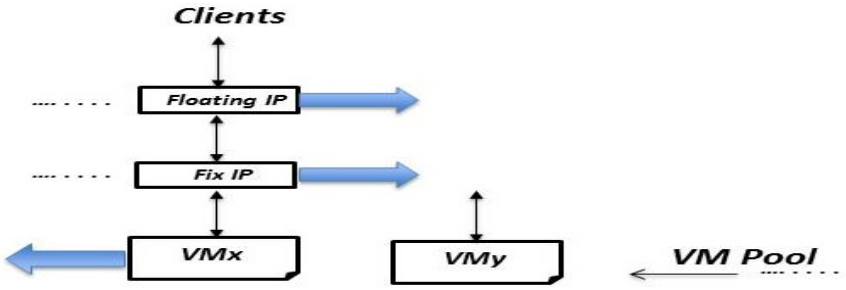


Fig. 3.3.: Illustration of VM compute and SDN interface interchanges. VM_x seamlessly replaces VM_y from a pool of VMs.

Algorithm 2 Network Interface Swap

Input: *nodeState, newNode*
Output: $newNode_{netInterface}$ ▷ new node with network interface

```
1: procedure InterfaceSwap()
2:     if nodeState_interface == NULL then
3:         newNode_netInterface ← NewNetInterface()
4:     else
5:         newNode_netInterface ← nodeState_netInterface
6:     end if
7: end procedure
```

The node Network interface swapping procedure is illustrated in Algorithm 2. The procedure simply creates a new interface if the node is created without an interface (an

stand by VM), otherwise, attaches from the interface from the old node. The method of creating a new network interface in NewNetInterface() or swapping with existing one is described in details in the implementation section. Figure 3.3 illustrates the process of swapping the network interface between two VMs (VM_x and VM_y). This is achieved with the *Software Defined Networking (SDN). SDN* is a programmable networking fabric that decouples the control plane from the data plane (i.e., switches). The OpenStack *neutron* component implements the SDN interfaces and others are enabled indirectly through the *nova* component.

Node Reincarnation

Reincarnation is a techniques of enhancing the resiliency of a system by terminating a running node and starting a fresh new one for its place on (possibly) a different platform/OS as it dropped off of the network and reconnected it. The node reincarnation procedure is illustrated in Algorithm 3. In REINCARNATE() procedure, we first save the nodes application state then destroy (deleting the VM) it in lines 2 and 3. We get a new node for the pool in line 4, then, swap its network interface in line 5, and transfer its state in line 6. The *GetNewNode()* method can be implemented in two different ways; by selecting a new VM from a pool of VMs or freshly booting a new instance on demand. We discuss the pros and cons of each method in section 5.3.

Algorithm 3 Node Reincarnation Procedure

Input: *targetNode*
Output: Substitute *targetNode* with a *newNode*

1: **procedure** REINCARNATE()
2: $nodeState \leftarrow targetNode_{state}$
3: $DestroyTarget()$
4: $newNode \leftarrow GetNewNode()$
5: $InterfaceSwap(nodeState, newNode)$
6: $newNode_{state} \leftarrow nodeState$
7: **end procedure**

Formally, let n be a node in *Mayflies* as defined in definition 3.2.2, a reincarnation 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.
- $n_\rho \in [n_{start}, < \rho | O_i^t >]$, represent the *lifespan* of the node 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_i^t time-interval t due to attacks. Observations $O_i \in [0, 1]$ represent the node 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 reincarnated to n' with a new predefined life expectancy ρ', thus, it's n_j tuple; $n_j = \langle n_{start}, n_\rho \rangle$

3.3 Mayflies System Model

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) [20].

3.3.1 Model Description

A Markov chain or process is a sequence of events or states Q=$\{q_1, q_2, \ldots q_n\}$, and a Hidden Markov Model (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

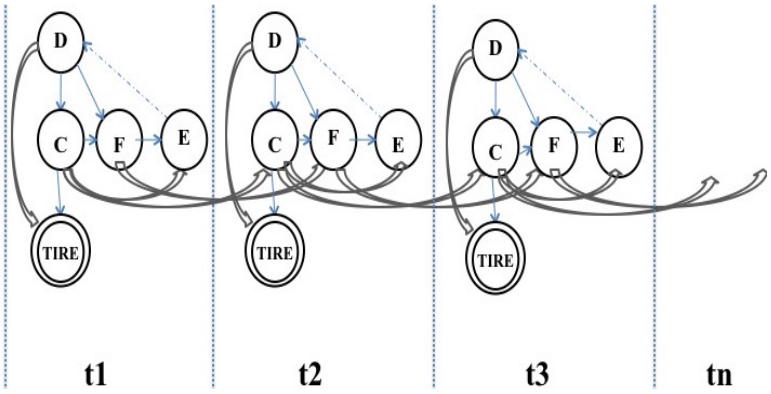


Fig. 3.4.: Mayflies DBN System Model – system states are Desired, Compromised and Failed labelled as D, C, and F, followed by the Exit state E. The dotted lines on E depict for the control returning to the parent node. TIRE is the observing state in double circles.

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 [54].

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 3.4, we construct an HHMM in which the hidden states S are *Desired, Compromise, Failed* and *TIRE* as the omitting/observable state (discussed next). We define the topology of the hierarchy as follows: The *Desired* state as the root state(i.e., initial state), the *Compromised and Failed* states in level II, and *TIRE* as the leaf state in level III, referred as (S_D, S_C, S_F, S_T) respectively

We define S_T *emissions* as node status observation captured by the proactive monitoring tools 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 in the introduction section in Figure 1.1(c). 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_T emit *in-active*, or to a *Compromised (S_C)* state if S_T emit *dirty*. Otherwise, stays in *(S_D)*, i.e. *node (s) is active*.

With this HHMM construction, we model the system with *Dynamic Bayesian Network* [23]. As depicted in Fig 3.4, 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'* objective.

To illustrate how we map the VMI observations to the high-level system states, consider at time *t=1* in Figure 3.4, the system starts in a *desired (S_D)* state and consider *TIRE (S_T) emission=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 E signalling for its exit. At this point, we reincarnate the compromised node so the next time in *t=2*, the *emission=active* and we transition to S_D at *t=3*.

Thus, modelling *Mayflies* with HHMM and formulating and encoding it in this manner, we can reason the state *transitions* guided by the *(emissions)* in terms of the overall proportion of the time $\{t_2, t_i, t_k, \ldots\}$, the system was compromised.

3.3.2 Time-Interval Runtime Execution Model

The key defensive strategy of the proposed framework is to avoid threats in time-intervals rather than continuously defending the systems for its entire *runtime executions*. To achieve this, we introduce an abstraction layer on top of the traditional runtime execution, dubbed *Time-Interval Runtime Execution (TIRE)*. *TIRE* is to break the runtime execution into intervals where in any interval we re-evaluate the system and determine its high-level state based on the observation collected by the monitoring component in that time-interval . This evaluation can result in the system to be still in the *desired* state or *compromised/failed* in which we can adjust to our *desired* state accordingly, thereby, guaranteeing to disrupt the attackers gain/loss balance and the unbounded time of controlling the system when it is compromised.

Definition 3.3.1 *Runtime Execution of distributed systems is typically defined as a set of infinite sequences of states in Q, donated by* Q^{ω}.

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 reincarnated 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 $\{Q_n^i, Q_n^j, Q_n^k, \ldots.\} \in Q^{\omega}$, thereby, allowing us to safeguard the individual nodes 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), then we reincarnate the comprised node(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 of whether a node status has changed or not is through an HMM, hence, makes *TIRE* the perfect choice of being the observable state in our HHMM construction described in the previous section.

HMM-based TIRE Transitions and Observations

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:

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

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* nodes. *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. 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 *TIRE (S_T)* 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 as depicted in Figure 3.4, 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.

3.3.3 Mayflies System States

As depicted in Figure 3.4, we defined three hidden states S_D, S_C and S_F and an observable state S_{TIRE} that omits node status described above. Note that, in this model, one can add many hidden and/or observing states in any levels; horizontally or vertically. 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.

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 ***Property 1***.

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 in ***Property 1***.

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. We will discuss these issues and our solution approaches in the evaluation section.

3.3.4 Calculating State Transitions

We assume nodes in *Mayflies* start with pristine status where they perform computations within a predefined *lifespan* and being reincarnated at specified time-intervals. The key objective is to ensure the system stays in *Desired* state as often as possible, and bounce back if ever transitions to any of the *Undesired* states. Thus, we are interested in the long-run distribution of the process/runtime execution, for example, the long-run proportion of the time that we are at *desired* state overtime.

Problem Formulation

Given our three hidden states S=(S_D, S_C, S_F) for $\{S_{Desired}, S_{Compromised}, S_{Failed}\}$ and we are initially starting at S_D, we formulate the problem as a *binary random walk* on the set of these 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_D in the first time-interval, then one time-interval later we will be at either S_C, S_F or stay in S_D according to the outcome of the observation collected in that time-interval *(T_i)*. These observations can be viewed of as rolling a fair die for every one minute, for example. We move to S_C if the die comes up 1 or 2, stay at S_D if the die comes up 3 or 4, and move to S_F in the case of a 5 or 6. Mapping the die throws in *TIRE* observation context, in each time-interval T_i, we collect observations about the nodes where they are either a) *clean*, reachable and

internal runtime virtual address space is intact, or b) *dirty*, could not be reached due to network drop or detected their runtime is corrupted. It's intuitive to see that these observations are also probabilistic in nature.

Example Solution

Let S_t represent the *state* of the system at time t where *t=1,2,3... minutes* and the time-intervals T_i where i = *0,1,2,...* n is in t. Note that the Markov process model is an exponential distribution, in that the decisions are dependent only in the current state. Probabilistically, if we are at *Desired* state now, the probability to any other state will be *1/3rd* no matter where we were *(Failed or Compromised)* in the past.

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:

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

where $\sum_{i=1}^{j} S_i$ is the total number n of visits the process makes to each state S_i over the time-intervals $T_t \in T_0, T_1, T_2 \ldots . T_n$. 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 *reincarnations* are effective.

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 $< x, y, z >$ time units for the three states, which means that in the long run we spent x amount of our time at the *desired* state S_D^x, y amount of our time at *compromised* state S_C^y, and z amount of our time at *failed* state S_F^z. Thus, we can easily reason about the high-level system states in time intervals, for instance, if we run the system for 1 hour, then, we can 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.

3.4 Implementation and Evaluations

Since *Mayflies* is designed as a generic MTD framework for distributed system, evaluating it against numerous types of systems is infeasible. As noted earlier, system classes *(i.e., Synchronous and Asynchronous)*, architecture and design are dependent on specific system model (i.e., BFT systems), thus, we discuss the use cases in a generic fashion and give general guiding principles of deploying distributed systems on *Mayflies*. We then discuss the prototype implementation details, algorithms, and the inherent challenges of extending the cloud management framework with a generic MTD capability. Finally, we present methodologies of evaluating the framework's resiliency to attacks. To illustrate the practicality of the framework, we present a *Synchronous* system class prototype on *Mayflies* in Chapter 4 and *Asynchronous* system class in Chapter 5.

3.4.1 Use Cases

The are two common classes of distributed systems; *Synchronous* and *Asynchronous* as discussed in details in the introduction chapter. The RESTful client/server model, request/response, push/pull, and event-based systems represent Asynchronous systems. The Byzantine fault-tolerant system is an example of a widely studied/deployed synchronous system class. These systems are typically replicated using chain replication, slave/master or quorum-based replication models, to name a few, for reliability reasons.

These replication models require different schemes of *reincarnation*, for instance, reincarnating a node in chain replication model, one must transfer all the data from the terminating node to the new node in order to proceed with the computations. For the slave/master replication model, the slave takes control when reincarnating the master in which their data is typically synchronized, thus, computation continues but the data should be handled accordingly. Most importantly, the quorum-based replication model require more than 2/3rd of the nodes to be running in synchrony

in addition to dealing with the their data during reincarnation. We consider this replication model in our use case to illustrate *Mayflies* prototype. This allow us to illustrate not only the randomization and diversification aspect of the our MTD solution approach, but also, the defensive aspect of the proactive monitoring for those nodes that are running at all times.

The core of our MTD scheme is the *time-interval* node *reincarnation* and node *observations/emissions*. The *observations/emissions* is the driving engine of the high-level system states (i.e., desired/undesired), however, the guiding principle for these state transitions are application dependent. Thus, we give a general guiding principle. In Chapter 4, we deploy a BFT system on *Mayflies* framework and show experiments using the system specific guiding principle.

General Guiding Principle

As noted earlier, nodes in *Mayflies* have a predefined lifespan. Generally, for a single node deployment or non replicated (i.e., web-server), we need to reincarnate/refresh that node in a pre-specified time intervals and observe the status within that time frame. Within each time interval, if anomaly detected, consider the system in *compromised* state else *failed* state, then reincarnate it (possibly) on a different cluster before reaching it's lifespan. Otherwise, consider the system in the *Desired* state.

3.4.2 Mayflies MTD Algorithm

The core of our MTD defensive tactic is to move nodes across platforms, referred to as *node reincarnation,* while we proactively monitoring at the hypervisor-level in order to guide the direction/safety of the move (i.e., avoid vulnerable platforms). There are different strategies to reincarnate a node, we give two examples; *round-robin* and *random*. Algorithm 4 below illustrate these strategies.

In Algorithm 4, lines 3 through 13 show the *round-robin* strategy. We continuously reincarnate nodes in round robin fashion, going through the list of nodes over and over again. This can be implemented, for example, a circular linked-list. The second strategy is reincarnating a node by simply selecting it randomly shown in lines 14 through 27. Assuming the node IDs are numbered *1...n*, we simply generate a random number within the range of the node IDs and reincarnate accordingly.

Algorithm 4 Mayflies Algorithm

```
1:  Initialize the replicas and time-interval x/lifespan
2:  while true do
3:      if strategy = RoundRobin then
4:          repeat
5:              isDirty ← INTROSPECT(replica_i)                ▷ any dirty node?
6:              if isDirty then                                  ▷ in algorithm 1
7:                  REINCARNATE(replica_i)                ▷ terminate the dirty node first
8:              else
9:                  targetNode ← GETNODE()            ▷ scheduled node in ordered list
10:                 REINCARNATE(targetNode)                     ▷ in algorithm 3
11:             end if
12:             WAIT(x)                              ▷ sleep for x minutes/transactions
13:         until stop MTD condition met
14:     else if strategy = Random then
15:         repeat
16:             isDirty ← INTROSPECT(replica_i)                ▷ any dirty node?
17:             if isClean then                                  ▷ in algorithm 1
18:                 REINCARNATE(replica_i)                ▷ terminate the dirty node first
19:             else
20:                 repeat        ▷ get a different node than the one just reincarnated
21:                     id ← RANDOMGEN()   ▷ get a random number within ID range
22:                 until id ≠ replica_iID
23:                 targetNode ← GETNODE(id)
24:                 REINCARNATE(targetNode)                     ▷ in algorithm 3
25:             end if
26:             WAIT(x)                              ▷ sleep for x minutes/transactions
27:         until stop MTD condition met
28:     end if                                                  ▷ many more strategies
29: end while
```

Note that INTROSPECT($replica_i$) in lines 5 and 16, described in Algorithm 1 in section 3.2.3, is an implementation dependent. For instance, we need to introspect the replica index i from the list in descending order and reincarnate in ascending order for the *round-robin* strategy. For the *random* strategy, we don't need to reincarnate the node that was just introspected.

3.4.3 Implementation

We implemented our algorithms with *bash* shell scripts tightly integrated into the OpenStack (Kilo) [7] framework, an open source cloud management software stack. As noted earlier int he design section above, OpenStack provides a modularized components (i.e., computing virtualizaiton and SDN) that simplify cloud management and ease of integration. With this, by orchestrating the interfaces implemented in these components, we extended the cloud framework with our *Mayflies* MTD framework. In Algorithm 4, there are five procedure calls: GETNODE(), INTROSPECT(), REINCARNATE(), RANDOM() and WAIT(). The INTROSPECT() and REINCARNATE() are described in algorithms 1 and 3 in section 3.2.3. The implementation is as follows:

GetNode() and Wait()

Depending on the data structure used to keep track of the nodes, this GETNODE() procedure is simply extracting a target node from the list, for instance, by index if it is a `list` or an `array`. The target node is selected randomly in the *random* strategy using a basic random generator function. Similarly, The WAIT() procedure is simply `sleep(x)` method call for x amount of time, else *lifespan* is used where the node self-terminate after x number of transactions/execution completes discussed in the use case illustrated in Chapter 4.

Introspect()

We leveraged *LibVMI* [37], an open source library for Virtual Machine Introspection. Algorithm 1 in section 3.2.3 illustrates the detection scheme, and the details of the implementation is discussed in Chapter 2. In summary, we first take a snapshot of the application's memory before we deploy/assign an IP address. We next take snapshots in time intervals and compare specific elements in the address block like the address offsets and alert if entries mismatch.

Reincarnate()

The reincarnation procedure is to reincarnate a target node if it is found dirty (i.e., compromised) by the introspection procedure, otherwise reincarnate as scheduled, illustrated in lines 7 and 10 for the *round-robin* strategy, and lines 18 and 24 for the *random* strategy in Algorithm 4. Assuming that the adversary can learn the tactics used for reincarnating nodes, for instance, when using *round-robin* strategy, the attacker can focus attacking those nodes that have longer exposure attack window or are last in the list/array. To balance, the introspection monitoring scheme should be constantly monitoring those nodes than those that are soon to be reincarnated.

There are different ways to implement node reincarnation in OpenStack as discussed in section 3.2.3. The `nova boot <options>` lets you create nodes, where the options specify the type of the node; cluster, OS type, etc. Depending on the time-criticality of the application, a node is booted on-demand or selected from prepared pool of VMs without network interface attached or prepared with temporary interfaces while their successor frees the network interface that is known to the client or the services in which is then the two interfaces are swapped.

The Reincarnate() procedure uses InterfaceSwap() procedure as illustrated in Algorithm 3. This procedure is implemented as follows: we first save the *port ID* associated to the terminating replica (the input *replica*). In SDN environment, the VM is attached to a virtual network interface that is referred to as *ports* with a *fixed* IP

similar to physical network interfaces. This interface is also associated with *floating* IP for external access as noted earlier in Figure 3.3. Thus, both of the IP addresses are part of the *port* even after it's separated from the VM, thereby, transferable to another VM. We detach the port off of the replica with `nova interface-detach <newReplica portID>`, we then get a *new replica* VM instance from the pool and attach the *port* to it. Note that depending on the OS image of the replica, a VM reboot is required after the `nova interface-attach <portID newReplica>`. At this point, the clients re-connect to this replica through it's floating IP (*128.x.x.x*) as the old server that dropped off of the network and came back.

The pseudo-code below reflect the implementation logic and a code snippets:

Algorithm 5 Reincarnate

1: **if** $nodeHasNetworkPort$ **then**
2: $nova\ interface - dis - associate < VM_{old}, FloatingIP >$ ▷ remove IP
3: $nova\ interface - associate < FloatingIP, VM_{new} >$ ▷ give IP
4: **else**
5: $neutron\ port - create < options >$ ▷ create virtual network card
6: $neutron\ port - attach < options >$ ▷ attach card
7: **end if**
8: **if** $nodeHasNetworkInterface$ **then**
9: $nova\ interface - detach < VM_{old}, VM_{old_{port_{ID}}} >$ ▷ remove network interface
10: **else**
11: $nova\ interface - attach < VM_{new} VM_{old_{port_{ID}}} >$ ▷ give network interface
12: **end if**

For the node without the interface, we use `neutron port-create <options>` to re-create the interface with attributes used by a terminating VM and then pass to another VM with `neutron port-attach <options>`, thereby, allowing the servers (if replicated) to continue using the known interface. With these capabilities, we can reincarnate nodes across subsets and networks, an MTD scheme known as *IP Hopping*.

3.4.4 Evaluations

Mayflies' defensive strategy is controlling the node's attack exposure window in time-intervals rather than the entire runtime through proactive *node reincarnation* and *node observations* schemes. Hence, the three competing time of the framework are the following:

- *T*he Reincarnation Cost RC(t_i) – the reincarnation time t_i of a node.
- The *Observation Time OT(t_i)* – time it takes to detect attacks with *virtual introspection*.
- The *Attack Success Time AS(t)* – is the time it takes for an attack to succeed.

To assess the upper bounds of our MTD attack resiliency capabilities under the constraints of the cloud framework (i.e., OpenStack), we quantify the attack success times *AS(t_i)* relative to *RC(t_i)* and *OT(t_i)*. Formally,

Let μ be the expected overhead time of *reincarnating* a node n and ν be the expected overhead time of node *observations* in one time interval t_i, then:

$$RC(t_i) = \sum_{i=1}^{j} \mu_n$$

and

$$OT(t_i) = \sum_{i=1}^{j} \nu_n$$

Thus, in order to stay in a *desired* state as often as possible, we need to keep:

$$RC(t) + OT(t) < AS(t)$$

where *RC(t)* and *OT(t)* is the cost of our defensive strategy of the overall time t against the attack success time.

We evaluate the performance impact of *Mayflies* with BFTSMaRT [55], a widely studied Byzantine Fault Tolerant system prototype in the literature in the next chap-

ter, and a publish and subscribe system in Chapter 5. These two systems represent two classes of different system models.

3.5 Conclusion

In this chapter, we presented *Mayflies*, a bio-inspired MTD framework, and discussed the formal model, design and implementations, and algorithms. The key challenge of designing and implementing the proposed generic MTD framework lie the partitioning of the runtime execution into time intervals, and synchronizing the OpenStack APIs (*nova/neutron*) to provision/deprovision VMs/nodes and reconfigure the networks in a dynamic fashion. We discuss next on how we addressed these challenges.

3.5.1 Partitioning Application Runtime

Partitioning the traditional runtime execution of the system seems a performance burden on the applications. Further, it has been reported that the constrained *VM* replacement requirements from the applications can result in temporary violation against the placement algorithms used by the underlying infrastructure [56] , thereby, possibly causing minor delays when such conflict happens. For a private cloud, we show in the next chapter that the performance impact of BFT system deployed on *Mayflies* framework is negligible. Our reincarnation scheme is similar to the *VMentry and VMexit* processes of the hypervisor for mapping the virtual computing environment to the physical resources (i.e., CPU, memory) by the scheduler where a VM is stopped and another one resumed.

The inherent performance issues in virtualized cloud platforms that are outside the users's control, for instance, is referred to as performance variability of the day, first reported in [57]. Furthermore, 22% of vital issues reported by cloud system developers relate to performance bugs [58]. Clearly, the defensive solution approaches

designed while considering the underlying computing fabric is far superior to traditional independent ad-hoc solutions.

3.5.2 Synchronizing Nova and Neutron

The process of reincarnating a node in our framework is greatly simplified by the combination of *nova* and *neutron*, however, the implementation of these capabilities are asynchronous by design, the functions have no return values to determine whether the call succeeded or failed. 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 these *nova* interfaces are implemented by the *neutron* component. In general, all inter component *(i.e., nova, neutron, horizon, glance, cinder, etc.)* calls in OpenStack software stack is done through RESTful messaging (i.e., AMQP).

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.

4. BYZANTINE FAULT-AVOIDANCE

The Byzantine Generals problem was first introduced by Lamport [59] as an abstract notion of constructing a provable reliability-guaranteed replicated distributed system. This replication model has the ability to cope with malfunctioning and byzantine/arbitrary faulty components. The State Machine Replication (SMR) approach is considered one of the effective ways of implementing a Byzantine Fault Tolerant (BFT) system [60]. SMR resolves the interactive consistency conditions for distributing single source of data to multiple channels by enforcing the replica to start in the same state, execute client requests, and unanimously respond to it in ordered fashion, thereby, enabling to reach agreement even in the face of few faulty replicas. This, satisfies the correctness condition properties; *safety* and *liveness*. The *safety* property asserts that some of the replicas remain consistent with one another, and *liveness* guarantees that the clients will eventually receive responses for their requests.

For decades, BFT research was considered theoretical due to its impracticality for implementing it in a real-world setting. A Practical Byzantine Fault Tolerant (PBFT) system [61] that achieves performance close to a non-replicated system was published and open sourced to help the ever-increasing need for reliable distributed systems prototype. PBFT has sparked a wide-array of research to further improve its performance, for example, reducing communication steps, replication costs, and addressing security issues which is the focus of this chapter.

4.1 Motivation

The performance improvement of BFT systems was mainly credited to the advancement of cloud computing, for example, virtualization techniques have been used on improving replications costs from *3f+1* where *f* is the faulty replica, to f*+1* in

ZZ [62], CheapBFT [63] and A2M [64]. While replicating systems on a highly dynamic virtualized elastic cloud environment is undeniably cost effective, it is increasingly challenging to guarantee reliability due to the inherent increase in attack surface [39] – the set of ways an adversary can exploit/penetrate the systems, due to the number of components built on these platforms.

Amir et al. first reported in "Byzantine Replication Under Attack" [28] that PBFT is vulnerable to performance degradation attacks. The core of such vulnerability is the quorum-based consensus protocol where n servers exchange messages to coordinate with a single selected leader node/server to reach consensus while there are some faulty ones in the mix. A compromised leader can increase latency and reduce throughput by delaying responses (state/view change or coordination messages) just in time to avoid detection or protocol time outs to make the system barely usable. They argued the insufficiency of the correctness condition for BFT, and introduced Prime [65], a new BFT protocol with bounded-delay performance criterion. Prime extends the existing BFT's agreement protocol with an additional step, *pre-agreement*, using reliable broadcast protocol.

The difficulty of determining the upper bound of the bounded-delay that defines an acceptable level of performance was later addressed in BFT-Mencius [66]. BFT-Mencius introduced an Abortable Timely Announced Bounded-Delay broadcast protocol that is on the order of real communication delay. Other notable works that address this issue include; Aadvark [67], which proposed a change in leadership when suspected, i.e., when the leaders' performance is slowing down. Along the same lines of work, Spinning [68] constantly rotates the leaders' role after every patch of accepted client request for execution, similar in spirit to BFA.

All of the above approaches have concentrated on the application protocol layer which is often defeated when the attack is originated outside the application (OS kernel). With the ever increasing sophisticated attacks in recent years, the computing landscape differences between the traditional computing platforms and the virtualized cloud environments have amplified these attacks. This is due to the inherent

complexity of the interdependent components and the programmable network model (SDN) on these platforms, thereby, defending replicated system (i.e., BFT) with the existing solutions is extremely challenging. For instance, in a cloud platform, a compromised BFT leader/server not only disrupt the reliability of the BFT protocol but can also wage several attacks such as attacks on the SDN controllers and the data plane forwarding flows by poisoning the entire network topology or even take control of the entire infrastructure [9].

The fundamental problem of BFT security issue is that *fault-tolerance* and *attack-tolerance* techniques is a double edge-sword. On one hand, replication is the ultimate solution for availability and *fault-tolerance*. On the other hand, replication increases the overall system attack vector (i.e., increased the number of nodes to be protected and resist attacks). Therefore, we believe shifting from a perceived over-emphasis on improving BFTs' protocols to designing architecturally resilient replicated systems that reflect on the underlying computing fabric is critical. We deploy a BFT prototype to the proposed *Mayflies* framework discussed in chapter 3 and evaluate its performance impact while the replicas on the move across platforms.

The framework enables the BFT system avoid byzantine faults through controlling the replicas' exposure attack window while simultaneously preserving the correctness condition of BFT properties; *safety* and *liveness*. Such control is achieved by allowing replicas (including the leader) to exist only for a short period to complete n client requests on a given underlying computing platform, then vanish and appear on a different platform with different characteristics, i.e., guest OS, Host OS, hypervisor, hardware, etc. As a result, enable a tight architectural-level integration of *attack-tolerant* to *fault-tolerant* protocols through avoidance. Thus, we view our approach as Byzantine Fault-Avoidance (BFA).

4.2 Background

Byzantine Fault Tolerance (BFT) is a well-established reliability guaranteed distributed system based on state-machine replication model. PBFT [69] is the first practical implementation of a leader-based BFT replication model that has been widely studied in the literature. PBFT systems consist of a *primary* node and n replicas; the basic operation (sketch) of the algorithm is as follows:

A primary node, referred as the leader, exchanges consensus messages to n replicas. The primary's main task is to assign monotonically increasing sequence numbers to each of the client's requests and start a three-phase agreement protocol; *propose, prepare, and commit* respectively. Initially, the leader assigns a sequence number for every request (from the clients), then multicasts to the other predefined replicas in the *propose* phase. The replicas confirm the receipt of the request back to the leader and transitions to the *prepare* phase. The leader then sends the execution approval back to all the replicas to transition to the *commit* phase, thus, reliably completing the request. Upon the completion, all the replicas send the response to the client, thereby, guaranteeing task completion even in the case of a few faulty replicas.

4.3 BFA System Model

BFT systems are typically implemented using State Machine Replication (SMR) model [60] and formalized with I/O automaton [70], therefore, it is a natural fit to model BFA as an automaton.

Consider system B as a BFT's finite state automata (*FSA*) system model. Typically, system B consists of four tuple automaton, B = *(sig(B), states(B), start(B), steps(B))*, where *sig(B)* are independent actions *acts(B)* which consist of *in(B), out(B)*, and *internal(B)*, of input, output, and internal actions respectively. A set of *states(B)* consists of a *non-empty* set, and a start state *start(B)* $\subseteq$ *states(B)* of states. A transition relation, *steps(B)* $\subseteq$ *states(B)* X *acts(B)* where the actions *acts(B)* are

the client request, the *propose, write, and accept* consensus protocol messages, and consensus decision as *in(B), out(B)*, and *internal(B)* respectively.

Similarly, we model BFA with *FSA* and call it system A. System A will also have the same four tuples, the set of states and a transition relation, however, the difference between the two *FSA* are their *input, output and internal* actions. For BFT, the client requests and the responses are considered the input and the output, and the consensus state transition protocols *propose, write, and commit* are the *internal* actions. We are interested one of the *internal* action *(accept)* from system B (BFT) to send to system A (BFA) as an *input* action.

4.3.1 BFT to BFA Transitions

In this work we are considering quorum-based also known leader-based BFT systems. Typically, leader-based BFT protocols consist of *prepare/propose, write, and accept* state transitions where the leader exchange messages ordering replicas to execute client requests. Upon the completion of each client request, a decision is reached depending on the non-faulty replicas participating the vote. The system transitions to an *accepting* state if the number of voted replicas are within the acceptable majority. We refer this decision as *BFTCommit*.

In BFT, the state machine is initially triggered by the client requests which is considered as *input* action, and the consensus transitions which are considered as *internal* actions happens next. One of these *internal* actions is the *BFTCommit* transition which happens upon successfully completing/committing the transaction. We need to pass this transition action to our BFA system as an input to trigger its internal actions. However, for an I/O automaton, the *internal* actions of SMR-based system are not visible to other systems in the same environment [70].

State transitions in BFT systems are just an abstract notion of implementation dependent persistence workspace, i.e., continuous memory region. In order to eliminate state synchronization complexities, BFT systems require logging every accepted

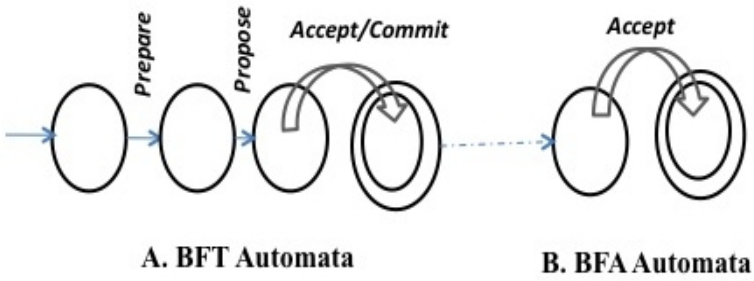


Fig. 4.1.: Illustration of finite state automata composition for BFT in A, and BFA in B. The transition of the BFT *accept/commit* state triggers as an input to BFA (dotted arrow) and transitions it to an *accept* state for refreshing a replica.

execution for the recovery process/server in the event of natural crashes. We consider every log event as the *output* action from system B that can be used as an *input* action to our system A (BFA). We assign this checkpoint event of the BFT system as the *output* action (*out(B)*) and use it as an *input* action (*in(A)*) for BFA.

Formally, the BFA's state transition steps can be defined as *steps(A)*$\subseteq$ *acts(B)*, where *out(B)* is the output action *(BFTCommit)* in *act(B)*. This eliminates state transfer complexities and effectively allow us to easily reason about the preservation of the BFT's reliability properties (*safety and liveliness*).

In order to perform useful computations, we consider a transition after n accepted *BFTCommits*, thus, indirectly trigger BFA system to transition to an accepting state as depicted in the dotted lines in Figure 4.1. Thus, we consider this transition point to be used as the application state transfer checkpoint between the terminating and the newly created replicas, discussed in section 4.5.2. The input enabled action of BFT's I/O automation shows clearly that BFA is suitable for any SMR-based deterministic system with state and some operations without any modification.

4.3.2 Correctness Proof

The rationale behind modelling BFA with I/O automata was due to the underlying SMR-based BFT system which is typically modeled with automata, therefore, it

is natural to frame our theoretical discussion in terms of automata composition. The automata composition property of I/O automaton allows an output action π of one automaton performs π, all automata having π as input action perform π simultaneously.

Formally, we consider the composition of BFA system A with the underlying BFT system B as parallel composition P= B‖A. For any transition $<s,\ \pi,\ s'>$ of B which is an *accepting* transition, there is a corresponding transition $<s,\ \pi,\ s'>$ of A. Note that we only consider the accepting state, which we refer it *BFTCommit* as illustrated in Figure 4.1.

Hence, for each accepting state in system B, there is a transition state in system A as an input *action* which results it to transition to an *accept* state, thereby, a replica node is refreshed. Therefore, the transition path becomes the *invariant* that must be preserved in every replica refreshes. it is intuitive to see if these *invariants* hold in one replica refresh round, then, we assert that the next replica refresh round will be identical to the previous round. Since our system A is driven by the underlying system B, it will not be the first one that violate the protocol. Thus, preserve the system *B's* correctness condition, *safety and liveliness*.

As discussed the *Time-Interval Runtime Execution (TIRE)* in the previous chapter as the driving engine for the high-level state transitions *i.e. desired vs. undesired*, composing $P \parallel TIRE$ as illustrated in Figure 4.2, then, clearly, BFA will not the first one to violate the automata transitions. Figure 4.2 illustrates the automata composition of BFT in A (left automata) to BFA in B (middle automata), described in the previous section, and *Mayflies'* TIRE in C (stacked time-intervals on right) discussed in Chapter 3.

In general, by ensuring n correct live replicas are in sync (weak synchrony) given that at least one of the replicas is being refreshed each time in a timely manner, ensures the preservation of the *liveness* properties. It has been noted in [71] the impossibility of achieving *safety* with synchrony, however, the SMR-based I/O au-

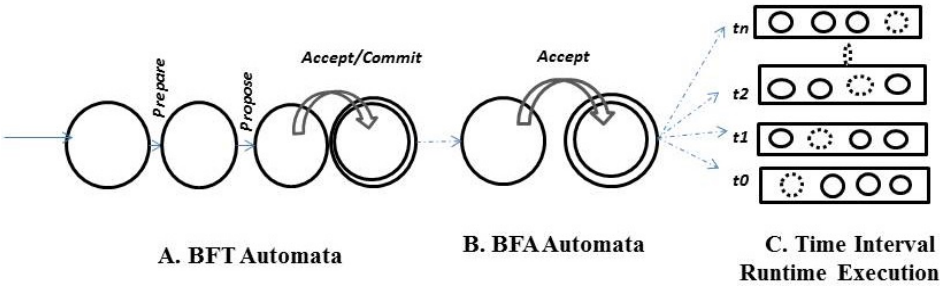


Fig. 4.2.: FSA automata composition of BFT to BFA with *Mayflies* TIRE.

tomaton model guarantees that the replica responses will be correct according to Linearizability [72].

4.4 BFA System Design

In this section we discuss the design approach and the building blocks of the proposed BFA solution.

4.4.1 System Design

Our design is motivated by the modularized, pluggable and structured cloud computing fabric, i.e, stacked hardware, host OS, guest VM/OS's, and reconfigurable networks (SDN) as depicted in the logical system view of 4 node replica deployment use case in Figure 4.3. In Figure 4.3, from the bottom up, at the core of each hardware (Hardware1...n), there is a Host OS with hypervisors (i.e., KVM/QEMU or XEN) and a cloud software stack *(i.e., OpenStack)* as depicted on the bottom three leyers of the stack. There are *n VMs* on each Host OS that is controlled with the *nova compute* (label). Note that the *VMI* introspection is enabled at this layer (between the hypervisor and the VMs).

To illustrate, we deployed four BFT replicas *(BFT-0, BFT-1, BFT-2 and BFT-3)* on the *VMs*. These *VMs* are interconnected with LAN address (192.x.x.x), referred as *fix* IP, and externally exposed with WAN address (128.x.x.x.), referred as *floating*

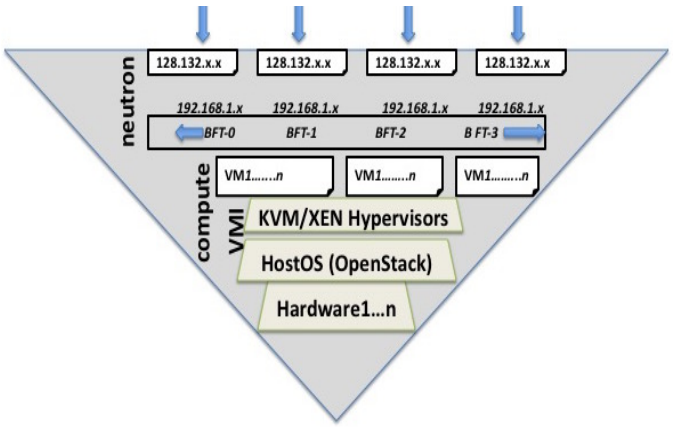


Fig. 4.3.: BFT system logic view on a cloud platform

IP. This mapping is handled by the *neutron* component, an implementation of Software Defined Networking (SDN). The arrows at both ends of the BFT replica stack depicts the *elastic computing model* to dynamically add/remove computing resources *(VM1... VMn)* below it. The cloud software management stack (i.e., *openstack*) implements all these capabilities through a wide range of open source projects such; *nova, neutron, horizon, glance, etc.* We leveraged these capabilities to not only build on-demand scalable platforms but also for a defensive security strategy at system runtime.

We adopted a cross layer vertical design that simultaneously operate on two logical layers of the cloud platform to enable the failure and attack resiliency of BFT systems; a *nova compute* at the application layer (guest VM/Os) and *nova neutron* at the networking layer. The *nova compute* is for the VM provisioning/de-provisioning and the *neutron* enables the dynamic network reconfiguration capabilities, thereby, used for reincarnate/refresh the *VMs* by continuously provisioning/de-provisioning at runtime. Thus, creating mechanically-generated diversity which is almost as powerful a defense as type-checking [52].

4.4.2 Replica Reincarnation

Refreshing a system/entity state is simply resetting its initial state. Refreshing techniques has been widely adopted with a proven success in the access control domain, specially, on passwords. The refreshing scheme of these systems is typically implemented by setting a predefined *lifespan* x for the password to exist/used, and enforce system wide policy for the user to create a new password when x expires and the system deletes it.

Along the same lines, replica/VM Refresh, referred as *node reincarnation*, is simply terminating the VM instance after x amount of time or after completing x number of transactions, then starting another one possibly with different characteristics (i.e., on a hardware, hypervisor, host and/or guest OS) to replace it. This VM substitution can be viewed in real-time with the network topology view of the *horizon* dashboard, a browser-based visualization tool for managing the cloud instances as shown in Figures 4.4(a and b).

There are two different ways to refresh a BFT replica in virtualized cloud platforms. By terminating the replica and selecting its replacement from either:

1. creating a new *VM* instance on-demand, or

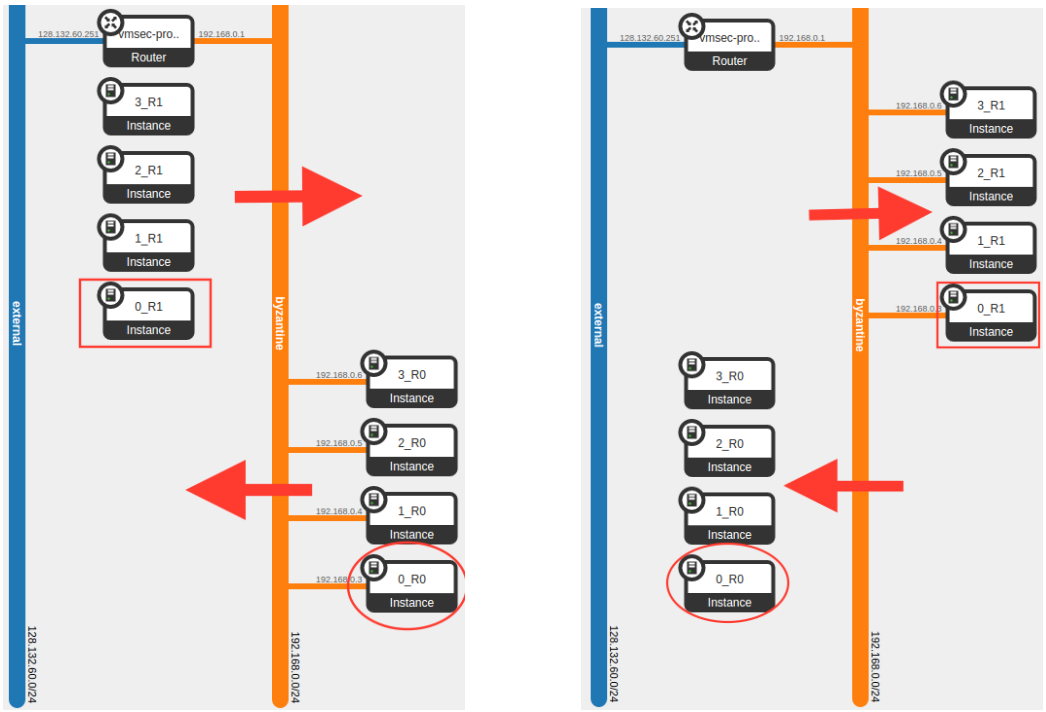
2. from a pre-prepared pool of standby *VMs*.

In this work, we give spacial emphasis on the second replacement strategy (discussed next). We then discuss the implementation approaches and the pros/cons of each replacement strategy for this replication model next.

Prepared VM Pool

On way to prepare a pool of standby *VM* replicas is through the `nova boot <options>` command. The `<options>` include; the OS type (i.e., Linux, freeBSD, windows, etc.), 32/64 bit OS, cluster geographic location, server apps/scripts to active upon booting the instance, network configurations, etc. Another way is through the

horizon dashboard, the virtualization tool for OpenStack,a virtualization tool for managing platform instances in Openstack. Note that the VMs from the standby pool *(x_R1)* are not associated with any network in order to limit their exposure prior using them as servers, thereby, not reachable in anyway. This is similar to booting a server machine without a network card installed, and then we manipulate the interfaces at runtime.



(a) *BFT* replicas pre-transformation

(b) *BFT* replicas post-transformation

Fig. 4.4.: Prepackaged VM pool reincarnation topology view

Figure 4.4(a) shows the network topology for our use case of the 4 *BFT* replicas *0_R0 ... 3_R0* on the right vertical bar (*byzantine* subnet) and 4 isolated standby replica pool *0_R1 ... 3_R1* between the vertical bars/subnets. A virtual router *(vmsec-proj)* interconnects the two subnets with *192.x.x.x* IP on the *byzantine* subnet side and *128.x.x.x* on the externally visible subnet (*external*). The arrows show the refresh direction where the *x_R0* replicas (circle box) will be replaced with the *x_R1*

(rectangle box) replicas. The result after the replicas reincarnated, all *x_R0* replicas are removed from the subnet *byzantine* and replaced with the standby replica from the pool *x_R1* (rectangle box to the oval box) while serving clients as shown in Figure 4.4(b).

Figure 4.4(b) shows the result after performing one round of refresh, all the replicas in the 0th round *(x_R0)* are thrown off of the *byzantine* subnet and one at a time replaced with those *(x_R1)* in the standby pool while serving clients. The cycle can continue to the next round *(x_R2)* replicas for round 2, and so on.

The naming convention used in our prototype *x_Rx* stands for the replica ID x and the round R it is operating, for instance, for the replica *0_R0* in the *byzantine* subnet is the replica ID 0 operating in round 0 (0_R0), and its counter part standby replica *0_R1* is for replica ID 0 in round 1, and so on. The basic idea of our refreshing scheme is to remove a replica x operating in round 0 *(x_R0)* from the *byzantine* subnet (oval) and replace it with the one of the same ID x from the pool designed to operate in the next round *R1* (rectangle).

Note that we manipulate the nodes' network interfaces as they are taking over the role of a server *(x_R0)* in the *byzantine* subnet. The generic scheme of such node transformation, dubbed *node reincarnation with network interface swap* is discussed in details in the previous Chapter. In this Chapter, we show a specific implementation of node reincarnation which we call it *replica refresh* and is illustrated in the following algorithm.

Algorithm 6 Replica Refresh Algorithm

1: **Input:** *replica*
2: **Output:** *newReplica* ▷ Substitute replica
3: **procedure** REFERESH(*replica*)
4: $portID \leftarrow intrface - list < replica_{ID} >$
5: $nova\ \ interface - detach < replica_{ID}\ \ portID >$
6: $newReplica \leftarrow VM_{Pool}$ ▷ standby VM
7: $nova\ \ interface - attach < portID\ \ newReplica >$
8: **end procedure**

In Algorithm 6, we first save the *port ID* associated to the terminating replica (the input *replica*). In SDN environment, the VM is attached to a virtual network interface that is referred to as *ports* with a *fixed* IP similar to physical network interfaces. This interface is also associated with *floating* IP for external access as noted earlier. Thus, both of the IP addresses are part of the *port* even after it is separated from the VM, thereby, transferable to another VM. We detach the port off of the replica in line 5, we then get a *new replica* VM instance from the pool in line 6 and attach the *port* to it in line 7. Note that depending on the OS image of the replica, a VM reboot is required after the `nova interface-attach <portID newReplica>`. At this point, the clients re-connect to this replica through its floating IP (*128.x.x.x*) as the old server that dropped off of the network and came back. We show a 4 replica BFT use case scenario with this refresh algorithm in the experiments section.

Pros & Cons

In general, one of the key advantage of refreshing a replica is the mechanism to control its *lifespan* in order to reduce its exposure attack window. The refresh time, the time it takes to swap a replica, is critical to the effectiveness of the proposed defensive security solution. The longer the refresh time, the longer the replica is absent from the quorum, thereby, violating the systems reliability properties (i.e., sufficient number of replica synchronized). The replacement choices (i.e., prepared vs. on-demand) of the replica dictates how fast a replica can be refreshed.

Creating a new VM instance on-demand takes roughly a minute and selecting one from a prepared pool of VMs takes less than 10 seconds. As a result, the *on-demand boot* replacement strategy is not suitable for BFT replication model, specially, for a 4 replica with 1 faulty settings. The main reason is that, in SMR-based BFT replication model, the absence of a node from the quorum contributes the faulty replicas *(f)* to fall below the threshold when an additional node fails (naturally or compromised), thereby, violate the preservation of the reliability properties *(safety and liveness)*.

However, such a replacement strategy is highly effective for non-SMR based replicated systems, for example, the high-availability slave/master replication models found in event-based middleware (i.e., HA RabitMQ's) where the *slave* node replaces the *master* node when it fails discussed in the next Chapter. For *quorum-based BFT* systems, the refresh transition time should appear to all of the servers and clients as the replica dropped off of the network and came back in order to preserve the system's reliability. To achieve this, having the replicas in standby mode and dynamically manipulating the network interfaces is the most efficient method for refreshing a replica.

There are two different ways to prepare the standby VM pool, the *isolated/detached* pool or *attached* pool. As the names imply, the pool is created in isolation or *detached* off of the network as our replacement strategy discussed in the previous section. The *attached* scheme is when the pool is prepared on the network similar to the post-transformation depicted in Figure 4.4(b). Having the pool within the subnet tend to be a little faster than our *isolated* replacement scheme if both the clients and replicas/servers communicate with the *floating* IP, however, this require a different network topology than the one depicted in Figure 4.4(a).

The standby VM pool can be prepared on the externally visible subnet than the internal (byzantine) subnet of the (192.x.x.x) when using *floating* IP for both the clients and servers, or perhaps, setting it in flat network topology than SDN. The main reason is that the servers in the (byzantine) subnet have no knowledge of the floating IP, therefore, cannot bound to a specific port with that IP. Replicas in the BFT-SMaRT bound to a port number *xyz* on the *fix* IP where they communicate among them, and the clients use the *floating* IP and the same port number *xyz*. The SDN seamlessly handles this mappings.

Regardless, the core issue in this scheme is that the entire standby pool are vulnerable as any other exposed BFT replicas due to the fact that they are reachable within the LAN prior joining to be part of the servers. Consequently, the pool increases the

systems' overall attack vector (i.e., more nodes to protect) and the newly refreshed replica will have higher exposure window as well.

Overall, creating a new instance on-demand is the most secure way that guarantees a zero exposure window as the VM instances are freshly created each time, however, its slow refresh time makes the least favourable scheme for BFT replication model. Using a pre-pared pool of VMs and dynamically manipulating the network interfaces is effective in time critical replication models as it offers faster refresh time. The *attached* pool scheme have some major security issues and it requires a different network configuration to support our use case BFT prototype. Therefore, the *isolated/detached* VM selection scheme offers the best of both worlds, given that the replicas are in standby mode unlike the fully exposed pool of the *attached* replicas or the *on-demand boot* scheme which requires up to a minute in preparation.

4.5 Implementation

The implementation details of the *Mayflies* MTD framework in a generic way is given in the previous Chapter, in this section we give a specific implemetation for BFT systems. We implemented our algorithms with *bash* shell script using OpenStack (Kilo) [7] components, an open source cloud management software stack. To illustrate our proposed BFA, we used BFT-SMaRT [53] prototype downloaded from [73], a widely studied Byzantine Fault Tolerant system in the literature.

We selected OpenStack due to its popularity in commercial clouds. For instance, RackSpace [8], is a public cloud platform built with OpenStack used by many well-established businesses like Netflix. Further, OpenStack provides a modularized components that simplify cloud management. We leverage the *nova, neutron and horizon* components. Similarly, we selected BFT-Smart due to its modern multi-core aware architecture and modularized Java based implementation that is widely studied in the literature in recent years. We have evaluated a number of open source BFT pro-

totypes and none come close to BFT-SMaRT given the fact that BFT research has been around for decades.

We deploy a BFT system to the *Mayflies* framework, we then apply node reincarnation which we call it *replica refresh*. The replica refresh scheme in this system model is to control their existence of the replicas in the hope of reducing their exposure window of attacks and avoid faults, thus, transforming the system into *Byzantine Fault-Avoidance (BFA)*. The fundamental question arise in such transformation approach (from BFT system to BFA) while preserving the systems' reliability properties is *how to deal with the applications' refresh points, dubbed lifespan, and its state transfer between the terminating and the starting replica?* We answer this question with implementation details below, we then present our transformation algorithm.

4.5.1 Replica State Management

The *state* in BFT-SMaRT system consist of two parts; first, is the dynamic part which is created at server start up time with information like; the replica id's, IPs and current leader ID, last executed ID or the committed transaction. This information is typically written in a file called *currentView* to assist the recovering replica upon natural crashes. The second part is the static system configuration files (`system.config` and `hosts`) which contains the security keys/certificates, total number of servers, the faulty model (i.e., *1f*), host IP and port, etc. These files are loaded only once at the server start up and not get updated, however, for spacial settings that supports servers to leave and join in order for the system to grow or shrink, the `hosts` file gets updated with the new server information at runtime.

As noted in section 4.1, the quorum-based state machine replication systems, the *accept* state transition happens upon the replica reaching a consensus. At this point, we can refresh the replica without violating the correctness conditions. The idea is terminating the replica after committing the transaction, given the fact that the system will progress with the majority without the temporarily terminated replica, we

then re-initiate a new one at some point in the future transactions. This guarantees a safe automata transitions, however, the challenge is *how to intercept this check point at runtime (discussed next) and transfer it to the new replica*?.

For transferring the system state, there are two different logical layers of the cloud platforms that can be implemented, either at the *Hypervisor* layer or at the *VM/ Application Protocol* layer. For the *hypervisor* layer approach, one can inject the state information into the servers' memory space. In this process, the VM is paused in which the state information becomes stale. Upon resuming the VM, given that other replicas are continuously processing client requests as long as the faulty-level is below the threshold, the state becomes stale, thereby, the replica has to send requests to others for the current state and update its state upon verifying it with more than one replica.

For the *application* layer, the process is to simply stop the replica, save its state, activate a new server and inject the state, then start the new replica as the old one. Upon resuming, the state information becomes stale, thus, acquire the state updates from others as well, similar to the hypervisor approach. We implement our state transfer scheme using this approach since it is simple and faster than the *hypervisor* approach.

With the three files used for the replica state management, first, we introduce a new configuration property in the `system.config` configuration file, called the `system.server.lifespan`. We execute a `sed` command at the server side for the configuration property substitution as illustrated in the code snippet below.

```
#/bin/bash
...
sed -i 's/\^system.server.lifespan=[^ ]*/
system.server.lifespan=new_value/
config/system.config
...
```

The `sed` command simply looks for the first part of the string, substitutes with the second part of the string with our desired `lifespan` *(new_value)* to the file name provided. This illuminates one round trip of `scp` to the server for injecting the entire configuration file, given that we need to update a single entry. We use the same `ssh` connection to first execute the *sed* command to update the *lifespan* property and then start the server afterwards.

Second, for the `hosts` file, as noted earlier, it only gets updates in spacial dynamic case settings. For simplicity, we illustrate the use case with only 4 BFT replica and keep the system size fixed (i.e., not allowing other replicas to join or leave). However, in dynamic setting like that, the same `system.config` configuration file update method can also be used to update the `hosts` file prior starting the newly replaced replica.

Finally, we save the `currentView` file from the terminating replica, and inject into the new replica using `scp`, then `ssh` to update the configuration files and start the server. Since replicas get the latest state information, especially, the last committed transaction from the others when they reconnect, the state information in the `currentView` file is critical for assisting the leader change protocol. We observed that when we terminate a leader, the reconnected replica further complicates the decision process of the new leader selection if the `currentView` file is not injected. We will discuss our improvements of this issue in the discussion section 4.6.4.

4.5.2 BFT-SMaRT Replica Lifespan

As noted earlier, our key objective is to reduce the exposure attack window of the replica by allowing replicas to run with a pre-defined time frame, dubbed, *lifespan*, on a variable platforms with different characteristics. As illustrated in Figure 4.5, it is intuitive to see the replicas in the initial window are vulnerable to attacks if they stand still for the entire time. We refresh one replica (pointing arrow depicted as

enlarged platform) for every x number of transactions completed, thus, reducing the attack exposure window in the consequent windows.

The key rationale behind this approach is that if a replica is compromised and undetected, the attacker will control within that *lifespan* time frame, thereby, eliminating the chance for the servers to collude. Most importantly, attacks crafted for a replica in one of the windows *(i.e., 0-x)* will not work against the same replicas as it changes its characteristics in the upcoming window x_i time frame.

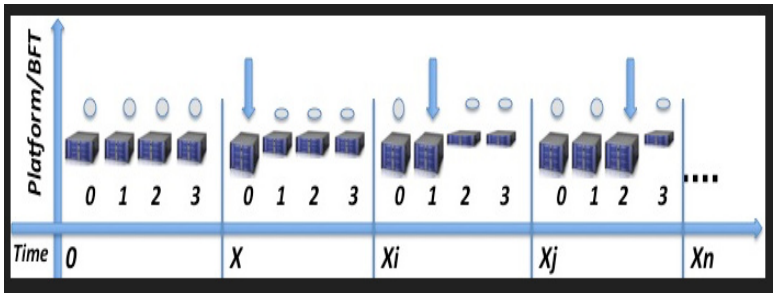


Fig. 4.5.: Illustration of exposure attack window time line of 4 replicas.

In Figure 4.5, the four computer/platforms with IDs $(0 \ldots 3)$ represents the host OSs, and the circle above it represent the BFT servers (guest OSs). Note each platform can host multiple guest *VM* instances. The initial 4 node use case show in the first block and refreshing a replica (pointing arrow) after a predefined *lifespan (x)* shown at the (x-axis) and $x_i, \ldots x_n$ in the consequence blocks with enlarged platforms, and other nodes shrinking, depicting that they are weakening (i.e., longer exposure time). The *lifespan* of the replica can be either a predefined fix system time as low a minute or after completing x number of client requests. This marks the transition from x to x_i in the time line window shown in the (x-axis). Precisely setting the replicas' *lifespan* is critical in order to guarantee safe state transition between the two composed automata (BFT and BFA) discussed in section 4.1. Intercepting exactly when the consensus is committed/accepted at runtime and the refreshed replica continues the process guarantees a safe transition.

To illustrate, we decided to set the *lifespan* of the replica to exist after completing x number of requests (decided consensus) which can be easily translated into a system time (i.e., x number of transactions takes x amount of time). We discuss the rationale behind our decision in the discussion section in section 4.6.3. We inserted our interested x value in the static configuration file `system.config` as discussed in the previous section, and edited the consensus decision method in `TOMLayer.java` class found in the Total Ordering Messages (TOM) module. The java code snippet below shows these changes.

```
package bftsmart.tom.core;
// TOMLayer.java
public final class TOMLayer extends Thread
implements RequestReceiver {
...
/* Called by the current consensus's
* execution, to notify the TOM layer that
* a value was decided
* @param cons The decided consensus */

public void decided(Consensus cons) {
/*Delivers the consensus to
* the delivery thread*/
  this.dt.delivery(cons);

 /*Lifespan detection and self termination*/
if(this.controller.getStaticConf().
   getReEntryPoint() == cons.getId()){
  Logger.println("Reached Life Expectency");
  try{
   PrintWriter writer = new
```

```
    PrintWriter("exitcertificate", "UTF-8");
   writer.close();
   try{
     Runtime runtime = Runtime.getRuntime();
     runtime.exec(new String[] {
     "/bin/bash", "-c","path/terminator.sh"});
   } catch (Exception e) {
         Logger.println(e.getMessage());}
  } catch (IOException ex) {
   Logger.println(ex.getMessage());}
 }//end if
 ...
```

In this class, the `decided(Consensus cons)` method calls the message delivery thread to deliver the consensus `cons` (i.e., committed/accepted the execution of the client response). We inserted our code after that method call as shown below the comment `/*Lifespan detection and self termination/`. We simply check every decision transaction number against the allowed system *lifespan* x loaded from the static configuration file. Once the replica reach its `lifespan`, it creates a file which we call it `exitcertificate`, then self-terminates by calling a bash script `terminator.sh` which kills all the java processes. This assures that the replica existed only up to its designated *lifespan*, thereby, guaranteeing a smooth automata transition between BFT to BFA, thus, preserve the *safety* property of the BFT.

Clearly, as the code snippets show that our changes have no impact on the correctness of the application protocol, however, one concern is for slow consuming clients where the decided messages are kept in the delivery queue or the batch process model where requests are delivered in batches (bulk). This should be simply addressed by disconnecting the communication of the replica from the rest of the servers first and delaying the termination call for the `terminator.sh` script until the delivery queue is cleared.

The monitoring bash shell script code snippet show the monitoring process is as follows: we *ssh* to the VM replica while suppressing the *ssh* trust warnings with the *-o* options, change to the directory to where the file will appear and keep checking every second. The loop breaks once the file is created, server terminated and then we perform the transformation process illustrated in Algorithm 7 discussed in the next section.

```
#/bin/bash
...
ssh -i $key -o UserKnownHostsFile=/dev/null
-o StrictHostKeyChecking=no
'ubuntu@'$ip "cd $path;
 while [ ! -f exitcertificate ];
 do sleep 1; done"
...
```

Note that the `ssh` protocol verifies the trust of the replica for the first time another host tries to connect. In our case, it is always the first time whenever we refresh a replica, therefore, we added the following options in order to skip the confirmation message:

```
[ssh --o UserKnownHostsFile=//dev//null -o StrictHostKeyChecking=no
...]
```

4.5.3 BFT to BFA Transformation

At a high level, the process of the transformation is as follows: we start the servers/replicas with a predefined *lifespan* in their initial static configuration file, then they create an `exitcertificate` file to signal the monitoring application for reaching their *lifespan* and self terminate. While we continuously monitoring the existence of this file, we refresh the replica once the file is detected. Algorithm 7 below illustrates the logical implementation of the proposed architectural transformation.

In Algorithm 7, given the set $\mathbb{S}$ of all the participating replicas, we first initialize the life expectancy x, counter i and the file name in line 2. In lines 6-14, we check if the *exitcertificate* file is issued/created in line 6, we save its state file for the new replica with `GetCurrentViewFile()` in line 7, apply the refreshing process in line 8. Note that this procedure is implemented in Algorithm 1 above. We then update the configuration file of the newly refreshed replica with the desired *lifespan* x and inject the *currentView*/state file and start the server in line 9 and 10. We create a new VM instance *x_Rn* to replace the one just used in line 11. The counter i in line 12 is to update the *lifespan* of the next upcoming replica, for instance, if the initial replica termination *lifespan* is 20K increments, then *x=20* and *i=1*, thus, we terminate the first replica after completing (20K), (40K), and so on.

Algorithm 7 BFT to BFA transformer Algorithm

```
1:  Input: 𝕊                                   ▷ Set of BFT replica servers (S_1 ... S_n)
2:  Initialize x, i, exitcertificate           ▷ lifespan, next, file name.
3:  while condition do
4:      for replica in 𝕊 do
5:          repeat
6:              if exitcertificate file exisits then
7:                  GetCurrentViewFile()
8:                  Refresh(replica)           ▷ Algorithm 6 above
9:                  InjectState()              ▷ scp
10:                 StartServer(x)             ▷ ssh
11:                 VM_Pool ← nova boot < opts >   ▷ refill
12:                 x ← x * i                  ▷ next lifespan
13:             else
14:                 do nothing                 ▷ keep waiting
15:             end if
16:         until                              ▷ until exit certificate issued
17:     end for
18:     condition = false                      ▷ one round refresh only
19: end while
```

The `GetCurrentViewFile()`, `InjectState()` and `StartServer()` in lines 7, 9, and 10 are implemented used secure copy `scp` and secure shell `ssh` commands as described in the previous section (sec. 6.1). To illustrate the concept, in our experi-

ment, we set the *condition* to *false* in line 18 to terminate the main `while` loop after performing only one refresh round to all the replicas.

4.6 Evaluations

As depicted in the BFT system logic view diagram in Figure 4.2 in section 4.3.2, each hardware/machine *(HW1... HWn)* lie a Host OS with a hypervisor *(HV... HVn) or containers*, and guest OS's/servers *(VM1... VMn)* on top of it. For each replica refreshed, we will start a new one with a different guest (VM) OS and hardware platform for its place. Thus, our experiments is targeted on evaluating the runtime execution gap while the replicas are on a constant move across these hardware (*HW1... HWn*), and at the same time processing ordered messages.

We deploy BFT-SMaRT' s *CounterServer()* and *CounterClients()* demo application on OpenStack cloud platform and report the transformation results. In this demo, the clients send requests that has a number to all the replicas/servers, then the servers respond the number incremented by a predefined x value in ordered fashion. This demo illustrates the SMR-based replication model that's mathematically proven to guarantee reliability even in the presence of some faulty ones. We are interested in the runtime execution gap (i.e., missed messages) between the terminating server and the new server in order to asses the transformation impact on the replicas reliability properties and throughput.

4.6.1 Guiding Principle for Quorum-based Replicated Systems

In this system model, there are typically n nodes, where $n \geq 4$, and the system tolerates upto f faulty nodes, where *f=1/3 of n*. For example, when n=4, then if one node is *compromised/fails*, the system should be still considered in a *desired* state. To represent this behavior in our model, we need an additional hidden state with the *compromised and failed* states in level II, call it, *Turbulence*, for example. Note that

the HHMM model is extensible horizontally and vertically, and the state transition probabilities are equally likely as discussed in section 3.3.3.

Then, the guiding principle for this system model can be governed by the following rules:

- if less than or equal to *1/3rd* of the nodes are *dirty*, then the system transitions to *Turbulence* state.
- if more than *1/3rd* of the nodes are *dirty*, then the system transitions to *Compromised* state.
- if more than *2/3rd* nodes detected *dirty*, then the system transitions to *Failed* state (i.e., application crashes).

4.6.2 Experimental Setup

Our experimental platform uses a private cloud built on OpenStack software on a cluster of 10 machines of Dell Z400 with Intel Xeon 3.2 GHz Quad-Core and 8GB of memory running Fedora 23 host OSs. We used a Gigabit Ethernet switch between the machines. We set up one of the machines as a controller and networking (SDN) node, and 9 were used as compute nodes. The 9 compute nodes allow us provisioning 36 virtual CPU's (vCPU) which equals upto 18 small vm instances/servers, 2 vCPU per instance. The client is installed in a separate node from the cluster to mimic the realistic setting of clients.

We used Ubuntu 14.04 for the clients and the replicas/servers in all our experiments to illustrate the concept. However, the idea applies to any cloud images/OS's formats (COW, EC2, etc.) available in the public repositories that OpenStack supports.

4.6.3 Experimental Results

We run BFT-SMaRT *CounterServer()* and *CounterClient()* for 4 servers with 1*(f)*, where f is the number of faulty replica the system tolerate, and a single client sending 100K request messages. We set a 100K messages to be published in order to cover a full round of refresh on all replicas. We set a refresh point of 20K increments for each replica *lifespan* starting at replica 1 on 20K, replica ID 0 at 40K, replica ID 2 at 60K and on 80K at replica ID 3. In BFT-SMaRT, the leader ID is typically number 0 and the candidate leader is number 1 and (+1) for the next candidate leader and so on. Refreshing the leader first will lead to refreshing a leader in each round, therefore, we start with server 1, then 0, and so on.

Table 4.1.: Counter Demo with Normal Re-start

Server	Re-Start	Transition Gap
0	0, 0	40000, 40037
1	1, 1	20000, 23650
2	2, 2	60000, 62286
3	3, 3	80000, 82105

Table I show the results of the normal server restarts using the scripts `smartrun.sh` and `killall.sh` included in the demo. The first column show the server ID, refreshing it to itself (i.e., server IDs 0 and 0 again) is shown in column 2. The third column shows the transition gap (i.e., the messages completed by the terminating server and the start of the new server where its predecessor left off) after the replica fully recovers and updates its state. In this experiment, we simply monitor the *exitcertifacate* file from the terminating replica, then we *ssh* back and restart the server again. This is to capture the recovery protocol timing in terms of transition gaps and the overall process time.

Table 2 show the results of the demo servers while on the move across platforms with the same settings above (i.e., 100K client requests and 20K increments). Similar to Table 1 layout, column 1 show the original replica ID's and the transformed ID

Table 4.2.: Counter Demo with BFA Transformation

Server	Refresh	Transition Gap
0	0_R0, 0_R1	40000, 40037
1	1_R0, 1_R1	20000, 25852
2	2_R0, 2_R1	60000, 66176
3	3_R0, 3_R1	80000, 86773

shown in the second column where the server id pairs *(x_R0)*, *(x_R1)* is for server id x in round *R0* and then to x in round *R1*, and so on. The third column show the message transitions gap between *(x_R0)* and *(x_R1)* replica group.

Note that the transition gap of the leader replica (Server 0) in both experiments are identical, this is due to the fact that the leader recovery time is about 20 seconds [53] and the elapsed time between the termination of the replica and the start of a new replica is typically less than 10 seconds as shown in Table III (column 2). The transition gap starts after the replica updates its state (installs the last execution ID) and resumes processing client requests which is ∼700 messages passed from the time it reconnects, and 0 messages for the leader replica.

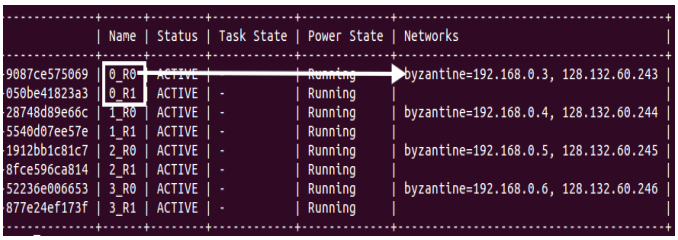
Table 4.3.: Comparisons of Generic, Re-starts and Refreshes

Use Case	Avg. Lapse Time (sec)	100K Time (sec)
Generic	0	185.655
Re-Start	0.120 ± 0.010	221.527
Refresh	11 ± 3	322.902

Table 3 shows the three use cases side–by–side: the *Generic* case were we start the servers without stopping them, *Re-Start* and *Refresh* experiments reported in Tables I and II with the 100K client requests. Column 1 is the use case scenario names. Column 2 show the average lapse time, the time it takes for the server to put back in business when *Re-Started* or *Refreshed*. The lapse time starts when we detect the *exitcertificate* file and *ssh* back to *re-start* the server or manipulate the interface to start a new and different replica for the case of the *Refresh* experiment. The lapse

time is 0 for the *Generic* case since the server is not stopped. Column 3 show the total process time of the 100K requests averaged across 5 experiments.

The total process time shows that it takes a little over 3 minutes to process a 100K client requests in the *Generic* case and little over 5 minutes when refreshing a replica in every 20K requests for the *Refresh* case. This illustrates that we can refresh a replica in as low as a minute while performing useful computation which is necessary when operating in contested environments, however, in a normal situation, the performance impact is negligible if randomly refreshing a replica (say for every 5-6 minutes or more which is $\sim$ 200K+ requests) to disrupt attacks.

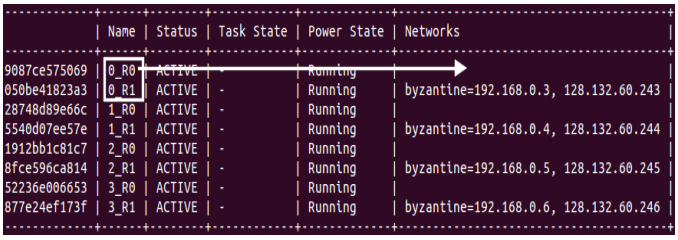


	Name	Status	Task State	Power State	Networks
9087ce575069	0_R0	ACTIVE	-	Running	byzantine=192.168.0.3, 128.132.60.243
050be41823a3	0_R1	ACTIVE	-	Running	
28748d89e66c	1_R0	ACTIVE	-	Running	byzantine=192.168.0.4, 128.132.60.244
5540d07ee57e	1_R1	ACTIVE	-	Running	
1912bb1c81c7	2_R0	ACTIVE	-	Running	byzantine=192.168.0.5, 128.132.60.245
8fce596ca814	2_R1	ACTIVE	-	Running	
52236e006653	3_R0	ACTIVE	-	Running	byzantine=192.168.0.6, 128.132.60.246
877e24ef173f	3_R1	ACTIVE	-	Running	

Fig. 4.6.: An output of `nova list` command showing the BFT and the standby replicas with their network mappings before the transformation. The arrow shows the (x_R0) replica groups have network interfaces and x_R1 have none.

Figure 4.6 reflect the graphical *horizon* dashboard network topology shown in Figure 4.4(a) where our 4 BFT replicas on the *byzantine* subnet *(x_R0)* with IP *(fix and floating)* addresses and *(x_R1)* with blank entries. To illustrate, the white square box shows one of each of these replicas *(0_R0)* and *(0_R1)* where the network interface is attached to *(0_R0)* as the arrow points and none to *(0_R1)*.

Figure 4.7 shows the result after the transformation algorithm completes for one round. The white square box shows the same two replicas as depicted in Figure 4.6, *(0_R0)* and *(0_R1)* and this time the interface is attached to *(0_R1)* and *(0_R0)* has no network entry as the arrow points. This transformation can continue as (*x_R2*) for round 2 with newer VMs created/refilled instances in line 12 of the Algorithm 7, and so on.



	Name	Status	Task State	Power State	Networks
9087ce575069	0_R0	ACTIVE	-	Running	
050be41823a3	0_R1	ACTIVE	-	Running	byzantine=192.168.0.3, 128.132.60.243
28748d89e66c	1_R0	ACTIVE	-	Running	
5540d07ee57e	1_R1	ACTIVE	-	Running	byzantine=192.168.0.4, 128.132.60.244
1912bb1c81c7	2_R0	ACTIVE	-	Running	
8fce596ca814	2_R1	ACTIVE	-	Running	byzantine=192.168.0.5, 128.132.60.245
52236e006653	3_R0	ACTIVE	-	Running	
877e24ef173f	3_R1	ACTIVE	-	Running	byzantine=192.168.0.6, 128.132.60.246

Fig. 4.7.: Post BFA transformation results. The network interfaces of (x_R0) group are seamlessly transferred to the (x_R1) group. Note (x_R0) entries are blank.

Each our 4 BFT use case replicas in the initial round *(x_R0)* clearly reached the 20K refresh increments as we see those in the standby *(x_R1)* round continued processing from where their predecessors left off as shown in Tables I and II, and the SDN results of Figure 4.6 and 4.7. Thus, this guarantees the safe state transition between the replica *(x_R0 and x_R1)* groups, as a result, preserved the reliability properties of the protocol *(safety and liveness)*. We showed the total process time of a 100K messages from a single client, we consider evaluating our algorithm with large number of clients while the servers are geographically distributed and under attack.

4.6.4 Discussion

In this section, we will discuss the key challenges on extending cloud framework to support MTD-based attack resiliency defensive strategy. The key challenges we address are the behaviour of the BFT-SMaRT leader change recovery protocol and dealing with the replica *lifespan* of the nodes to accommodate the dynamics of the framework.

Improving BFT-SMaRT Recovery Protocol

The absence of the node due to the re-start is less than a second, and the refresh process is between 8-11 seconds. During this time, the replica misses over thousands of messages as shown in Table I and Table II. In both cases, we observed that occa-

sionally the system crashes when the recovering replica is catching up with the missed messages (timeouts). This was due to the fact that terminating another replica that reached its *lifespan* leads the system to fall below the allowed faulty number.

As the logs show, we believe that the replica is re-connected but has not yet been fully recognized by the majority in the quorum process, thereby, the system crashes upon terminating a second one. This issue also appears when terminating a replica at the start of the experiments which was due to the replicas proceeding to process client requests once 3/4 of the replicas are connected, thus, putting the 4th replica (i.e., 4 replica use case) in a catch-up mode early in the game. We set the replica *lifespan* to 20K increments to space out the termination and considering to thoroughly analyse the code and systematically solve this issue in the future.

For the leader change recovery protocol, we discovered (from the logs and the code) that upon the return/reconnected leader at specific point of the protocol message exchanges, the returned leader further contributes more messages and delays the leader selection process even further, for example, the reconnecting replica receives a message from a certain replica which shows that his id is the known leader for this replica. In contrast to the messages from other replicas that show a new leader is considered (regency number incremented) but not decided and other replica sending for unknown leader *(regency -1)*. This causes many more messages to be exchanged in order to settle the differences among all the servers until a new leader is selected which eventually happens.

Occasionally, the system enters into an infinite loop. The logs show that it is due to the de-conflicting regency numbers among the replicas where each replica is incrementing the next regency (leader) id beyond the number of participants, thereby, a dead-lock in deciding a new leader. On one hand, these additional messages have helped our refreshed replica to catch up to the point where its predecessor left off. On the other hand, it affects the system stability and the overall performance.

We know that the recovery time for a non-leader replica in BFT is negligible and about 20 seconds for the leader due to the leader change protocol messages

exchanged in order to unanimously decide for a new leader [53]. Given the fact that our failure inducing defensive techniques, repeatedly refreshing (terminating/re-starting) replicas, takes only between 8-11 seconds, we considered improving the leader recovery protocol in a systematic fashion than the existing approaches in order to greatly benefit the refreshing defensive solution.

Some of the previous attempts to remedy the vulnerability of the leader (the key motivating factor to our work) and the leader change protocol include: extending the leader change time-outs in the configuration file to delay the leader change protocol activation [28]. The bounded acceptable delay time has been argued and proposed improvements in [66]. Experiences with Fault-Injection in BFT Protocol in BFT-SMaRT [74] revealed high time outs (i.e., client side) leads to high recovery times. Furthermore, they showed injecting attacks to the next candidate leader (regency), leads the system to revolve around malicious leaders.

To reduce the inherent issues in leader's recovery time in a systematic approach, we introduced a reclaim leadership method in the protocol in which the recovering leader sends a new message called *ignoreLeaderChange (LC_IGNORE)* as long as a new leader is not yet decided. Once this messages is received, all the replicas simply check if the sender was actually the leader, current leader has not yet been decided, then cancel all the messages in the pipe (i.e., STOP, STOP_DATA, SYNC, etc.). This greatly improved the recovery process by eliminating the exchange of the additional messages and the stability of the system in general. Another possible solution is to perform a parallel post recovery techniques for the *catchup* process.

Replica Lifespan

The *lifespan* for the server can be either a predefined fixed time (as low a minute) or after completing x number of client requests. The goal is to refresh replicas without modifying the BFT code. To set the *lifespan* using the system clock time requires that all the VM servers system clocks to be accurate at all times. This can be simply

achieved through the use of *NTP* or other methods, however, the *lifespan* of the replica may never be reached if a server is compromised and its system clock time is altered.

Setting the *lifespan* of the replica upon completing x number of transactions which is technically be translated to *time* is secure and it does not require system clock maintenance, however, the assumption is that the network flows are always synchronized and the replicas process the client requests in the order it was received and respond in that order. This assumption does not hold in virtualized environments, thereby, is challenging to detect the exact transaction completion point.

Our first attempt to set the replica *lifespan* was to monitor the log file entries as it is written for our interested value x, however, we discovered that there are 60 $\pm$ messages processed by the time we extract the system state in order to terminate the replica. This is possibly due to the IO disk read requests from our monitoring script and the write requests of the server. Therefore, we decided to insert the *lifespan* value x in the configuration file `system.config` and slightly edit the code to intercept when the value x is reached and issue an exit/create certificate file, thus, assuring the replicas to exist only with their intended *lifespan*. As a result, we monitor the existence of the *exitcertificate* file instead of the log file entries and upon detecting it, the replica gets refreshed safely as described in section 4.5.2.

4.7 Conclusion

In this Chapter we illustrated *Mayflies* MTD framework with a *fault-tolerent* replicated system prototype. The criticality of incorporating *attack-tolerance* to *fault-tolerance* protocols as an integral part of distributed system's architecture and protocols was first addresed in [52] for over a decade. To the best of our knowledge, this work is the first to attempt an architectural-level integration of *attack-tolerance* and *fault-tolerance* on virtualized cloud platforms. Furthermore, the proposed *Mayflies*

MTD framework not only enable such integration on *fault-tolerance* protocols, but also to other replicated systems.

Experiences over the period of this work revealed major improvements between the OpenStack software releases (semi annual). For instance, in *Icehouse* release, detaching an interface from the terminating VM with `nova interface-detach` to free the resources (i.e., IPs) in order to re-use it in `nova interface-attach` call required a new interface to be created first with `neutron port-create` because the interface is deleted. Then, the new interface has to be associated with the *floating* IP. This resulted a slower refresh time than swapping the interface with just the two steps *(detach and attach)* supported by the current version *(kilo)* used in our experiments.

Therefore, as new versions of cloud software stack and SDN implementations emerge, and the BFT protocols re-engineered to adopt to such platforms, we believe the cloud computing fabric can advance MTD-based security solutions to defend systems against sophisticated attacks than the traditional defensive approaches.

5. DISRUPTION-RESILIENT PUBLISH AND SUBSCRIBE

The Publish and Subscribe (pub/sub) dissemination paradigm has emerged as a popular means of disseminating filtered messages across large numbers of subscribers and publishers. This dissemination paradigm has attracted many applications: financial trading systems, cloud infrastructures to interconnect components, and distributed clustering (i.e., RabitMQ), as service buses used in Service Oriented Architectures, Yahoo Message Broker, OracleJMS, IBM-Websphere, JBoss, and many others.

In pub/sub, typically, a broker(s) mediates the exchange of topic or content-based messages between the producers (publishers) and consumers (subscribers). Subscribers register their topic of interest to the broker in which is then filtered against the incoming messages from the publishers and forwarded to them upon match, thereby, eliminating the need for a prior connection between the message publishers and subscribers.

5.1 Motivation

The many–to–many loose coupling data sharing model between the subscribers and the publishers mediated by a broker have a major security issue. Once the broker is compromised, messages can be dropped or not delivered at all, delayed or delivered unfiltered. Most importantly, replicated brokers can collude to disrupt the entire operation. These malicious behaviors are known as Byzantine faults; a faulty model where the system deviates from the protocol specification and enters into undesired states.

Mayer et. al. [75] evaluated the robustness of pub/sub systems in eight architectural dimensions and argued the criticality of the rational behaviour. For decades,

replication has been the corner stone for achieving reliability and robustness of the brokers. For example, crash failure resiliency issues and reliability in pub/sub systems using replication have been studied in [76].

With the growing trend of cloud computing adaptations, replicating brokers on a highly dynamic virtualized cloud environment is undeniably cost effective. However, it's increasingly challenging to guarantee the reliability and robustness of the brokers on these platforms due to the increase in the attack surface [39] – the set of ways/entries an adversary can exploit/penetrate the systems.

Chang et al, in their position paper [77], noted the lack of studies of BFT-based pub/sub systems in the literature and pointed out that building such as system is difficult, perhaps even impossible. The key challenge is that the BFT system's run time execution model is ordered (client request are processed in persistent FIFO model) in contrast to the loosely coupled nature of pub/sub system.

A crash tolerant Paxos-based system, referred as P2S, was recently proposed by the same authors [78]. Others have approached this problem using overlay networks. To name a few, a consensus replication model is proposed in [79] which was noted that the protocol somewhat deviates from the traditional message forwarding standards. Another overlay networks based on neighbourhoods is proposed in [80].

With the ever-increasing sophisticated targeted attacks that employ zero-day exploits, protocol-level solutions tend to be defeated when attacks originate outside the application, i.e., OS kernel. We believe shifting from a perceived over-emphasis on improving existing protocol-centric solutions, to better enable fault–resiliency while reflecting the underlying computing fabric, is critical. Thus, we deploy widely adopted pub/sub system prototype on *Mayflies* framework to illustrate the MTD scheme of randomizing and diversifying brokers over variable platforms to combat against the aforementioned threats.

5.2 Background

Eugster et. al. [81] gave a fair treatment of the pub/sub models and their interaction patterns, highlighting the decoupled nature of publishers and subscribers in time, space, and synchronization. Many variants of the paradigm are especially adapted a variety of applications and network models. In this work, we are interested in highly available replicated brokers/pub/sub systems distributed across heterogeneous clusters.

Pub/sub ecosystem consists of three agent interactions: subscribers, publishers, and broker. Typically, a broker mediates the exchange of topic or content-based messages between the producers (publishers) and consumers (subscribers). In general, pub/sub systems handle aperiodic and periodic messages of heterogeneous sizes and formats in near-real time.

One of the proven methods of implementing a fault-tolerant (i.e., Byzantine Faults) replicated services is through State Machine (SM) approach [60]. However, the main challenge of implementing Byzantine Fault Tolerant (BFT) protocol in pub/sub systems is that BFT protocols are based on SM which complies a well ordered finite state automata transitions, for example, client requests enter the system which is then executed in ordered fashion among the replica and then a consensus is reached for the response. Unlike pub/sub, the client requests/events are processed in loosely coupled fashion, for example, events are published out of order and brokered by the broker independent of the subscribers that are interested in these events.

5.2.1 Scope and Threat Model

Many variants of the pub/sub paradigm have been proposed and each is being specially adapted to specific application and network models [81]. We consider replicated n brokers where $n>1$ nodes/replica typically used for high availability cluster settings.

We consider the standard assumptions of Byzantine fault models [82] . Typically, replicated brokers with Byzantine fault models demonstrate arbitrary faults that deviate from the correctness protocol. We consider a Byzantine broker misbehaviour described in [77]. A compromised broker can: 1) impact the performance by delaying publications, 2) effect system integrity by tempering published contents, violating message reordering and corrupting forwarding tables, and 3) even cause system outage. We consider publishers and subscribers (clients) are trusted.

5.3 System Design and Implementation

Our design is motivated by the modularized, pluggable and structured cloud computing fabric, i.e, stacked hardware, host OS, guest VM/OS's, and reconfigurable networks. In this section we will describe our system design approach and discuss our algorithms.

5.3.1 System Design

The logical system view depicted in Figure 5.1 illustrates the building blocks of a cloud platform and the 3 Brokers for our use case. These blocks can be viewed as three logical layers; i) the bottom three blocks (Hardware1..n, HostOS, and Hypervisors) which we call it the foundation layer, ii) *guest/VM* layer which consists of the VM1..n blocks and the impeded applications (Broker1, Broker2, and Broker3), and iii) the networking layer.

In Figure 5.1, the arrows on top represent the entry point of the application, and those on the side of the application layer (*Broker 1..n*) represent the elastic computing model of the brokers provisioned on any of the underlying VM.We adopted a cross layer vertical design that simultaneously operate on two logical layers of the cloud platform; a *guest/VM* and the network layer. The *guest/VM* layer aims for broker VM instance refreshes while the networking component aims to dynamically reconfigure the network at runtime. It is intuitive to see that such scheme has the

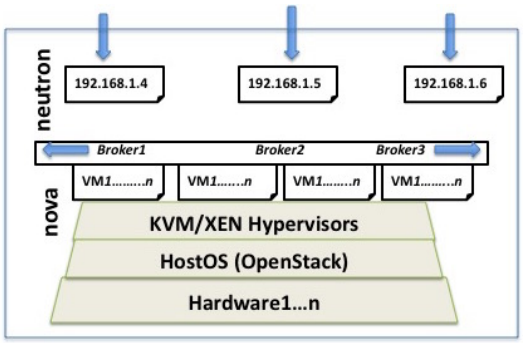


Fig. 5.1.: Logical pub/sub system view

benefit of terminating a compromised broker within a given time frame, therefore, a successful and/or in progress attack will have a limited impact on the system.

Since attacks originate at the entry point, externally visible *192.168.1.x* IP for clients, by refreshing the underlying VM (*broker1...n*), we circumvent any attack crafted or vulnerability exploited to a given system (i.e., hardware, Host and guest OSs and the Hypervisor). Note that the brokers typically communicate with local IP address similar to those found in LAN settings.

5.3.2 Implementation

We implemented our algorithms with *bash* shell script using OpenStack [7] *nova* api, an open source cloud management software stack. As described in details in chapter 3, OpenStack provides modularized components that simplify cloud management. In this work, we leveraged *nova compute* for provisioning the VMs/servers, *neutron* for networking, *glance* for the VM image management, and *horizon* dashboard for visualization.

Algorithm 8 illustrates the broker refresh procedure. The procedure works as follows: we first save the broker's externally feasible IP address known as *floating IP* in line 4. We then delete the broker in line 5, and in line 6 we create a new VM instance with *options* like; specific port ID with selected *fix* IP address (LAN), OS type, cluster, geographic location, file to run after boot, etc. We finally associate the

floating IP to the new VM instance *(newBroker)* in line 7. This algorithm can be used in a random refresh fashion or timely based (i.e., 5 minute) intervals as we illustrate in the previous Chapter. In this Chapter, we illustrate on-demand VM instance boot to reincarnate nodes. Unlike the BFT system illustrated in the previous Chapter, this reincarnation process does not require network interface swapping.

Algorithm 8 Replica Refresh Algorithm

1: **Input:** *replica*
2: **Output:** *newReplica* ▷ substitute targetBroker with a newBroker
3: **procedure** BROKERREFERESH(*replica*)
4: $flIP \leftarrow targetBroker_{floatIP}$
5: *nova delete targetBroker*
6: $targetBroker \leftarrow$ *nova boot* $<$ *options* $>$
7: *nova floating-ip-associate*$(newBroker_{ID}, flIP)$
8: **end procedure**

Clearly, this algorithm is also suitable for any replicated and non-replicated system deployed on virtualized cloud platform using some form of *nova* implementation for *provisioning* and *de-provisioning* VM instances, and an SDN implementation.

5.4 Evaluation

Our experiment is targeted on evaluating how fast we can refresh brokers in order to reduce the exposure window time of an attack to succeed and disrupt the system. We deploy an open source pub/sub system on *Mayflies* framework and illustrate how we move the brokers while under attack. We use a simple application to illustrate how we employ VMI to detect attacks while the brokers are on the move.

5.4.1 Experimental Setup

Our experimental cloud platform uses a private cloud built on OpenStack software on a cluster of 10 machines (Dell Z400) with Intel Xeon 3.2 GHz Quad-Core with 8GB of memory connected with 1 gigabit Ethernet switch. The 10 machines were used as

one controller and networking node, and 9 compute nodes. The 9 compute nodes allow us provisioning 36 virtual CPU's (vCPU) which equals upto a pool of 18 small VM instances, 2 vCPU per instance.

We used Ubuntu 14.04 for clients and the replicas/servers in all our experiments. Note that the OS's of both replicas and clients can be any OS image that OpenStack supports. We deployed *RabitMQ* [83] pub/sub brokers in distributed fashion. *RabitMQ* is a widely adopted open source and commercially supported content-based message brokering. RabitMQ is used in may applications such the financial trading systems, SOA Service buses, inter cloud component interconnections, etc.

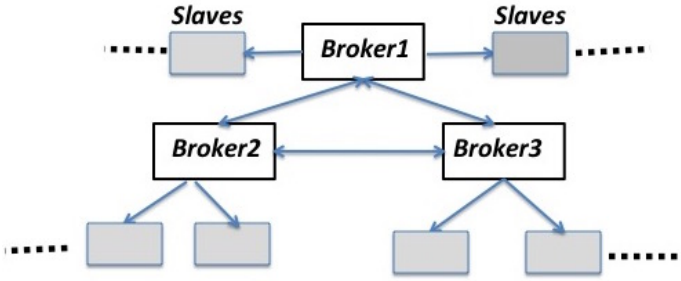


Fig. 5.2.: Three broker replication model

There are numerous ways on setting highly available replicated brokering. For simplicity, we set up 3 brokers with 6 slaves (2 each master) as depicted in Figure 5.2. Within a RabbitMQ cluster, queues (message topics) are singular structures that exists only on one node (the master node) to which is then mirrored (replicated) across multiple nodes to address high availability. Each mirrored queue consists of one master and one or more slaves that can be synchronized, with the idea of the slave replacing the master when it fails (built-in reliability scheme). Thus, the key motivating factor for our solution approach, the existence of the built-in reliability schemes in the protocol enables the brokers and clients to reconnect after a short disconnect.

5.4.2 Experimental Results

Figure 5.3 show the experimental platform. On the left, we show the 6 nodes of experiment (three master and 3 slaves) topology. We reincarnate nodes every 5 minutes while under attack. In this Chapter, we show the tight integration of the VMI-based proactive monitoring with OpenStack platform. As an illustration, we use a simple application shown in Figure 5.3 top right shell window, and the VMI detection scheme shown in the bottom shell window.

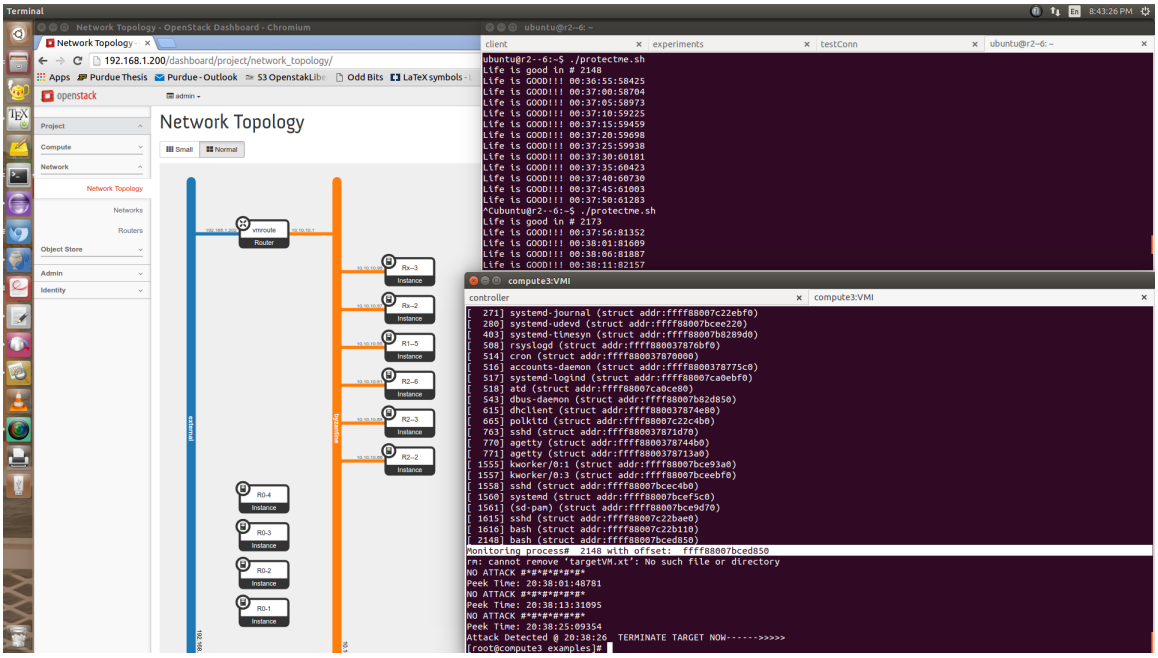


Fig. 5.3.: Experimental platform: broker topology view with OpenStack Horzon Dashboard (left browser window), and attack detection illustration shell windows (top right) for protecting application runtime integrity and detection window (bottom)

We omit evaluating the performance impact on node reincarnation, this is because each replica manages queues for registered subscriptions in which the queue size of these subscriptions is implementation dependent. The reincarnation process is similar to the one described in the previous Chapter. In this experiment, we show a simple application *(protectme.sh)* rather than RabitMQ outputs. The simple *protectme.sh*

is running on the RabitMQ broker node prints its process id. We alter its integrity by re-starting it, presumably, a malicious version.

As shown in Figure 5.3, were able to detect due it's address space offset changes. However, this type of detection can be bypassed with code injection attacks without altering the processes running memory block. In that case, our next use case monitors the internal structure of the applications memory space. The process is as follows: We use the process's ID to extract it's memory region and monitor that region by comparing certain offset value for faster response time. We were able to detect the changes in the application runtime as shown in Figure 5.3 (shell window). The details of the detection implementation is discussed in Chapter 2. This illustrates how the proactive monitoring is performing the detection of attacks of the RabitMQ brokers while on the move across platforms.

Figure 5.4 shows the exposure window time line. The *x-axis* shows five minute blocks. In each block we have 3 physical nodes numbered *1, 2, 3* in which the applications/brokers are deployed on (depicted as circles). In the first block, we show 3 nodes depicting the 3 replicas deployed for the experiment. The 3 replicas/VM can be on any hardware, say, hardware #1, #2 and #3 out of the 9 nodes of our private cloud infrastructure setting. To illustrate the concept, we refreshed a brokers in 5 minute intervals marked *0 to 5min* by the end of each block.

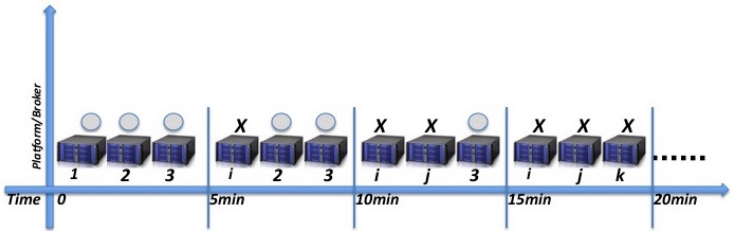


Fig. 5.4.: Illustration of three Broker/VM exposure window of 5 minute intervals.

At the start of the experiment, the brokers $broker_1$, $broker_2$ and $broker_3$ represented in circles are on their respected hardware among our cloud nodes, node#1, node#2 and node#3 respectively. Note that each node can host more than one bro-

ker instance. After 5 minutes, we refreshed $broker_1$ on one of the other 6 nodes, $node_i$ in this case. At the end of the next 5 minutes (10 minute block), we did similarly to $broker_2$ and mapped on the other 6 nodes of the platforms, $node_j$ in this case. Similarly in the consequent 5 minute blocks. The process of refreshing a new booted VM took only 50 to 60 seconds, unlike previous chapter where we refreshed the VM/node from a pool of prepared VMs and swap the network interface took less than 20 seconds. This is to illustrate the different schemes of VM diversifications across platforms for different systems models.

It's intuitive to see that defending the 3 replicas in the first block for it's entire run time is extremely challenging compared to when defending them in one of the 5 minute blocks. The rationale behind this is that in each block there is at least one broker replica is on yet unknown (to the attacker) platform, and another one (or more) soon to be refreshed, thus, reducing the exposure attack window of the overall system.

5.4.3 Conclusion

In this chapter, we deployed a pub/sub system on *Mayflies* framework. We showed how *Mayflies* seamlessly adds visibility on system's runtime to prevent disruptive faulty behavior. The combination of *node reincarnation* and *proactive monitoring with VMI* show the systematic approach of solving a long standing security issue in pub/sub without changes to the applications or protocols.

6. CONCLUSION AND FUTURE WORK

Moving Target Defense (MTD) offers a promising defensive strategy to combat against sophisticated exploits, however, existing MTD solutions are ad-hoc and designed to address for specific threats. We introduced a bio-inspired generic MTD framework for distributed systems, referred to as *Mayflies*. *Mayflies* controls the exposure attack windows of the nodes through proactive monitoring and continuously refreshing the VM, referred to as *node reincarnation*, across diverse platforms (i.e., hardware, OSs) in time intervals (as low as a minute).

We formally modeled and discussed its effectiveness in terms of the proportion of the time the system being in a desired or undesired/compromised state. We introduced a new abstraction layer on the traditional runtime execution, dubbed *Time Interval Runtime Execution (TIRE)*. *TIRE* abstraction model allows a non-deterministic system to be structured to deterministic type to mathematically reason using *Finite State Automata (FSA)*. We showed the effectiveness of the model when coupled with *Virtual Machine Introspection (VMI)* to achieve the desired defensive security objective, a generic MTD framework to combat against sophisticated attacks. Then, we discussed in details on the design and a prototype implementation. Finally, we illustrated the framework with two different classes of systems, a Byzantine Fault-Tolerant and Event-based Publish and Subscribe systems.

6.1 Future Work

In recent years, a lot of attention has been given on the security and the performance issues on the core of the building blocks of the cloud frameworks (i.e., *SDN and VMI*), for example, attacks on hypervisors by malicious VMs and the effects on VMI-based solutions and techniques have been reported in [84]. Similarly, SDN

related security issues and counter measures are reported in [9]. We first consider improving *Mayflies* algorithms to accommodate these countermeasures and extend the framework with IP-hopping, reincarnating nodes across sub-nets or diverse network partitions.

Second, since *Mayflies* framework model enables to conduct *inference* over the system observations O by the VMI component and the transition choices, therefore, we can plan and prioritize the transitions (i.e., aggressively reincarnate nodes when necessary and avoid compromised clusters/platforms). We conisder extending the framework to support inference and learning from the observations. With these inferences, one can formulate the fundamental HMM problem on maximizing the probability [54] of starting in a *desired* state and staying there as often as possible. With this type of inference and learning, we can anticipate attacks before they occur which is critical to defend against modern sophisticated attacks.

Third, the *Virtual Machine Introspection* is an effective method of detecting runtime integrity violations. LibVMI [37] and DRAKVUF [85] are the two open-source introspection libraries currently available with their own pros and cons. For example, LibVMI is originally designed for Xen-based cloud platforms, using it in KVM/QEMU-based platform which is well integrated into Openstack framework require some modifications in which causes stability and performance issues comparing to when used without the modification (i.e., through GDB debugging scheme) [37]. In this work, we leveraged LibVMI in a dynamic runtime environment using the GDB debugging scheme to illustrate the framework's node observation mechanism. Since our defensive solution approach competes with the attack success time, reducing the *observation* time is critical. We consider developing a lightweight introspection library tightly integrated into the framework with improved performance in our future work.

Furthermore, the core of *Mayflies'* MTD defensive strategy is node *reincarnation and observation*. We achieve reincarnation between 8 and 12 seconds. Depending on the granular-level of the observations needed, there are two ways of implementing

node/VM *observations* in the cloud; at the memory address observations using VMI or at the CPU registers. With the CPU layer approach, one can listen an event of an specific registry entry (i.e., CR3) when an application that is not known to the user (i.e., white listing) is scheduled/loaded for execution. We consider exploring CPU observation capabilities in the future work.

Finally, we consider introducing a new computing model, in which we call it *Fragmented Computing*, designed to perform computation over fragmented data on decomposed systems using *Mayflies* to make extremely difficult to compromise a system.

REFERENCES

REFERENCES

[1] T. Hnetynka, L. Murphy, and M. J, "Comparing The Service Component Architecture and Fractal Component Model," *The Computer Journal*, vol. 54.7, pp. 1026–1037, 2011.

[2] P. Oreizy, *Issues in The Runtime Modification of Software Architectures*. Citeseer, 1996.

[3] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An Architecture-based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, no. 3, pp. 54–62, 1999.

[4] D. Garlan, S.-W. Cheng, and B. Schmerl, "Increasing System Dependability Through Architecture-based Self-Repair," *Lecture Notes in Computer Science*, pp. 61–89, 2003.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison Wesley, Reading, USA, 1998.

[6] T. Garfinkel, M. Rosenblum *et al.*, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *The Proceedings of The Network and Distributed System Security (NDSS)*, vol. 3, 2003, pp. 191–206.

[7] "Openstack," http://www.openstack.org, Accessed April 20, 2016.

[8] Rackspace, http://www.rackspace.org, Accessed April 19, 2016.

[9] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," in *Network and Distributed System Security (NDSS)*, 2015.

[10] S. Forrest, A. Somayaji, and D. H. Ackley, "Building Diverse Computer Systems," in *The 6th Workshop on Hot Topics in Operating Systems*. IEEE, 1997, pp. 67–72.

[11] P. K. Manadhata and J. M. Wing, "A Formal Model for a Systems Attack Surface," in *Moving Target Defense*. Springer, 2011, pp. 1–28.

[12] L. Chen and A. Avizienis, "N-version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Digest of Papers FTCS-8: 8th Annual International Conference on Fault Tolerant Computing*, 1978, pp. 3–9.

[13] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant Systems: A Secretless Framework for Security Through Diversity," in *Usenix Security*, vol. 6, 2006, pp. 105–120.

[14] G. Portokalidis and A. D. Keromytis, "Fast and Practical Instruction-set Randomization for Commodity Systems," in *The Proceedings of The 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 41–48.

[15] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced Operating System Security Through Efficient and Fine-Grained Address Space Randomization," in *Presented as part of The 21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 475–490.

[16] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand Live Randomization," in *The Proceedings of The 6th ACM on Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 50–61.

[17] S. Rauti, S. Laurén, S. Hosseinzadeh, J.-M. Mäkelä, S. Hyrynsalmi, and V. Leppänen, "Diversification of System Calls in Linux Binaries," in *Trusted Systems*. Springer, 2014, pp. 15–35.

[18] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow Random Host Mutation: Transparent Moving Target Defense Using Software Defined Networking," in *The Proceedings of The 1st Workshop on Hot Topics in Software Defined Networks*. ACM, 2012, pp. 127–132.

[19] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer Science & Business Media, 2011, vol. 54.

[20] S. Fine, Y. Singer, and N. Tishby, "The Hierarchical Hidden Markov Model: Analysis and Applications," *Machine Learning*, vol. 32, no. 1, pp. 41–62, 1998.

[21] F. T. Sheldon and C. Vishik, "Moving Toward Trustworthy Systems: R&D Essentials," *Computer*, no. 9, pp. 31–40, 2010.

[22] J. Buchi, "On a Decision Method in Restricted Second-order Arithmetic," in *Proceedings of The 1960 Congress on Logic, Methdology and Philosophy of Science*, 1962, p. 111.

[23] K. P. Murphy and M. A. Paskin, "Linear-Time Inference in Hierarchical HMMs," *Advances in Neural Information Processing Systems*, vol. 2, pp. 833–840, 2002.

[24] L. De Alfaro and H. Thomas, "Interface Automata," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, 2001.

[25] J. Pfoh, C. Schneider, and C. Eckert, "A Formal Model for Virtual Machine Introspection," in *The Proceedings of The 1st ACM Workshop on Virtual Machine Security*. ACM, 2009, pp. 1–10.

[26] S. U. R. Malik, K. Samee, and S. Sudarshan, "Modeling and Analysis of State-of-the-art VM-based Cloud Management Platforms," *IEEE Transactions on Cloud Computing*, 2013.

[27] N. Ahmed and B. Bhargava, "Towards Targeted Intrusion Detection Deployments in Cloud Computing," *International Journal of Next-Generation Computing*, vol. 6, no. 2, 2015.

[28] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine Replication Under Attack," in *Proceedings of IEEE International Conference on Dependable Systems and Networks*. IEEE, 2008, pp. 197–206.

[29] N. Ahmed and B. Bhargava, "From Byzantine Fault-Tolerant to Fault-Avoidance: An Architectural Transformation to Attack-Resilliency," *Under Review – IEEE Transactions on Cloud Computing*, 2016.

[30] ——, "Disruption-Resilient Publish and Subscribe," in *The Proceedings of The 6th International Conference on Cloud Computing and Services Science, CLOSER 2016*. CITESEER, 2016.

[31] C. A. Catania and C. G. Garino, "Automatic Network Intrusion Detection: Current Techniques and Open Issues," *Computers & Electrical Engineering*, vol. 38, no. 5, pp. 1062–1072, 2012.

[32] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online Detection of Utility Cloud Anomalies Using Metric Distributions," in *IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2010, pp. 96–103.

[33] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, "A Survey of Intrusion Detection Techniques in Cloud," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 42–57, 2013.

[34] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion Detection System: A Comprehensive Review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[35] M. Goldszmidt, D. Woodard, and P. Bodik, *Real-Time Identification of Performance Problems in Large Distributed Systems*. Taylor and Francis, 2011.

[36] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," in *Proceedings of The 22nd Symposium on Operating Systems Principles (SIGOPS)*. ACM, 2009, pp. 117–132.

[37] "LibVMI: Library For Virtual Introspection," http://libvmi.com, Accessed April 19, 2016.

[38] B. D. Payne, "Simplifying Virtual Machine Introspection using LibVMI," *Sandia report*, 2012.

[39] P. K. Manadhata and J. M. Wing, "An Attack Surface Metric," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, 2011.

[40] B. Sweeney, *Mayflies and Stoneflies: Life Histories and Biology*. Kluwer Academic Publisher, 1987.

[41] B. Sweeney and R. Vannote, "Population Synchrony in Mayflies: A Predator Satiation Hypothesis," *Evolution*, vol. 36, pp. 810–821, 1982.

[42] F. Salfner and M. Malek, "Using Hidden Semi-Markov Models for Effective Online Failure Prediction," in *26th IEEE International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2007, pp. 161–174.

[43] K. D. Bowers, M. Van Dijk, R. Griffin, A. Juels, A. Oprea, R. L. Rivest, and N. Triandopoulos, "Defending Against The Unknown Enemy: Applying FlipIt to System Security," in *Decision and Game Theory for Security*. Springer, 2012, pp. 248–263.

[44] H. Okhravi, E. I. Robinson, S. Yannalfo, P. W. Michaleas, J. Haines, and A. Comella, "TALENT: Dynamic Platform Heterogeneity for Cyber Survivability of Mission Critical Applications," in *Secure and Resilient Cyber Architecture Conference (SRCA'10)*, 2010.

[45] M. M. Carvalho, T. C. Eskridge, L. Bunch, J. M. Bradshaw, A. Dalton, P. Feltovich, J. Lott, and D. Kidwell, "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*. ACM, 2013, p. 38.

[46] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-based Platform for Trusted Computing," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 193–206.

[47] I. Ahmed, A. Zoranic, S. Javaid, and G. G. Richard, "ModChecker: Kernel Module Integrity Checking in The Cloud Environment," in *41st International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 2012, pp. 306–313.

[48] K. Bhaduri, K. Das, and B. L. Matthews, "Detecting Abnormal Machine Characteristics in Cloud Infrastructures," in *The 11th International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2011, pp. 137–144.

[49] A. Boin and M. J. Van Eeten, "The Resilient Organization," *Public Management Review*, vol. 15, no. 3, pp. 429–445, 2013.

[50] E. Hollnagel, D. D. Woods, and N. Leveson, *Resilience Engineering: Concepts and Precepts*. Ashgate Publishing, Ltd., 2007.

[51] F. Adler and R. Karban, "Defended Fortresses or Moving Targets? Another Model of Inducible Defenses Inspired by Military Metaphors," *American Naturalist*, pp. 813–832, 1994.

[52] F. B. Schneider, "From Fault-tolerance to Attack Tolerance," http://www.dtic.mil/dtic/tr/fulltext/u2/a548748.pdf, (2011). Accessed April 19, 2016.

[53] A. Bessani, J. Sousa, and E. E. Alchieri, "State Machine Replication for The Masses with BFT-SMaRT," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 355–362.

[54] L. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of The IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[55] "BFT-SMaRT: High-Performance Byzantine Fault-Tolerant State Machine Replication," http://bft-smart.github.io/library/, Accessed April 20, 2016.

[56] H. T. Dang and F. Hermenier, "Higher SLA Satisfaction in Datacenters with Continuous VM Placement Constraints," in *Proceedings of The 9th Workshop on Hot Topics in Dependable Systems*. ACM, 2013, p. 1.

[57] Y. Mei, L. Liu, X. Pu, S. Sivathanu, and X. Dong, "Performance Analysis of Network I/O Workloads in Virtualized Data Centers," *IEEE Transactions on Services Computing*, vol. 6, no. 1, pp. 48–63, 2013.

[58] R. O. Suminto, A. Laksono, A. D. Satria, T. Do, and H. S. Gunawi, "Towards Pre-deployment Detection of Performance Failures in Cloud Distributed Systems," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[59] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[60] F. B. Schneider, "Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[61] M. Castro, B. Liskov *et al.*, "Practical Byzantine Fault-Tolerance," in *USENIX Symposium on Operating Systems Design and Implementation*, vol. 99, 1999, pp. 173–186.

[62] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and The Art of Practical BFT Execution," in *Proceedings of The 6th Conference on Computer Systems*. ACM, 2011, pp. 123–138.

[63] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-Efficient Byzantine Fault Tolerance," in *Proceedings of The 7th ACM European Conference on Computer Systems*. ACM, 2012, pp. 295–308.

[64] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested Append-Only Memory: Making Adversaries Stick to Their Word," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 189–204.

[65] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine Replication Under Attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, 2011.

[66] Z. Milosevic, M. Biely, and A. Schiper, "Bounded Delay in Byzantine-Tolerant State Machine Replication," in *IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2013, pp. 61–70.

[67] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults," in *The Proceedings of The 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 9, 2009, pp. 153–168.

[68] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin One's Wheels? Byzantine Fault-Tolerance With a Spinning Primary," in *Proceedings of The 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 135–144.

[69] M. Castro and B. Liskov, "Practical Byzantine Fault-Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[70] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI Quarterly*, vol. 2, no. 3, pp. 219–246, 1989.

[71] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of The ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[72] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[73] "BFT-SMaRT: High-Performance Byzantine Fault-Tolerant State Machine Replication," http://bft-smart.github.io/library/, Accessed April 20, 2016.

[74] R. Martins, R. Gandhi, P. Narasimhan, S. Pertet, A. Casimiro, D. Kreutz, and P. Veríssimo, "Experiences With Fault-Injection in a Byzantine Fault-Tolerant Protocol," in *Middleware 2013*. Springer, 2013, pp. 41–61.

[75] T. R. Mayer, L. Brunie, D. Coquil, and H. Kosch, "Evaluating The Robustness of Publish/Subscribe Systems," in *P2P, International Conference on Parallel, Grid, Cloud and Internet Computing*. IEEE, 2011, pp. 75–82.

[76] R. S. Kazemzadeh and H.-A. Jacobsen, "Reliable and Highly Available Distributed Publish/Subscribe Service," in *28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 41–50.

[77] T. Chang and H. Meling, "Byzantine Fault-Tolerant Publish/Subscribe: A Cloud Computing Infrastructure," in *IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2012, pp. 454–456.

[78] T. Chang, S. Duan, H. Meling, S. Peisert, and H. Zhang, "P2S: A Fault-Tolerant Publish/Subscribe Infrastructure," in *The Proceedings of The 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 189–197.

[79] L. Jehl and H. Meling, "Towards Byzantine Fault-Tolerant Publish/Subscribe: A State Machine Approach," in *Proceedings of The 9th Workshop on Hot Topics in Dependable Systems*. ACM, 2013, p. 5.

[80] R. S. Kazemzadeh and H.-A. Jacobsen, "PubliyPrime: Exploiting Overlay Neighborhoods to Defeat Byzantine Publish/Subscribe Brokers," in *Proceedings of The ACM/IFIP/USENIX 13th International Conference on Middleware Demos and Posters*, 2012.

[81] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[83] "Rabitmq," http://www.rabbitmq.com/ha.html, Accessed April 19, 2016.

[84] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, and S. Kounev, "HInjector: Injecting Hypercall Attacks for Evaluating VMI-based Intrusion Detection Systems," in *Poster Reception at The 2013 Annual Computer Security Applications Conference (ACSAC 2013)*, 2013.

[85] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, Fidelity and Stealth in The Drakvuf Dynamic Malware Analysis System," in *Proceedings of The 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 386–395.

VITA

VITA

Noor Ahmed received his BS degree from Utica College in 2002, MS degree from Syracuse University in 2006 and PhD from Purdue University in 2016, all in Computer Science.

Noor (aka Norman) Ahmed is a Computer Scientist at the Air Force Research Laboratory's Information Directorate (AFRL/RI) since 2003. His research interests include: Security in Cloud Computing with special emphasis on Anomalous-based Intrusion Detection on virtualized cloud platforms, Security and QoS in Service Oriented Architectures, Semantic Computing, Reliability and Resiliency in Distributed Systems, and Moving Target Defense (MTD). He serves as a session chair and program committee in conferences and workshops in these research areas, and a reviewer in IEEE transactions on Cloud Computing. His research is focused on building an MTD-based attack and disruption resilient framework for distributed systems.