From Byzantine Fault-Tolerance to Fault-Avoidance: An Architectural Transformation to Attack and Failure Resiliency

Noor O. Ahmed and Bharat Bhargava, *Fellow, IEEE*

Abstract—We present Byzantine Fault-Avoidance (BFA), a fault-resilient architecture designed for Byzantine Fault Tolerant (BFT) systems to withstand against attacks and failures. BFA allows replicas to short live on a given computing platform, i.e., hardware, hypervisors and OS, to thwart successful and in-progress attacks while simultaneously preserving the correctness condition of BFT properties; *safety* and *liveness*. BFA combines the cloud management software stack of OpenStack (*nova)* and the *Software Defined Network* (SDN) implementation *(neutron)* to control the replicas susceptibility window of attack in order to avoid Byzantine faults. The proposed fault-avoidance scheme illustrates the defensive security solutions enabled by the underlying cloud computing fabric are far superior than the ones implemented at the application/protocol level. Preliminary results of widely studied BFT system (*BFT-SMaRT*) deployed in a cloud infrastructure (*OpenStack-Kilo*) indicate that BFA achieves desired BFT reliability properties and throughput over contested environments.

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

1 Introduction

The Byzantine Generals problem was first introduced by Lamport [3] as an abstract notion of constructing a provable reliability-guaranteed replicated distributed system with 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 [4]. 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 ones. Thus, satisfying the correctness condition properties; *safety* and *liveness*. The *safety* property asserts that some of the replicas remain consistent of 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 real-world setting. A Practical Byzantine Fault Tolerant (PBFT) system [5] that achieves performance close to a non-replicated system was published and open sourced to help the ever-increasing need for reliable distributed systems. PBFT has sparked a wide-array of research to further improve its performance, i.e., reducing communication steps, replication costs, and addressing security issues which is the focus of our work.

Performance improvement was mainly credited to the advancement of cloud computing, for example, virtualization techniques have been used on improving replication costs from *3f+1* where *f* is the faulty replica, to f+*1* in ZZ [12], CheapBFT [13] and A2M [24]. While replicating systems on a highly dynamic virtualized elastic cloud environment is undeniably cost effective, it's increasingly challenging to guarantee reliability due to the inherent increase in attack surface [15]—the set of ways an adversary can exploit/penetrate the systems, due to the number of components built on the virtualized cloud environment.

Amir et al. first reported in "Byzantine Replication Under Attack" [6] 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 when some of the servers are faulty. 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 protocol and introduced Prime [7], 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 [8]. BFT-Mencius

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

Manuscript received 11 Feb. 2016; revised 21 Nov. 2016; accepted 22 Dec. 2016. Date of publication 12 Mar. 2018; date of current version 8 Sept. 2020. (Corresponding author: Noor O. Ahmed.)
Recommended for acceptance by J. Cao.
Digital Object Identifier no. 10.1109/TCC.2018.2814989

introduced an Abortable Timely Announced Bounded-Delay broadcast protocol that is on the order of real communication delays. Other notable works that address this issue include; Aadvark [9], which proposed a change in leadership when suspected, i.e., when the leaders' performance is slowing down. Along the same lines of work, Spinning [10] constantly rotates the leaders' role after every patch of accepted client request for execution, similar in spirit to BFA.

However, all of these 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 due to the computing landscape differences between the traditional computing platforms and the virtualized cloud environments (i.e., many moving parts) with its programmable network model (i.e., SDN), defending replicated system (i.e., BFT) with the existing solution approaches is extremely challenging. For instance, in a cloud platform, a compromised BFT leader/server not only disrupts the reliability of the BFT protocol but can also wage several attacks on the SDN controllers and the data plane forwarding flows, specifically, a compromised host can poison the entire network topology or even take control of the entire infrastructure [31].

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 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 present an attack and failure resilient architecture designed for BFT systems to avoid byzantine faults through controlling their exposure attack window while simultaneously preserving the correctness condition of BFT properties; *safety* and *liveness*. We accomplish such control 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, we 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). This solution approach is commonly referred as a Moving Target Defense (MTD) [25].

Our approach combines recent advances in cloud platform management capabilities, specifically, VM provisioning/de-provisioning and the programmable networking model (software defined networking) to reduce the node's exposure time to attack. We deploy BFTSMaRT [32], an open source BFT implementation, on Mayflies [2], a bio-inspired generic MTD framework for distributed systems introduced by the same authors, on a private cloud setting built with OpenStack-Kilo [33], and discuss the implementation details of BFT to BFA transformation. To illustrate the practical effectiveness of BFA, we show the protocol evolving with the correct behavior with a negligible overhead while on the move across platforms.

We make three contributions in this work:

1) We propose the first practical Byzantine Fault-Avoidance (BFA) architecture using the cloud platform technologies.
2) We provide a dynamic BFT VM/replica refresh algorithms for replicas on virtualized cloud platforms to control their exposure to attacks.
3) We introduce an attack/failure avoidance scheme for any deterministic system with state and some operations without any modification while preserving it's correctness conditions.

We have organized the paper as follows: we first give a brief overview of a BFT system and cloud ecosystem in Section 2, followed by the threat scope and assumptions that we consider in Section 3. We present BFA system model in Section 4, followed by the design and implementation in Sections 5 and 6. We present preliminary experimental evaluations in Section 7 . Finally, we discuss the related work and the conclusion in Sections 8 and 9 respectively.

2 Background

In this section we give a brief overview of BFT architecture and cloud eco-systems to set the context of our architectural transformation scheme to attack and failure resiliency.

2.1 Byzantine Fault-Tolerance Systems

Byzantine Fault Tolerance (BFT) is a well-established reliability guaranteed distributed system based on state-machine replication model. PBFT [30] 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 clients'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 multicast to the other pre-defined replicas in the *propose* phase. The replicas confirm the receipt of the request back to the leader and transition 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.

2.2 Cloud Eco-System

The key promise of the cloud adaptation is the cost benefits of the computing resources that can scale up/down on demand, referred as pay-as-you-go elastic computing. Infrastructure (IaaS), Platform (PaaS), and Software (SaaS) -as-a-Service are the three service deployment models for cloud environments. Each deployment model offer different protection schemes for the applications. Since our work does not involve infrastructure-level interaction, we developed our solutions in PaaS model.

While the on-demand elastic computing and the simplified service deployment models of the cloud is undeniably useful, it's increasingly challenging to guarantee provable reliability due to the sophisticated cyber attacks in recent years. The transient hardware and software design faults inherent on the commercial-off-the-shelf hardware built on such infrastructures have amplified such threats.

On the other hand, by design, such infrastructures also provide unprecedented security capabilities enabled by the structured underlying computing architecture and virtualization. The two key capabilities are the *isolation* and *introspection*, the ability to safeguard systems below the OS that they are on. 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 they are on a separate OSs. OS-level *isolation* allows running multiple OS instances on the same physical hardware, referred as *multi-tenancy* credited to the Virtualization technologies. At the core of the Virtualization world is the Virtual Machine Monitors (VMMs), referred to as a *hypervisor*.

There are two types of virtualization techniques for building cloud infrastructures: type I and type II. Type I enables bare-metal hardware virtualization mechanism within the hosting OS kernel, i.e., Xen and other commercial ones. Quick Emulator (KVM/QEMU) is the second type capable of virtualizing hardware resources. In this work, our MTD solution is built on OpenStack [33], an open source cloud software stack. OpenStack supports all types of hypervisors, and is widely used in the commercial space, for example, RackSpace [36], a commercial cloud provider that serve many well established businesses like Netflix.

Recent computing advances on cloud technologies enables hardware assisted security. These include; Intel's SGX processor which divides the CPU into many secure enclaves/sandboxes to protect applications from each other or even from a compromised OS. The ARM TrustZone processor where the CPU is divided into two halves, the insecure and secure worlds that communicate via a Secure Monitor Call instruction. In this work, we are interested in applying BFT solutions using only existing publicly available cloud software stack and commodity architectures without hardware customization.

3 Threat Scope and Assumptions

We consider an adversary that gains full control/privilege of a virtual machine undetected by the traditional defensive mechanisms, a valid assumption in cyber space. The adversary can be a user with physical access to the guest VM node/replica (not the hypervisors), escalated privilege, or even an escaped tenant from the neighbouring VM. The adversaries' advantage, in this case, is the unbounded time to disrupt the reliability of the system. The fundamental premise of BFA is to eliminate such advantage of time and complicate the attackers gain/loss balance.

We 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, since our scheme is designed for BFT systems avoid faults, we consider the standard assumptions of existing BFT systems and fault models [3] on both clients and servers. Typically, these systems consist of N identical replicated servers with Byzantine faulty model, i.e., they demonstrate correctness or arbitrary faults that deviate from the protocol. We assume the clients's faulty behaviors (i.e., replay attacks) are handled by the BFT protocol using crypto and digital signature techniques that are computationally bound to subvert.

4 BFA System Model

BFT systems are typically implemented using State Machine Replication (SMR) model [4] and formalized with I/O automaton [11], therefore, it's 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, $steps(B) \subseteq states(B) \times acts(B)$ where the actions *acts(B)* are the client request, the *propose, write, and accept* consensus protocol are messages and consensus decision as *in(B), internal(B) and out(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.

4.1 BFT to BFA Transitions

In this work, we are considering quorum-based also known as 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 internal actions are the consensus state transitions (*propose/prepare, write, and accept*) where the accept/BFTCommit is the last state transition action that happens upon successfully completing/committing a client request. We pass this transition action to our BFA system as an input to trigger its internal actions as illustrated in Fig. 1. However, for an I/O automaton, the *internal* actions of SMR-based system are not visible to other systems in the same environment [11].

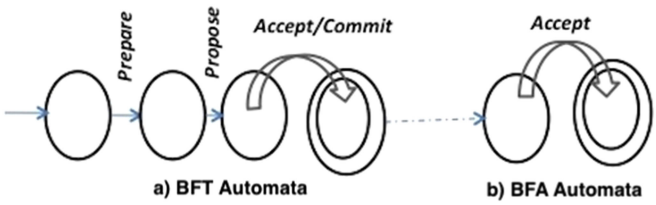

Fig. 1. Illustration of Finite State Automata Composition for BFT in a, and BFA in b. The transition of the BFT *accept/commit* state triggers as an input to BFA (dotted arrow) and transitions it to an *accept* state for refreshing a replica.

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 execution for the recovering replica/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 liveness*).

In order to perform useful computations, we consider a transition after n accepted *BFTCommits* to indirectly trigger BFA system to transition to an accepting state as depicted in the dotted lines in Fig. 1. Thus, we consider this transition point as the application state transfer checkpoint between the terminating and the newly created replicas, discussed in Section 6.2. The input enabled action of BFT's I/O automaton clearly shows that BFA is suitable for any SMR-based deterministic system with state and some operations without any modification.

4.2 (Sketch) Correctness Proof

The rationale behind modelling BFA with I/O automata was due to the underlying SMR-based BFT system which is typically modelled with automata. Therefore, it's 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 Fig. 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's intuitive to see if these *invariants* hold in one replica refresh, 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 liveness*.

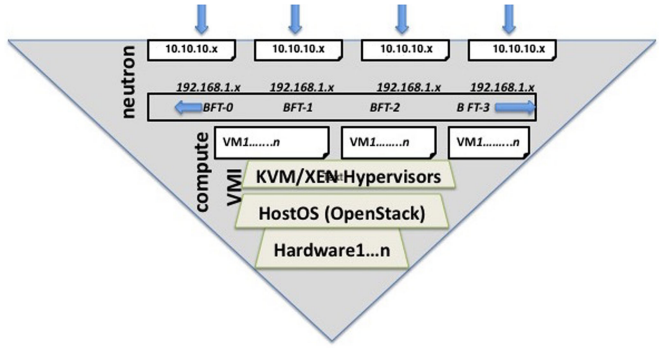

Fig. 2. BFT system logic view on a cloud platform.

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 [16] the impossibility of achieving *safety* with synchrony, however, the SMR-based I/O automaton model guarantees that the replica responses will be correct according to Linearizability [14].

5 BFA System Design

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

5.1 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 in Fig. 2.

In Fig. 2, 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 layers of the stack. There are *n VMs* on each Host OS that is controlled with the *nova compute*. Note that the Virtual Machine Introspection (VMI) is used for proactive monitoring of the VMs at the hypervisor-level.

To illustrate, we deployed four BFT replicas *(BFT-0, BFT-1, BFT-2 and BFT-3)* on the *VMs*. 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. These *VMs* are interconnected with LAN address (192.x.x.x), referred as *fix* IP, and externally exposed with WAN address (10.x.x.x.), referred as *floating* IP. This mapping is controlled with the *neutron* component, an implementation of Software Defined Networking (SDN). 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* allow VM provisioning/de-provisioning and the *neutron* enables

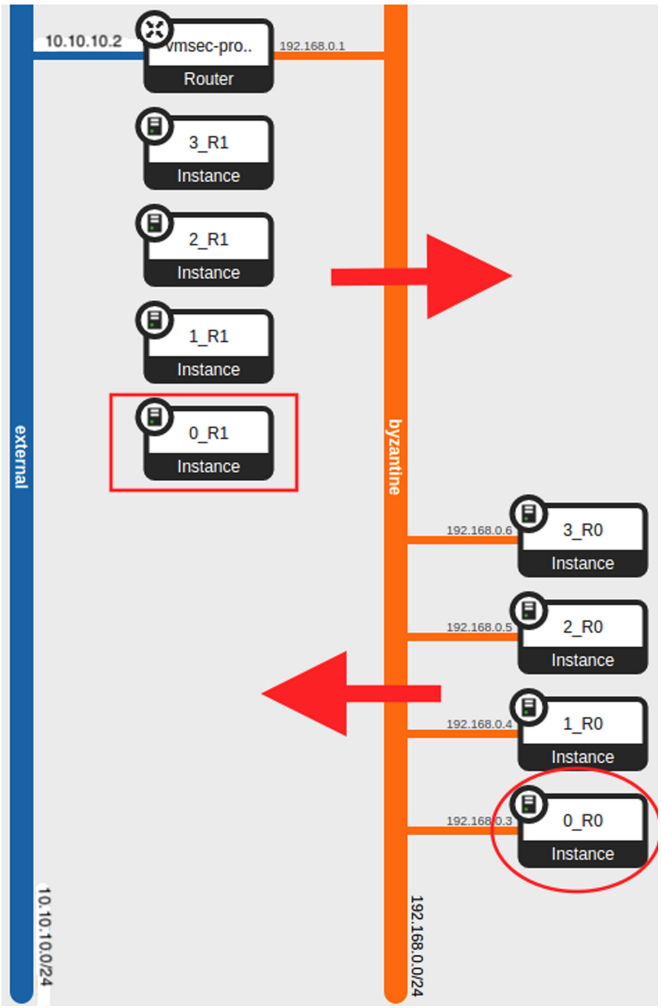

Fig. 3. Horizon Dashboard view of the 4 *BFT* replicas *0_R0 ...3_R0* on the right vertical bar (*byzantine* subnet) and 4 isolated standby replica pool *0_R1 ...3_R1* between the vertical bars/subnets. A virtual router *(vmsec-proj)* interconnects the two subnets with *192.x.x.x* IP on the *byzantine* subnet side and *10.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.

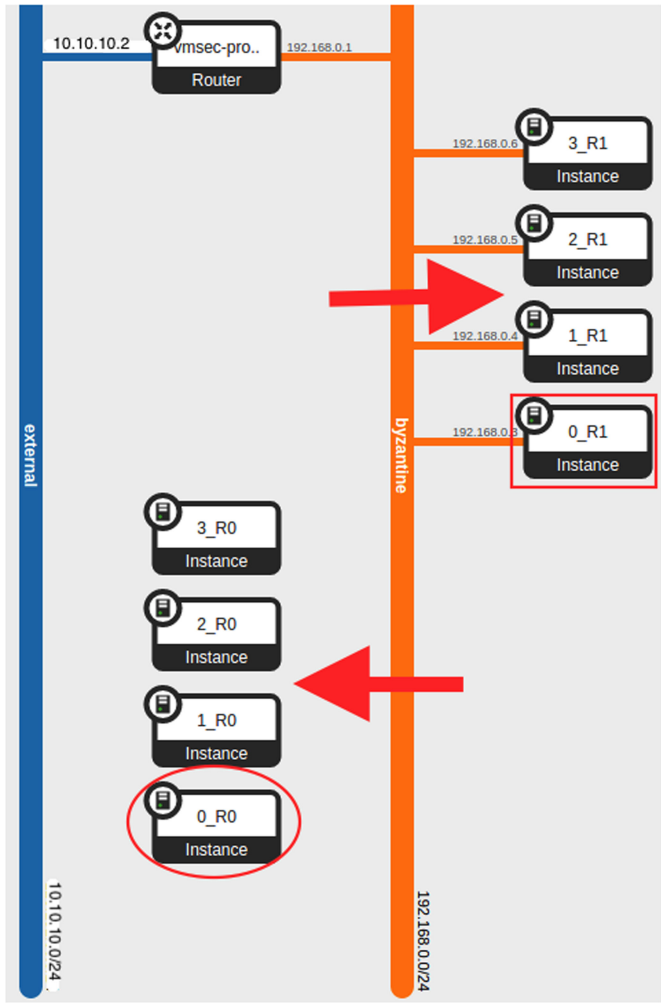

Fig. 4. The result after the replicas refreshed, all *x_R0* replicas are removed from the subnet *byzantine* and replaced with the standby replica from the pool *x_R1* (rectangle box to the oval box) while serving clients.

the dynamic network reconfiguration capabilities, thereby, used for refreshing the *VMs* (discussed next) by continuously provisioning/de-provisioning at runtime. Thus, creating mechanically-generated diversity which is almost as powerful a defense as typechecking [23].

5.2 Replica Refresh

For decades, 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* or VM Refresh 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 different hardware, hypervisor, host and 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 Figs. 3 and 4.

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 in Section 5.2.2.

5.2.1 Pre-Prepared VM Pool

One 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 activate upon booting the instance, network configurations, etc. Another way is through the *horizon* dashboard. We prepared 4 standby replicas *(0_R1, 1_R1, 2_R1, 3_R1)* using the command line as shown between the vertical bars (subnets) and 4 BFT use case replicas *(0_R0, 1_R0, 2_R0, 3_R0)* on the *byzantine* subnet (left vertical bar), depicted in Fig. 3.

The naming convention used in our prototype *x_Rx* stands for the replica ID *x* and the round *R* it's 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). Fig. 4 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 continues to the next round *(x_R2)* of replicas for round 2, and so on.

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. We manipulate the interfaces at runtime when they are taking over the role of a server *(x_R0)* in the *byzantine* subnet. Algorithm 1 illustrates the replica refresh procedure REFRESH() and the implementation details of the transformation is discussed in Algorithm 2 in Section 6.3.

Algorithm 1. 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 1, 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 *fix* 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's 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 reconnect 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.

5.2.2 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 versus 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 acceptable threshold when an additional node fails (naturally or compromised), thereby, violating the preservation of the reliability properties *(safety and liveness)*.

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 our 4 replica use case depicted in Fig. 4. 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 (i.e., flat) that is ineffective to dynamically manipulate at runtime.

The standby VM pool can be prepared on the externally visible subnet than the internal *byzantine* subnet (192.x.x.x) when using *floating* IP for both the clients and servers, or perhaps, setting it in flat network topology than using 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 that is mapped to the same port number *xyz*. This mappings are seamlessly handled by the SDN.

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 prepared 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 upto a minute in preparation.

6 Implementation

We implemented our algorithms with *bash* shell script using *Mayflies* [2] MTD framework. *Mayflies* is tightly integrated

into *OpenStack* (Kilo) [33] cloud framework, an open source cloud management software stack widely adopted in commercial clouds. For instance, RackSpace [36], 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. The core components that enables the MTD techniques are *nova, neutron and horizon*.

To illustrate our proposed BFA, we used BFT-SMaRT [32] prototype downloaded from [29], an opensource Byzantine Fault Tolerant system. 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 prototypes and none come close to BFT-SMaRT given the fact that BFT research has been around for decades.

We deploy a BFT system to an OpenStack cloud platform and continuously refresh the VMs in order to control their existence for the hope of reducing their exposure window of attacks and avoid faults, thus, transforming into 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.

6.1 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 client request or the committed transaction number. 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 static files are loaded only once at the server start up, 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 using VMI. 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 with secure shell (ssh), 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's 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. 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 eliminates 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 replica/server afterwards.

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

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 the improvements of this issue in Section 7.3.2.

6.2 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 Fig. 5, it's intuitive to see the replicas in the initial

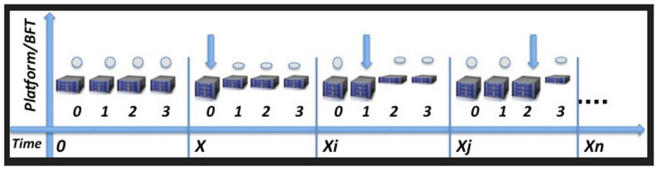

Fig. 5. Illustration of exposure attack window time line. The four computer/platforms with IDs (0...3) represents the host OSs, and the circle above it represent the BFT servers (guest OSs). Note each platform can host multiple guest *VM* instances. The initial 4 node use case show in the first block and refreshing a replica (pointing arrow) after a predefined *lifespan (x)* shown at the (x-axis) and x_i, ...x_n in the consequence blocks with enlarged platforms depicting as a stronger replica.

window are vulnerable to attacks if they stand still for the entire time (static). 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.

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 Fig. 5 (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 lessons learned section (Section 7.3). 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.

Below is the monitoring bash shell script code snippet. 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 refresh process implemented in Algorithm 2, 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's always the first time whenever we refresh a replica, therefore, we added these options [ssh –o UserKnownHosts-File=//dev//null -o StrictHostKeyChecking=no ...] in order to skip the confirmation message. Also, note that the script code is broken down to fit the column.

6.3 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 monitor the existence of this file, we refresh the replica once the file is detected. Algorithm 2 below illustrates the logical implementation of the proposed architectural transformation.

Algorithm 2. 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 1 above
9:                 InjectState()                 ▷ scp
10:                StartServer()                 ▷ 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
```

In Algorithm 2, 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.

The GetCurrentViewFile() in line 7, the InjectState() in line 9 and StartServer() in lines 10 are implemented with secure copy scp and secure shell ssh commands as described in Section 6.1. To illustrate the concept, in our experiment, 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.

7 Evaluations

As depicted in the BFT system logic view diagran in Fig. 2 in Section 4.2, each hardware/machine *(HW1...HWn)* lie a Host OS with a hypervisor *(HV...HVn)*, 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 *CounterClient()* 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.

7.1 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 8 GB 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.

7.2 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 tolerates, 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 ID 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 leads to refreshing the newly selected leader at each round. Therefore, we start with server 1 to complete 20K, then 0 for 40K, and so on.

Table 1 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

TABLE 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 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

TABLE 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

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 *exitcertificate* 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 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 [32] and the elapse time between the termination of the replica and the start of a new replica is typically less than 10 seconds as shown in Table 3 (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 3 shows *Generic* case were we start the servers without stopping, *Re-Start* and *Refresh* experiments reported in Tables 1 and 2 with the same 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

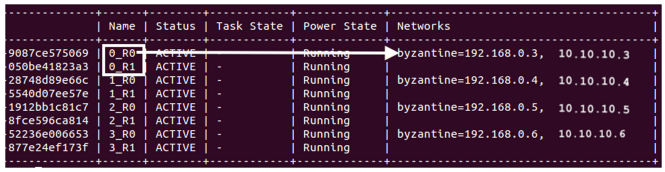

Fig. 6. An output of `nova list` command showing the BFT and the standby replicas with their network mappings before the transformation. The arrow shows the (x_R0) replica groups have network interfaces and x_R1 have none.

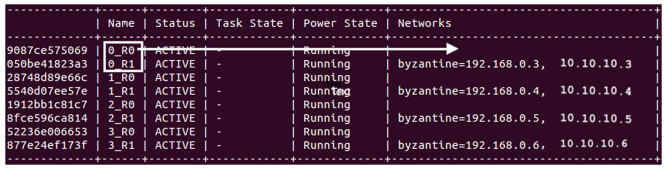

Fig. 7. Post BFA transformation results. The network interfaces of (x_R0) group are seamlessly transferred to the (x_R1) group. Note (x_R0) enteries are blank.

to restart 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 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 ~ 200K+ requests) to disrupt attacks.

Fig. 6 reflect the graphical *horizon* dashboard network topology shown in Fig. 3 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)*.

Fig. 7 shows the result after the transformation algorithm completes for one round. The white square box shows the same two replicas R0 and R1 depicted in Fig. 6, now the network interface for R0 is attached to R1 as shown the arrow pointing R0 with no network entry as well as all the *(x_R0)* group. This transformation can continue as (*x_R2*) for round 2 with newer VMs created/refilled instances in line 12 of the Algorithm 2, and so on.

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 1 and 2, and the SDN results of Figs. 6 and 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 in our future work.

7.3 Lessons Learned

In this section, we will discuss the key lessons learned during this project. These include: 1) adopting the cloud software stack (OpenStack) *nova and neutron* components in a dynamic fashion, 2) the behaviour of the BFT-SMaRT leader change recovery protocol, 3) dealing with the replica *lifespan*, and 4) the limitations of our approach.

7.3.1 Synchronizing Nova and Neutron

The process of refreshing replica in a cloud platform 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 its *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 (crashing) 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 for function returns in such environment.

7.3.2 Improving BFT-SMaRT Recovery Protocol

We timed the BFT recovery protocol by restarting the same server, and also refreshing it with a different VM as shown in Tables 1 and 2 (column 2) in the previous section. The recovery time is less than a second for restarting the same server, and refreshing with a new VM is between 8-11 seconds. In both cases, we observed that occasionally the system crashes when we restart/refresh another node given the fact that the first restarted/refreshed node has joined back the replica group. This is due to the node trying to catch up processing the missed messages (timeouts), as a result, causing the system to fall below the faulty allowed threshold when another node is terminated.

Similar 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 catchup mode early in the game. We set the replica *lifespan* to 20K increments to space out the termination and considering to thoroughly analyze the code and systematically solve this issue in the future.

Another major issue of transforming BFT to BFA is the recovery of the leader upon crash or failure. It has been reported that the recovery time for a non-leader replica in BFT is negligible and about 20 seconds for the leader [32]. This is due to the leader change protocol messages (*regency x, where x is the replica/server ID*) exchanged in order to unanimously decide for a new leader when the leader fails/crashes. Given the fact that our failure inducing (terminating/re-starting) replicas as a defensive technique (MTD) takes only between 8-11 seconds for all the replicas as noted earlier, we considered improving the leader recovery protocol for BFA and the system stability in a systematic fashion than the existing approaches described in the introduction section, as well as eliminate threats that makes the system barely usable [6] or even revolve around malicious leaders [20].

We observed (logs/code) that the recovering/refreshed leader complicates the leader selection process when it joins at specific point in the process. This is due to the reconnected leader sending messages that show its ID *(regency x)* as the known leader and others sending for unknown leader message *(regency-1)* to activate the leader change protocol. Upon activating a leader change, all the replicas exchange other messages (STOP, STOP_DATA, SYNC, etc.) to synchronize the last committed transaction (client request) number before deciding for a new leader.

Once the last transaction is synchronized, the replicas then exchange the next candidate leader *(regency+1)* selection message and agree with a majority vote. However, with the reconnected leader message claiming as a known leader is in the pipe, and some of the replicas verify/accept that claim (additional messages to be exchanged), getting a majority vote makes it difficult. In some cases, we observed that the *regency+1* increments goes beyond the participants IDs, thus, causing a dead-lock in deciding for a new leader.

To reduce the leaders' recovery time and its inherent issue in a systematic approach, we introduced a reclaim leadership method in the protocol in which the recovering/refreshed 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 their known leader and the current leader has not yet been decided, then, cancel all

the synchronization messages in the pipe. This greatly improved the recovery process of the leader.

We are currently evaluating parallel *catchup* process and considering to evaluate BFA with the improved BFT-SMaRT and report results in the future. We consider submitting the improved leader recovery and the catchup code to the BFT-SMaRT authors to validate and include in their future releases.

7.3.3 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's written for our interested value x, however, we discovered that there are 60 ± 10 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 create a file called *exitcertificate* to signal that point. We monitor the existence of that file instead of the log file entries. Upon detecting it, the replica gets refreshed safely as described in Section 6.2, thus, assuring the replicas to exist only with their intended *lifespan*.

7.3.4 Limitations

The criticality of incorporating *attack-tolerance* to *fault-tolerance* protocols as an integral part of distributed system's architecture and protocols was first addressed in [21] 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. With the advances of cloud software stack and SDN implementations emerge, and the BFT protocols re-engineered to adopt to such platforms, we believe MTD-based security solutions for BFT on cloud computing is far superior than the traditional defensive approaches.

One of the major improvement, for instance, in *OpenStack-Icehouse* release, is 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 gets deleted once detached from the VM. 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.

With this minor improvement, the VM replica refresh time decreased from 20 seconds to between 8-11 seconds depending on the network service workload. As a result, narrows the time needed for the adversaries craft an effective attack when the pattern of our re-deployment time frame is discovered. However, the current BFA solution approach fails against attacks that take less than 8 seconds.

In this work, we focussed on the following three parts; 1) determining and improving the MTD cost inherent on the cloud software stack to reduce the replicas window of attack, 2) replicas state transfers, and 3) the replica lifespan. To fully achieve BFA on cloud computing, some of the limitations need to be addressed include:

1) Balancing between the performance impact on delivery latency, replica refresh time, and the security robustness, when the protocols' cryptographic message digests and certificates (i.e., X509 certs) is enabled.
2) Determining an optimal lifespan x in order to prevent for an attacker causing byzantine faults during the replicas lifespan.
3) Dealing with the application-level vulnerabilities, and the security risks inherent on blindly selecting VM/replica candidates and platform destinations without informed decision (i.e., avoiding platforms/configurations that are susceptible to attacks or have known vulnerabilities).

8 Related Work

To the best of our knowledge, BFA is the first to leverage cloud management software stack to address BFT security issues, therefore, we divide our related work section into two parts; we first discuss works on BFT attack and fault detection/prevention and system diversification (MTD) approaches, we then consider research approaches that leverage virtualization that are relevant to our work.

8.1 BFA Fault Detection

Supporting *fault* and *attack* tolerance in a distributed trust by coupling the replication with threshold cryptography was first proposed in [21]. An alternative approach that aims to detect faults by monitoring rather than masking is proposed in [35] where each node is equipped with a detector that monitors each other for signs of faulty behavior and isolated if suspected. Another scheme that continuously monitors the leaders' behavior and changes upon suspicious behavior is proposed in Aadvark [9]. Similarly, Spinning [10], develops a techniques for constantly rotating the leader after every patch of request processed and punish the faulty ones.

The monitoring schemes of all the above systems are within the application layer, thereby, easily defeated once the system is compromised by only infecting even a single replica. In addition, the other difference is that BFA eliminates the maintenance cost for dealing with the fault nodes

by constantly refreshing replicas, thereby preventing future faults in the event of a compromised undetected node in the replicated domain.

Application level diversification or program synthesis is first introduced in N-version programming [18]. They propose methods of developing different implementations of the same specification by different development groups while anticipating to produce versions without common faults. A fault-tolerant approach of this scheme was later introduced in [17]. However, experiments have shown that common flaws in the independently implemented binaries do occur [19].

Furthermore, N-variant [26] introduced program synthesis or program re-writing, where computationally variable binary forms of the same program are produced to enable diversity. A new ordered broadcast fault-tolerant service was recently proposed [27] that accomplishes diversity in time by switching between 2/3 consensus protocol and Paxos, and diversity in space by running synthesized program in two different evaluators. However, BFA enables diversity in time and space to a wide range of existing BFT SMR-based systems without modifying the application protocol while leveraging the underlying computing fabric.

Note that BFT-SMaRT allows a replica to dynamically leave and join at runtime for the system to shrink or grow through a reconfiguration protocol run as a client application. The application notifies the joining or leaving replica to all other replicas, thus, not suitable to use as a refreshing scheme for MTD strategy. BFA transformation scheme seamlessly removes the replica and adds it back without notifying to the rest of the replicas and all is done outside the protocol, however, it can supplement to the dynamic BFT work on selecting replicas from the prepared *VM* pool for faster leave/join process.

8.2 Virtualization Techniques

Although BFA solution is built around the cloud platform, it's not the only work that leverages virtualization (VM) techniques, for example, ZZ [12], CheapBFT [13] and A2M [24] leverage VM for BFT performance improvements, i.e., replica costs.

Others have used VM for security, for example, an intrusion tolerance BFT for web services that stops functioning when a security violation is detected [34], and MAS [28] that rotate web-services with versions of application implementation, webservers on different OSs and hypervisors between on-line and off-line to deceive adversaries. A diversified guest OS's and binary randomization of single-machine BFT is proposed in [37]. An architecture for secure outsourcing of data and arbitrary computation between trusted cloud that serves as a proxy between the client and untrusted commodity clouds is proposed in [38].

These improvements have been credited to the advances in computing, especially, virtualization on commodity hardware and trusted subsystems, i.e., Trusted Computing, TPM, FPGA, Gurbled Circuts, and Homomorphic Encryption. Unlike other solutions, BFA neither depends on trusted subsystem nor globally shared space, and the system continues to evolve and remain online.

Overall, the main difference is that BFA is not a new BFT protocol to mitigate attacks, it's an architectural integration of *fault-tolerant* protocols and *attack-tolerance*. BFA enables agility to existing rigid SMR-based well established BFT systems in a real world setting by leveraging simple but effective mechanisms enabled by the infrastructure. While all these approaches are affective to improve replica costs and defend against class of attacks, our work is complementary to those that implement monitoring faulty behavior and change/rotate leader.

9 Conclusion and Future Work

We proposed a Byzantine Fault Avoidance (BFA) techniques built around the cloud's software management stack *(VM provisioning/de-provisioning, neutron, horizon, and SDN)*. By synchronizing the cloud software stack in a dynamic fashion to fully control the existence of the replicas, we were able to control the replicas attack exposure window in order to complicate the attackers' system control balance.

We showed the practicality and effectiveness of our scheme with a widely studied byzantine fault-tolerant system (BFT-SMaRT) in the literature deployed on a private cloud built with OpenStack and *Mayflies* MTD framework techniques. BFA demonstrated that the capabilities enabled by the underlying computing fabric are simpler and more effective than the ones implemented at the application-level protocol to introduce dynamicity into the system in order to disrupt against modern sophisticated attacks.

Future works will address the following; secure diversification using TPM, IP-Hopping, trace-based performance analysis across multiple public cloud platforms while the system is under attack, and practical experiments using Micro/Andrew benchmarks. We consider integrating a Virtual Machine Introspection-based system runtime integrity violation detection scheme introduced in our previous work [1] to address one of the limitations (3) described in Section 7.3.4. Furthermore, we will extend the VM refresh algorithm on container technologies (i.e., docker).

Acknowledgments

Authors would like to sincerely thank Jim Hanna for his support on the experimental cloud platform. Special thanks to Dr. Mark Linderman and Steven Farr at AFRL for their continuous support and guidance, and the reviewers for their constructive comments and making this paper more readable. Note: the external IP address 10.x.x.x on figures 3,4,6, and 7 was edited for privacy.

References

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

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

[3] L. Lamport, R. Shostaka, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Languages Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[4] F. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proc. 3rd Symp. Operating Syst. Des. Implementation*, 1999, pp. 173–186.

[6] Y. Amir, J. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *Int. Conf. Dependable Syst. Netw.*, Jun. 2008, pp. 197–206.

[7] Y. Amir, J. Yair, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack, "Dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 8, no. 4, pp. 564–577, Jul./Aug. 2011.
[8] Z. Milosevic, M. Biely, and A. Schiper, "Bounded delay in byzantine-tolerant state machine replication," in *Proc. 32nd IEEE Int. Symp. Reliable Distrib. Syst.*, 2013, pp. 61–70.
[9] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proc. 6th USENIX Symp. Netw. Syst. Des. Implementation*, 2009, vol. 9, pp. 153–168.
[10] G. S. Veronese, M. Correia, A. N. Bessani, and L. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *Proc. 28th IEEE Int. Symp. Reliable Distrib. Syst.*, 2009, pp. 135–144.
[11] N. Lynch and M. Tuttle, "An introduction to input/output automata," in *Publisher PN*. MIT Technical Memo MIT/LCS/TM-373, 1988. [Online]. Available: http://groups.csail.mit.edu/tds/papers/Lynch/CWI89.pdf
[12] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and the art of practical BFT execution," in *Proc. 6th EuroSys Conf.*, 2011, pp. 123–138.
[13] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, and K. Stengel, "CheapBFT: Resource-efficient byzantine fault tolerance," in *Proc. 7th ACM Eur. conference Comput. Syst.*, 2012, pp. 295–30.
[14] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Languages Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.
[15] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 371–386, May/Jun., 2011.
[16] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. Assoc. Comput. Machinery*, vol. 32, no. 2, pp. 374–382, Apr. 1985.
[17] A. Avizienis and L. Chen, "On the Implementation of N-version programming for software fault-tolerance during program execution," in *Proc. Int. Comput. Softw. Appl. Conf.*, 1977, pp. 149–155.
[18] L. Chen and A. Avizienis, "N-version programming: A fault tolerance approach to reliability of software operation," in *Proc. 8th Int. Symp. Fault-Tolerant Comput.*, 1978, pp. 3–9.
[19] J. Knight and N. Leveson, "An experimental evaluation of the assumption of independence in multi-version programming," in *Proc. IEEE Trans. Softw. Eng.*, vol. SE-12, no. 1. pp. 96–109, Jan. 1986.
[20] R. Martins, et al., "Experiences with fault–injection in a Byzantine fault-tolerant protocol," in *Middleware*, Berlin, Germany: Springer, 2013, pp. 41–61.
[21] F. Schneider and L. Zhou, "Distributed trust: Supporting fault-tolerance and attack-tolerance," Cornell University, Ithaca, NY, Rep. TR 2004–1924, Jan. 2004.
[22] T. Garfinkel and M. Rosenblum, "A virtual machine-based architecture for intrusion detection," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2003, pp. 191–206.
[23] F. Schneider, From Fault–Tolerance to Attack Tolerance, 2010. [Online]. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a548748.pdf
[24] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, pp. 189–204, 2007.
[25] S. Jajodia, et al., *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, vol. 54, Berlin, Germany: Springer, 2011.
[26] B. Cox, D. Evans, A. Fillipi, J. Rowanhill, and W. Hu, N-variant systems: A secretless framework for security through diversity, Fort Belvoir, VA, United States: Defense Technical Information Center, 2006.
[27] V. Rahli, N. Schiper, R. Van Renesse, M. Bickford, and R. Constable, "A diversified and correct-by-construction broadcast service," in *Proc. 20th IEEE Int. Conf. Netw. Protocols*, 2012, pp. 1–6.
[28] S. Jajodia, et al., *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, vol. 54, Chapter 8, Berlin, Germany: Springer, 2011.
[29] BFTSMaRT, 2015. [Online]. Available: https://code.google.com/p/bft-smart/wiki/GettingStarted
[30] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.
[31] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and counter measures," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2015, pp. 8–11.
[32] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMaRT," in *Proc. 44th Annu. IEEE/IFIP Int. Depend. Syst. Netw.*, 2014, pp. 355–362.
[33] OpenStack, 2014. [Online]. Available: OpenStack. https://www.openstack.org/
[34] J. Lau, L. Barreto, and J. D. Fraga, "An infrastructure based in virtualization for intrusion tolerant services," in *Proc. 19th IEEE Conf. Web Serv.*, 2012, pp. 170–177.
[35] A. Haeberlen, P. Kouznetsov, and P. Druschel, "The case for byzantine fault detection," in *Proc. 2nd Workshop Hot Topics Syst. Dependability*, 2006, p. 5.
[36] RackSpace, 2014. [Online]. Available: https://www.rackSpace.com/
[37] B. G. Chun, P. Maniatis, and S. Shenker, "Diverse replication for single-machine byzantine fault-tolerance," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 287–292.
[38] S. Bugiel, S. Nurnberger, A. Sadeghi, and T. Schneider, "Twin clouds: An architecture for secure cloud computing," in *Proc. Workshop Cryptogrsaphy Security Clouds*, 2011, pp. 32–34.

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

Bharat Bhargava is a professor of computer science with the Purdue University. His research work deals with the security and privacy issues in Service Oriented Architectures and Cloud Computing, and secure Internet-scale routing and mobile networks. He is the editor-in-chief of four journals and serves on over ten editorial boards of international journals. He is the founder of the IEEE Symposium on Reliable and Distributed Systems, IEEE conference on Digital Library, and the ACM Conference on Information and Knowledge Management. He is a fellow of the IEEE.