Mayflies: A Moving Target Defense Framework for Distributed Systems

Noor Ahmed
Air Force Research Laboratory/RI
525 Brooks RD
Rome, NY 13114. USA
norman.ahmed@us.af.mil

Bharat Bhargava
Purdue University
Computer Science Dept.
West Lafayette, IN 47906. USA
bbshail@purdue.edu

ABSTRACT

The traditional defensive security strategy for distributed systems is to prevent attackers from gaining control of the system using well established techniques such as; perimeter-based fire walls, redundancy and replications, and encryption. However, given sufficient time and resources, all these methods can be defeated. Moving Target Defense (MTD), is a defensive strategy that aims to reduce the need to continuously fight against attacks by disrupting attackers gain-loss balance. We present *Mayflies*, a bio-inspired generic MTD framework for distributed systems on virtualized cloud platforms. The framework enables systems designed to defend against attacks for their entire runtime to systems that avoid attacks in *time intervals*. We discuss the design, algorithms and the implementation of the framework prototype. We illustrate the prototype with a quorum-based Byzantime Fault Tolerant system and report the preliminary results.

Keywords

Cloud Computing; Software Defined Networks; OpenStack; Moving Target Defense; Byzantine Fault Tolerant

1. INTRODUCTION

While the techniques of constructing a resilient distributed systems for arbitrary system faults and crashes have been around for over decades, defending against sophisticated threats from intelligent adversaries still remains a challenging issue. Replication and redundancy have been the corner stone for achieving the faulty and crash resiliency. Because of the same code base across the replicas, adversaries with unbounded time need to exploit just a single replica and take control of the entire system due to the increased in the attack surface [18] – the set of ways/entries an adversary can exploit/penetrate the systems. The criticality of diversity as a defensive strategy in addition to replication/redundancy was first proposed in [12].

Diversity and randomization allow the system defender to deceive adversaries by continuously shifting the system's attack surface. Early works of replica diversification include: N-version programming [9] that proposed techniques to produce computationally variable binary forms of the same program. Running multiple variants of the same system in synchrony with a given input and monitoring for divergent is introduced in N-Varient Systems [11]. In general, the idea of proactive diversity is to periodically randomize replicas in the hope of reducing windows of vulnerability.

Modern sophisticated attacks, for instance, code injection attacks, target runtime execution exploits such as; buffer and heap overflows and control flow of the application (i.e., JIT-ROP), rather than the applications level code base. Diversification and randomization techniques fails to prevent these type of attacks. Techniques such as; Instruction Set Randomization [21], Address Space Randomization [14], randomizing runtime [10], and system calls [22] have been used to effectively combat against system-level *return-oriented* attacks. These techniques are considered mature and tightly integrated into some operating systems.

In general, all of the above defensive security mechanisms, both at the high-level (i.e., application/replica) and low-level (i.e., system level memory address layout), are known as Moving Target Defense (MTD) [17]. MTD promises new defensive strategies against modern sophisticated attacks by continuously shifting the systems attack surface to disrupt the attackers gain/loss balance. A unified generic MTD framework for distributed systems on virtualized cloud platforms have yet been addressed in the literature, thus, the focus of this work.

In this work, we propose *Mayflies*, a bio inspired MTD framework for distributed systems. Mayflies [23] is a short-lived (minutes to 24 hours) winged insects known in the literature as the *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 [24].

The basic idea of our framework is for a node (i.e., servers or replicas) to exist in short time intervals, participate in the overall computation, then vanish and appear 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.

We designed our *MTD* framework on top of a cloud framework. The framework emphasizes time (as lows a minute) and space diversification and randomization across heterogeneous cloud platforms (i.e., OS, Hypervisors).

MTD'16, October 24 2016, Vienna, Austria.
© 2016 ACM. ISBN 978-1-4503-4570-5/16/10…$15.00.
DOI: http://dx.doi.org/10.1145/2995272.2995283

The frameworks' chief defensive strategy is to increase the cost of an attack of the system, and lower the likelihood of successful attacks. We achieve this by 1) controlling the systems runtime execution in *time intervals* through node *reincarnations*, and 2) pro-actively monitoring attacks even in face of OS compromise with *Virtual Introspection*. Thus, our main contribution is as follows:

1. We developed a generic MTD framework that enables time and space system diversification on virtualized cloud platforms without changes to the system.
2. We introduced an abstraction scheme to separate the application runtime with the virtual instance, dubbed, *Time Interval Runtime Execution* model.
3. We introduced node reincarnation algorithm for system agility in order to reduce the exposure window of attack.

We have organized the paper as follows; we first give the threat model and assumptions in section 2. We then discuss the framework design, algorithms and the prototype implementation in section 3 followed by the evaluations in sections 4. Finally, discuss the related work in section 5 and the conclusion in sections 6.

2. 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 at any time. 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.

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, the adversary can employ arbitrary attacks on the node (s). We do not consider attacks that targets the networking fabric of the compute instances *(i.e., SDN)* that can disrupt the routing table [15].

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.

3. FRAMEWORK DESIGN, ALGORITHMS AND IMPLEMENTATION

3.1 Design

Mayflies adopts a cross-vertical design that operate on three different logical layers of the cloud framework; the *compute* at the application layer (GuestOS layer), the *VMI* at the hypervisor layer (HostOS layer), and at the networking layer (SDN). In this section, we describe the architecture, then discuss the core components of the framework. We built *Mayflies* framework on top of Openstack cloud framework [20], a widely adopted open source cloud management software stack. Figure 1 shows the high-level architecture of *Mayflies* framework on top right and Openstack cloud framework on the left.

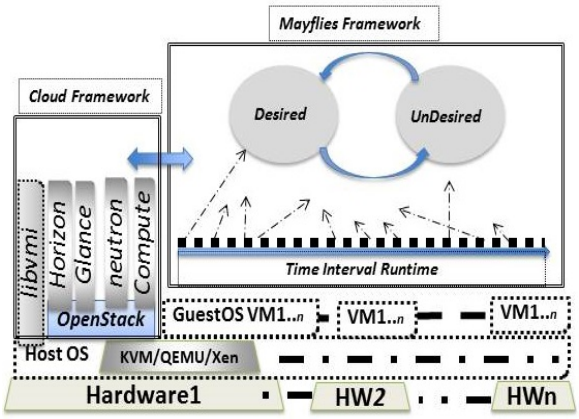

Figure 1: High level Mayflies architecture – Mayflies Framework (top right) and the Cloud Framework (bottom and top left)

In the cloud framework, starting from the infrastructure at the bottom layer lie the hardware (Hardware1 ... HWn). Each hardware has a host OS, a hypervisor (KVM/Xen) to virtualize the guest VMs using the cloud management 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* [1], a library for virtual introspection. This library interacts with the hypervisor to intercept the resource mapping request (i.e., memory) from the VMs for the physical available resource, thereby, allowing to detect anomalous behaviour even in the event of VM/OS compromise.

In *Mayfilies* framework, we introduce two abstraction layers; a high-level *System State* (top) and the *Time Interval Runtime* (bottom). 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 the high-level states are the observations collected at the *time-interval runtime* using *libvmi* depicted as the dotted arrows.

3.2 Algorithms

Algorithm 1 illustrates *Mayflies* prototype. In Algorithm 1, we simply choose an strategy (i.e., round-robin, random) to refresh nodes accordingly. The two core procedures of the algorithm are INTROSPECT() and REINCARNATE() to perform the *Proactive Node Monitoring* and the *Node Reincarnation* respectively.

3.2.1 Proactive Node Monitoring

For proactive node monitoring, we leveraged *libvmi* [1], an open source implementation of Virtual Machine Intro-

Algorithm 1 Mayflies Algorithm

```
1: Initialize replicas n and time-interval x/lifespan
2: while true do
3:     if strategy = RoundRobin then
4:         repeat
5:             targetNode ← GetNode()
6:             isClean ← Introspect(targetNode)
7:             if isClean then
8:                 targetNode ← GetNode()
9:                 Reincarnate(targetNode)        ▷ scheduled
10:            else
11:                Reincarnate(targetNode)        ▷ dirty
12:            end if
13:            wait(x)                            ▷ sleep for x minutes
14:        until no more replicas left
15:    end if                                     ▷ can be randomized next
16: end while
```

spection to monitor host VMs below th hypervisor. In INTROSPECT() procedure, we first take a snapshot of the process's live memory before going online, then, we compare key/value pairs of the subsequent snapshots. The result can be either *true* if anomaly is detected in the memory structure, otherwise *false*. We gave a detailed attacks and detection schemes with *libvmi* in our previous work [5].

3.2.2 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 *nova compute* is designed for provisioning and de-provisioning VM instances on the cloud platforms. The main idea is to continuously provisioning/de-provisioning (spawn/delete VM instance) nodes in time intervals at run time, dubbed *Node Reincarnation*.

Algorithm 2 Node Reincarnation Procedure

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

```
1: procedure Reincarnate()
2:     nodeState ← targetNode_state
3:     DestroyTarget()
4:     newNode ←GetNewNode()
5:     if nodeState_interface == NULL then
6:         newNode_netInterface ← NewNetInterface()
7:     else
8:         newNode_netInterface ← nodeState_netInterface
9:     end if
10:    newNode_state ← nodeState
11: end procedure
```

The two fundamental problems of reincarnating/refreshing nodes are dealing with the network dynamics and the application state transfers between the terminating node and the newly activated node to take over it's role, discussed next. The network dynamics is achieved by simply swapping the network interfaces at runtime as described in details in the implementation section.

3.2.3 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. We assume most of the applications are designed to reconnect when a connection is lost for very short period of time, typically, within 8-12 seconds which is the reincarnation time of *Mayflies*.

In most applications, there is a configuration file(s) which is typically saved in a file `(system.config` or `hosts)` that is static. 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, and security keys/certificates, We simply inject these configuration files before we start the application, however, the dynamic state information poses a challenge.

Devising a generic method for managing the dynamic part of the application state for *Mayflies* is not feasible since it's application dependent. For instance, the quorum-based Byzantine Fault-Tolerant system discussed in the experiment section, the recovering node requests the updated system state (current transaction number) from the quorum. We consider evaluating the framework with other applications with complex state information (i.e., sessions, security tokens, etc.) in our future work.

3.3 Implementation

We implemented our algorithms with *bash* shell scripts tightly integrated into the OpenStack (Kilo) [3] framework, an open source cloud management software stack. As noted earlier int the 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.

There are different ways to implement node reincarnation in OpenStack. 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 can be booted on-demand. Another approach is to select from a pool of prepared VMs without network interface or with a temporary interface while their successor frees the interface known to the client or the services. Once the node reaches its lifespan, it signals to be terminated and the two interfaces are swapped using `nova interface-detach <newReplica portID>` to remove the interface, and `nova interface-attach <portID newReplica>` to re-attach the old interface to the new VM. At this point, we inject the application state and other configuration files.

For the node created without network 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 reincar-

nate nodes across subsets and networks, an MTD scheme known as *IP Hopping*.

4. EVALUATION

We evaluate *Mayflies* with BFTSMaRT [6], a widely studied quorum-based Byzantine Fault Tolerant system prototype in the literature. The main reason we selected this system is that it requires n nodes to be running in synchrony in order for the quorum to be reached. We deploy BFT-SMaRT' s *CounterServer()* and *CounterClients()* demo application downloaded from [2] 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.

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 and throughput on *Mayflies*. This given us the lower bounds of our defensive cost, *reincarnation* and *observation* (virtual introspection) costs in each time interval t while preserving the reliability properties of the deployed system. We consider evaluating our defensive cost relative to attack success rate in our future work.

4.1 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 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. We used Ubuntu 14.04 for the client and 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.

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.

4.2 Preliminary Results

Figure 2 reflect the graphical *horizon* dashboard of the network topology in OpenStack, and Figure 3 shows the topology result after the transformation algorithm completes for one round. *(x_Ry)* denotes node ID x operating in round y *(Ry)*.

Figures 4 and 5 reflect the underlying node status and the network interfaces swaps using `nova list` command. Note the *(x_R0)* with IP *(fix and floating)* addresses and *(x_R1)* with blank entries as shown with the white square box and the pointing arrow in *(0_R0)* and *(0_R1)*. Figure 8 shows *(x_R0)* and *(x_R1)* interfaces are swapped while the replicas serving clients. This transformation can continue as (*x_R2*) for round 2 with newer VMs and so on. This transformation took only 8-12 seconds per replica.

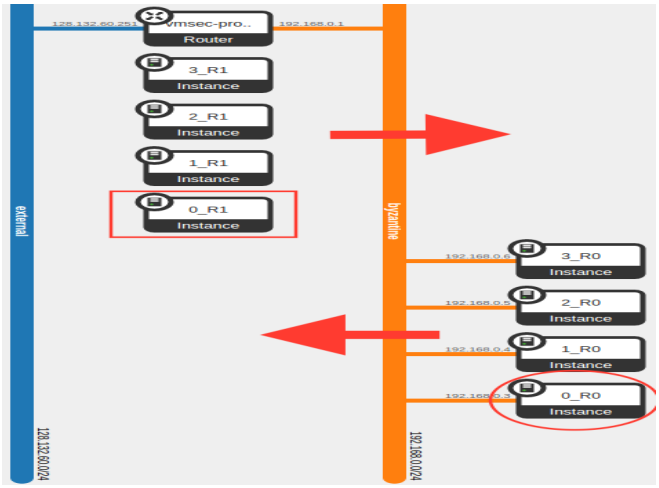

Figure 2: Horizon Dashboard view of the 4 *BFT* replicas *0_R0 ... 3_R0* on the right vertical bar (*byzantine* subnet) and 4 isolated standby replica pool *0_R1 ... 3_R1* between the vertical bars/subnets. A virtual router *(vmsec-proj)* interconnects the two subnets with *192.x.x.x* IP on the *byzantine* subnet side and *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.

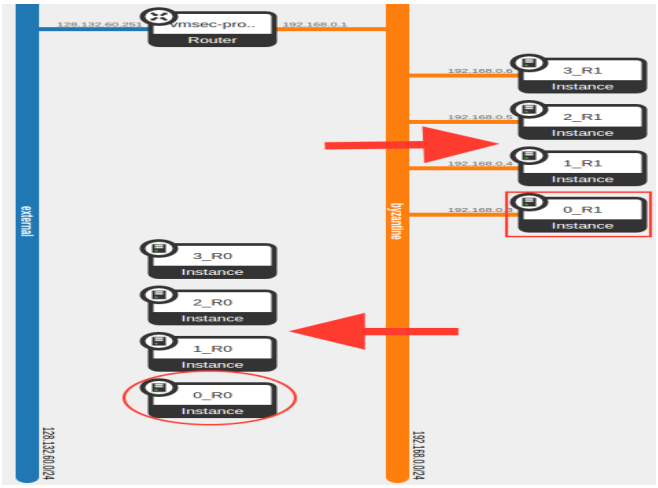

Figure 3: The result after the replicas refreshed, all *x_R0* replicas are removed from the subnet *byzantine* and replaced with the standby replica from the pool *x_R1* (rectangle box to the oval box) while serving clients.

Table 1 show the results of the normal server restarts using the scripts `smartrun.sh` and `killall.sh` included in the demo. This is to capture the recovery protocol timing in terms of transition gaps and the overall process time without MTD-enabled. 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 of the messages completed by the terminating server and the restarted same server.

Table 2 show the results of the demo servers while on the move across platforms with the same settings above (i.e.,

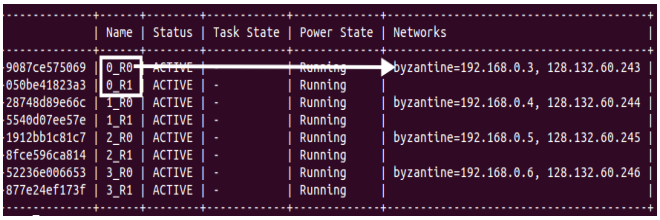

Figure 4: An output of `nova list` command showing the BFT and the standby replicas with their network mappings before the transformation. The arrow shows the (x_R0) replica groups have network interfaces and x_R1 have none.

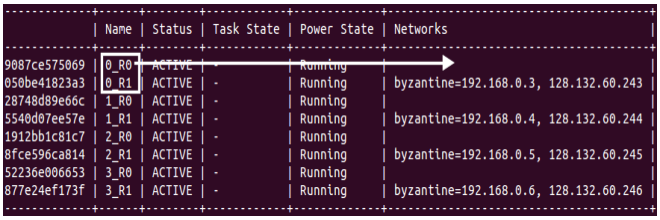

Figure 5: Post BFT transformation results. The network interfaces of (x_R0) group are seamlessly transferred to the (x_R1) group. Note (x_R0) entries are blank.

100K clint 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 [6].

Table 3 shows the average lapse time of the *Generic* case were we start the servers without stopping, *Re-Starts* and *MTD-enabled Refresh* as reported in Tables 1 and 2. Column 1 is the use case scenario names. Column 2 show the average lapse time, the time it takes for normal situation, naturally recovering if restarted, and *MTD-enabled Refreshed*. Column 3 show the total process time of the 100K requests averaged across 5 experiments. It takes a little over 3 minutes to process a 100K client requests in the *Generic* case and little over 5 minutes when MTD-enabled in every 20K transaction/client requests completion.

This preliminary results illustrate that we can refresh a replica in *Mayflies* as low as a minute with proactive monitoring while performing useful computation as well as preserving the reliability properties of the deployed systems protocol *(safety and liveness)*. These experiments was limited to a single client and 4 replicas. We consider evaluating the system with large number of clients and servers and assess the portion of time the system can be in *Desired* state while in constant attack. Further, we consider evaluating applications with complex state information.

5. RELATED WORK

To the best of our knowledge, *Mayflies* is the first generic MTD framework for distributed systems. However, the ideas employed, specially, the model and the proactive monitoring with virtual introspection schemes, are not unique to the framework. For example, a game theocratic approach is introduced in Flipit [7] 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 [19], 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 [8]. Along the same lines of our cloud-based framework, a network focussed MTD architecture is introduced in [16] 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 [13] was the first VMI-based architecture for intrusion detection for the integrity of user programs in Linux, i.e., sshd, syslogd, etc. Early works include protecting Unix program such as; ls, ps, etc. Modcheker, an OS Windows kernel module integrity violation detection approach using VMI [4]. Monitoring system calls between the VMs and its host kernel using HMM to classify VM behaviour is proposed in [15]. 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.

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: MTD-enabled/Refresh Counter Demo

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 MTD-enabled/Refresh

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

6. CONCLUSIONS

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

hardware, OSs) in time intervals (as low as a minute). We formally modelled and discussed the core components of the framework and the performance impact of BFT. In our future work, we consider improving the performance of the virtual introspection capabilities for the proactive monitoring, and extending the node reincarnation with *IP-hopping*, reincarnating nodes across subnet or networks while we assess the proportion of the time the system being in a *desired* or *undesired/compromised* states.

Acknowledgments

The first author would like to sincerely thank Dr. Mark Linderman, Jim Hanna, Steven Farr and Lt. Col. Mike Halick at AFRL for their continuous guidance and support.

7. REFERENCES

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

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

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

[4] 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)*, pages 306–313. IEEE, 2012.

[5] N. Ahmed and B. Bhargava. Towards targeted intrusion detection deployments in cloud computing. *International Journal of Next-Generation Computing*, 6(2), 2015.

[6] 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*, pages 355–362. IEEE, 2014.

[7] 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*, pages 248–263. Springer, 2012.

[8] 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*, page 38. ACM, 2013.

[9] 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*, pages 3–9, 1978.

[10] 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*, pages 50–61. ACM, 2016.

[11] 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*, volume 6, pages 105–120, 2006.

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

[13] 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)*, volume 3, pages 191–206, 2003.

[14] 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)*, pages 475–490, 2012.

[15] 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.

[16] 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*, pages 127–132. ACM, 2012.

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

[18] P. K. Manadhata and J. M. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371–386, 2011.

[19] 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.

[20] Openstack.org. OpenStack cloud management framework, 2014.

[21] 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*, pages 41–48. ACM, 2010.

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

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

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