Combating the Bandits in the Cloud: A Moving Target Defense Approach

Terry Penner
Department of Computer Science
Texas State University
t_p68@txstate.edu

Mina Guirguis
Department of Computer Science
Texas State University
msg@txstate.edu

***Abstract*— Security and privacy in cloud computing are critical components for various organizations that depend on the cloud in their daily operations. Customers' data and the organizations' proprietary information have been subject to various attacks in the past. In this paper, we develop a set of Moving Target Defense (MTD) strategies that randomize the location of the Virtual Machines (VMs) to harden the cloud against a class of Multi-Armed Bandit (MAB) policy-based attacks. These attack policies capture the behavior of adversaries that seek to *explore* the allocation of VMs in the cloud and *exploit* the ones that provide the highest rewards (e.g., access to critical datasets, ability to observe credit card transactions, etc). We assess through simulation experiments the performance of our MTD strategies, showing that they can make MAB policy-based attacks no more effective than random attack policies. Additionally, we show the effects of critical parameters – such as discount factors, the time between randomizing the locations of the VMs and variance in the rewards obtained – on the performance of our defenses. We validate our results through simulations and a real OpenStack system implementation in our lab to assess migration times and down times under different system loads.**

I. Introduction

The cloud has been a major target of cyber attacks owing to the ever-increasing reliance on it by industry, private, and public sectors in their day-to-day operations. Attackers target the cloud to seek unauthorized access to sensitive and private data and/or intellectual property, or to render some functionality of the cloud unusable for legitimate users. While virtualization seeks to isolate virtual machines (VMs) from each other, recent attacks have shown that they can bypass such isolation [1], [2]. Moreover, due to their open-access nature, attackers can dynamically create their own VMs with malicious code that can target the host machines or the network of the cloud provider.

Due to the dynamic nature of the cloud, attackers may not necessarily know a priori which physical machine they will be hosted on nor the nature of other VMs collocated on that same machine. Such decisions are often left to the provider (subject to resource constraints enforced by the user) and typically are the result of various resource optimization problems. This means that rational attackers would likely try to *explore* the cloud environment seeking particular physical and virtual machines to *exploit*. Thus, in this paper, we model the behavior of the attacker as in a Multi-Armed Bandit (MAB) problem. In a MAB problem, the bandit is presented with K slot machines and in each turn, he/she chooses a slot machine in order to maximize their reward. Initially, the bandit tries to explore the rewards obtained from all of the machines and then exploit the high paying ones. We believe that this adversarial model is critical in the cloud security domain as it reflects four important components: (1) initially, the attacker does not have a lot of knowledge when their first VM is created, (2) the ability of the attacker to create more opportunities to potentially explore other physical machines, (3) other VMs will be migrated by the reallocation strategies of the provider, enabling the attacker to explore them (and explore other physical machines once their own VMs are migrated), and (4) the attacker can trigger a move by manipulating their own resource constraints until they observe a change or alternatively, exploit a placement vulnerability through manipulating the cloud provider into creating their VM on the same physical node that hosts the VM they want to target [3].

As a simple attack scenario, consider an attacker who has loaded malicious code onto their VM and used a virtualization vulnerability to access information that belongs to other VMs on the same physical node [1]. The attacker can snoop the memory for sensitive information and as other VMs are migrated into this physical machine (or the attacker's VM is migrated to another physical machine) the attacker can continue doing so until they find the critical information.

Moving Target Defense (MTD) strategies has been proposed to allow randomization to harden the system against attackers [4]. The idea is that the defender makes some changes to its system's configuration every so often (e.g., migrating VMs) to make it harder for the attacker to succeed. In this paper, we develop MTD strategies and assess their effectiveness against attackers using a wide range of MAB-based policies. We show that our defense can effectively waste the attacker's effort – making it no better than a random attack in which no knowledge is exploited. While it may not completely prevent the attacker from achieving some small successes, it will greatly reduce the potential damage that can be caused. This will help protect the users' sensitive and intellectual property information in enterprise cloud environments.

In this paper we make the following contributions:

1) Develop a set of MTD strategies that introduce randomization to counter MAB policy-based attacks.
2) Assess the impact of our defenses against a variety of MAB algorithms and show that it can make them no

more effective than a randomized attack policy.
3) Study the effect of critical parameters (e.g., time to switch VMs, variance in rewards, reward saturation, etc) on the performance of our defense.
4) Validate our mechanisms using a real OpenStack system to collect data on migration times and VM down times under different system loads.

Paper organization: In Section II we discuss related work. In Section III we present the MTD strategies against MAB policy-based attacks. We present our results from simulations and real system implementation in Section IV and conclude the paper in Section V.

II. Related Work

This work relates to the following three main areas of research:

Security in Cloud Computing: The authors in [5] provide a detailed list of security flows that can occur in the cloud from the perspective of a software developer and the work in [6] lists seven major ways cloud services can be at risk from the perspective of a business executive, giving recommendations for how companies can prepare to leverage cloud services. The work in [7] provides several techniques to protect data through encryption while the work in [8] describes a new architecture for cloud systems designed with security as the primary concern. The authors in [2] proposed a migration based method for detecting and avoiding Denial of Service attacks in a cloud environment. The main difference is that their solution is to be implemented by the clients running on the VM on the cloud, whereas our solution is built into the cloud system itself. Similarly, the authors of [9] propose a system for cloud defense based on the actual migration of VMs. While it shares many similarities with our defensive strategy, our contributions are not the same. They focus exclusively on preventing Denial of Service attacks, while we show the effectiveness against other types of attacks as well, such as packet sniffing and memory snooping. In addition, their evaluation was conducted using PlanetLab, a large scale worldwide network research environment. We performed our evaluation on a deployment of OpenStack, a cloud system that is currently used by actual cloud provider companies, such as HP, IBM, and Oracle.

Our solution addresses a few of the security concerns listed in the above papers, mostly focusing on the Network Security flaw from [5] and the Malicious Insiders flaw from [6]. Our MTD strategy can be used in tandem with encryption for extra protection, and does not require a redesign of the existing cloud, so it could be implemented efficiently.

Multi-Armed Bandit: The standard version of the MAB problem that we will define in Section III-A was described in [10]. Many variants of the problem have been created over the years, which modify the process that determines how the arms give rewards. There are two main versions of the MAB problem based on how its rewards are generated: stochastic and non-stochastic. In the stochastic problem, the rewards are generated based on some logical process, such as a probability distribution, while in the non-stochastic problem there may not be any logic to the choice of reward values. The traditional stochastic MAB problem simply uses a probability distribution – as we consider here – but other ideas have been proposed, such as the one by [11], where each arm's rewards are given by a Markov Decision Process (MDP). Whenever an arm is pulled, it gives some reward and causes the MDP to transition to the next state. A further modification of this version is called Restless Bandits [12], where all the arms transition state every turn, not just the arm that was pulled. There are also MAB variants that, like our work, modify the state of the game over time. In [13], they define a problem where more arms appear over time, growing the number of choices the gambler is presented with. In [14], they define a variant where arms have a lifespan and will "die" after a number of turns, to be replaced by a completely new arm. Many solutions to the stochastic problem have been proposed over the years. One of the first popular ones was an optimal policy called the Gittins Index [15]. In more recent years, the UCB algorithm from [16] has been a standard, forming the base of many other variations. In Section III, we will study those solutions in more details.

Some of the MAB variants that modify the game state (e.g., [14]) are close to our strategy. There are differences between their work and ours however, such as how they model their scenario as a non-stochastic problem, while we have intentionally avoided doing that. In addition, when they talk about an arm "dying", they mean that its reward distribution is replaced with a completely new one. In our system, rather than replace old distributions we move them around, so that our system remains constant aside from the mapping of reward distributions to arms.

Moving Target Defense: The work in [4] gives detailed information about the definition of a MTD, various MTD strategies, and the general effectiveness of these strategies against different classes of attacks and exploits. Instead of the comprehensive overview that they gave, this paper delves into the effectiveness of MTD strategies against the specific MAB policy-based attacks. A formalized theory of MTD systems is laid out in [17]. The authors in [18] give an overview of several different types of MTD strategies and compare how they perform under different attacking scenarios. In their work, they study migration based defense strategies against an attacker that acts on a genetic algorithm, rather than a MAB policy. In [19], the authors look at the effectiveness of a network based defense. They adapt their system through the complete refresh of the VMs, where all prior state information is lost. In our scenario, we are simply migrating the VM from a node to another with no information lost and almost zero expected downtime. They are also using a configuration manager component to decide when to refresh the VMs, whereas we are migrating them at random.

III. Methodology

A. Multi-Armed Bandit (MAB)

In a MAB problem, the gambler is presented with K slot machines to choose between over a number of turns T (the

horizon). In turn $t \in \{1, ..., T\}$, the gambler selects one of the slot machines $m \in \{1, ..., K\}$, and pulls its arm receiving some reward $r_{m,t}$. The reward obtained from slot machine m is chosen based on some probability distribution D_m that is not known to the gambler a priori and is independent from the distributions of the other slot machines. The goal is to maximize the aggregate reward R over the T turns through choosing m at each turn. The aggregate reward is given by:

$$R = \sum_{t=1}^{T} r_{m,t}. \tag{1}$$

To maximize the rewards obtained, the gambler needs to create a policy that selects which m to pull in each turn. Because the gambler has no foreknowledge of how the machines give their rewards, the gambler must explore the arms before choosing which one(s) to exploit. Policies, in general, are designed to choose when to explore and when to exploit based on the previous rewards earned.

Policies are not evaluated in terms of maximizing rewards, but instead in terms of minimizing the regret ρ. Regret is defined as the cumulative total of the difference between the optimal arm and the arm that was actually pulled by the policy. Because the arms can give rewards with some variance, regret is calculated using the expected rewards [20]. We let μ_m denote the expected reward of pulling the arm of machine m and μ^* denote the highest expected reward (i.e., $\mu^* = \max_{1 \leq m \leq K} \mu_m$. We define regret, ρ, after T turns as:

$$\rho = T * \mu^* - \sum_{m=1}^{K} \mu_m * P_m(T), \tag{2}$$

where $P_m(t)$ is the number of times machine m's arm has been pulled by time t. Equation 2 can be reformulated in terms of T, to yield:

$$\rho = \sum_{t=1}^{T} (\mu^* - \mu_{m_t}), \tag{3}$$

where m_t is the index of the arm that was pulled at time t.

In many situations, it becomes important to value earlier rewards higher – specially from an adversary's standpoint that wishes to minimize the time they spend in the system. Equations 1 and 3 can be stated with a discount factor, γ : $0 < \gamma < 1$, [21] as follows:

$$R = \sum_{t=1}^{T} \gamma^t * r_{m,t} \tag{4}$$

$$\rho = \sum_{t=1}^{T} \gamma^t * (\mu^* - \mu_{m_t}). \tag{5}$$

B. Attack Policies as Solutions to the MAB Problem

Many policies can be applied to solve the MAB problem. To capture a wide range of possible attack policies, we present the following list of policies that vary in the way they tradeoff between exploration and exploitation. The full details of each policy can be found in the respective references, but here we give a brief description of each method.

- **Upper Confidence Bounds (UCB)**: This policy is one of the most basic ones [16]. It pulls every arm once, and then it chooses the arm that maximizes:

$$\bar{\mu}_m + \sqrt{\frac{c * \log t}{P_m(t)}} \tag{6}$$

  where $\bar{\mu}_m$ is the current sample expected reward, c is a positive constant, t is the current turn number, and $P_m(t)$ is the number of times arm m has been pulled at time t. The $\log t$ term ensures a non-decreasing sequence of values that are an order of magnitude below t, which is what allows it to explore again over time if the rewards received are not very large.
- **UCB-Tuned**: This policy is from [16], and is a variant of the UCB policy. The upper confidence bound is modified so that instead of the constant c, a variable based on the the previous variance of the arm is used. This allows it to function the same way, but hopefully make choices at a smarter time.
- **UCB-V**: A straightforward modification of the UCB policy is to account for variance [22]. It chooses the arm that maximizes:

$$\bar{\mu}_m + \sqrt{\frac{2 * \log t * \bar{v}_m}{s}} + c * \frac{3 * b * logt}{s} \tag{7}$$

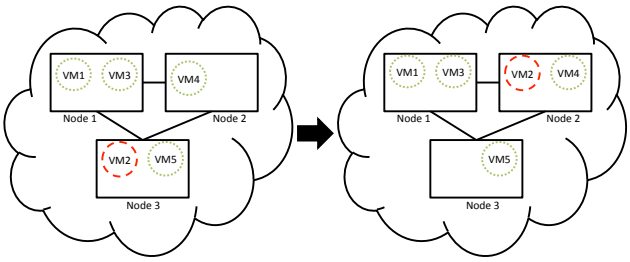
  where $\bar{v}_m$ is the current sample variance, c and s are constant positive numbers (s usually is $P_m(t)$), and b is the bound on the rewards. It adds in the information that it knows about the bound on the rewards to try and fine-tune which arm is the most likely to pay out well.
- **KL-UCB**: This policy is from [23]. It always selects the arm with the maximizes:

$$P_m(t) * BKLD(\bar{\mu}_m, \log t + c * \log \log t) \tag{8}$$

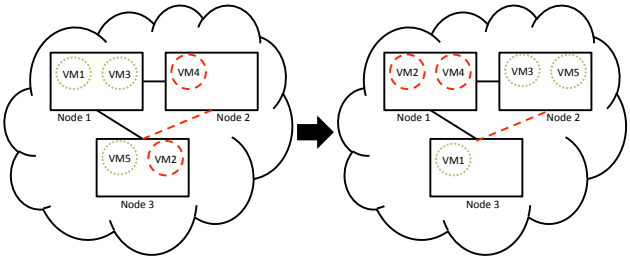
  where $BKLD$ is the Bernoulli Kullback-Leibler divergence – a measure of information gain – so it tries to select the arm with the most likely gain.
- **MOSS (Minimax Optimal Strategy in the Stochastic case)**: This policy from [24] selects the arm that maximizes:

$$\bar{\mu}_m + \sqrt{\frac{\max(\log(\frac{T}{P_m(t) * m}), 0)}{P_m(t)}} \tag{9}$$

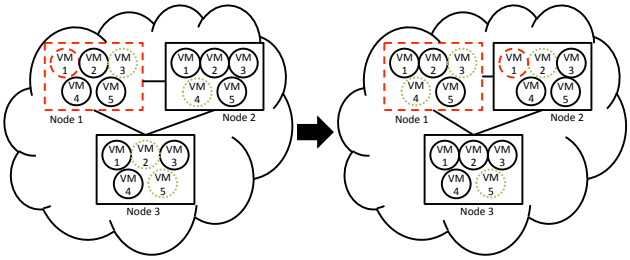
  where T is the horizon. It is inspired by the UCB algorithm, and it looks for the arm with the highest upper confidence bound.
- **Empirical Likelihood UCB**: This policy is from [25]. It is a variation on the KLUCB policy, in which the goal is to try to pick the arm with the greatest information gain.
- **KL-UCB-exp**: This policy is found in [23] and is a variation of the KL-UCB. It uses a divergence that expects an exponentially distributed reward input, but



(a) Credit Card Sniffing



(b) Network Mapping / Flood



(c) Memory Snooping

Fig. 1. Moving Target Defense Scenarios

has the same goal of selecting the arm with the most likely gain.

- **Random**: This is a simple policy that we added as a baseline. Instead of making any complicated choices, it simply picks an arm at random from the ones currently available to it.

C. Moving Target Defense (MTD)

A MTD is a defensive strategy applied to a configurable system with the goal of adding some randomness to invalidate any knowledge that a potential attacker could have gained over time. A configurable system Γ consists of a set of states S, a set of actions Λ, and a transition function τ that maps $S \times \Lambda \rightarrow S$. A state $s_i \in S$ is a unique system setting, and an action $\alpha \in \Lambda$ is a set of steps that will change one state into another valid state. A MTD system Σ is thus defined as a configurable system Γ, a set of goals G (including goals for both the system's proper operation g_o, and its security g_s), and a set of policies P (rules for what constitutes a valid system configuration). The set of all valid states S_v is referred to as the configuration space, and a MTD aims to make things more difficult for an attacker by moving the current state throughout the configuration space. For a more thorough definition, see [17].

To implement a MTD strategy, two essential components must be added – an adaptation engine and a configuration manager [19]. The adaptation engine decides what changes should be made to the system, and how often they should be made. The configuration manager makes and enforces the changes. If desired, an additional analysis engine component can be added that feeds current system information into the adaptation engine to help make more informed decisions.

To apply the formal definition to our cloud system, S_i describes a mapping of each VM to a physical node with S being the set of all possible permutations of the mapping. α is the migration commands to move a VM from one node to another, with Λ being the set of all such possible migrations. The transition function τ would be where the details of a particular strategy would be encoded. The set of goals G would include things like g_{o1}, "allow customer access to VM," and g_{s1}, "prevent customer traffic from being intercepted." Finally, the set of policies P would include rules/constraints such as p_1: "The sum of the disk space required by all VM on a node must not exceed the disk space of that node." With this, it is easy to see that a cloud environment works well as a configurable system.

We developed three MTD strategies for use with our system. The first one we call **Complete Restructure**, because it has the goal of changing the location of every single VM in the system. In this strategy, the transition function τ would consist of only tuples $(s_i, a_k) \rightarrow s_j$ that result in a new configuration where none of the VMs in s_j are located on the same physical node as they were in s_i.

We also use a more relaxed version which we denote **Hide Max**, where we only migrate the VM that rewards the attacker the highest (e.g., the critical VM the processes credit card transactions), assuming it is known to the defender. The transition function τ would consist of only tuples $(s_i, a_k) \rightarrow s_j$ that result in a new configuration where the only change is that the maximum rewarding VM has swapped locations with another VM.

Our third strategy is denoted **Duplicate and Deactivate**, because it keeps a copy of every VM on every node, and deactivates all but one of each at any given time. In this case, the transition function τ would consist of only tuples $(s_i, a_k) \rightarrow s_j$ that result in a new configuration where every VM in s_j is listed only once along with the node it is activated on.

Since the adaptation engine is responsible for deciding when to trigger the defense, we set it as a fixed interval that we can manually vary to assess its effect on the performance. The configuration manager is responsible for initiating the migrations, which is a tool that is already provided by most cloud systems.

D. Sample Attacks and Defenses

For illustration purposes, we consider the following three MAB attack scenarios along with the MTD strategies to combat them.

Scenario I: Consider an attacker that has loaded malicious code directly onto a physical node in the cloud – an attack that can be done through a VM Escape exploit [1]. This would allow the attacker to sniff for packets transmitted

and received and identify other nodes that he/she can target (e.g., the node that is generating traffic with credit card transactions). The MTD strategy would change the layout of the system and change which packets the attacker can see, since the VM it was sniffing the packets of will no longer be located on the same physical node. This will then invalidate all of the knowledge that the attacker has gained about where the packets with the credit card information are located. Additionally, because the attacker doesn't know that the VM have been moved, it may not equate the lack of packets with a move for quite a while. Figure 1 (top) illustrates the outcomes of this MTD strategy in a setting in which the target VM – marked as a red dashed VM – has been migrated to a different physical machine, so the attacker cannot sniff its packets anymore.

Scenario II: In this scenario, we consider an attacker who has placed their code on multiple VMs that they have legally created. They can send traffic between those VMs – which may or may not be located on the same physical node – in an attempt to map the underlying physical network topology. Once this is achieved, the attacker can determine the bottleneck links and attack them – even from outside the cloud – to degrade the performance of the entire cloud [1]. A process for carrying out this type of attack is given in [2]. The MTD can once again create a moving target by migrating the VM between hosts every so often. In this scenario, the attacker's knowledge is invalidated by the fact that it itself was moved. The new virtual network structure would be completely different; connections that formerly went over physical links could be on the same physical node now, resulting in no congestion at all. Figure 1 (middle) illustrates the outcomes of a possible MTD strategy in which the defense prevented the attacker from inferring the physical links since both VMs are migrated to the same machine.

Scenario III: Here, the attacker has managed to load malicious code onto their VM and use a virtualization vulnerability to access information that belong to other VMs on the same host (e.g., a VM monitoring attack [1]). Unlike scenario I, the attacker does not have full access to the cloud's network; instead, they have full access to the physical node's memory. They can snoop around the live memory of the other VMs, looking for sensitive information to steal from the currently running processes. The idea of this MTD strategy is to create a copy of every VM on multiple hosts, but have all of them suspended, except for one. This means that the attacker can see all of the VMs even without access to the network. However, they cannot see all of the information on currently running processes, only the information for the VMs active on that node. The MTD strategy is to change which node each VM is currently active on, thus changing the processes that can be snooped on. Unlike scenario I, the attacker does not have control of the network, so it cannot follow the VM when it moves. Figure 1 (bottom) illustrates this case, where the green dotted VMs are the active ones, the black solid VMs are the suspended ones, the red dashed VMs is the one the attacker is looking for, and the red dashed node is the one the attacker is located on.

IV. Evaluation

In this section, we report of the effectiveness of our MTD strategies against MAB attack policies through simulation experiments as well as implementation experiments with OpenStack in our lab.

A. The setup

To study a good range of attack policies, we used the open source maBandits package [25] as a base and modified the implementation to run our MTD strategies. The maBandits package assumes a finite horizon and takes advantage of that to pre-calculate all reward payouts for each of the arms at every step, if they are chosen at that point. The program then proceeds to test each policy in order, letting it choose which arm to pull for each turn, and returning the appropriate reward from the table it has pre-calculated. The policy then updates its internal state and comes back for the next turn. The program continues the game this way until the horizon is reached. It then starts a new game with the same policy and repeats it all, doing this for every policy. Once this has completed, it averages the results from each policy in terms of regret and how many times the attacker chose a sub-optimal arm to pull. The full details and original source are available from [25].

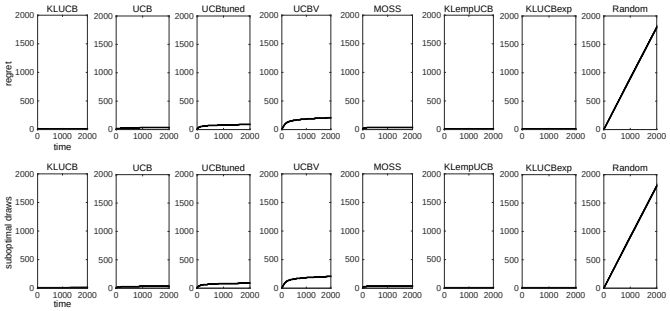
We simulated our MTD strategies through an adaptation engine that, when triggered, would swap the rewards obtained by the MAB policy. This would cause the distribution and reward payout of each arm to swap as well. The adaptation engine is triggered at a set interval which we varied manually to assess the impact of the swap frequency on the affect of the MAB policies.

In our experiments, we set the horizon to 2000 turns and results are averaged over 20 independent runs. We ran tests with 10 arms with normalized rewards in which the sum of the expected rewards for all the arms was equal to 1.

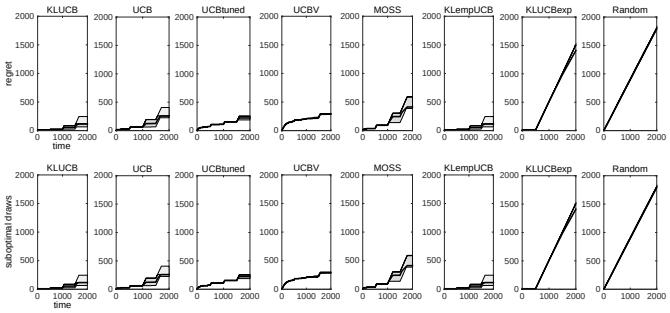
B. Simulation experiments

As a base case to see the impact of our defense strategies, we removed the variance from the rewards and computed the regret as in Equation 3. Figure 2 shows the results when we had one machine pay a reward of 1 while the other ones paid 0, and we implemented our **Hide Max** MTD to hide that machine. The results in Figure 2 show the effect of no defense (top) in comparison to the defense being triggered every 500 turns, 50 and 5, turns, respectively. One can see that as we increase the frequency of our swapping defense, the effectiveness of the attack strategies all decrease dramatically (captured by the increase in the regret). There is a noticeable change between shuffling every 500 and every 5 turns, at which point the attacks are nearly indistinguishable from a random strategy.

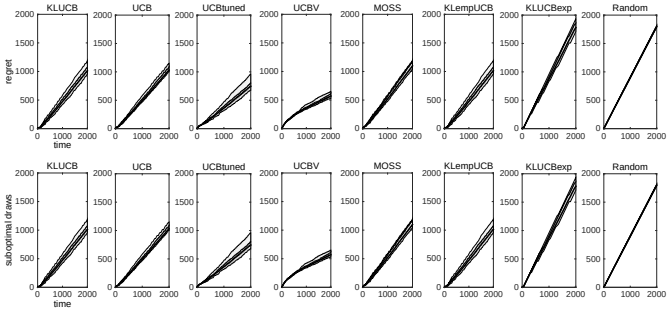
Figure 3 shows the effect of the swapping frequency on the average regret under each policy at the end of the game. With a MTD applied every 50 turns, the regret has the highest decrease and from the attacker's perspective, the more frequently the defense is activated, the more similar



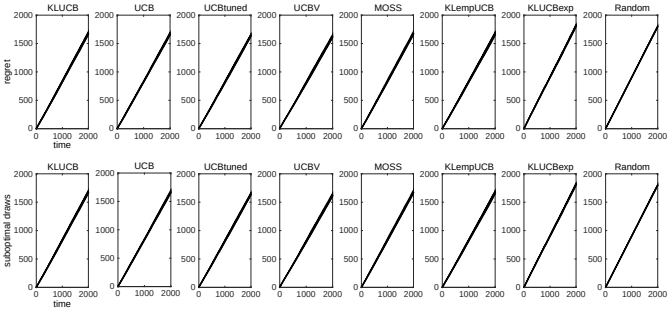
(a) No Defense



(b) Defense Every 500



(c) Defense Every 50



(d) Defense Every 5

Fig. 2. Hide Max; No Variance; No Discount

the attack policies' performances become, with all of them approaching the performance of the random strategy.

In our next set of experiments, we consider the effect of the discount factor γ as in Equation 5 where the attacker is on a time sensitive schedule and must collect rewards as near the start of the game as possible. An example of this would be an attacker that is being actively monitored by a security program, and the more attacks it commits, the more likely it will be caught and removed from the system. We set our discount to be 0.999. Figure 4 shows the results with the discount factor. One can see that the effect of the defense

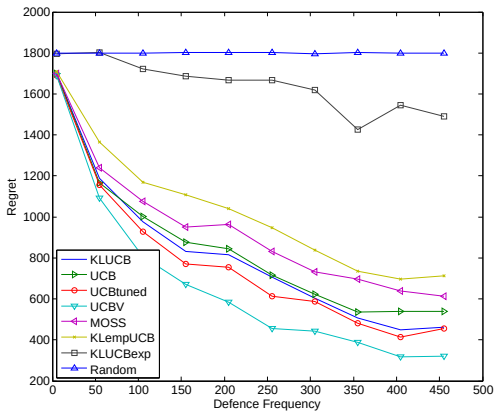


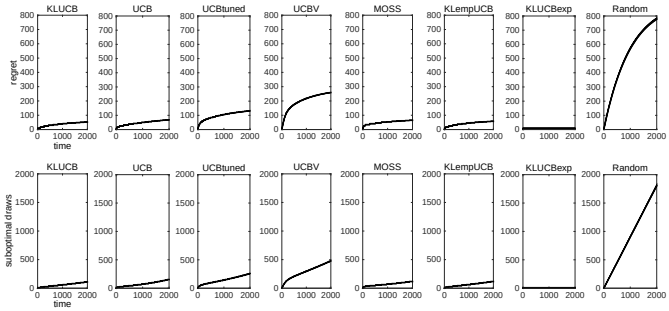
Fig. 3. Final Regrets of Hide Max; No Variance; No Discount

is even more pronounced in this situation, with even the defense every 500 turns showing a significant advantage over no defense at all. Once again, swapping every 5 turns lead to all the attacks performing on par with a random strategy.

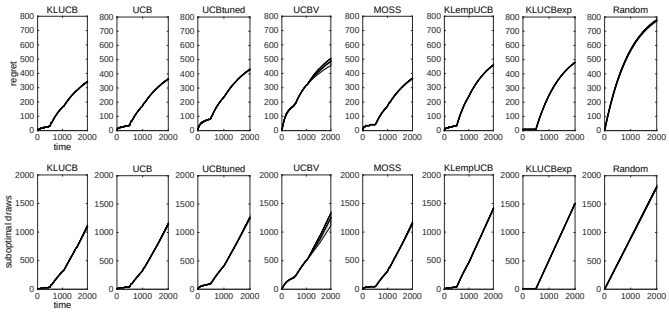
For the next experiments, we removed the discount factor and introduced variance in the rewards. This was done through generating Poisson distributed random values with a mean (and variance) of 1, just like in the first experiment. As shown in Figure 5, the average case performance is nearly identical or better than that of the case with no variance from 2. In fact, the most noticeable difference is the UCB-V algorithm with a defense frequency of 50, which shows a marked improvement for the defence when the variance is added into the data. The other best and worst case results can vary a little, but not dramatically.

To further explore the effect of variance on our defense, we decided to look at a range of variances. We did this by making sure that the mean of the rewards was always 1, but we changed the variance to be 1, 0.1, and 0.01 (the distribution is no longer Poisson). Figure 6 shows the final regret under different variance values when one arm is paying a reward with mean 1 and the Hide Max MTD strategy is used every 50 turns. One can see that the the more varied the data is, the better our defense performs. The Worst Regret line refers to the policy that had the lowest regret against our defense, while the Best Regret refers to the one with the highest regret (excluding the exponentially tuned KL-UCB-exp, since it uniformly performed as poorly as the random policy). Looking at the figure, it quickly becomes apparent that as the variance is increasing, the worst and best performances are approaching each other, and are also approaching the random performance. This means that, since most every real world situation will involve some degree of variance, our defense will perform even better than under normal variant conditions.

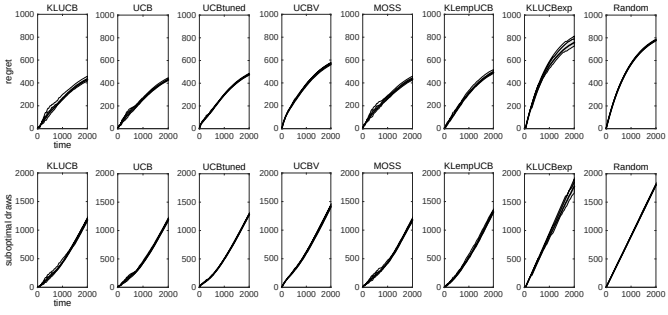
Another aspect we investigated is how the saturation of rewards affects the effectiveness of the Complete Restructure strategy. By saturation of rewards, we mean what percentage of the potential arms actually give a reward. To do this, instead of having just 1 arm with an expected reward of 1, we had 2 arms with an expected reward of 0.5, or 3 arms with



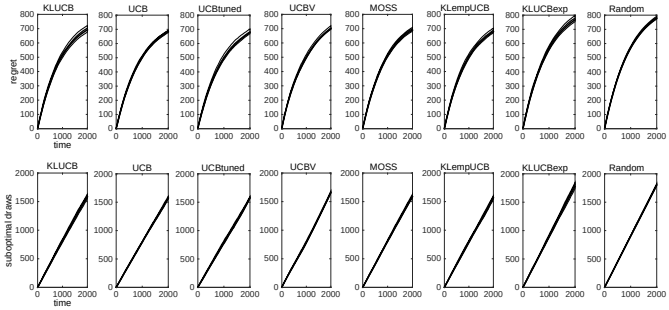
(a) No Defense



(b) Defense Every 500

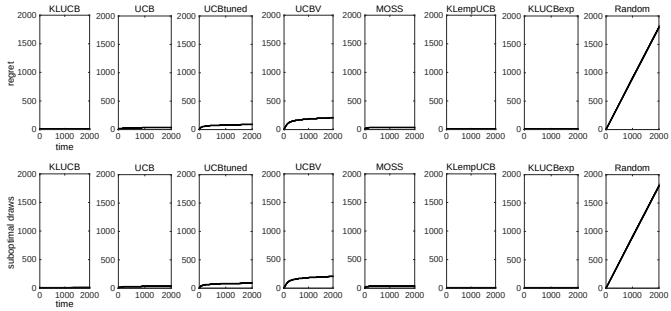


(c) Defense Every 50

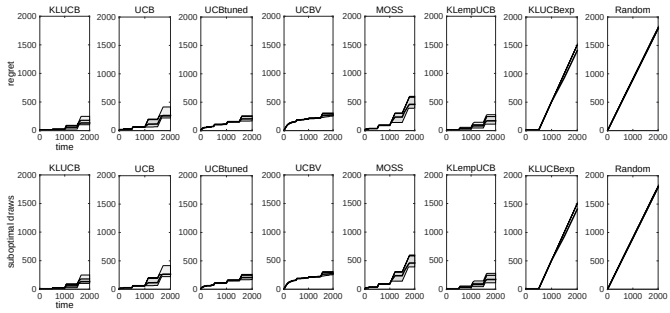


(d) Defense Every 5

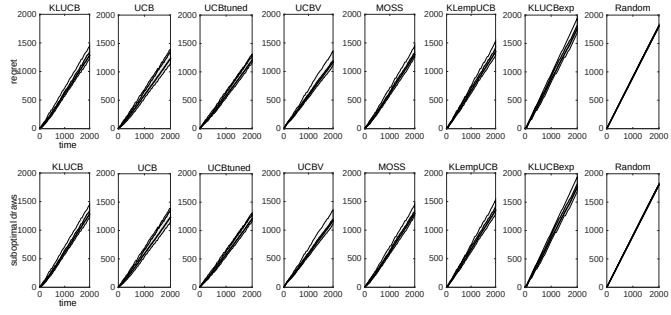
Fig. 4. Hide Max; No Variance; Discount factor = 0.999



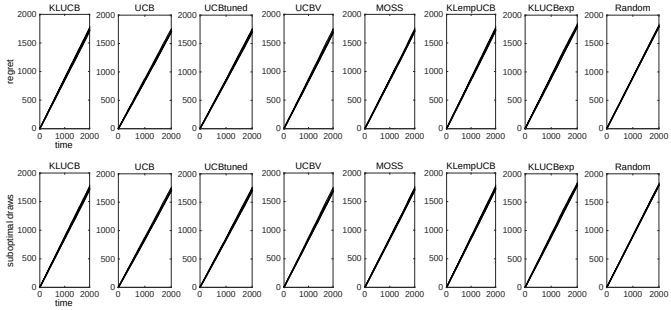
(a) No Defense



(b) Defense Every 500



(c) Defense Every 50

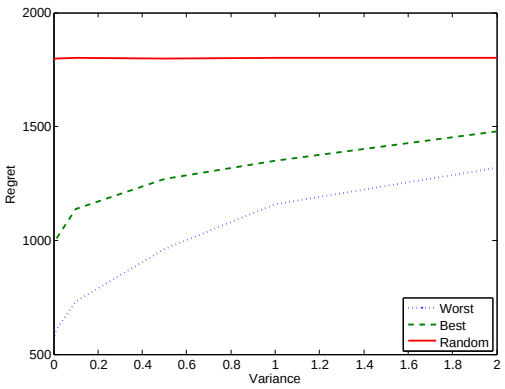


(d) Defense Every 5

Fig. 5. Hide Max; Variance of 1; No Discount

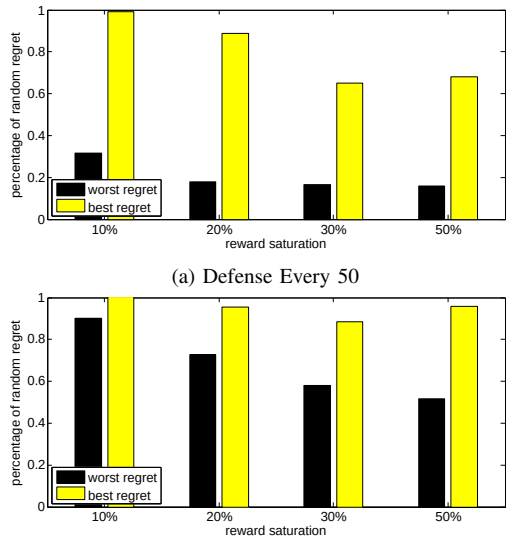
an expected reward of 0.33, etc. The results are presented in Figure 7. It is clear that as the reward saturation increases, the effectiveness of our strategy decreases. Figure 7 (bottom) shows for the defense frequency of 5 that the regret of the policy that our defense performs the worst against goes from 90% of the random strategy at 10% saturation down to 50% of the random at 50% saturation. That's a drastic change, and not in the defense's favor. However, this is not as much of a problem as it might seem at first. If we compare the results of a defense frequency of 5 to that of 50, we see that both the policy that we perform the best against and the policy we perform the worst against show a significant improvement as the defense frequency gets smaller. This means that, while our defense may not be as effective at higher saturations, it is still significantly more effective than doing nothing.

We consider the case in which there is an uneven distribution of rewards. Figure 8 shows results when the arms have different payouts: 1 arm gave an expected reward of 0.6, and 2 arms gave an expected reward of 0.2 each time. We did so to observe differences between the Complete Restructure and Hide Max strategies in this unbalanced situation. One can see that the two defense strategies have nearly identical results.



(a) Variance 0, 0.1, 0.2, 0.3, 0.5

Fig. 6. Hide Max with Defense Every 50; No Discount



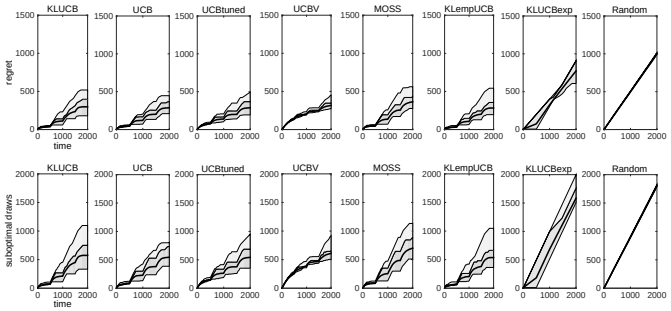
(a) Defense Every 50

(b) Defense Every 5

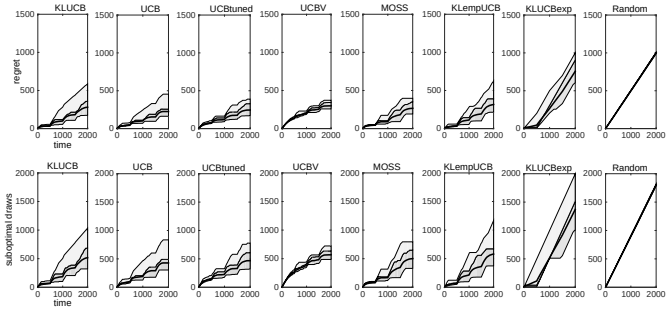
Fig. 7. Complete Restructure; Variance of 1; No Discount

The only difference seems to be the best and worst case performances (shown by the lighter grey areas), which are slightly larger for the Complete Restructure strategy, though even that could simply be the product of different random numbers generated between the runs. In fact, it does not seem to make much of a difference whether you shuffle all the arms or just the maximum valued one, at least in a case where one VM pays significantly higher rewards than all the others. Likewise, if you do not know which VM is the most desirable target, it will not hurt your effectiveness to simply shuffle them all to be safe.

Figure 9 shows the results for our Duplicate and Deactivate strategy. It is apparent that the Duplicate and Deactivate strategy is not as effective as the Hide Max or Complete Restructure strategies. It is, however, more effective than no defense at all. One of the most interesting things from Figure 9 is that the average regret seems nearly constant from changing the activated VM every 500 turns all the way down to 15. The main thing that seems to change is the range of best and worst case scenarios (the light-gray areas). This is most likely because the less frequently the system changes,



(a) Hide Max



(b) Complete Restructure

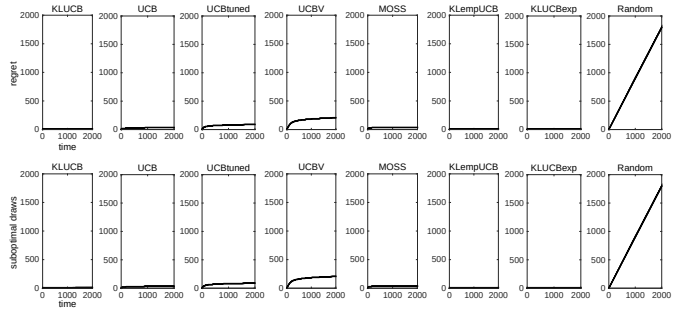
Fig. 8. Defense Every 500; Variance of 1; No Discount

the easier it is to be either extremely lucky (or unlucky) for a long period of time with the randomized configuration.

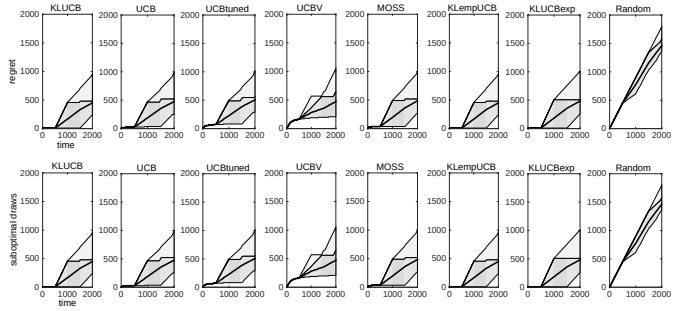
C. Implementation Experiments

To demonstrate the feasibility of our defense strategies and assess its performance in a real-world setup, we created a cloud using OpenStack Kilo devstack running across 3 machines, each with 4 Intel Xeon 2.66GHz processors and 4GB of RAM. The network speed between the nodes was 940Mb/s, measured at 380Mb/s in practice. We tested to see how long live migrations took to complete, as well as the memory and network usage of the physical nodes during migrations. The VM image we used was Ubuntu Trusty 14.04, and it was given 100GB of ephemeral storage and 256MB RAM.

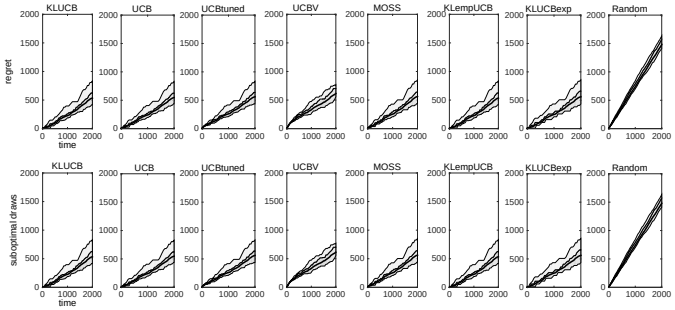
According to [26], the way that OpenStack implements live migrations is by taking the current memory of the VM on the physical node it is on and copying it over to the new node it is moving to. It copies it over as quickly as it can, but since the VM is still in use, the state of the memory is still changing even while it is being copied. Therefore, by the time that the entire memory has been copied, it is no longer the same across the two physical nodes. To fix this, the parts of the memory that have been changed, called "dirty pages", are then copied over. Of course, the memory is still changing while this is going on as well, so it must find more dirty pages to move. This continues until the remaining dirty pages are small enough that they can be moved all at once in a very small amount of time. The VM is suspended during this time and when it finishes, the migration is complete and the VM is resumed on the new node and deleted from the previous node. It is important to note that the network speed can be a limiting factor. If the dirty pages cannot be



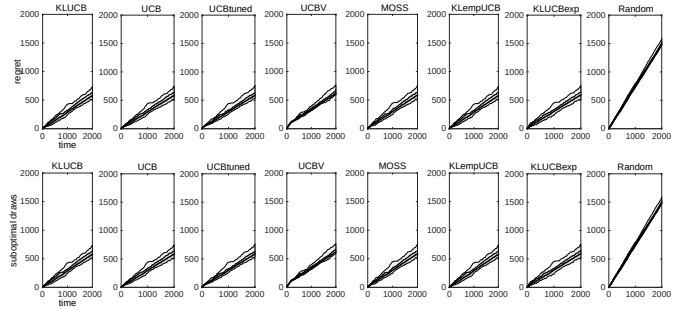
(a) No Defense



(b) Defense Every 500



(c) Defense Every 50



(d) Defense Every 15

Fig. 9. Duplicate and Deactivate; No Variance; No Discount

transferred between physical nodes faster than the VM is creating them, the migration will never complete.

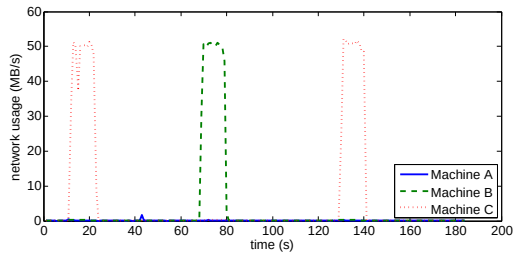
It is clear from this process that the length of time the migration takes is dependent on the size of the VM memory, and on how long it took to get the memory synced between the machines. To test this length of time, we used a "stress" program to specify how much memory we wanted to be used on the VM at any given time. The process to discover if a migration has been completed has a slight delay built into it, so there could be up to 2 seconds of variance between the results. To help minimize this, we ran each configuration 3 times and computed the average times. We used the "ping" command with a 0.1 second interval to test how long the VM was unreachable during the migration.

TABLE I

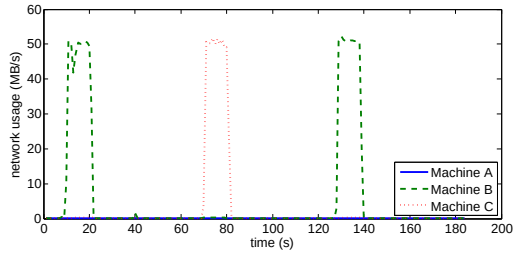
TIME FOR LIVE MIGRATION FOR STRESSED MEMORY (256 MB TOTAL)

Stress (MB)	Migration Time (s)	Down Time (s)
0	22.6	2.0
16	23.4	2.5
32	22.8	2.7
64	27.5	3.3
128	29.4	3.2
200	29.4	2.8

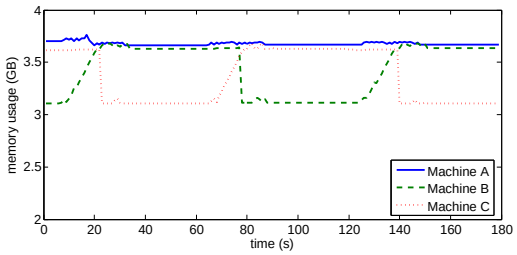
Table I illustrates our results. One can see that as the stress on the memory increases, so too does the length of time it takes to complete the migration. However, even with the stress levels being as high as 200MB, most of the memory of the VM, the migration took less than 30 seconds to complete on average with around 3 seconds of down time, which are very reasonable numbers.



(a) Outbound Network Traffic



(b) Inbound Network Traffic



(c) Memory Usage

Fig. 10. Physical Nodes During Unstressed VM Migration

We also collected some data from the physical nodes during the migration of an unstressed VM. We tested migrating the VM back and forth from Machine C to Machine B, starting on C, once every minute. During this process, we tracked both the inbound and outbound network traffic, and the usage of the memory on all of the nodes, with data

points collected once every second. The results can be seen in Figure 10.

What these results show is that during the migration process there is a sudden flurry of activity, and as soon as the migration has completed the system returns to a more stable state. A large amount of bandwidth is used to transfer the data, but only between the nodes that are the endpoints of the migration. The third node in this scenario doesn't show any increase in network traffic or memory usage at all. The reason Machine A's base memory usage is higher than the other two nodes is because it is also functioning as the controller node, so is responsible for other functions of the cloud, while the other two nodes in our cloud were only responsible for running VMs. The memory plot also shows how the memory of the node the VM is running on is constant until the migration is nearly complete, at which point it is deleted and the memory freed, while the memory of the node that is being migrated to slowly grows throughout the migration. Our results well matche those reported in [27] where the down time was less than 1 second for all sizes of VMs during live migration, and their actual migration time was almost always less than 20 seconds, presumably due to the fact that their cloud was running on more powerful servers and networking components.

V. Conclusions

In this paper, we developed a set of MTD strategies against a class of MAB policy-based attacks. The MAB policies capture the adversarial nature of attackers in the cloud in terms of exploring the VMs and exploiting the ones that yield high rewards. Our results show that the MTD strategies are indeed effective against the various types of MAB policies. With frequent changes to the system, the MAB policies become indistinguishable from a random strategy – one in which the attacker has gained no knowledge to exploit. We have investigated the performance under different parameters such as the discount factor and the variance in the rewards obtained. The presence of the discount factor made no changes to the effectiveness of our defenses where as the variance has made our defenses even more effective. To tie our defense frequency to real-world scenarios, we created an OpenStack setup to show the feasibility of our defense mechanisms. Our setup illustrates that migration times are short enough to implement effective MTD strategies with almost no downtime for the customers. Since network traffic only exists between the nodes directly involved in the migration, we can see that in a system with more physical nodes, multiple migrations can be carried out simultaneously between any nodes not currently involved in a migration, reducing the time it would take to migrate all VMs.

Acknowledgment

This research is funded in part by NSF CNS award #1149397.

References

[1] J. S. Reuben, "A survey on virtual machine security," *Helsinki University of Technology*, vol. 2, 2007.

[2] H. Liu, "A new form of dos attack in a cloud and its avoidance mechanism," in *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop*, Chicago, IL, October 2010.

[3] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A placement vulnerability study in multi-tenant public clouds," *Proceedings of the 24th USENIX Security Symposium*, August 2015.

[4] E. Al-Shaer, "Moving target defense: Creating asymmetric uncertainty for cyber threats," in *Moving Target Defense*, S. Jajodia, K. A. Ghosh, V. Swarup, C. Wang, and S. X. Wang, Eds. New York, NY: Springer New York, 2011.

[5] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, January 2011.

[6] A. Bisong and S. M. Rahman, "An overview of the security concerns in enterprise cloud computing," *International Journal of Network Security and Its Applications*, January 2011.

[7] J. Kaur, M. Gobindgarh, and S. Garg, "Survey paper on security in cloud computing," *International Journal in Applied Studies and Production Management*, May 2015.

[8] J. He, M. Dong, K. Ota, M. Fan, and G. Wang, "Netseccc: A scalable and fault-tolerant architecture for cloud computing security," *Peer-to-Peer Networking and Applications*, January 2014.

[9] F. Gillani, E. Al-Shaer, S. Lo, Q. Duan, M. Ammar, and E. Zegura, "Agile virtualized infrastructure to proactively defend against cyber attacks," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, Hong Kong, April 2015.

[10] H. Robbins, "Some aspects of the sequential design of experiments," *Bulletin of the American Mathematical Society*, September 1952.

[11] R. Bellman, "A problem in the sequential design of experiments," *Sankhya: The Indian Journal of Statistics (1933-1960)*, April 1956.

[12] P. Whittle, "Restless bandits: Activity allocation in a changing world," *Journal of Applied Probability*, January 1988.

[13] ——, "Arm-acquiring bandits," *The Annals of Probability*, April 1981.

[14] D. Chakrabarti, R. Kumar, F. Radlinski, and E. Upfal, "Mortal multi-armed bandits," in *Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems*, Vancouver, BC, December 2008.

[15] J. C. Gittins, "Bandit processes and dynamic allocation indices," *Journal of the Royal Statistical Society, Series B*, July 1979.

[16] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, May 2002.

[17] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense," in *Proceedings of the First ACM Workshop on Moving Target Defense*, Scottsdale, AZ, November 2014.

[18] M. L. Winterrose and K. M. Carter, "Strategic evolution of adversaries against temporal platform diversity active cyber defenses," in *Proceedings of the 2014 Symposium on Agent Directed Simulation*, Tampa, FL, April 2014.

[19] R. Zhuang, S. Zhang, S. A. DeLoach, X. Ou, and A. Singhal, "Simulation-based approaches to studying effectiveness of moving-target network defense," in *National Symposium on Moving Target Research*, Annapolis, MD, June 2012.

[20] S. Bubeck, V. Perchet, and P. Rigollet, "Bounded regret in stochastic multi-armed bandits," *Journal of Machine Learning Research*, February 2013.

[21] P. Varaiya, J. Walrand, and C. Buyukkoc, "Extensions of the multi-armed bandit problem: The discounted case," *IEEE Transactions on Automatic Control*, May 1985.

[22] J.-Y. Audibert, R. Munos, and C. Szepesvari, "Exploration-exploitation tradeoff using variance estimates in multi-armed bandits," *Theoretical Computer Science*, January 2009.

[23] A. Garivier and O. Cappé, "The kl-ucb algorithm for bounded stochastic bandits and beyond," in *Proceedings of the 24th Annual Conference on Learning Theory*, Budapest, Hungary, June 2011.

[24] J.-Y. Audibert and S. Bubeck, "Minimax rolicies for adversarial and stochastic bandits," in *Proceedings of the 22nd Annual Conference on Learning Theory*, Montreal, QC, June 2009.

[25] O. Cappe, A. Garivier, and E. Kaufmann, "pymabandits," 2012, http://mloss.org/software/view/415/.

[26] P. K. Michal Jastrzebski, Michal Dulko, "Dive into vm live migration," 2015, https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/dive-into-vm-live-migration.

[27] V. Cima, "An analysis of the performance of live migration in openstack," 2014, https://blog.zhaw.ch/icclab/an-analysis-of-the-performance-of-live-migration-in-openstack/.