BANDITS IN THE CLOUD: A MOVING TARGET DEFENSE AGAINST MULTI-ARMED BANDIT ATTACK POLICIES

by

Terrence Penner

A thesis submitted to the Graduate College of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2016

Committee Members:

Mina Guirguis, Chair

Qijun Gu

Xiao Chen

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGEMENTS

I would like to thank my advisor and committee chair, Dr. Mina Guirguis, for working with me and helping me these past two years. I would also like to thank the rest of my committee members, Dr. Qijun Gu and Dr. Xiao Chen.
This work was supported in part by NSF awards CNS-1149397 and CNS-1156712.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF ABBREVIATIONS

IaaS Infrastructure as a Service

MAB Multi-Armed Bandit

MDP Markov Decision Process

MTD Moving Target Defence

PaaS Platform as a Service

SaaS Software as a Service

VM Virtual Machine

ABSTRACT

The cloud is a very popular field in both business and computing right now, with many companies starting to move their data and operations into clouds hosted over the public Internet. Both the data stored on the hosts' servers and the operations on it are the customers' proprietary information, so they want assurance that their data will be safe, which makes the security of cloud computing critical for its adoption. Given the complexity of cloud systems, many different attack policies have been created, some of which are for the Multi-Armed Bandit (MAB) problem. In this thesis, we develop a set of Moving Target Defence (MTD) strategies that randomize the location of a cloud's Virtual Machines (VMs) to counter attacks from a MAB policy and we assess through simulation the effect our defense has on a variety of MAB algorithms, showing that it can make them no more effective than a randomized attack policy. Additionally, we show the effect of the critical parameters (e.g. time between randomizations of VM locations, variance in the effectiveness of an attack, etc.) on the performance of our defense, and use a real OpenStack system to validate our defense strategy through the collection of migration times and VM down times for different VM memory loads.

I. INTRODUCTION

One of the areas of computer technology that has had the most explosive growth in recent years is cloud computing. According to Shetty (2013), by 2016 cloud computing will be the the single largest new expense in the field, with "nearly half of large enterprises" having some form of cloud deployment by the end of 2017. The amount of money in the cloud market is staggering, with an estimated $191 Billion by 2020 according to Andrew Bartels (2014). Cloud deployments are created and hosted by big name companies that offer cloud solutions to other corporations and private individuals, such as Amazon, Microsoft, Google, IBM, and Oracle.

Cloud Definitions

In broad terms, a cloud deployment is an application suite that provides customers with various features in a highly scalable fashion through the use of virtualization. These features are offered "as a service" over the Internet in a few different variants: Software as a Service (SaaS), Infrastructure as a Service (IaaS), and Platform as a Service (PaaS). SaaS is where the cloud provider hosts the software on its servers and the subscribers simply use a local client to interface with the remote server over the Internet. On the other side of things, IaaS is much more open with what it can be used for, by giving subscribers the ability to create their own VMs on the hardware the cloud provider has given them. PaaS offers a sort of middle ground between SaaS and PaaS, where subscribers can build their own applications, but also have access to functionality given by the cloud provider.

These "as a service" offerings are possible through the power of virtualization. The cloud consists of several physical nodes connected together in a network, and each node runs VMs for the subscribers. For a traditional network, in order

to add a new machine, it would need to be purchased and attached to the network, an expensive and time consuming process that also uses up a lot of space in a data center. Virtualization changes all of that. Instead of each physical machine simply being just one machine in the network, they can each have multiple VMs that each take a portion of their resources, so each VM can pretend to be multiple machines in the network. This makes an IaaS situation of adding a machine to a network as easy as starting a new VM. This can be leveraged for SaaS and PaaS situations as well: if a subscriber finds they need a more powerful application, they can ask the cloud provider to upgrade their server to have more RAM or access more CPUs, simple changes to make in a virtualized system. There are three main types of clouds: private, public, and hybrid. Private clouds are built with machines given for the sole use of one subscriber, and are often run from data centers that the subscriber themselves own and are responsible for maintaining. A public cloud is one that is provided over the Internet for the use of anyone who will subscribe, hosted on the cloud provider's servers. A hybrid cloud is, as the name suggests, using a combination of a private and public clouds.

Cloud Security Flaws

For any computer system, security must be taken into consideration during its design, which is especially true for a new and rapidly growing area such as cloud computing. A detailed overview of common security issues in cloud computing is found in Subashini and Kavitha (2011), who describe flaws in all three variants of service provided by clouds: SaaS, IaaS, and PaaS. We will be focusing on IaaS systems, and the flaws we find most interesting are Network security, Data breaches, and Vulnerability in Virtualization. Additionally, we are interested in the VM Escape, Denial of Service, and VM Monitoring from Another VM attacks from Reuben (2007). These are all facets of the cloud that can be targeted in order to either gain access to the sensitive data of other subscribers,

or to render the cloud functionally unusable.

There are two main types of attacks on a cloud: internal and external. External attacks are primarily network focused, sometimes with the goal of infiltrating the system and becoming an internal attack. Because clouds offer access through the Internet, they are vulnerable to the same types of attacks that other web applications are, such as SQL Injection or Denial of Service, the attack type that has taken down Twitter and Facebook, as per McCarthy (2009). Internal attacks, on the other hand, can only be done on systems that the attacker already has access to. This is the type of attack that was used in the Heartland Payment Systems attack Press (2009), where malicious software was uploaded to the company's network and credit card information from some 130 million customers was stolen, costing Heartland $110 Million. In cloud systems, because the VMs of different subscribers can be located on the same node as each other, it is important to keep their data separate. Some of these subscribers are companies that are uploading very sensitive data, including customer data and the company's own intellectual property. Naturally, they do not want anyone else being able to access this information freely.

Consider the following attack scenario where the attacker uses the Data Breach and Network Security flaws from Subashini and Kavitha (2011): there is an attacker that has managed to load malicious code directly onto the physical nodes of a cloud, something that can be done through a VM Escape exploit Reuben (2007). This means that the attacker has the ability to sniff for packets being sent to or from any of the VMs on the cloud. The attacker may value finding certain information more than others; perhaps they are trying to find credit card transactions that are all processed from one VM, located on an unknown node. The attacker just needs to find which node has the VM that is sending the packets containing the information that it is looking for, and sniff for those packets on the node that VM resides on.

Another possible scenario that uses the Network Security flaw from Subashini

and Kavitha (2011) is an attacker who has placed their code on one of the VMs that they have legally created. They can send packets to any of the other VMs they have created, which may or may not be located on the same physical node, in an attempt to map the underlying physical network. Once they have mapped out as much of the network as they can see, the attacker can then determine which of the virtual links it should attack in order to have the greatest effect on the performance of the physical network. It can then flood that link with traffic in an effort to degrade the performance of the entire cloud, an example of a Denial of Service attack from Reuben (2007). A process for carrying out this type of attack is given in Liu (2010).

Finally, consider a third scenario that uses the Vulnerability in Virturalization security flaw from Subashini and Kavitha (2011): an attacker has loaded malicious code onto their VM and uses a virtualization vulnerability that allows them to read information from any of the VMs located on the same physical node as theirs, an example of a VM Monioring from Another VM attack from Reuben (2007). Unlike the credit card sniffing scenario, 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 use this to snoop around the live memory of the other machines, looking for sensitive information to steal from the currently running processes.

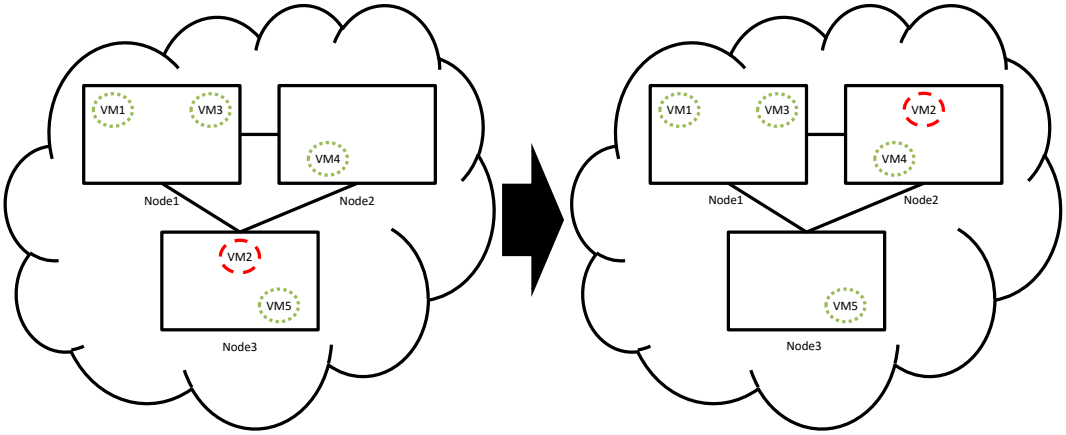
Attack and Defense Strategies

What do the scenarios given in the previous section have in common? For one, the attacker can solve them all with a Multi-Armed Bandit (MAB) policy. MAB is a well known statistical problem where a player (the gambler) is presented with multiple slot machines. Each turn, they can pull the arm of one of the slot machines, and receive some payout. Before starting the game, they have no idea how much reward they can expect to get from each of the slot machines, which all pay out with independent rewards. The gambler must choose every turn

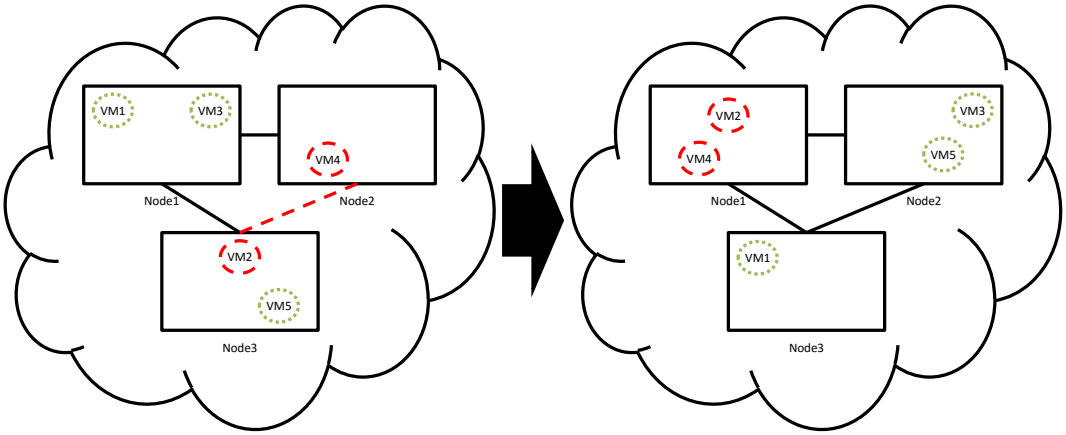
whether they should exploit a well paying arm they have already found, or if they should explore the other arms to see if there is a better paying one that exists. The other similarity that these scenarios have is that they are good candidates for the defender to apply a Moving Target Defence (MTD) strategy, from Al-Shaer (2011). This is a defense strategy based on the exploitation of randomization, where the idea is that the defender makes some changes to its system's configuration every so often. This should cause the attacker to have a much more difficult time trying to be successful.

This MTD strategy matches our scenarios very well. For example, in the credit card sniffing scenario, the attacker can apply a MAB policy to determine which of the VMs it should sniff the packets of. However, we can create a literal moving target by changing the layout of the system. We can use a MTD strategy of migrating the VMs from one physical node to another, which will also 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, but because the attacker doesn't know that the VMs have been moved, it may not equate the lack of packets with the VM being moved for quite a while. A visual representation of this strategy can be seen in Figure I.1, where the green dotted VMs are running, and the red dashed VM is the one with the credit card information.

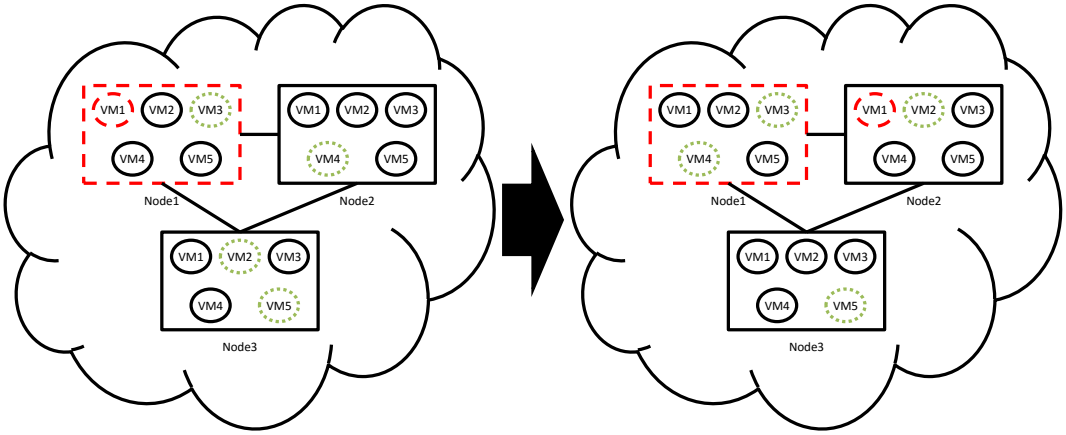
Our network mapping / flooding scenario also fits nicely with these strategies. The attacker must choose between each of the links it can see, where each one will have a different likelihood of getting congested based on how much traffic it is responsible for at any given time. A MAB policy can be applied to help choose which of these links to target. The MTD can once again create a moving target by migrating the VMs 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



(a) Credit Card Sniffing



(b) Network Mapping / Flood



(c) Memory Snooping

Figure I.1: Moving Target Defense Scenarios

formerly went over physical links could be on the same physical node now, resulting in no congestion at all. A visual representation of this strategy can be seen in Figure I.1, where the green dotted VMs are running, the red dashed VMs are the ones the attacker has created, and the red dashed network links are the ones the attacker can see.

Our memory snooping scenario can also have these strategies applied to it. As in the previous two scenarios, it is easy to use a MAB policy to choose which of the VMs to attack. In addition, we decided to use a slightly different system design and MTD strategy, for clouds that are worried about the potential failure of nodes. This new system works by creating a copy of every VM on every physical node, but having all of them suspended except for one of each at any given time. 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 the credit card sniffing scenario, the attacker does not have control of the network, so it cannot follow the VM when it moves. A visual representation of this strategy can be seen in Figure I.1, where the green dotted VMs are the active ones, the black solid VMs are the suspended ones, the red dashed VM is the one the attacker is looking for, and the red dashed node is the one the attacker is located on.

With this thesis, we aim to show the effectiveness of our MTD strategy against an attacker using a MAB policy. As we have just shown, this is a strategy that is very relevant to our modern computer systems, and in particular to the field of cloud computing. This thesis will show that using our proposed defense strategy can make an attacker's policy be effectively wasted effort compared to simply choosing a random target to attack each turn. While it will not completely block the attacker from having some small success, it will greatly reduce the potential damage that can be caused. This will help protect customers' data in enterprise

cloud environments.

The contributions of this thesis are as follows:

1. Develop a set of MTD strategies that introduce randomization to counter attacks from a MAB policy.

2. Assess the impact of our defense on 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, 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 for different memory loads.

Outline

In Section II we will cover what related work has been done in this field. Section III will explain the methodology behind this problem and strategy in detail, before Section IV introduces the results we gathered from our theoretical and real system evaluations, and we conclude in Section V.

II. RELATED WORK

Cloud Environment Security

The main area we would like to apply our MTD strategy to is that of cloud environments. Like any other fast growing area, security is a primary concern. Earlier in Section I, we mentioned Subashini and Kavitha (2011), who give a detailed list of types of security flaws that can be found in clouds from the perspective of a software developer. Bisong and Rahman (2011) list seven major ways cloud services can be at risk from the perspective of a business executive, and give recommendations for how companies can prepare to leverage cloud services.

Some solutions are given in Kaur et al. (2015), who give an overview of several current techniques for protecting data through encryption, and by He et al. (2014) who describe a new architecture for cloud systems designed with security as the primary concern.

The authors of Liu (2010) proposed a migration based method for detecting and avoiding Denial of Service attacks in a cloud environment that sounds similar to our solution on the surface. The main difference is that their solution is to be implemented by the clients running on the VMs on the cloud, whereas our solution is built into the cloud system itself.

Similarly, the authors of Gillani et al. (2015) 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. In addition, in their evaluation they gathered their results by 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 is one that aims to address a few of the security concerns listed in the above papers, mostly focusing on the Network Security flaw from Subashini and Kavitha (2011) and the Malicious Insiders flaw from Bisong and Rahman (2011). Our MTD strategy can be used in tandem with encryption for extra protection, and does not require a redesign of the existing cloud, so could be implemented with a minimum of effort.

Multi-Armed Bandit

The MAB problem that we mentioned in Section I has been around for many years. The standard version of the problem that we will define in Section III was described in Robbins (1952). However, 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 defines the rewards as being given by a probability distribution like we use, but other ideas have been proposed, such as the one by Bellman (1956), 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, defined in Whittle (1988), where all the arms transition state each 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 Whittle (1981), they define a problem where more arms appear over time, growing the number of choices the gambler is presented with. In Chakrabarti et al. (2009), 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, published by Gittins and Gittins (1979). In more recent years, the UCB algorithm from Auer et al. (2002b) has been a standard, with it forming the base of many other variations. We will go into more detail on solutions in Section III. Solutions to the non-stochastic problem exist as well, such as the adversarial bandit from Auer et al. (2002a). While they claim to make no assumptions about there being any logic to the reward distributions of the arms, this also means that they make fairly weak claims about what performance they can achieve.

As we showed in Section I, an attacker could very easily be applying a MAB policy to decide where to target next, since it fits the cloud environment quite well. The rewards that each arm pays out in our scenarios are all best modeled by a standard probability distribution since it is trying to model the fluctuations of the usage of the VMs, thus we are modeling a stochastic problem, and will spend our effort competing against those policies.

Due to the nature of our MTD moving the reward distributions around at random, it could be argued that the non-stochastic problem would be a better fit, but the bounds on that are well known and weak, and we decided that it would be more interesting to see how policies developed for the stochastic problem would fare against our MTD, since our base scenarios without any defense applied are stochastic themselves.

Some of the MAB variants that modify the game state (like Chakrabarti et al. (2009)) are similar 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

In order to combat the MAB policy, we are applying a MTD strategy. One of the definitive works in this area is by Al-Shaer (2011). This book 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 thesis will be delving into the effectiveness of a MTD against one specific attack strategy, the MAB. In a shorter paper, Winterrose and Carter (2014) also give an overview of several different types of Moving Target defense strategies and compare how they perform in different attacking scenarios. A formalized theory of MTD systems is laid out in Zhuang et al. (2014).

In their paper, Winterrose et al. (2014) go deeper into the migration based defense strategy, which is the same type of strategy that we are using in this thesis. However, they compare it to an attacker that acts based on a genetic algorithm, rather than a MAB policy like we do.

In Zhuang et al. (2012), 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 VMs from node to node, with no information lost, and almost zero expected downtime. They are also using a configuration manager component to decide where to refresh the VM, whereas we are simply migrating them at random, since all the physical nodes in our network are equivalent.

Although all MTD strategies are based on the same fundamental idea (to increase the difficulty for the attacker by introducing change into the system's structure), the other strategies go about this in a different way than we do. We are presenting a novel idea of near continuous VM migration.

Additional Works

Another type of game that is very similar to the MAB problem is the Stakelberg game, first described in Kaldor (1936). These games also function by having the attacker and the defender take turns making their move, one the "leader" and one the "follower". The main difference between this and our work is that the leader and follower can both see each other's moves, in a way making it a more general version of our problem. These have been applied to the security of physical locations already, as seen in Jain et al. (2011) and Kiekintveld et al. (2008).

III. METHODOLOGY

Multi-Armed Bandit

Let us now expand on our definition of the MAB problem from Section I more formally. In the game, the gambler is presented with K slot machines to choose between, and it will last for T turns. Each turn $t \in \{1, ..., T\}$, the gambler will select one of the slot machines $m \in \{1, ..., K\}$, and will pull its arm, receiving some reward $r_{m,t}$. This reward is decided upon by the slot machine m, and the gambler is not privy to any knowledge about how this reward was generated. The goal of the gambler is gain as much total reward as possible, maximizing Equation III.1 through the clever choice of m at each turn.

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

Traditionally, as well as in our case, the rewards that each machine m arm gives are chosen based on some probability distribution d_m, where each of the K machines have their own distribution independent of the others.

In order to win this game, the gambler creates a policy to select which m to pull each turn. Because the gambler has no foreknowledge of how the machines select their rewards, they must explore the arms before they can choose which to exploit. Policies are designed to choose when to explore and when to exploit based on the rewards that were earned in previous turns. Policies are not evaluated in terms of maximizing rewards, but instead in terms of minimizing the regret ρ.

Generally speaking, 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, a common way to decide the regret is by calculating the difference in the expected rewards, as in Bubeck

et al. (2013). We call the expected value of pulling machine m's arm μ_m, and the best expectation $\mu^* = \max_{1 \leq m \leq K} \mu_m$. This then defines regret after T turns as:

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

where $P_m(t)$ is the number of times machine m's arm has been pulled by time t. This equation can also be reformulated in terms of T, becoming:

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

where m_t is the index of the arm that was pulled at time t.
Sometimes the situation arises where we would like to model the attacker valuing successful attacks earlier more. To do this, we can introduce a discount factor $0 < d < 1$, as in Varaiya et al. (1985). Equation III.1 would become:

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

and Equation III.3 would become:

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

Attack Strategies

One important note to make about the MAB is that it has many different policies that can be applied. In order to truly gauge the success of our MTD, we will need to compare it to several of these policies. The policies that we chose are those implemented in the maBandits library by Cappe et al. (2012). Full details for each of these policies can be found in their respective sources, but a brief description of each follows:

- UCB: This is the most basic of the strategies, from Auer et al. (2002b). It

is very straightforward: it pulls every arm once, and then it chooses the arm that maximizes $\bar{\mu}_m + \sqrt{\frac{c*\log t}{P_m(t)}}$ where $\bar{\mu}_m$ is the current sample expected reward, c is a constant positive number, 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 is included to have 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 it is receiving are not very large.

- UCB-V: A fairly straightforward modification of the UCB policy, from Audibert et al. (2009). It chooses the arm that maximizes $\bar{\mu}_m + \sqrt{\frac{2*\log t*\bar{v}_m}{s}} + c * \frac{3*b*logt}{s}$, 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 Garivier (2011). It always selects the arm with the maximum $P_m(t) * BKLD(\bar{\mu}_m, \log t + c * \log\log t)$, where $BKLD$ is the Bernoulli Kullback-Leibler divergence, a measure of information gain, so it is trying to select the arm with the most gain likely. Also used was a variant tuned for use specifically with exponentially distributed rewards.

- MOSS: This policy from yves Audibert et al. (2004) selects the arm that will maximize $\bar{\mu}_m + \sqrt{\frac{\max(\log(\frac{h}{P_m(t)*n}),0)}{P_m(t)}}$, where h is the horizon and n is the number of arms. 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 Cappe et al. (2012), written by the authors of the maBandits package. It is a variation on the KLUCB policy, so is also trying to pick the arm with the greatest information gain.

Moving Target Defense

At this point, a more detailed definition of a MTD strategy is required. At its heart, 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 Zhuang et al. (2014).

Of course, in order to actually use a MTD strategy, some additional components must be added to the configurable system to facilitate the making of the changes. As per Zhuang et al. (2012), the two essential components that must be added are the adaption engine and the configuration manager (the logical mission model they mention is the same as our polices P). The adaption engine is the component that decides what changes should be made to the system, as well as how often they should be made. The configuration manager is the component that actually makes and enforces the changes. If desired, an additional analysis engine component can be added that feeds current system information into the adaption engine to help make more informed decisions.

To apply the formal definition to our example cloud system,
$s_i = \{(VM1, Node1), (VM2, Node4), ...\}$, defined by its mapping of each VM to a physical node, with S being the set of all possible permutations of this mapping. α would be migration commands to move VMs 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 VMs," and g_{s1}, "prevent customer traffic from being intercepted." Finally, the set of policies P would include rules such as p_1: "The sum of the disk space required by all VMs 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 of this we call Hide Max, where the only VM we migrate is the one that rewards the attacker the most, 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 a single other VM.

Our third strategy is what we call 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 (see the memory snooping scenario in Section I). 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 with the node it is activated on.

Since the only changes we are making in our strategy are migrations of VMs and it can be safely assumed that all the physical nodes that they are being migrated between are identical, we decided that our adaption engine needed to be only as complex as a random number generator used to decide the new nodes the VMs would move to. Since the adaption engine is also responsible for deciding when to trigger the defense, we chose to set it as a fixed interval that we would be able

to vary to better see how it effects the performance. Because of this simplicity in the adaption engine, we had no need of an analysis engine at all. The configuration manager is only responsible for initiating the migrations, which is a tool that is already provided by most cloud systems.

IV. RESULTS

Theoretical Analysis

As we mentioned in section III, there are several different policies for finding solutions to the MAB problem. In order to get a good range of techniques, we used the open source maBandits package from Cappe et al. (2012) as a base and modified the MATLAB implementation to allow for our MTD strategies.

The base maBandits implementation is a fairly straightforward simulation. It 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, at which point the game ends. 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 Cappe et al. (2012).

In order to add our MTD strategy to this code, we had to make some modifications. We simulated our adaption engine by adding a section of code that would, when triggered, swap the rows in the reward table. This would therefore cause the distribution and reward payout of each arm to swap as well. We set it so it would be triggered at a set interval, and we varied the interval to see how the frequency of swaps would affect the MAB policies.

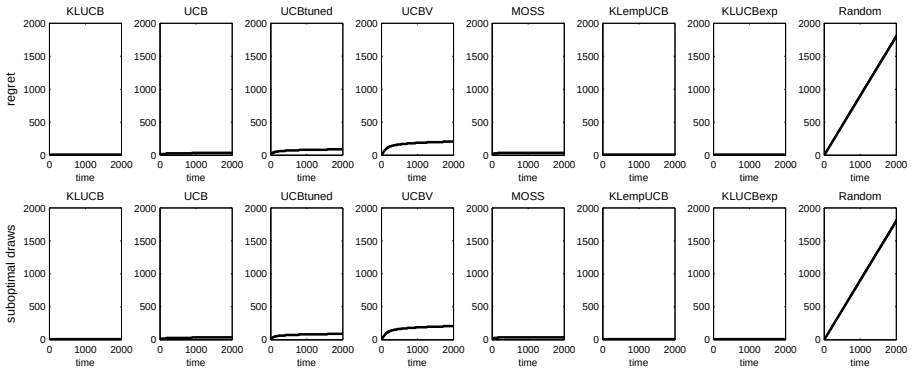
For all of our experiments, we set our horizon to be 2000 turns and 20 tests to be repeated and averaged. We ran tests with 10 arms, where the sum of the expected rewards for all the arms was equal to 1. We also made sure to have a

control test where our added MTD code would not be triggered, in order to better see what effect our strategy had.

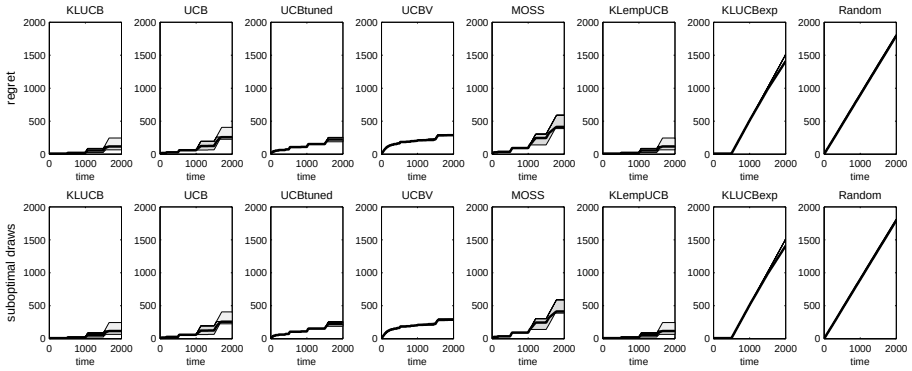
For our first experiment we created the simplest situation where the regret was calculated by Equation III.3, and all the variance in the rewards was removed so each arm would always pay out the exact same amount each time it was pulled. This allowed us to more clearly see the effect of our defense, since we effectively removed any randomness of the data. We decided to use our simplest defense strategy first, Hide Max, and we created a situation where there are 10 arms, and only one of them pays out any reward for the attacker. It had a paid 1, while all other arms paid 0. The detailed results for this are shown in Figure IV.1. What you can see from this sequence of figures is that as we increase the frequency of our swapping defense, the effectiveness of the attack strategies all decrease dramatically. We decided to set the defense frequency to every 500, 50, and 5 turns to see the results on a logarithmic scale. There is a noticeable effect by shuffling every 500 turns, and by the time it reaches every 5 turns, the attacks are nearly indistinguishable from a random strategy.

In order to explore the effect the frequency of swapping had in more detail, we created plots that combined all those results into one. This is seen in Figure IV.2, where the x values are the number of turns the defense waits between swapping, and the y values are the average regret of the policies at the end of the game. The first plot shows the regret when the defense frequency varies from 5 to 455 in steps of 50. It shows that at around 50, the attack policies all start to bunch up somewhat, so we ran a second set of tests where we varied the frequency from 5 to 43 in steps of 2, shown in the second plot. The takeaway from these results is clear: from the attacker's perspective, the more frequently the defense is activated, the more similar the attack policies' performances become, with all of them approaching the performance of the random strategy.

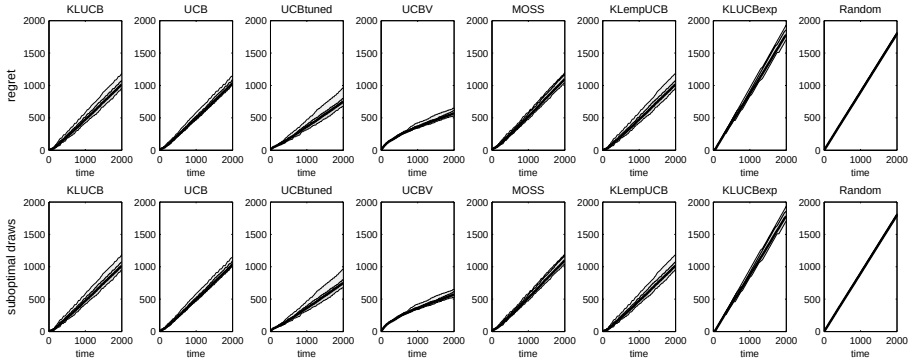
For our next experiment, we changed our regret calculation to use Equation III.5, adding the discount factor in for situations where the attacker is on a time



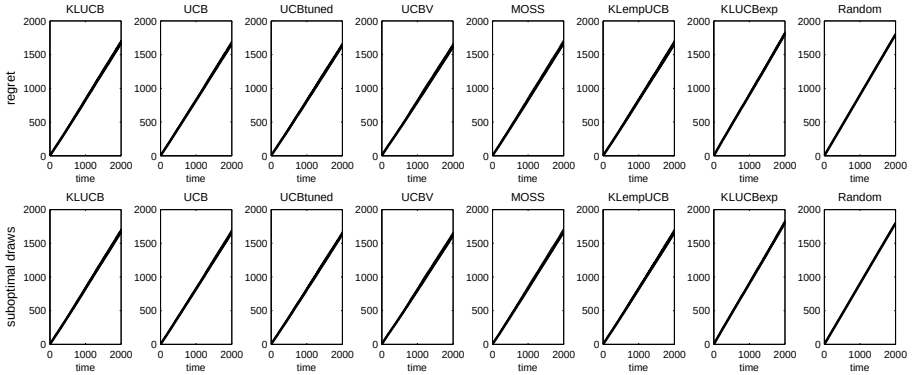
(a) No Defense



(b) Defense Every 500

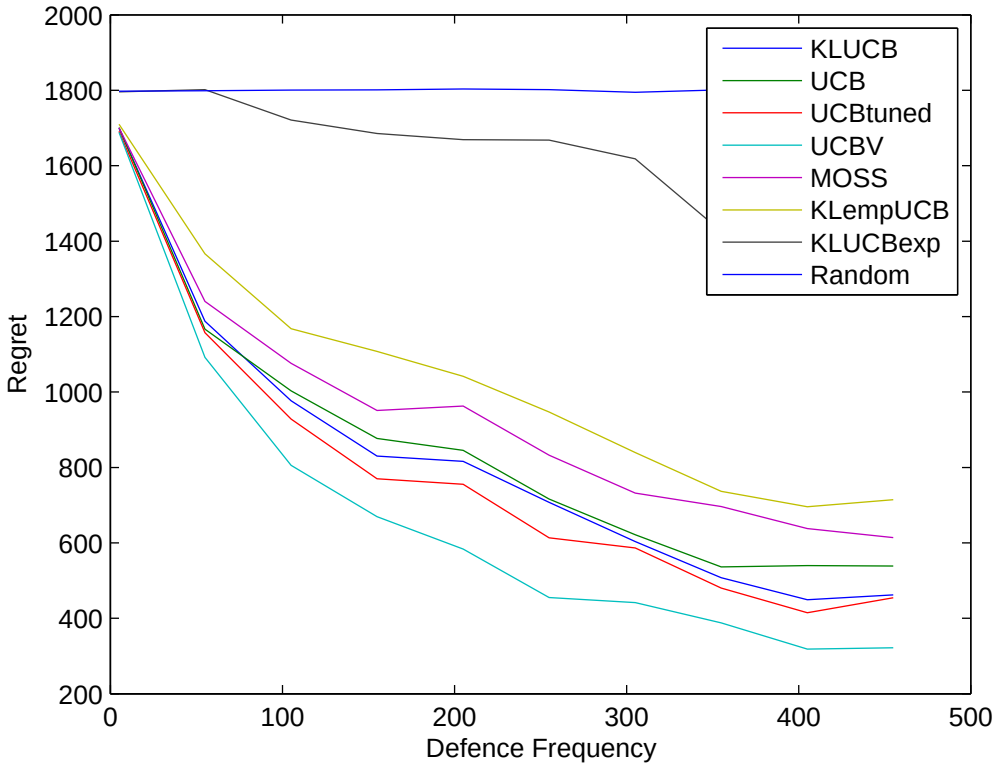


(c) Defense Every 50

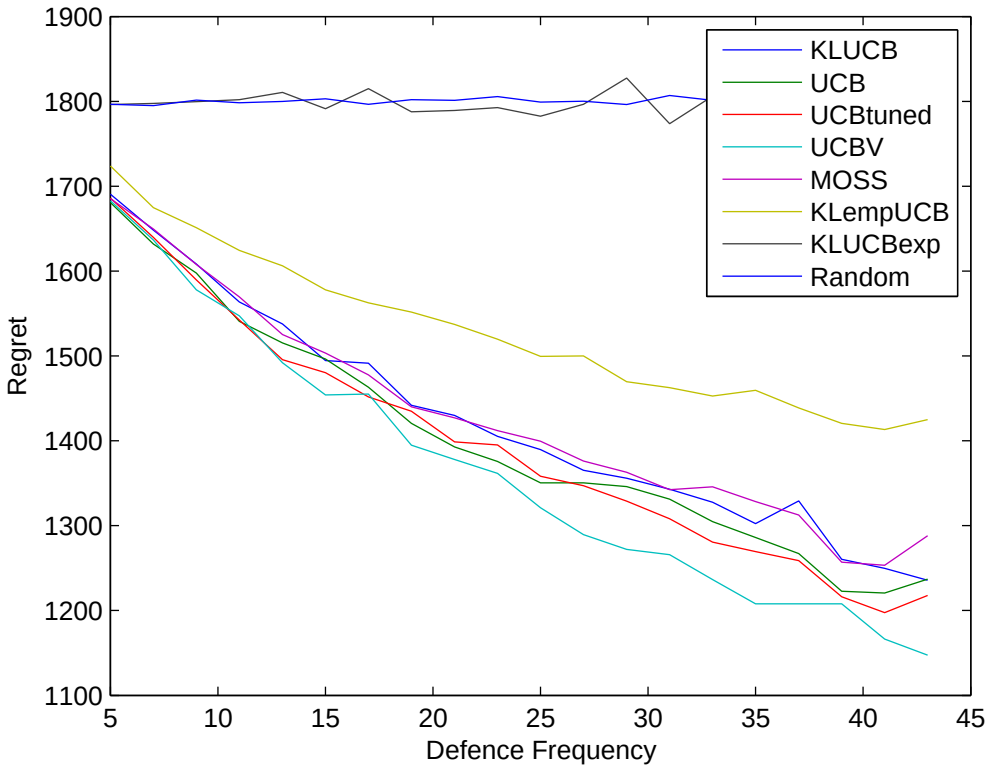


(d) Defense Every 5

Figure IV.1: 1-1 9-0; No Variance; No Discount; Hide Max

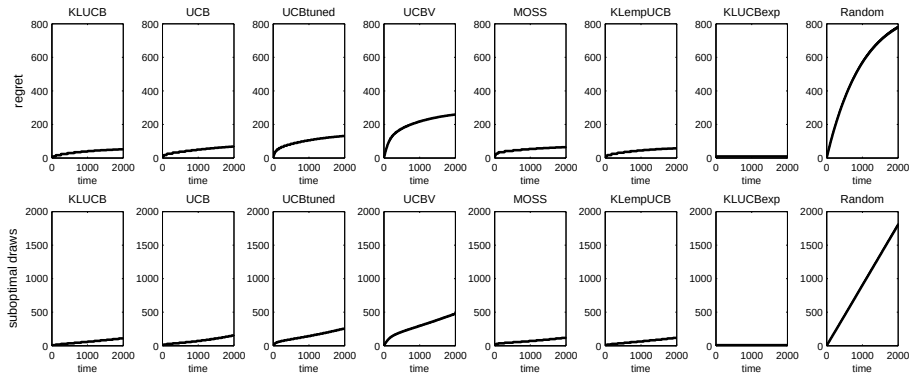


(a) Defense Frequency from 5 to 455; Step of 50

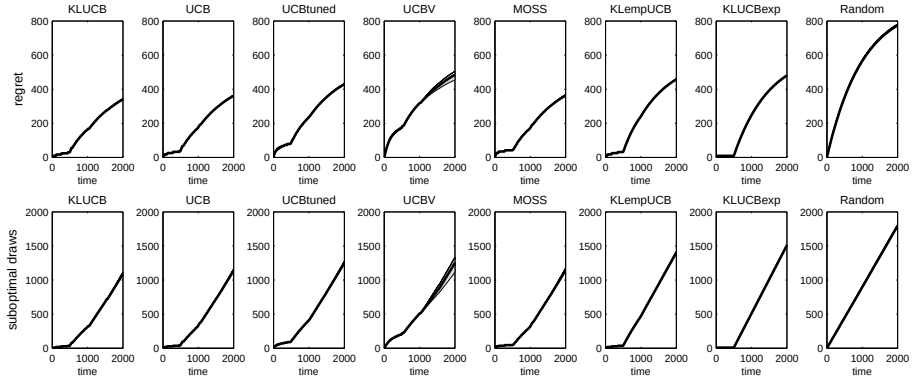


(b) Defense Frequency from 5 to 43; Step of 2

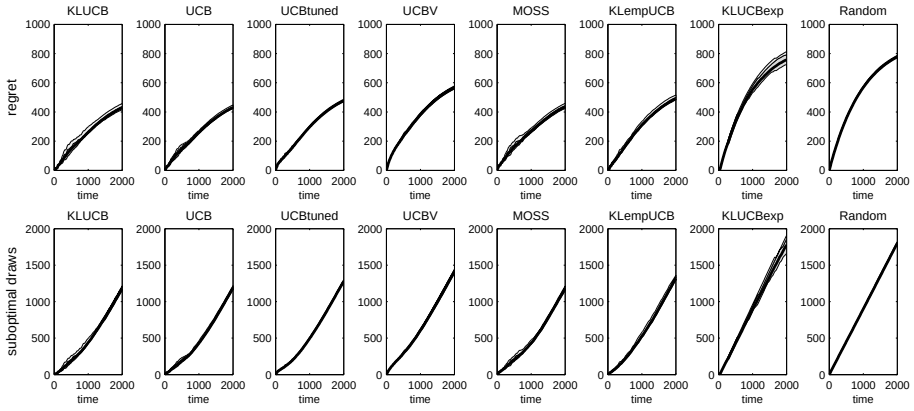
Figure IV.2: 1-1 9-0; No Variance; No Discount; Hide Max; Final Regrets



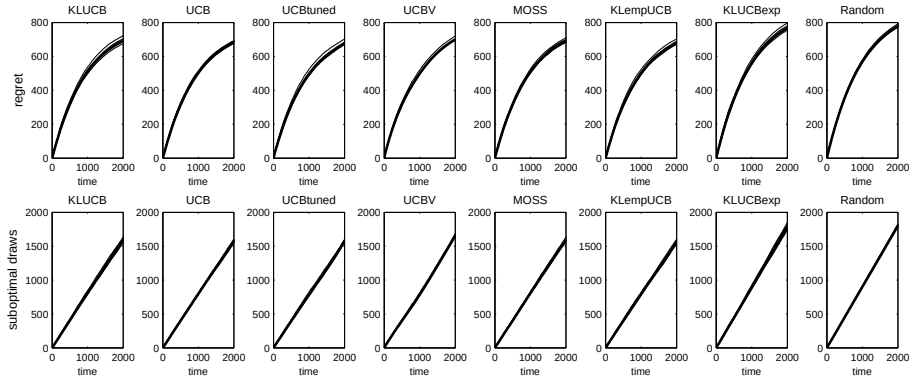
(a) No Defense



(b) Defense Every 500



(c) Defense Every 50



(d) Defense Every 5

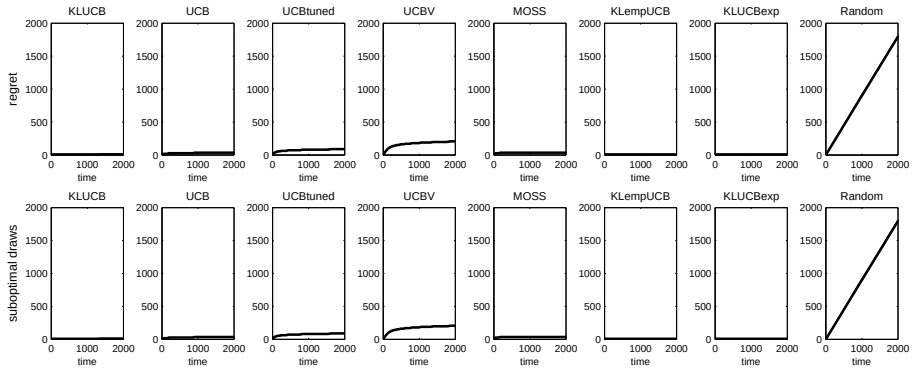
Figure IV.3: 1-1 9-0; No Variance; Discount .999; Hide Max

sensitive schedule and must collect rewards as near the start of the game as possible. An example of this could be when the attacker in on a system 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 $d = 0.999$, or a tenth of a percent decrease each game. The effect of the defense 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. This is all shown in Figure IV.3.

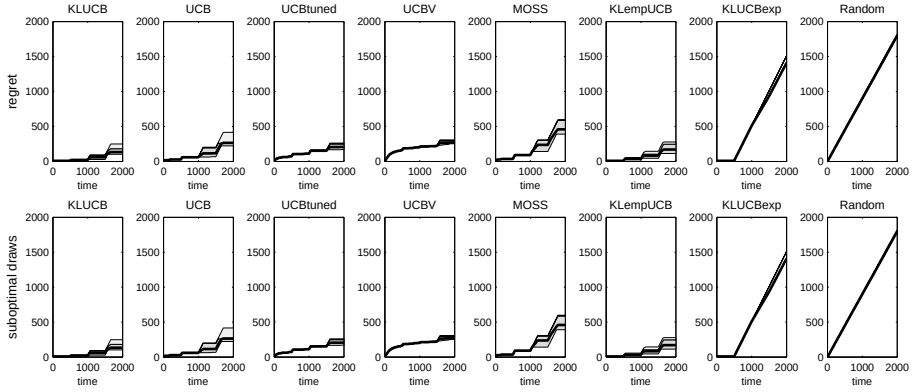
Our next experiment removed the discount factor, switching our regret calculation back to Equation III.3, and we added variance to the data. This was done through generating Poisson distributed random values with a mean of 1, just like in the first experiment. As shown in Figure IV.4, the average case performance is nearly identical or better than that of the case with no variance from IV.1. In fact, the most noticeable difference is the UCB-V algorithm with a defense frequency of 50, which shows a marked improvement when the variance is added into the data. The other best and worst case results can vary a little, but not dramatically; all in all, it is quite consistent.

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. As you can see in Figure IV.5, particularly with the UCB-V algorithm, as the variance increases, the defense gets more and more effective.

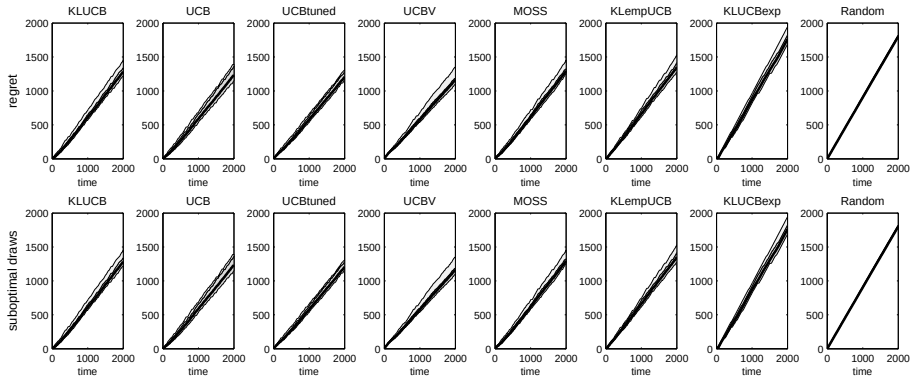
The effect of variance on the MAB policies is laid out fairly clearly in Figure IV.6. For all of the tests in it, the mean reward was fixed at 1, the arms were set up with only one paying out anything, and the Hide Max strategy was used every 50 turns. The figure shows what the final regret was after all 2000 turns. Since the only thing that changed was the variance, it's quite apparent that the more varied the data is, the better our defense performs. The Worst Regret line refers



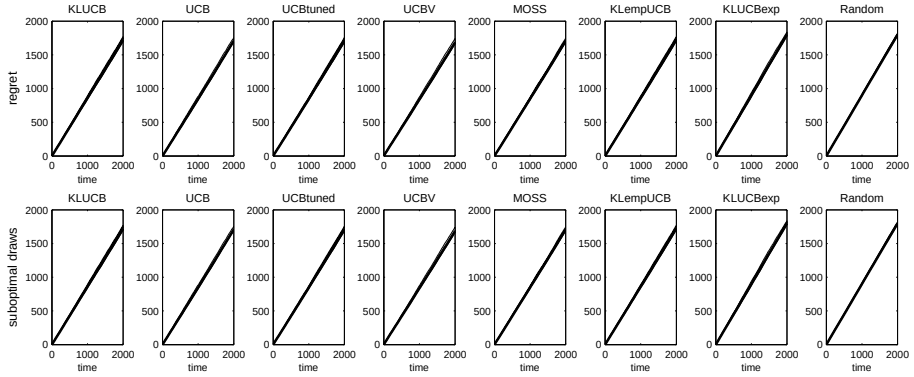
(a) No Defense



(b) Defense Every 500

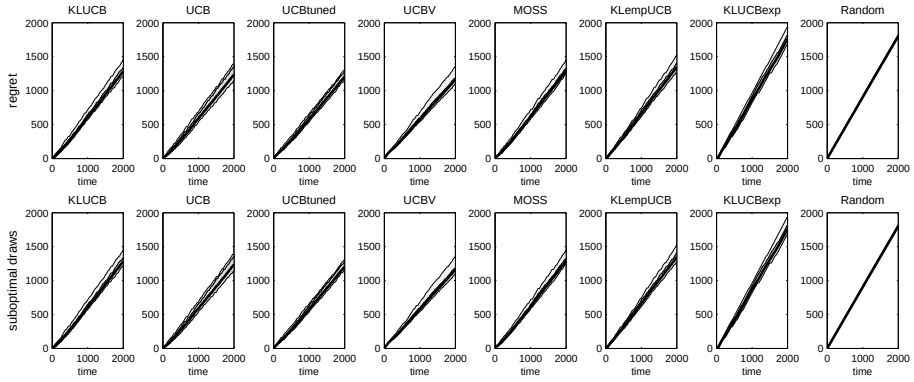


(c) Defense Every 50

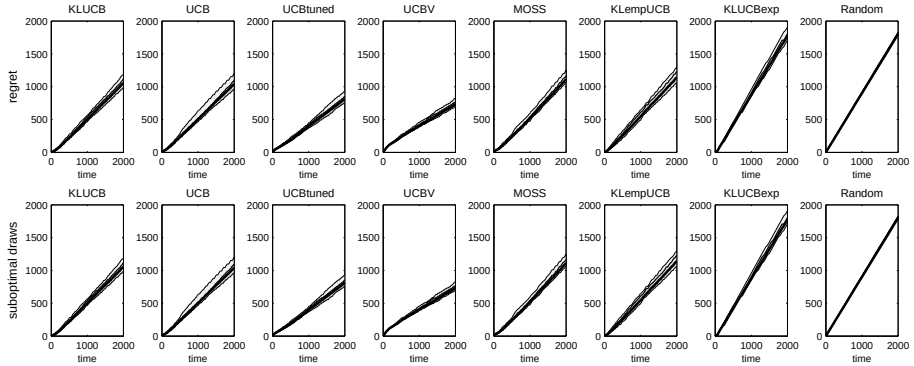


(d) Defense Every 5

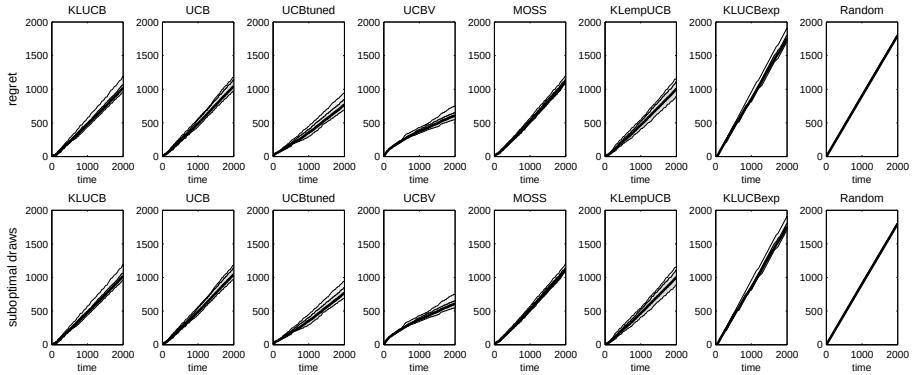
Figure IV.4: 1-1 9-0; Poisson Variance of 1; No Discount; Hide Max



(a) Poisson Variance of 1

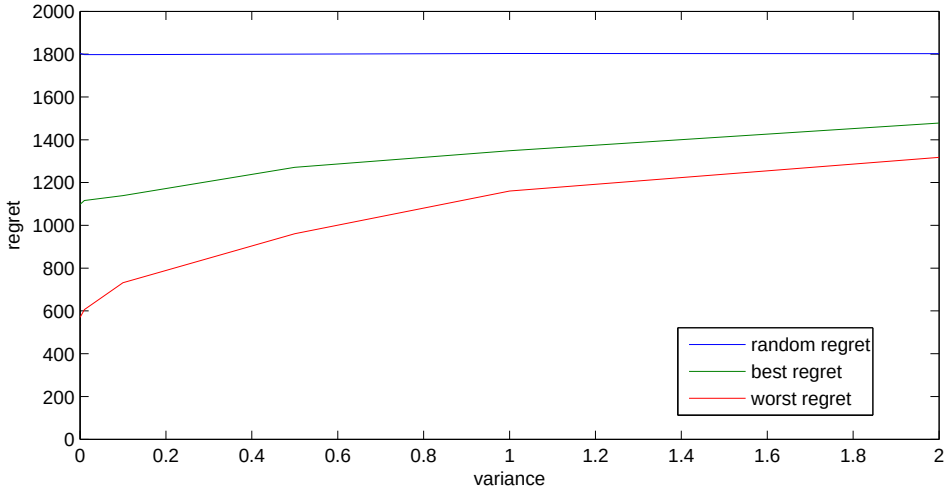


(b) Poisson Variance of 0.1



(c) Poisson Variance of 0.01

Figure IV.5: 1-1 9-0; No Discount; Hide Max; Defense Every 50



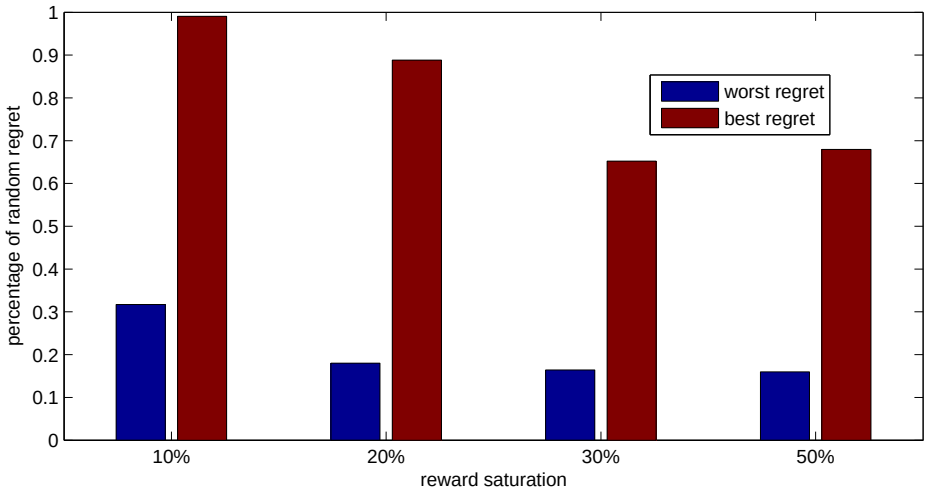
(a) Variance 0, 0.1, 0.2, 0.3, 0.5

Figure IV.6: 1-1 9-0; No Discount; Hide Max; Defense Every 50

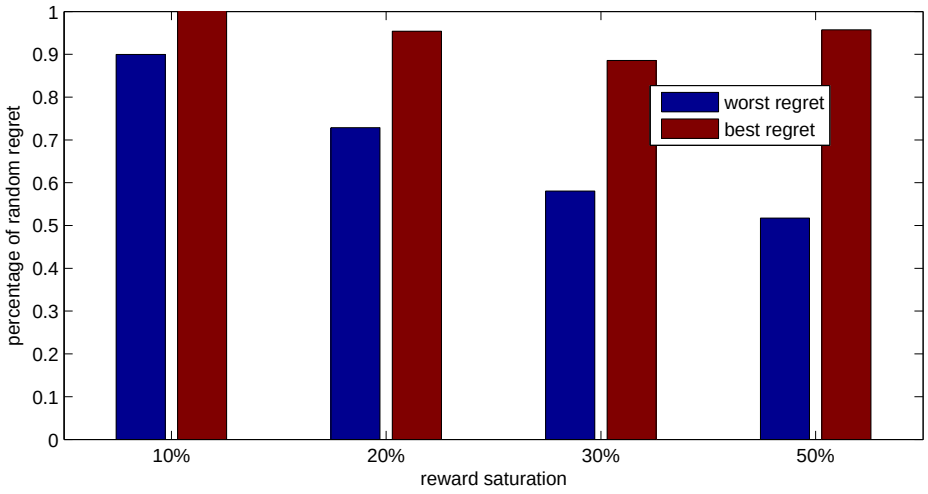
to the policy that had the lowest regret against our defense, while the Best Regret refers to the one with the highest regret (excluding the exponentially tuned KL-UCB, since it uniformly performed as poorly as the random policy). Looking at the figure, it quickly becomes apparent that as the variance 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 amount of variance in how much reward the attacher receives, our defense will perform even better than under static lab conditions.

Another aspect we looked into was how the saturation of rewards affected our Complete Restructure strategy's effectiveness. 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 an expected reward of 0.33, etc. The results can be seen with the raw data in Table IV.1 and scaled data in Figure IV.7.

Several things can be learned from these results. First off, it is quite clear that as the reward saturation increases, the effectiveness of our strategy decreases.



(a) Defense Every 50



(b) Defense Every 5

Figure IV.7: Poisson Variance of 1; No Discount; Complete Restructure

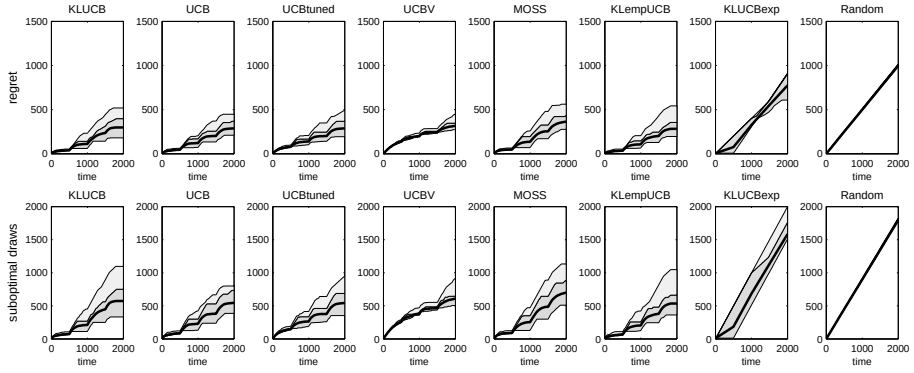
Table IV.1: Effect of Reward Saturation on Regret (Complete Restructure; Every 50,5 Turns)

Shuffle Freq	**Saturation**	**Worst Regret**	**Best Regret**	**Random Regret**
5	10%	1617	1808	1797
5	20%	582	762	799
5	30%	270	412	465
5	50%	103	190	199
50	10%	570	1780	1797
50	20%	144	711	801
50	30%	76	301	462
50	50%	32	136	200

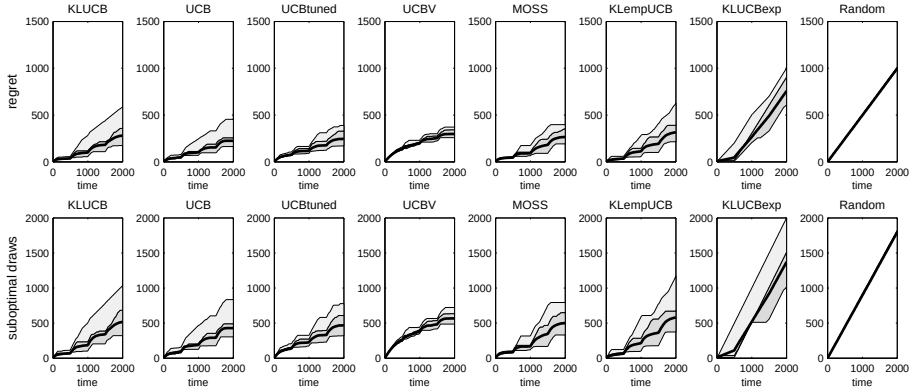
Figure IV.7 shows for the defense frequency of 5 that the regret of the policy that our defense performs the worst against goes from 90% of the random strategy at 10% saturation down to 50% of the random at 50% saturation. That's a drastic change, and not in the defense's favor. However, this is not 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.

The final experiment we conducted at with the simulation was the situation where there was an uneven distribution of rewards. In the previous example, when we increased the saturation of rewards, we gave all the arms the same payout. In this case, we gave them slightly 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 this so we could try and see any differences between the Complete Restructure and Hide Max strategies in this unbalanced situation. The results can be seen in Figure IV.8.

It quickly becomes apparent by looking at the results that there isn't really



(a) Hide Max



(b) Complete Restructure

Figure IV.8: 1-0.6 2-0.2 7-0; Poisson Variance of 1; No Discount; Defense Every 500

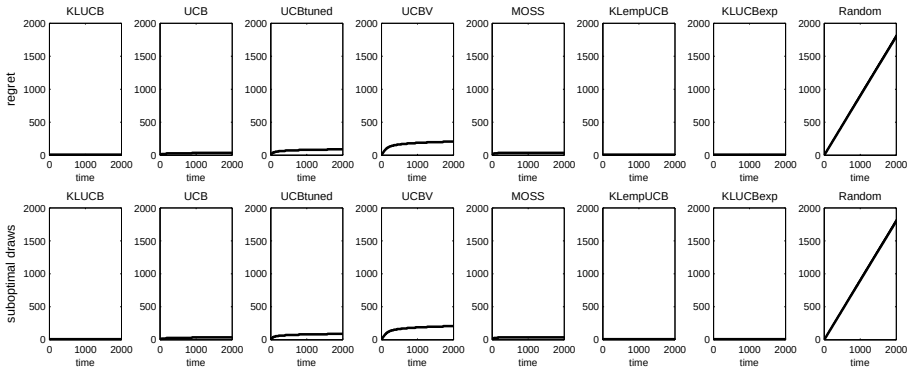
much of a difference at all. In fact, the two defense strategies are nearly identical. The only real difference seems to be the best and worst case performances (shown by the lighter grey areas), which are slightly larger for the Complete Restructure strategy, though even that could simply be the product of different random numbers generated between the runs.

The reason for this is most likely because in the situation where we compared the two strategies, there was one VM that paid out more than the others combined, meaning it was nearly guaranteed to be targeted by the attacker. Since all shuffles in both strategies are completely random, both strategies have the same likelihood of swapping a lesser paying VM or a non-paying VM into the location that the max had been in before.

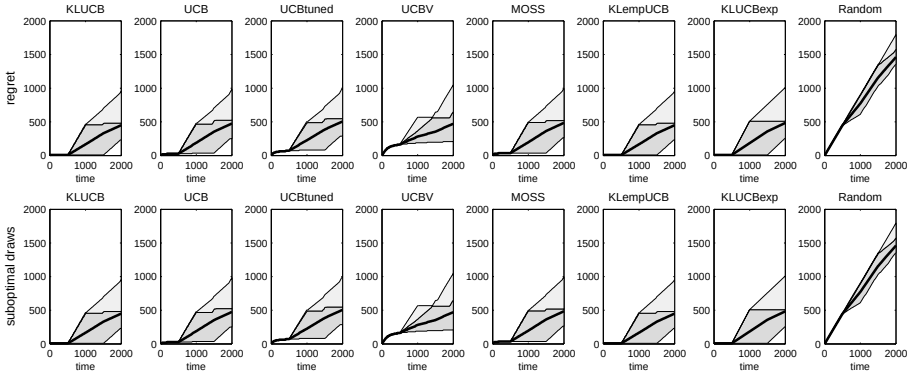
In short, it doesn't seem to make much of a difference whether you shuffle all the arms around or just the maximum valued one, at least in a case where one VM pays significantly higher rewards than all the others. Likewise, if you do not know which VM is the most desirable target, it will not hurt your effectiveness to simply shuffle them all to be safe.

Unfortunately, our Duplicate and Deactivate strategy did not fare as well. We tried doing the same as our firsts experiment with the Hide Max strategy, where the regret was calculated by Equation III.3 and all the variance in the rewards was removed, the results of which can be seen in Figure IV.9.

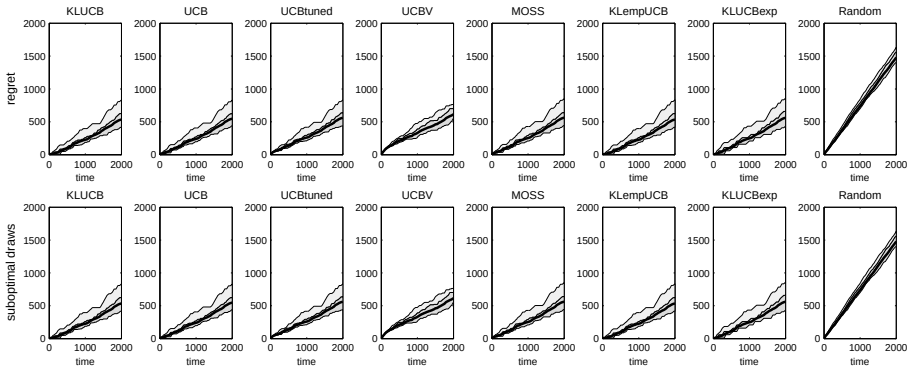
It is very quickly made apparent from the results that the Duplicate and Deactivate strategy is not as effective as the Hide Max or Complete Restructure strategies. It is, however, more effective than no defense at all. One of the most interesting things from Figure IV.9 is that the average regret seems nearly constant from changing the activated VMs 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, the easier it is to be either extremely lucky (or unlucky) for a long period of time with the randomized configuration.



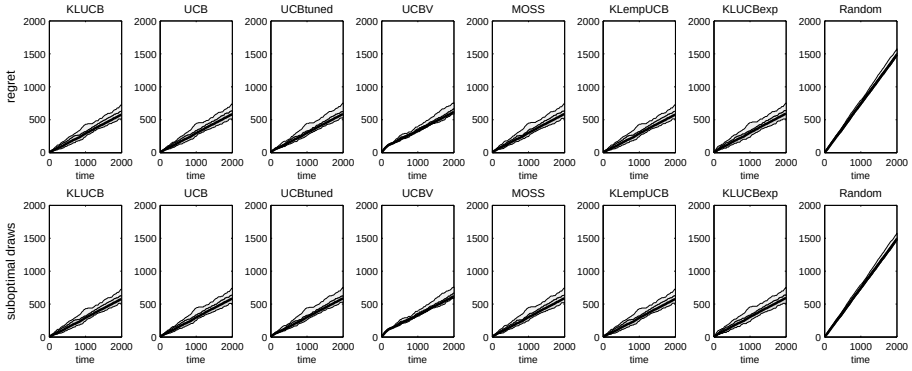
(a) No Defense



(b) Defense Every 500



(c) Defense Every 50



(d) Defense Every 15

Figure IV.9: 1-1 9-0; No Variance; No Discount; Duplicate and Deactivate

Example System

In order to collect real world data to show the feasibility of this defense strategy, we created a cloud for our use locally. Our setup was 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 Michal Jastrzebski (2015), 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. So, 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 memory to be moved is small enough that it can be done 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 transfered 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 the

Table IV.2: Time for Live Migration for Stressed Memory (256 MB total)

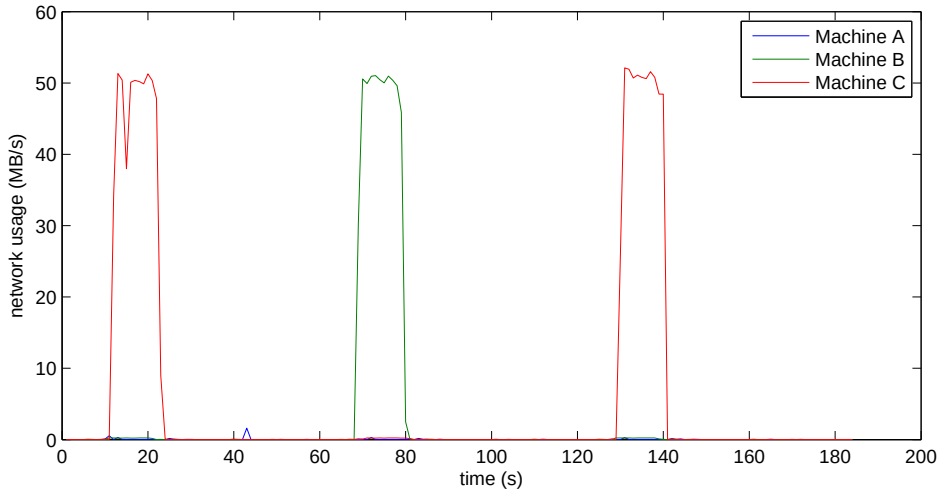
Stress (MB)	**Migration Time (s)**	**Down Time (s)**
0	22.6	2.0
16	23.4	2.5
32	22.8	2.7
64	27.5	3.3
128	29.4	3.2
200	29.4	2.8

"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. Our results can be seen in Table IV.2.

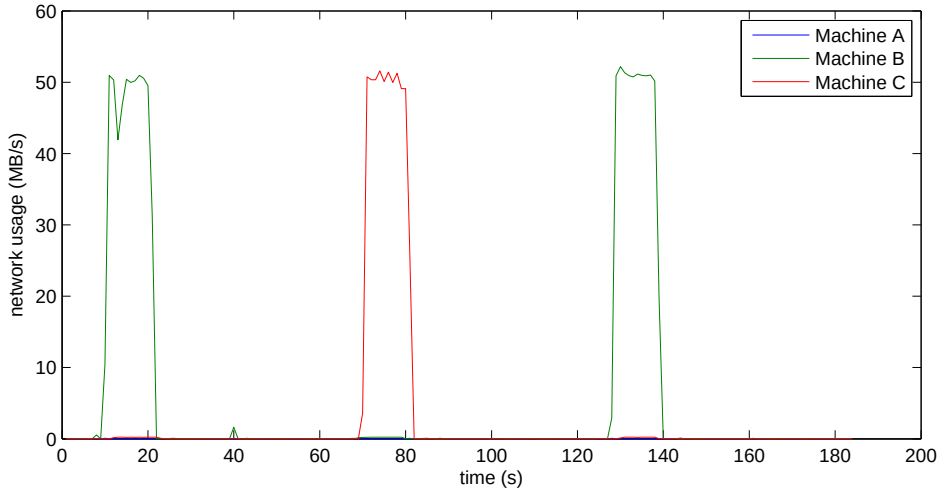
Looking at these results, we can see that as the stress on the memory increases, so too does the length of time it takes to complete the migration. However, even with the stress levels being as high as 200MB, most of the memory of the VM, the migration took less than 30 seconds to complete on average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

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 IV.10.

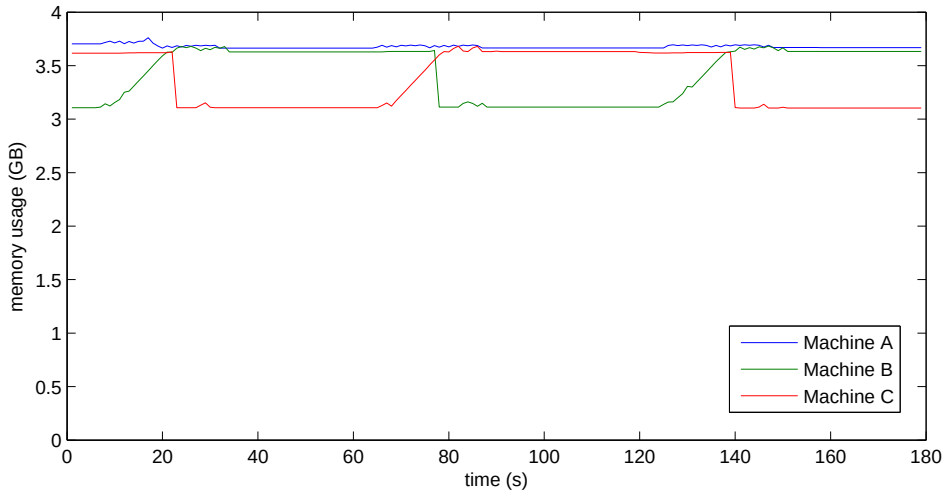
What these results show is that during the migration process, there is a sudden flurry of activity, and as soon as the migration has completed, the system returns to a more stable state. A large amount of bandwidth is used to transfer the data,



(a) Outbound Network Traffic



(b) Inbound Network Traffic



(c) Memory Usage

Figure IV.10: Physical Nodes During Unstressed VM Migration

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 in 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.

A more thorough examination of the downtime and network usage during a live migration can be found from the authors at cima and Bohnert (2014). We followed their example to collect the timing data for the migrations. They found that their down time was less than 1 second for all sizes of virtual machines 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.

V. CONCLUSION

The result of this thesis is very promising. Through our theoretical analysis, we found that our MTD is indeed effective against the various types of MAB attack policies. Obviously, the more frequently the system is changed, the more effective the defense is. What is especially encouraging is that if we make changes often enough, all of the policies become indistinguishable from a random strategy, meaning that the attacker has no hope of ever really learning anything useful about the system we are defending.

Another positive result is how the introduction of a discount factor made no change to the effectiveness of our defense, and how we found that the addition of variance into our data only served to improve our defense. In other words, the more varied the data, the better our defense performs. This is because our defensive strategy to move everything around takes advantage of the variance already in the data and adds even more unpredictable variance on top of it. The MTD policies are written in part to attempt to compensate for the variance that they assume is in the data, so when we subtly change what they are expecting to find, they may not notice immediately that they are suddenly receiving sub-optimal rewards.

One final set of conclusions we can take from our theoretical experiments is that although our MTD strategy gets less effective when there are more targets that give rewards to the attacker, we still found that it was still better to use our defense than to completely ignore it, even in cases where the arms did not pay out with equal expectations. Clearly doing something, *anything*, to confuse and invalidate the attacker's knowledge is better than not trying at all.

Our real world OpenStack example shows that this is a very feasible strategy to implement on actual systems. Since the time it takes to migrate virtual machines isn't very large, and there is essentially no downtime for the customer, there is no reason to not spend that little bit of time every so often and shuffle the

system a bit in order to confuse any attacks that may be happening. By noticing that 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. Security is essential in the modern world, and attacks are getting more and more clever. As we have shown, a moving target defense is one way to make those attacks less clever than they think they are.

REFERENCES

Al-Shaer, E. (2011). *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, chapter Toward Network Configuration Randomization for Moving Target Defense, pages 153–159. Springer New York, New York, NY.

Andrew Bartels, John R. Rymer, J. S. w. K. K. J. C. D. W. (2014). The public cloud market is now in hypergrowth. `https://www.forrester.com/report/The+Public+Cloud+Market+Is+Now+In+Hypergrowth/-/E-RES113365`.

Audibert, J.-Y., Munos, R., and Szepesvari, C. (2009). Exploration-exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19):1876–1902.

Auer, P., Cesa-Bianchi, N., Freund, Y., and Schapire, R. E. (2002a). The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77.

Auer, P., Fischer, P., and Kivinen, J. (2002b). Finite-time analysis of the multiarmed bandit problem. In *Machine Learning*.

Bellman, R. (1956). A problem in the sequential design of experiments. *Sankhya: The Indian Journal of Statistics (1933-1960)*, 16(3/4):221–229.

Bisong, A. and Rahman, S. M. (2011). An overview of the security concerns in enterprise cloud computing. *CoRR*, abs/1101.5613.

Bubeck, S., Perchet, V., and Rigollet, P. (2013). Bounded regret in stochastic multi-armed bandits. *ArXiv e-prints*.

Cappe, O., Garivier, A., and Kaufmann, E. (2012). pymabandits. `http://mloss.org/software/view/415/`.

Chakrabarti, D., Kumar, R., Radlinski, F., and Upfal, E. (2009). Mortal multi-armed bandits. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L., editors, *Advances in Neural Information Processing Systems 21*, pages 273–280. Curran Associates, Inc.

cima and Bohnert, T. M. (2014). An analysis of the performance of live migration in openstack. `https://blog.zhaw.ch/icclab/an-analysis-of-the-performance-of-live-migration-in-openstack/`.

Garivier, A. (2011). The kl-ucb algorithm for bounded stochastic bandits and beyond. In *In Proceedings of COLT*.

Gillani, F., Al-Shaer, E., Lo, S., Duan, Q., Ammar, M., and Zegura, E. (2015). Agile virtualized infrastructure to proactively defend against cyber attacks. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 729–737.

Gittins, A. J. C. and Gittins, J. C. (1979). Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society, Series B*, pages 148–177.

He, J., Dong, M., Ota, K., Fan, M., and Wang, G. (2014). Netseccc: A scalable and fault-tolerant architecture for cloud computing security. *Peer-to-Peer Networking and Applications*, 9(1):67–81.

Jain, M., Kardeş, E., Kiekintveld, C., Tambe, M., and nez, F. O. (2011). Security games with arbitrary schedules: A branch-and-price approach. In *Security and Game Theory*, pages 177–190. Cambridge University Press. Cambridge Books Online.

Kaldor, N. (1936). *Economica*, 3(10):227–230.

Kaur, J., Garg, S., Principal, R., and Gobindgarh, M. (2015). Survey paper on security in cloud computing.

Kiekintveld, C., Jain, M., Tsai, J., Pita, J., nez, O. O., and Tambe, M. (2008). Computing optimal randomized resource allocations for massive security games.

Liu, H. (2010). 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*, pages 65–76. ACM.

McCarthy, C. (2009). Twitter crippled by denial-of-service attack. `http://www.cnet.com/news/twitter-crippled-by-denial-of-service-attack/`.

Michal Jastrzebski, Michal Dulko, P. K. (2015). Dive into vm live migration. `https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/dive-into-vm-live-migration`.

Press, A. (2009). Heartland payment systems hacked. `http://www.nbcnews.com/id/28758856/ns/technology_and_science-security/t/heartland-payment-systems-hacked/`.

Reuben, J. S. (2007). A survey on virtual machine security. *Helsinki University of Technology*, 2:36.

Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc.*, 58(5):527–535.

Shetty, S. (2013). Gartner says cloud computing will become the bulk of new it spend by 2016. `http://www.gartner.com/newsroom/id/2613015`.

Subashini, S. and Kavitha, V. (2011). A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1 – 11.

Varaiya, P., Walrand, J., and Buyukkoc, C. (1985). Extensions of the multiarmed bandit problem: The discounted case. *IEEE Transactions on Automatic Control*, 30(5):426–439.

Whittle, P. (1981). Arm-acquiring bandits. *Ann. Probab.*, 9(2):284–292.

Whittle, P. (1988). Restless bandits: Activity allocation in a changing world. *Journal of Applied Probability*, 25:287–298.

Winterrose, M. L. and Carter, K. M. (2014). Strategic evolution of adversaries against temporal platform diversity active cyber defenses. *CoRR*, abs/1408.0023.

Winterrose, M. L., Carter, K. M., Wagner, N., and Streilein, W. W. (2014). Adaptive attacker strategy development against moving target cyber defenses. *CoRR*, abs/1407.8540.

yves Audibert, J., Est, U. P., and Bubeck, S. (2004). Minimax policies for adversarial and stochastic bandits, in. In *Proceedings of the 22nd Annual Conference on Learning Theory, Omnipress*, pages 773–818.

Zhuang, R., DeLoach, S. A., and Ou, X. (2014). Towards a theory of moving target defense. In *Proceedings of the First ACM Workshop on Moving Target Defense*, MTD '14, pages 31–40, New York, NY, USA. ACM.

Zhuang, R., Zhang, S., DeLoach, S. A., Ou, X., and Singhal, A. (2012). Simulation-based approaches to studying effectiveness of moving-target network defense. In *National symposium on moving target research*.