The AAAI-17 Workshop on
Artificial Intelligence for Cyber Security
WS-17-04

Adapting Honeypot Configurations to Detect Evolving Exploits

Marcus Gutierrez, Christopher Kiekintveld
mgutierrez22@miners.utep.edu and cdkiekintveld@utep.edu
University of Texas at El Paso
500 W University Ave
El Paso, Texas 79902

Abstract

Honeypots are fake resources that gain value in being probed and attacked. They deceive network intruders into detailing the intruder's behavior and the nature of an intended attack. A honeypot's success relies on the quality of its deception and the perceived value to the attacker. In this paper, we emphasize the latter. We model a repeated game where a defender must select from a list of honeypot configurations to detect an adversary's attack. The adversary's attacks each contain their own unique value function and required features to execute an exploit. Each exploits "evolves" by having its value decreases with the number of detections and new attacks may be added to the adversary's arsenal as the game progresses. We show that this model demands the defender to act strategically, by showing the adversary can exploit naive defense strategies. To solve this problem, we leverage the Multi-Armed Bandit (MAB) framework, a class of machine learning problems that demand balance between exploration and exploitation.

Introduction

Cybersecurity is a rapidly evolving battleground between offensive and defensive cyber capabilities. As new exploits are discovered and used to launch new types of attacks, cyber defenders must detect these changes and develop countermeasures as quickly as possible. The *exploit detection problem* is therefore of critical importance to maintaining effective cyber defenses. Detecting any type of exploit attempt in the wild provides valuable information about attacker capabilities and patterns of behavior, but a particularly important case is detecting novel exploits that have not been observed before. Such exploits are known as *zero-day* attacks and they have the potential to be especially damaging because there may be no known defense (e.g., a software patch) when the exploit is first detected (Bilge and Dumitras 2012). Zero-day exploits can cause very significant damage in the time before they are detected and before effective countermeasures are developed.

Intrusion Detection and Prevention Systems (IDPS) are commonly used to detect suspicious activities based on traffic analysis, packet inspection, and other methods. However, these systems are often ineffective at detecting zero-day attacks because they rely on hard-coded rules or pattern matching based on known attacks. A common tactic that is used to improve detection capabilities especially for unknown types of attacks is the use of fake network hosts or services, known as *honeypots* (Spitzner 2003). Honeypots are designed to look like legitimate computers, servers, or services, but since the network administrator knows that they are decoys any attempt to interact with one of these hosts can be unambiguously identified as malicious, even if it does not match known attack patterns. In addition, honeypots are set up to carefully monitor and log all of the interactions with an attacker, providing valuable information to diagnose an attack and develop countermeasures such as new signatures or patches. A large variety of different types of honeypots exist, but in this paper we focus on honeypots that masquerade as servers on a network. We examine how these deceptive hosts can improve intrusion detection for both known and zero-day attacks.

Honeypots also have limitations. They can only log activity if attacked directly (Spitzner 2003) and an attacker is likely to detect a poorly designed honeypots and actively avoid it. For example, a honeypot running old, highly vulnerable software/services pretending to contain highly valuable data (such as credit card information) is unrealistic and will most likely be avoided by attackers. There are two broad categories of honeypots depending on the level of realism and interaction they support: low-interaction and high-interaction. Low-interaction honeypots mimic small services and require little overhead or upkeep. A low-interaction honeypot may act as the log-in service for a File Transfer Protocol (FTP) server, but might not actually implement any features of FTP (such as authentication). A high-interaction honeypot will go to greater lengths to simulate real features, such as mimicking all features of an FTP server of a specific version. High-interaction honeypots are more believable, but are much more costly to design and maintain.

In this paper we address one of the most important limitations of honeypots for detecting exploits, particularly novel ones. Any honeypot must represent itself as a particular type of host or service (e.g., a server running a particular version of an operating system, with certain open ports, installed software packages, etc.). This means that any particular honeypot presents a limited attack surface, and will only attract attention from a certain subset of potential attackers. For example, an attacker seeking to exploit a new vulnerability

in the Android operating system will not choose to interact with a Windows file server. To detect a wider variety of attackers and exploits, we would ideally like to deploy all (or at least many) possible configurations of honeypots to cover many facets of the full network attack surface. However, deploying high-quality honeypots is costly.

To address this challenge we propose a new model for the exploit detection problem that is based on a repeated game. In each round of the game, the defender choose a set of honeypot configurations to deploy (sometimes called a honeynet). Each configuration exposes a specific attack surface, and there is an exponential number of possible configurations due to different combinations of features. In each round the attacker will also choose to attack the network using a specific exploit from a set known only to the attacker. The exploits have different levels of severity, and require different sets of features to be used to target a victim system. In addition, the set of exploits available to the attacker evolves over time to model the lifecycle of attack exploits (Frei et al. 2006; McQueen et al. 2009). The goal for the defender is to maximize the total value of the exploits detected over time.

We propose several methods for optimizing the defender's selection of honeypots to maximize the value of exploits detected. These methods are based on online learning algorithms (UCB and EXP3) developed for the multi-armed bandit problem (MAB). We show that the basic learning methods are more effective than several baseline approaches, including the static "set it and forget it" approach often used in the real world. However, the underlying problem in this case has a complex combinatorial structure, so we also introduce modified versions of UCB and EXP3 that take advantage of this structure. We demonstrate empirically that these methods offer faster learning, and significantly improve the ability of the system to detect a dynamically evolving space of exploits used by an intelligent attacker.

Motivating Example: KFSensor

To aid in building a realistic model, we focus on KFSensor, a commercial intrusion detection system that deploys high-interaction honeypots on a network (Focus 2003). Though our model should be applicable to other types of honeypot software such as HoneyD (Provos 2003), we have selected KFSensor for its highly customizable and easy-to-use nature. In addition, KeyFocus, the developers of KFSensor, still maintain the software and provide updates regularly, offering a current and relevant look into honeypot selection decision making.

At its core, KFSensor operates by simulating high level systems services in a network. It achieves this through the use of Sim Servers, simulated servers that define how KFSensor should act in order to masquerade as a real server. KFSensor imposes no restriction on the amount of Sim Servers that can run in a given time, but the network defender must maintain a degree of believability, limiting the amount of playable Sim Servers and combinations. Creating Sim Servers for protocols that leave open ports that are not used on your network's real servers or utilizing numerous Sim Servers of the same type may hint to the attacker as to which hosts on the network are honeypots.

KFSensor splits Sim Servers into 2 types: Sim Banners and Sim Standard Servers. Sim Banners offer a basic receive/response service with little room for complexity. For instance, a Sim Banner can easily simulate a basic echo server that receives some text and responds with a copy of the text. This differs from the Sim Standard Servers, which offer a more sophisticated imitation of real servers. These have a server type (e.g. Telnet, FTP, HTTP, MySql, etc.), port number, description, version number, and more. All of these features of a Sim Standard Server are customizable. KFSensor currently supports a wide variety of the most commonly seen servers as Sim Servers, including, but not limited to, HTTP, Telnet, FTP, SSH, and MySQL servers.

To capture network traffic, the network defender defines listen definitions. These are instructions for what actions KFSensor should perform on the designated open or bound port. KFSensor offers 3 types of actions when the port listener receives a connection: close the connection, perform actions determined by the selected Sim Server, or connect to a native service.

At the highest level of decision making in KFSensor lies scenario selection. A scenario is a collection of listen definitions. The network administrator can only run 1 active scenario at a given time, but switching between existing scenarios happens quickly and effortlessly. In essence, the network defender using KFSensor must decide between the combination of deployed servers in a given round. Selecting the active scenario happens quickly, but the development of intricate and believable scenarios will cost significant time to develop. As mentioned previously, crafting a scenario that has every port open may capture some activity, but an intelligent adversary will take note of this and actively try to avoid the deployed honeypots. In addition, detecting an attempted attack on an SQL server when the network in question contains no SQL server may not provide value. Since well-designed scenarios will offer a limited set of entry points into the network, the network defender will need to select an appropriate scenario otherwise running the risk of not detecting attacks.

Related Work

Honeypots are have received significant attention since their initial description by Stoll in the "Cuckoo's Egg" (Stoll 2005) and Berferd in "An Evening with Berferd" (Cheswick 1992). Pasman evaluates various virtual honeypot software (Pasman 2007). Bringer et. al. provides a more recent and extensive survey on the latest honeypot software and analyzes future trends (Bringer, Chelmecki, and Fujinoki 2012). Most honeypot work emphasizes how to build more believable and deceptive honeypots, ranging from general to specific domains, whereas we focus on honeypot selection in this paper.

Pibíl et. al. perform a similar approach as the one taken in this paper, but take a game-theoretic approach (Píbil et al. 2012). They look at the network and problem more statically. In our case, due to the lack of assumptions about the adversary and its arsenal of attacks, we take an online machine learning approach, where we discover the adversary's targets and behavior over time. Pibíl et. al. expand upon their

work where they formalize their game models and add to them, such as solving their honeypot selection game when the attacker utilizes attack graphs, series of steps to execute a chosen attack (Kiekintveld, Lisỳ, and Píbil 2015)

Model

We now introduce our model in more detail. An attacker and defender play a repeated game where the defender selects a honeypot configuration (e.g., a KFSensor scenario) in each round. The defender must select from a list of K_d pre-defined scenarios. Each scenario contains a collection of hand-crafted listen definitions. We assume all listen definitions connect to pre-defined Sim Servers. If a listen definition closes after a connection, KFSensor will lose the details of an attack. Furthermore, if listen definitions connect to native services, the attack may cause damage to the network as a whole. Therefore, we assume that all scenarios involve a combination of various Sim Servers. This will provide details about the entirety of an attack.

KFSensor offers an enormous amount of possible configurations for a single listen definition/Sim Server connection, including server name, server type, port number, version, description, responses, and services, leading to an astronomical amount of different configuration combinations. To simplify this concept, we define a honeypot as the connection of a listen definition to its designated Sim Server. So, we now say that a KFSensor scenario contains a combination of honeypots. To represent these combinations abstractly in our model, we use a binary vector of features to represent a possible honeypot configuration. We define D as the set of all scenarios/actions available to the defender in each round, where $D_i \in \{0,1\}^f$, such that $1 \leq i \leq f$ and f is the total number of existing honeypot configurations in the collection of pre-defined scenarios. A scenario is said to contain honeypot j if $D_i^j = 1$ and to not contain honeypot j when $D_i^j = 0$ where $1 \leq i \leq K_d$ and $1 \leq j \leq f$. In each round, the defender must select a scenario D_i to add to the network. If the adversary's chosen attack requires a server that the selected scenario emulates, then the defender detects the attack and gains positive utility.

The defender's ultimate goal is to protect its network from the adversary, while the attacker's goal is to deploy attacks that exploit vulnerabilities on the existing network. The defender could play a scenario that contains an HTTP server, but if the network does not have an HTTP server, this provides little value, and similarly for the attacker. We define the network similarly to the definition of KFSensor scenarios. The network N is the set of all hosts on the network, such that $N_s \in \{0,1\}^f$ represents a single host on the network where $1 \leq s \leq K_n$ and K_n is the total number of hosts on the network. Since a host can be viewed as a server with one configuration, we note that $|N_s| = 1$. Also note that we bound the set of total host configurations by f. If an attack were to target a host on the network and the set D was incapable of detecting such an attack (perhaps the attack is for the newest version of MySQL that is not represented in the set D), we view this as an issue with developing new scenarios, and outside of the scope of our game. Unlike the scenarios in D, every host N_s has some value associated with it, defined by a value function HostVal(N_s). These values will determine the utility received by the defender and adversary in each round.

We now define the adversary's role in the game. The adversary starts with a set of attacks A. We define $A_u \in \{0,1\}^f$ to represent the required configuration features of a server for attack u where $1 \leq u \leq K_a$, such that K_a is the total number of attacks available to the adversary. Often attacks consist of multiple stages, meaning that the attack exploits several vulnerabilities in order to achieve some ultimate goal. In other words, an attack may require multiple server features (or even multiple servers) in order to execute. For example, a complex attack may require an Apache HTTP server v2.0 and a FileZilla FTP server v0.9.59 to fully execute. In order for the defender to successfully detect the adversary's selected attack, the chosen scenario needs to contain a honeypot configuration for every required configuration feature of the attack. In other words, if $A_u \cdot D_i = |A_u|$, then the defender detects the attack. Similar to how we restrict hosts in the network N, note that we bound the required configuration features of attacks in A to f features, because an attack that bypasses all pre-defined scenarios is out of scope of our current model.

Not all attacks have the same purpose and and level of severity. The National Institute of Standards and Technology uses the Common Vulnerability Scoring System (CVSS) to assess the severity, difficulty of implementation, and impact of exploits (Mell, Kent, and Romanosky 2007). We assign a unique utility function to different attacks, consistent with the real-world use of a scoring system like CVSS to provide risk assessments. We use functions because exploits inherently do not have static values over time. As an exploit becomes older and more commonly seen, awareness and defenses increase to mitigate this exploit from causing the same degree of harm it might have done when it was first discovered. To model this dynamic life cycle of an exploit, we restrict the value functions of each attack to monotonically decreasing functions with respect to the number of detections by the defender. This captures the spirit of a zero-day attack, as an unknown zero-day attack will provide its most value in its infancy. As the number of detections increase for each attack, the value for future detections will decrease as potential patches for the vulnerability being exploited by the attack will arise. Every attack has a unique value function $AttackVal(A_u)$. For instance, a SQL Injection attack will provide high value while undetected for its severity, but may be patched quickly in the system when revealed, leading to low value for future detections. Meanwhile, a brute force password attack may provide little value to begin with, as little can be done to prevent it, but may only decrease slightly, maintaining consistent value in future detections.

The issue with these monotonically decreasing value functions is that as time $t \to \inf$, the value of all attacks such that

$$AttackVal(A_u) \to 0$$

and eventually become worthless to both the defender and adversary. To account for this, at the end of each round t, the

adversary may discover a new attack with some probability p. This new attack will have a randomly-generated set of required features and a randomized attack value function with probability p. This creates a realistic model of dynamically evolving cyberattacks. As the defender learns of attacks and detects them repeatedly, learning how to patch them, new attacks are being developed and added to the adversary's arsenal of attacks A, also increasing the total number of attacks $K_{a,t+1} = K_{a,t} + 1$ with probability p.

To summarize, in each round t, the defender must select a scenario D_i and the adversary selects an attack A_u and both are rewarded a utility based on three factors: if D_i can detect A_u, the current value of the attack A_u, and the total value of all the hosts in the network N affected by the attack. At the end of each round, a new attack has a probability of being added to the adversary's collection of attacks. In this game, the defender must protect the hosts on the network and the various attacks employed by the attacker, particularly protecting from known, dangerous attacks, while still exploring the possibility of potentially, new, dangerous attacks. In essence, the defender must balance exploiting the scenarios that defend from known dangerous attacks, while exploring scenarios that may offer attack surfaces that detect new, unknown attacks. To solve this game, we look to a machine learning problem framework known as the Multi-Armed Bandit Problem attackers may use.

Attacker and Defender Algorithms

In this section we detail the adversary and 5 defender solutions used in the experiments.

Adversarial Attacker

We model our attacker model using the "Adversarial Attacker" from Klima et al. (Klíma, Lisỳ, and Kiekintveld 2015), which is based on fictitious play. This attacker determines a value for each exploit by keeping a running average of the utility values from previous play, adding in the value from each new round as it is played and discounting the previous value. The actions are selected proportionally based on their values. This policy gradually moves towards playing actions with higher values with a learning rate λ. This style of agent provides a general decision making strategy that balances potential reward with historical gain, but with a gradual learning rate. We can think of this as either a single agent that learns from experience over time, or a collection of attackers that gradually learns based on shared knowledge. In cybersecurity, attacks typically happen in campaigns with the same type of attack. When a new vulnerability is discovered, many hackers exploit the vulnerability with varying attacks until patches come out, leading to gradual shifts in vulnerability use instead of drastic and erratic shifts. Our Adversarial Attacker assesses each attack u's Utility U_u in time t using the formula:

$$U_u(t) = AttackVal(u, t-1) + c * \frac{AttackTotal(u, t-1)}{t}$$

$AttackVal(u, t)$ provides current value of attack u, depending on its own generated monotonically decreasing value function and the number of defender detections. Attacks provide their highest value when they have no defender detections, otherwise known as zero-day status. c is a caution parameter that determines how important the history of the attack is. For our experiments, we used a caution value $c = 1.0$, such that the value of the attack and its past success are roughly worth the same. The $AttackTotal(u, t)$ provides the total rewards received from attack u.

Algorithm 1 Adversarial Attacker

Input: λ, c
Initialization: $P_0(u) = \frac{1}{K_a}$
for $t = 1, 2, 3, \cdots$ **do**
  for All Exploits $u = 1, 2, \cdots, K_a$ **do**
    Evaluate $U_u(t)$
    $M_{u,t} = 1 \iff u = \underset{u*}{argmax}(U_{u*}(t))$ else, $M_{u,t} = 0 \iff u \neq \underset{u*}{argmax}(U_{u*}(t))$
    $P_t(u) = (1 - \lambda) * P_{t-1}(u) + \lambda M_{u,t} U_u(t)$
  Play Attack $U \sim P_t$
  Observe Reward

This attacker differs from the Adversarial Attacker described by Klima in that it is not deterministic. Instead, this agent uses a distribution that it updates according its beliefs and randomly selects from this distribution each round to play an attack. Note that the learning rate λ affects how quickly the attacker shifts towards the best response attack. We implement this learning rate to simulate the collective knowledge and actions of hackers and malware. When hackers discover a new vulnerability, repeated exploits of the vulnerability from different hackers and malware may be launched. Similarly, if a new worm is released, it will jump from system to system and may attempt to breach a network multiple times from many of its affected hosts, even after defensive patches are released.

Naïve Defense Strategy

The simplest defense follows the "set and forget" mentality. The defender chooses a honeynet scenario to play at the beginning of the game and purely plays this scenario for the remainder of the game. This limits the defender's possible attack surface coverage drastically and we aim to show that even simple scenario selection changes provides a stronger strategy and better coverage to detect attacks. For the remainder of the paper, we refer to this defense strategy as a Pure strategy.

Algorithm 2 Pure Defender

Initialization: Select random arm i s.t. $1 \leq i \leq K_d$
for $t = 1, 2, 3, \cdots$ **do**
  Play Arm i

Random defense strategies should improve over the Pure strategy. A Uniform Random defense agent will equally distribute, but randomize its playing of each honeynet scenario, making it impossible to fully predict, but still exploit. Next

is the Fixed Random agent that randomizes its distribution of random scenario selections. Similar to Uniform Random, this agent offers a random, but exploitable defense, as the attacker can determine the random distributions then minimize the expected detection.

Multi-Armed Bandits

MABs target a specific type of problem: balancing exploration vs exploitation. This translates naturally to the high-level problem in our model. Should the defender deploy honeynet scenarios that detect well-known, dangerous exploits, or explore other scenarios to try to detect new zero-day attacks? The term Multi-Armed Bandit comes from a gambler playing many slot machines (one-armed bandits) and trying to maximize his payoffs as each slot machine has its own, unknown expected payoff. The gambler agent must select one slot machine, known as an action or arm, in each round and only receives a random reward from the selected arm. Because we are dealing with randomization and uncertainty, we evaluate the performance of these agents differently that usual. Instead of looking at accuracy, we utilize a metric known as regret. Regret is the difference in expected rewards between the chosen strategy vs the optimal strategy. If the gambler finds the best slot machine and plays it but loses money, he would have no regrets, because he made the best move prior. The two popular categories MAB solutions fall under are stochastic bandits and adversarial bandits.

In the stochastic setting, there exists K_d arms to choose from, each with their own independent reward distributions on $[0, 1]$. We use the most notable stochastic solution, the Upper Confidence Bound (UCB) first described by Auer et. al. (Auer, Cesa-Bianchi, and Fischer 2002). Though our defender plays against an adversary, the adversary remains somewhat controlled by the evolution of the exploits. As the defender detects exploits, the exploits lose value, eventually dying off while newly generated exploits with higher value, somewhat steering the adversary to prefer the new exploits. This may cause UCB to perform better than expected against the Adversarial Attacker.

The adversarial setting assumes that an opponent determines the expected rewards of each arm, potentially with some external randomization (Bubeck and Cesa-Bianchi 2012). We utilize the Exponentially weighted algorithm for Exploration and Exploitation (EXP3) is the most famous adversarial bandit algorithm, described by Auer et. al. (Auer et al. 1995). EXP3 differs from UCB in that it is not deterministic. At the end of each round, it develops a probability distribution function and polls a random arm from the distribution. This prevents an intelligent attack from using pattern recognition or simply knowing the algorithm. This adversarial-style assumption fits perfectly in a cybersecurity setting where hackers may discover the honeynet scenario deployment logic.

Combinatorial Bandits

In this section we address the combinatorial nature of the model. When the defender selects a honeynet scenario, it selects a combination of honeypots with varying configurations. Ideally, we could exploit the combinatorial structure of the problem as many scenarios will contain the same honeypots. If the selected honeynet scenario's FileZilla v2.0 FTP server honeypot captured the adversary, then the defender should update its beliefs about all honeynet scenarios that contain a FileZilla v2.0 FTP server honeypot.

Combinatorial MABs have received significant attention in the literature due to increased complexity and variety. Unfortunately, none of the existing combinatorial bandit methods proposed in the literature is a good fit for our model due to restrictive assumptions. Combinatorial bandits elevate the problem by allowing the player to select multiple "basic" arms, forming a "super arm", in each round. Many combinatorial bandit models restrict a specific selection of k basic arms for all "super arms", such that $k < f$. In our model, we do have this restriction; honeynet scenarios may contain differing numbers of honeypots. Furthermore, our model demands a specific style of non-linear rewards. If the defender successfully detects an attack, every deployed honeypot in the honeynet scenario receives a reward which is revealed to the defender. However, the defender may require multiple honeypots to detect some attack. In this case, the defender should update its beliefs about every honeynet scenario that could have detected the attack and not the scenarios that can only partially detect the attack.

We show that even a simple approach to converting EXP3 and UCB into combinatorial bandits provides an improvement in detection rates. When implementing EXP3 and UCB, we take the naïve approach of treating every honeynet scenario as independent basic arms on the bandit and play those, ignoring all combinatorial structure. Instead, we propose a simple approach that makes no changes to base algorithms of EXP3 nor UCB, but instead, when updating the rewards for playing the arm D_i that detected the played attack A_u, we update all super arms that are capable of detecting A_u. In other words, if the defender successfully detects an attack, we update every honeynet scenario, such that $|D_{i*} \cdot A_u| = |A_u| \forall i*, 1 \leq i \leq K_d$. We predict this simple combinatorial conversion to be suboptimal, but show in the Experiments section, that we do make improvements by exploiting the combinatorial nature of the problem.

Experiments

Our initial experiment consists of comparing all 5 defender strategies against the Adversarial Attacker with a $\lambda = 0.25$ learning rate. As a baseline, we use a defender oracle that can see the attacker's exploit distribution before selecting a honeynet scenario in order to make the optimal decision. We subtract the oracle's expected reward each round from the expected reward of a defender agent in question to obtain the defender's regret. This is metric we used to evaluate the performance of the 5 solutions.

We use four general monotonically decreasing types of value functions for the various attacks as seen in Figure 1. We start with a steep exploit, that starts with initially high value, but within the first few defender detections, drops drastically. This represents the simple to fix, but dangerous nonetheless exploits, such as an SQL Injection. We also utilize a constant exploit classification, that offers little value in detection to begin with, but does not decrease in value,

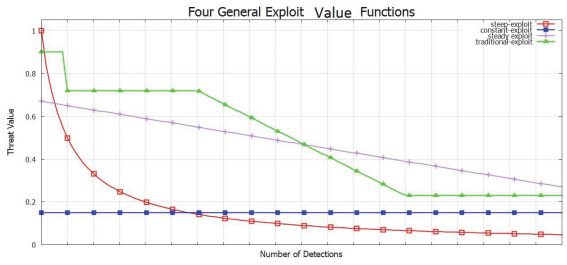

Figure 1: Example value functions for the 4 general exploit types: Steep, Traditional, Constant, and Steady.

because little can be done about them, such as brute force password attacks. Our traditional exploit function type after the vulnerability lifecycle studied by Frei (Frei et al. 2006).

The value for these type of exploits decreases in segments, depending on what stage of the lifecycle it is in. Initially, when an exploit is an undetected 0-day it provides the most value. Upon detection, when network defenders become aware of its existence but are unable to mitigate it, it begins decreasing in attack value. With each additional detection, the defender grows closer to patching the vulnerability. Eventually, the defender patches the vulnerability and the exploit provides its lowest value where it turns into a constant exploit thereafter. Lastly, we show the steady exploit, which is a linearly decreasing value that is a smoother version of the traditional exploit. Once low enough in value, the steady exploit shifts towards a constant exploit, much like the traditional.

Rewards are calculated based on a number of factors in our model. We assign a random value to each network server in the game initialization. These values are static and normalized such that the total value of the servers on the network sums to 1.0. If the defender successfully detects the adversary's attack, the attacker receives no reward, such that $R^{detected}_{A_u,t} = 0$ in round t. However, if the defender's selected honeynet scenario D_i fails to detect the adversary's attack A_u, the attacker receives a reward $R_{A_u,t}$ such that:

$$R^{\neg detected}_{A_u,t} = \sum_{s \in N} AttackVal(u,t)(A_u \cdot N_s) HostVal(s)$$

The defender's rewards are opposite of the attacker, so $R^{detected}_{D_i,t} = R^{\neg detected}_{A_u,t}$ and $R^{\neg detected}_{D_i,t} = R^{detected}_{A_u,t}$. As mentioned earlier, we use the concept of an oracle to evaluate each defender. In each round, the oracle observes the adversary's strategy and plays the arm that maximizes its expected payoff. We compare this expected payoff to the defender's expected payoff. We define the Oracle's expected utility as,

$$E[\theta] = \max_{\theta} \sum_{\theta \in D} \sum_{u \in A} X_t(\theta) * Y_t(u) * R^{detected}_{D_\theta,t}$$

and regret as,

$$Regret_t = E[\theta\,] - \sum_{i \in D} \sum_{u \in A} X_t(i) * Y_t(u) * R^{detected}_{D_i,t}$$

where $X_t(i)$ is the defender's probability for playing honeynet scenario i in round t. In the deterministic algorithms Pure and UCB, only a single honeynet scenario will be given $P_t(i) = 1.0$. $Y_t(u)$ is the probability that the adversary selects attack u. The closer the defender is to fully playing the optimal honeynet scenario that will best detect the adversary's attacks, the closer the defender moves to 0 regret. We can now measure performance through the measure of regret, with lower total regret indicating a better strategy.

In each match, the defender and attacker play for 20,000 rounds. We set the total number of possible configurations of a single Sim Server $f = 50$. In practice, we expect this will be significantly higher, but the results in Figure 2 suggest even this small number provides a compelling argument for strategically selecting honeynet scenarios. We set the number of honeynet scenarios to $K_d = 50$ where each scenario contains between 8 and 12 listeners. We feel this is well grounded, as a network defender hand-crafting 50 honeynet scenarios would provide a wide variety of attack surfaces. We have the attacker start with only $K_a = 10$ exploits, but each round the attacker has a 10% to gain a new exploit with a fresh value function starting from zero-day status.

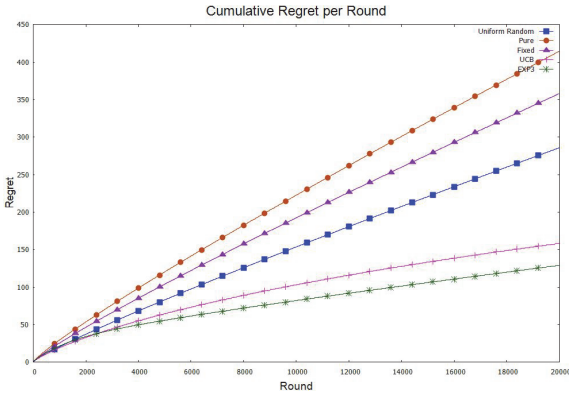

Figure 2: Cumulative Regret Over Time

As seen in Figure 2, the Pure strategy performs the worst, followed by the Fixed Random, then Uniform Random strategies. This further suggests that the "set and forget" Pure strategy offers minimal defenses and even simple changes between honeynet scenarios can improve overall network security. UCB and EXP3 perform the best, and begin to converge by the end of the game, despite new exploits being created often. This implies that both strategies strike the right balance between exploration and exploitation, despite the numerous variables and uncertainty, providing a strong case for MAB algorithms as a foundation for solving this problem.

One argument against using UCB is the fact that is deterministic. Our Adversarial Attacker does not perform any pattern recognition or make any assumptions about the defender, but if an attacker had knowledge that the defender is employing UCB, they could exploit the defender and avoid detection 100% of the time. On the other hand, EXP3 assumes that the opponent is actively trying to minimize the

agent's expected rewards. To combat this, EXP3 uses computed distributions to randomly select an arm in each round.

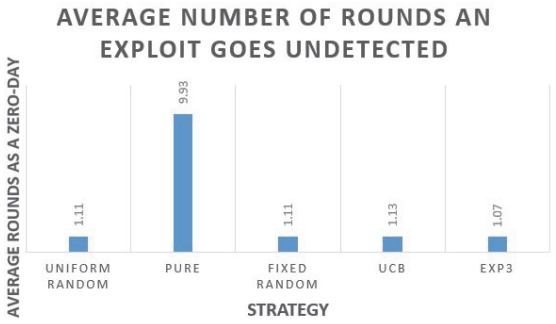

Figure 3: Average number of rounds new exploits go undetected (zero-day)

In our second experiment, we keep the same parameters, but increase the number of honeynet scenarios to $K_d = 100$ and the total number of honeypot configurations to $f = 100$. This increases the complexity and number of strategies for the defender. As seen in Figure 4, the naïve strategies perform similarly, continuously being exploited by the Adversarial Attacker. UCB and EXP3 also perform only slightly worse than their experiment 1 counterparts. In first 4000 rounds, EXP3 appears to perform worse than UCB, unlike in the first experiment 1. This is most likely caused by the increased number of scenarios and therefore, an increased amount of exploration.

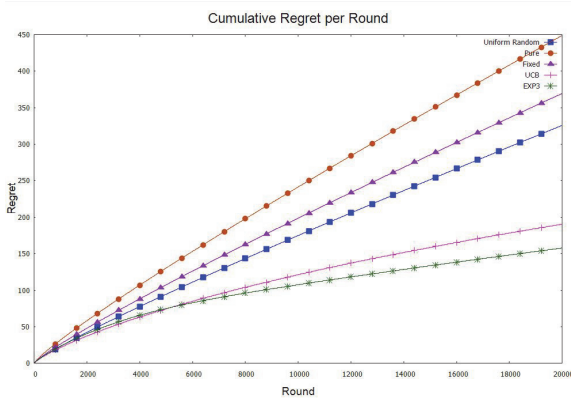

Figure 4: Second experiment, increasing the honeynet scenarios to $K_d = 100$ and honeypot configurations to $f = 100$.

Another performance metric we consider is the average number of rounds a zero-day goes undetected. This essentially measures exploration. As seen in Figure 3, nearly all defense algorithms average close to 1 round before detecting a never before-seen zero-day attack. The Pure strategy, on the other hand, averages close to 10 rounds before detecting zero-days. This is somewhat deceptive however, due to averaging. In reality, if the chosen zero-day attack bypasses Pure strategy's single selected scenario, it will always pass this scenario, maintaining its zero-day status. However, if the zero-day attack falls to selected scenario, then it will be captured immediately, going 0 rounds undetected. Though trivial, Figure 3 exemplifies why changing honeynet scenarios to expose different attack surfaces is necessary for proper network security.

As mentioned, measuring the average number of rounds an exploit goes undetected is also measuring exploration. This can be seen in Figure 3 where UCB has a higher averaged undetected rounds than Uniform Random and Fixed Random. This is because, as a deterministic algorithm developed for fixed random distributions, it exploits rather quickly after exploration. Compare this to EXP3 with the lowest average undetected rounds, because EXP3, as an adversarial setting algorithm, assumes the opponent is constantly learning from EXP3 and remains more pessimistic about the"optimal arm" than UCB does.

In the second experiment, we investigate how the attacker's learning rate impacts the MAB defenders. In the first experiment, we found that UCB and EXP3 performed exceptionally well against the Adversarial Attacker with a $\lambda = 0.25$ learn rate. Intuitively, one might argue that the slow learn rate allowed for smooth transitions between selecting new exploits. In the second experiment, we run UCB and EXP3 10 times each against an Adversarial Attacker with learn rates $\lambda = 0.0$, $\lambda = 0.20$, $\lambda = 0.40$, $\lambda = 0.60$, $\lambda = 0.80$, and $\lambda = 1.0$ to determine how the attacker's design affects the defender's performance.

Combinatorial Improvements

In our fourth experiment, we looked to exploit the combinatorial nature of the problem by implementing the simple modification detailed in the Combinatorial Bandits section. This conversion updates rewards such that every honeynet scenario that could potentially detect an attack is credited with detecting the attack. In other words, if the defender detects attack A_u, we update the rewards of every honeynet scenario such that A_u's required configuration features is a subset of the honeynet. We utilized the same game parameters described in Experiment 1 and the results are shown in Figure 5.

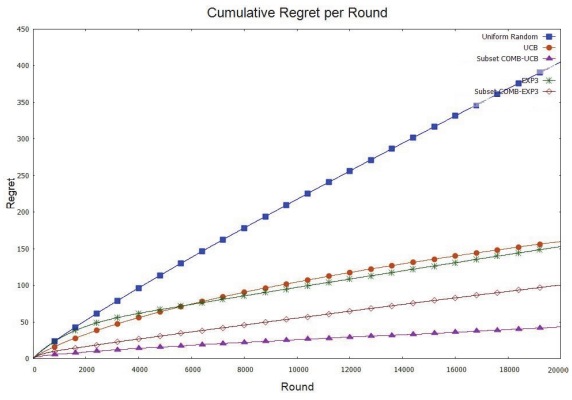

Figure 5: Cumulative Regret Over Time with the simple combinatorial versions of EXP3 and UCB.

In this combinatorial experiment, COMB-UCB and COMB-EXP3 outperform UCB and EXP3. We believe COMB-UCB performs so well due to its quick "exploitation", which it is able to do more effectively as its beliefs about other honeynet scenarios change quicker due to the combinatorial modifications. Though these simple modifications provide significant improvement, we believe there is room for improvement using new algorithms that better address all the nuances of this problem.

Future Work

In the current model we ignore network topology. Networks are typically more layered and intricate. Incorporating topology into the model would provide more a realistic challenge to the defender. Furthermore, the defender makes no use of the given network structure in the policies we have proposed. Since the reward structure is determined largely by the servers on the network, an intelligent defender could utilize this information to better detect future attacks from attackers looking to inflict the most damage on a network. Contextual bandits might provide solutions to problem. We also plan to investigate if more sophisticated ways of exploiting the combinatorial structure of the defender's strategy will yield better results. Due to our construction of the model, the rewards of the honeypot configurations are not linear, so we will need to account for this in a combinatorial bandit model.

Conclusion

We have introduced a new model for using dynamic honeypot configurations to improve exploit detection for cybersecurity. This model allows for automatically adapting the configuration of honeypots over time to better explore the attack surface and detect a wider variety of exploits, including more effective detection of zero-day attacks. We have proposed some initial methods for determining the defense strategy in these games based on online learning strategies for multi-armed bandits. A complication in this domain is that the underlying problem has a strong combinatorial structure. We propose some basic modifications to the multi-armed bandit problems suitable for this problem, but more sophisticated methods are needed to fully address the specific structure of this domain. We showed a broad set of empirical results based on simulation of this scenario. It is clear that a naïve, static honeypot selection does not provide effective coverage to detect exploits in our dynamic model. The learning algorithms demonstrate dramatically improved performance in detecting exploits, and especially the most dangerous category of zero-day exploits.

References

Auer, P.; Cesa-Bianchi, N.; Freund, Y.; and Schapire, R. E. 1995. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, 322–331. IEEE.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47(2-3):235–256.

Bilge, L., and Dumitras, T. 2012. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 833–844. ACM.

Bringer, M. L.; Chelmecki, C. A.; and Fujinoki, H. 2012. A survey: Recent advances and future trends in honeypot research. *International Journal of Computer Network and Information Security* 4(10):63.

Bubeck, S., and Cesa-Bianchi, N. 2012. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *arXiv preprint arXiv:1204.5721*.

Cheswick, B. 1992. An evening with berferd in which a cracker is lured, endured, and studied. In *Proc. Winter USENIX Conference, San Francisco*, 20–24.

Focus, K. 2003. Kfsensor overview.

Frei, S.; May, M.; Fiedler, U.; and Plattner, B. 2006. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, 131–138. ACM.

Kiekintveld, C.; Lisỳ, V.; and Píbil, R. 2015. Game-theoretic foundations for the strategic use of honeypots in network security. In *Cyber Warfare*. Springer. 81–101.

Klíma, R.; Lisỳ, V.; and Kiekintveld, C. 2015. Combining online learning and equilibrium computation in security games. In *International Conference on Decision and Game Theory for Security*, 130–149. Springer.

McQueen, M. A.; McQueen, T. A.; Boyer, W. F.; and Chaffin, M. R. 2009. Empirical estimates and observations of 0day vulnerabilities. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, 1–12. IEEE.

Mell, P.; Kent, K. A.; and Romanosky, S. 2007. *The common vulnerability scoring system (CVSS) and its applicability to federal agency systems*. Citeseer.

Pasman, D. 2007. Catching hackers using a virtual honeynet: A case study. In *6th Twente Conference on IT, Enschede*.

Píbil, R.; Lisỳ, V.; Kiekintveld, C.; Bošanskỳ, B.; and Pěchouček, M. 2012. Game theoretic model of strategic honeypot selection in computer networks. In *International Conference on Decision and Game Theory for Security*, 201–220. Springer.

Provos, N. 2003. Honeyd-a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, volume 2, 4.

Spitzner, L. 2003. *Honeypots: tracking hackers*, volume 1. Addison-Wesley Reading.

Stoll, C. 2005. *The cuckoo's egg: tracking a spy through the maze of computer espionage*. Simon and Schuster.