SHARE: A Stackelberg Honey-Based Adversarial Reasoning Engine

SUSHIL JAJODIA, George Mason University
NOSEONG PARK, University of North Carolina
EDOARDO SERRA, Boise State University
V. S. SUBRAHMANYAN, Dartmouth College

A “noisy-rich” (NR) cyber-attacker (Lippmann et al. 2012) is one who tries all available vulnerabilities until he or she successfully compromises the targeted network. We develop an adversarial foundation, based on Stackelberg games, for how NR-attackers will explore an enterprise network and how they will attack it, based on the concept of a system vulnerability dependency graph. We develop a mechanism by which the network can be modified by the defender to induce deception by placing honey nodes and apparent vulnerabilities into the network to minimize the expected impact of the NR-attacker’s attacks (according to multiple measures of impact). We also consider the case where the adversary learns from blocked attacks using reinforcement learning. We run detailed experiments with real network data (but with simulated attack data) and show that Stackelberg Honey-based Adversarial Reasoning Engine performs very well, even when the adversary deviates from the initial assumptions made about his or her behavior. We also develop a method for the attacker to use reinforcement learning when his or her activities are stopped by the defender. We propose two stopping policies for the defender: *Stop Upon Detection* allows the attacker to learn about the defender’s strategy and (according to our experiments) leads to significant damage in the long run, whereas *Stop After Delay* allows the defender to introduce greater uncertainty into the attacker, leading to better defendability in the long run.

Categories and Subject Descriptors: C.2.0 Security and Protection [D.4.6 Security and Protection]: H.2.0 Security, Integrity and Protection

General Terms: Protecting Enterprise Security, Pareto Optimality, Adversarial Modeling

Additional Key Words and Phrases: Enterprise systems, computer security, adversarial models

ACM Reference format:

Sushil Jajodia, Noseong Park, Edoardo Serra, and V. S. Subrahmanian. 2018. SHARE: A Stackelberg Honey-Based Adversarial Reasoning Engine. *ACM Trans. Internet Technol.* 18, 3, Article 30 (March 2018), 41 pages. <https://doi.org/10.1145/3137571>

Parts of this work were funded by ARO Grants W911NF11103, W911NF09102, W911NF-13-1-0421, and W911NF-13-1-0317; by ONR Grant N00014-13-1-0703; and by the Maryland Procurement Office under contract number H98230-14-C-0137.

Authors’ addresses: S. Jajodia, Center for Secure Information Systems, George Mason University, 4400 University Drive, Fairfax, VA, USA; email: jajodia@gmu.edu; N. Park, Department of Software and Information Systems, University of North Carolina, Charlotte, NC, USA; email: npark2@uncc.edu; E. Serra, Computer Science Department, Boise State University, Boise, ID, USA; email: edoardoserra@boisestate.edu; V. S. Subrahmanian, Computer Science Department, Dartmouth College, Hanover, NH, USA; email: vs@dartmouth.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 ACM 1533-5399/2018/03-ART30 \$15.00

<https://doi.org/10.1145/3137571>

1 INTRODUCTION

Though there has been much work on honey nodes and honey tokens (Bercovitch et al. 2011; Shabtai et al. 2012), most such work has either defended against specific known attacks or worked in the absence of a specific adversary model. In the real world, however, cyber-attackers have specific goals and objectives that can vary widely. Some attackers want to exfiltrate data via advanced persistent threat attacks (Tankard 2011), others want to be more “in your face” and deface websites (Aggarwal et al. 2015), while others want to deny service of one kind or the other (Mirkovic and Reiher 2004).

Lippman et al.’s security metrics overview (Lippmann et al. 2012) at IFIP 2014 classifies attackers into three models: noisy-rich (NR) attackers who keep exploiting vulnerabilities until they compromise a network, stealthy attackers who try the single attack having the highest probability of success, and random attackers who try a single attack at random. Noise-rich attackers pose the biggest risk, as they do not count on a single attack succeeding but are determined enough to hit a given target network over and over again. The recent attacks on the US insurer Anthem (formerly WellPoint) and the US Office of Personnel Management (OPM) during the 2014–2015 time frame fall within the noisy-rich attack category. For instance, the Anthem attackers were within the Anthem network for over 7 months¹ before detection, while the OPM attackers were inside for over a year² before detection. During this time, the attackers were probing and attacking the network in many ways in a manner consistent with the “noisy-rich” model.

In this article, we consider an enterprise network targeted by NR attackers where we (the defender) know how network nodes are interconnected and what software is present on each node. Defenders usually know the structure of their enterprise network and have a reasonable idea of the known vulnerabilities in their network (e.g., through penetration testing). In Section 2, we leverage the National Vulnerability Database (NVD) from NIST (NIST 2017), which tells us what vulnerabilities are present in what software (also available through Mitre’s Common Weakness Scoring System (The MITRE Corporation 2011), and we define the concept of a system vulnerability dependency graph (SVDG). Intuitively, an SVDG tells us how an attacker can navigate through an enterprise network and captures which vulnerabilities must be exploited prior to exploiting a new vulnerability.

We model the attacker–defender scenario via the well-known Stackelberg game paradigm (Başar and Olsder 1998) that has been extensively used in the field of security games (Korzhyk et al. 2011; Dickerson et al. 2010; Paruchuri et al. 2008; Kiekintveld et al. 2009b; Pita et al. 2008), including deployment for ensuring the security of Los Angeles International airport. Unlike classical two-player games where players make moves in a given order and in rounds, Stackelberg games make many simplifying assumptions. In particular, they assume that one player is the leader and the other is the follower. The leader must commit to his or her strategy and the follower then takes actions that are optimal for him or her, given the strategy of the leader. In this article, the defender is the leader, while the attacker is the follower. The defender has two broad classes of actions: inserting honeypots and honey vulnerabilities into the network to attract the attacker and removing vulnerabilities (e.g., through patching or deactivation of software). Stackelberg games are appropriate here for two reasons. First, the defender cannot keep reconfiguring the network, because designing realistic honeypots takes time and effort—and so does patching. Second, the attacker can make many, many moves before the defender even realizes the moves have been made. As a consequence, the traditional two-player game model is not appropriate, which is why we, and many others, have used Stackelberg games for security applications.

¹<http://krebsonsecurity.com/2015/02/anthem-breach-may-have-started-in-april-2014/>.

²<https://www.wired.com/2016/10/inside-cyberattack-shocked-us-government/>.

Assuming that the defender has chosen a strategy (introduction of apparent vulnerabilities through the introduction of honeypots and removal of vulnerabilities via patching), the attacker responds by choosing one of two types of actions: scanning a node for potential vulnerabilities and exploiting those vulnerabilities. Section 3 defines the NR-attacker’s behavior via a state transition function involving these two types of actions. Note that the “exploit” action is actually a shorthand for virtually every attack in use today, because it is parameterized by a vulnerability. Almost all exploits on the market today exploit a vulnerability, and hence, our “exploit(v)” action, which takes a vulnerability as input, characterizes virtually all types of attacks carried out today. We show how the state transition function captures both the changes in the NR-attacker’s knowledge of the network as he or she executes these two types of operations, as well as allows him or her to carry out exploits. We develop a utility-theoretic model for the attacker and suggest possible utility functions that are grounded in real-world vulnerability assessments from the US National Institute of Standards and Technology’s National Vulnerability Database (NIST 2017). Our framework can also build on vulnerability assessments from other organizations such as MITRE, whose Common Weakness Scoring System (The MITRE Corporation 2011) includes specific metrics such as Technical Impact (if the vulnerability is exploited), Business Impact (if the vulnerability is exploited), Likelihood of Discovery (of the exploit by the attacker), and Likelihood of Exploit (being successfully used by the attacker). We can combine these five vulnerability scores into a single score $sc(v)$ for a vulnerability v (e.g., through linear combination). The probability that an attacker exploits vulnerability v can then be set to $\frac{sc(v)}{\sum_{v' \in \mathcal{V}} sc(v')}$.

Section 4 examines the defender’s strategy—he can either insert honey nodes and/or honey vulnerabilities (i.e., modify the enterprise network and/or the software residing on network nodes to introduce apparent vulnerabilities) as well as remove some vulnerabilities (e.g., by patching). A honey node is a node in a network that appears vulnerable and enticing yet is intended to attract an attacker into a sandbox environment where the defender can learn more about him or her while presenting the illusion that the attacker is being successful. A honey network is a network that is combined with an existing enterprise network. A honey network contains both honey nodes as well as nodes corresponding to the enterprise network so that the combination of the honey network and the enterprise network causes attractive honey nodes to be strategically placed. We model the interaction between defender and attacker as a Stackelberg game where the defender is the leader and the attacker is the follower. Stackelberg games (Kiekintveld et al. 2009a; An et al. 2012) represent the most common paradigm used for game-theoretic analysis of cybersecurity (and indeed security) applications. This is especially true when, as in our case, it is difficult for the defender to know the actions of the attacker.

We define several measures of impact on the defense of the enterprise network and show that computing expected impact is computationally intractable. We then develop a Monte-Carlo algorithm to quickly approximate the expected impact. Section 5 sets up our problem as a Pareto optimization problem and shows that checking if a particular defender strategy is Pareto optimal or not is NP-hard. We also show how to adapt genetic algorithms to solve the defender problem efficiently. Though finding an exact Pareto optimal solution is NP-hard, we show that our algorithm polynomially finds a $(1 - 1/e)$ -approximate solution by leveraging sub-modularity and monotonicity properties of an expected negative impact function. Section 6 shows how an intelligent adversary can learn about the defender’s honey nodes/vulnerabilities as he or she observes the defender’s responses. As it is impossible to precisely predict how an adversary will learn from past experience, our framework assumes that adversaries have a range of learning skills based on reinforcement learning. Based on this, we consider two strategies the defender might use: The *Stop Upon Detection* strategy stops the attacker immediately on detection, while the *Stop After Delay* strategy delays when the attacker is stopped after detection. The latter has the problem that the

attacker could potentially go on to do more damage before being stopped (though it is possible, once detected, to move him or her to a synthetic fishbowl style environment where he or she can do little harm)—but it has the advantage that the attacker does not know exactly when he or she was detected. This increased uncertainty makes it harder for the attacker to learn from the defender’s actions as he or she does not know what triggered his or her detection. Section 7 describes a detailed set of experiments that we performed on small, medium, and large enterprise networks. We specifically show that (i) even though stopping the attacker immediately on detection leads to less damage in the near term, it allows the attacker to use reinforcement learning about defender strategy that allows him or her to cause greater damage in the long term—and hence using Stop After Delay is a better strategy against intelligent attackers than Stop Upon Detection. (ii) Our optimal defender strategies allow the defender to defend against even when the attacker strategies deviate from our initial assumptions. (iii) We assess the impact of attacker’s using scan vs. exploit operations and show that performing relatively few scans is helpful against networks with lots of honey vulnerabilities, but performing lots of scans is helpful against networks that are well defended. In our experiments, we also vary the knowledge the attacker has of the enterprise network and of the vulnerabilities in it, as well as the ability of the attacker to learn effectively from the network.

The appendix contains a summary table listing the main parameters in the Stackelberg Honey-based Adversarial Reasoning Engine (SHARE) framework, along with the location where the notation is introduced, and a brief explanation of the notation. In addition, the appendix provides ways in which the parameters can be estimated.

2 THE ENVIRONMENT

In this section, we will formally define enterprise networks and software vulnerability dependency graphs. We assume the existence of some set $\mathcal{S}$ of software packages and some set $\mathcal{V}$ of vulnerabilities that have been identified. A *software vulnerability mapping* $v : \mathcal{S} \rightarrow 2^{\mathcal{V}}$ associates a set of vulnerabilities with each piece of software.

Each vulnerability $v \in \mathcal{V}$ has an associated $\text{impact}(v)$ score that is the impact of vulnerability v on the enterprise network if it is successfully exploited by an attacker and an exploitability score $\text{es}(v)$ that quantifies how easily an attacker can exploit the vulnerability. The impact of a vulnerability can be measured in many ways. For instance, NIST’s National Vulnerability Database (NIST 2017) has an “impact” subscore that measures the impact of a vulnerability if exploited by an adversary as well as an “exploitability” score. MITRE’s Common Weakness Scoring System (The MITRE Corporation 2011) assigns a quantitative measure to the impact of a vulnerability.

We now define an enterprise network formally.

Definition 2.1 (Enterprise Network). An *enterprise network* $\mathcal{EN}$ is a triple $\mathcal{EN} = (N, E, \text{compr}, \text{PER})$ where (I) N is a set of nodes, (II) $E \subseteq N \times N$ is a set of edges, (III) $\text{compr} \subseteq N$ is the set of compromised nodes, and (IV) PER is a subset of N called *perimeter nodes*.

compr represents the set of all nodes that are already compromised in the network. In most real-world enterprise networks, some of these nodes are “external facing” and accept connections from nodes outside the network. These are often called “perimeter” or “edge” nodes and are denoted by the set PER above. The attacker starts from one of these edge nodes, which are the ones usually compromised. *Throughout this article, we will abuse notation somewhat by assuming that PER is arbitrary, but fixed, and, hence, we will often write an enterprise network $\mathcal{EN}$ as just the triple $\mathcal{EN} = (N, E, \text{compr})$.* The following simple example shows a toy enterprise network.

Example 2.2. Figure 1(a) shows a toy enterprise network in which node n_1 is compromised.

Fig. 1. (a) Example of a small enterprise network. (b) Example of System Vulnerabilities Dependency Graph.

We note that a system manager should know his or her enterprise network—what nodes are there, how they are interconnected, and what software runs on each node. On the other hand, the attacker does not necessarily know the enterprise network that he or she is attacking, at least at the beginning. To confuse the attacker, a defender may augment the enterprise network with some new “honey” nodes (or some software with apparent vulnerabilities) into an existing node. The first operation can be viewed as a union operation—generally speaking, it takes two enterprise networks (the enterprise network and a network of honey nodes) and merges them together.

Definition 2.3 (Union of Enterprise Network). Suppose $\mathcal{EN}^1, \mathcal{EN}^2$ are enterprise networks s.t. $\mathcal{EN}^i = (N^i, E^i, \text{compr}^i)$. The union of $\mathcal{EN}^1$ and $\mathcal{EN}^2$, denoted $\mathcal{EN}^1 \cup \mathcal{EN}^2$, is the enterprise network $\mathcal{EN} = (N, E, \text{compr})$, where (I) $N = V^1 \cup V^2$, (II) $E = E^1 \cup E^2$, and (III) $\text{compr} = \text{compr}^1 \cup \text{compr}^2$.

Note that Definition 2.3 explicitly allows the possibility that $\mathcal{EN}^1$ and $\mathcal{EN}^2$ contain nodes in common.

Attackers frequently attack networks by compromising one node (by exploiting some vulnerability in a piece of code running on that node) and then launching an attack on another vulnerability in a neighboring node (or within the same node). To capture this “hopping” behavior, we define the notion of a system vulnerability dependency graph (SVDG for short).

Definition 2.4 (System Vulnerability Dependency Graph). Given an enterprise network $\mathcal{EN} = (N, E, \text{compr})$, a system vulnerability dependency graph (SVDG for short) is a directed graph $G = (SV, EV)$, where $SV \subseteq \{(n, s, v) | n \in N, s \in \mathcal{S}, v \in \mathcal{V}\}$ is the set of system vulnerabilities (vertices) and $EV \subseteq SV \times SV$ is the set of edges.

Intuitively, an edge from sv_1 to sv_2 means that once the attacker has exploited sv_1 , he or she can exploit sv_2 . Given an SVDG $G = (SV, EV)$, we use $S(G)$ to denote the set of all “start” vertices, that is, vertices having in-degree 0, and $\text{in}(G, sv)$ and $\text{out}(G, sv)$, respectively, to denote the sets $\{sv' | (sv', sv) \in EV\}$ and $\{sv' | (sv, sv') \in EV\}$, respectively.

We note that there is a large body of work (see, for example, Jajodia and Noel (2010)) that describes how a system manager can construct the system vulnerability dependency graph of his or her organization, given that he or she has full visibility into his or her enterprise network. However, the adversary does not have this information and needs to learn it as he or she explores the network.

Example 2.5. Figure 1(b) shows an example system vulnerability dependency graph G built on top of the enterprise network of Figure 1(a).

In this case, we have $S(G) = \{(n_2, s_4, v_6), (n_4, s_2, v_2)\}$, $in(G, (n_2, s_4, v_4)) = \{(n_2, s_4, v_6), (n_4, s_3, v_4)\}$, and $out(G, (n_2, s_4, v_4)) = \{(n_4, s_2, v_3), (n_4, s_3, v_4)\}$.

Assumption. Throughout this article, we assume that given an enterprise network $\mathcal{EN}$ and an associated SVDG G , if there is an edge $((n_1, s_1, v_1), (n_2, s_2, v_2)) \in G$, then there exists an edge $(n_1, n_2) \in \mathcal{EN}$. This assumption merely ensures that G does not include bogus edges.

As mentioned earlier, the defender can modify the enterprise network by adding honey nodes and honey vulnerabilities. This means that the SVDG associated with an enterprise network $\mathcal{EN}$ changes when $\mathcal{EN}$ changes. To achieve this, we define two operators on SVDGs—union and difference.

Definition 2.6 (Union). Suppose $G^1 = (SV^1, EV^1)$ and $G^2 = (SV^2, EV^2)$ are two SVDG, where SV^1, EV^1 and SV^2, EV^2 are pairs of system vulnerabilities and edges in G^1 and G^2 , respectively. The *union* of G^1 and G^2 , denoted $G^1 \cup G^2$, is SVDG $G = (SV, EV)$, where $SV = SV^1 \cup SV^2$ and $EV = EV^1 \cup EV^2$.

The union operation captures the new vulnerabilities that result when an enterprise network is expanded via the addition of honey vulnerabilities (e.g., when the defender pretends a node on the network contains some software that has unpatched vulnerabilities). Similarly, a difference operation describes how the SVDG changes when, for instance, some true vulnerabilities are patched.

Definition 2.7 (Difference). Suppose $G^1 = (SV^1, EV^1)$ and $G^2 = (SV^2, EV^2)$ are two SVDG where SV^1, EV^1 and SV^2, EV^2 are the system vulnerabilities and the edges of G^1 and G^2 , respectively. The *difference* of G^1, G^2 , denoted $G^1 \setminus G^2$, is the SVDG $G = (SV, EV)$, where $SV = SV^1 \setminus SV^2$ and $EV = \{(sv_1, sv_2) \mid sv_1, sv_2 \in SV, (sv_1, sv_2) \in EV^1\} \setminus EV^2$.

3 THE ATTACKER'S STRATEGY

In this section, we first formally define the attacker's actions (scan a node, exploit a vulnerability). We then define the attacker's state—at any given point in time, the attacker has partially uncovered the network and learned an SVDG associated with the network he or she has learned. Given an attacker state, he or she can then decide which actions are feasible and which ones are not and which state he or she moves to if he or she takes a given action. Based on this, we define a valid attacker strategy to be a sequence (of actions) whose actions are feasible to execute in the state in which they are executed (formal definition will follow shortly). Because an attacker does not want to be fully predictable, and because he or she has goals, we introduce the notion of the probability of an attack strategy.

We start by defining the attacker's actions. The attacker sees an enterprise network and needs to probe the network, understand where the vulnerabilities are, and determine how to formulate an attack. For this, he or she has two actions.

Definition 3.1 (Attacker Actions). Given an enterprise network $\mathcal{EN} = (N, E \text{ compr})$ and an associated SVDG $G = (SV, EV)$, the set of all possible actions of an attacker is $\mathbb{A}(\mathcal{EN}, G) = SC(\mathcal{EN}) \cup EXP(G)$, where $SC(\mathcal{EN}) = \{\text{scan}(n) \mid n \in N\}$ is the set of all possible scan actions and $EXP(G) = \{\text{exploit}(n, s, v) \mid (n, s, v) \in SV\}$ is the set of all possible exploit actions.

It is important to note that although on the surface it seems as though there are only two actions available to the attacker, the $\text{exploit}(-, -, v)$ action has a vulnerability v as a parameter. Thus, for each v , $\text{exploit}(-, -, v)$ is a *different* action. For instance, the vulnerability v with the CVE number CVE-2017-3340 is a vulnerability in the Oracle Business Suite, while the vulnerability v with the CVE number CVE-2017-3841 is a vulnerability in CISCO's Secure Access Control System. Our $\text{exploit}(-, -, v)$ action would end up invoking two very different pieces of code to exploit these

two very different vulnerabilities. As of the date of submission of this article, NIST's National Vulnerability Database had 83,217 reported vulnerabilities.³ This means that, in practice, our formulation of the attackers' actions is shorthand for over 83K attacks in all.

An *attacker strategy* is a sequence (without repetition as scanning the same node twice or exploiting the same (n, s, v) triple twice leads to no improvement) of attacker actions in $\mathbb{A}(\mathcal{EN}, G)$. However, not all attacker strategies are feasible. For this, we first need to define the concept of an attacker's state.

Definition 3.2 (Attacker State). An attacker state st_i is a triple $(\text{Acts}_i, \mathcal{EN}_i, G_i)$, where Acts_i is a sequence (intended to be the sequence actions used to reach state st_i) and $\mathcal{EN}_i$ and G_i , respectively, represent the enterprise network and the SVDG that the attacker knows.

Note that an attacker's state (as opposed to the system state) has two parts: a state of the system (enterprise network and SVDG) that he or she has learned and the actions that he or she has taken to learn the system state. In the real world, a reasonable attacker would likely reflect concurrently on what he or she has done and what he or she learned about the system. For instance, if he or she is in a state where he or she is required to provide a password to access a node, then reflecting on the fact that he or she has already tried three passwords might influence his or her choice of what to do next—he may decide not to try to guess the password yet again, potentially triggering a security alert. It is for this reason that the attacker's state (really his or her state of mind) includes both what he or she is seeing about the network and what he or she has done to date. Hutchins et al. (2011) reported that identifying courses of actions that constitute a broader campaign is important to prevent "Advanced Persistent Threat." We note that extensive-form games (Hart 1992), which uses a game tree, allows explicit representation of a number of important aspects of game including the sequence of players' action selections, precisely for the reasons listed above. We now note an important assumption.

Assumption. We assume that the attacker does not initially know the entire enterprise network and its entire associated SVDG. He uses his or her actions to iteratively uncover more of the true enterprise network and the SVDG. This is a reasonable assumption, especially as the defender can account for the attacker's a priori knowledge of weaknesses in the network by having as many start nodes as he or she wishes in the associated SVDG.

Initial State. The *initial state* of the attacker is $st_0 = (\emptyset, \mathcal{EN}_0, G_0)$ —the set of actions the attacker has performed is $\emptyset$, and the attacker knows an initial network $\mathcal{EN}_0$ and an initial SVDG G_0 . We set $\mathcal{EN}_0 = (N_0, E_0, \text{compr}_0)$, where compr_0 is some initial set of entry points for the attacker, $N_0 = \text{compr}_0 \cup \{n | (n', n) \in E, n' \in \text{compr}\}$ consists of the initial set of enterprise node and nodes that are accessible from it in the true enterprise network, and $E_0 = \{(n, n') | n, n' \in N_0, (n, n') \in E\}$ consists of the obvious edges. We initially set $G_0 = (\emptyset, \emptyset)$, i.e., initially, his or her SVDG has no nodes and edges. Starting from this initial state, the attacker can take actions that build up his or her knowledge of the enterprise network, e.g., by scanning to identify vulnerabilities that he or she can exploit. We can now define the actions that an attacker can perform in a given state.

Definition 3.3 (Valid Attacker Actions). Given an attacker state $st_i = (\text{Acts}_i, (N_i, E_i, \text{compr}_i), (SV_i, EV_i))$, the set of valid attacker actions in state st_i is $\mathbb{VA}(st_i) = \text{VSC}(st_i) \cup \text{VEXP}(st_i)$, where $\text{VSC}(st_i) = \{\text{scan}(n) | n \in N_i \setminus \text{compr}, \text{scan}(n) \notin \text{Acts}_i\}$ and $\text{VEXP}(st_i) = \{\text{exploit}(n, s, v) | (n, s, v) \in SV_i \wedge \text{exploit}(n, s, v) \notin \text{Acts}_i \wedge \exists \text{exploit}(n', s', v') \in \text{Acts}_i : ((n', s', v'), (n, s, v)) \in EV_i\}$.

Thus, for a scan action to be feasible, it must not have been done before and the node being scanned must be in the network that the attacker has learned so far. Likewise, for an exploit action to be

³A more up to date number can be found in the "Resource Status" panel on the left portion of the <https://nvd.nist.gov/> web page.

successful, it must not have been done before, and there must be an (n', s', v') in the SVDG that is already known to the attacker that enables the exploit action (n, s, v) . After he or she takes an action, the attacker's state changes, reflecting new knowledge of the network that he or she might have learned.

Example 3.4. Consider the enterprise network in Figure 1(b), and suppose there is no direct link between n_1 and n_4 . At the beginning, the attacker is at node n_1 and knows of the existence of n_2 . By scanning n_2 , he or she discovers the vulnerability (n_2, s_4, v_6) , allowing him or her to access n_2 . He then uses the vulnerability (n_2, s_3, v_4) to access n_4 . Finally, by scanning node n_4 , he or she also obtains information about vulnerabilities in n_4 (see Figure 7 in the appendix).

We now define how the attacker's state changes after he or she performs an action.

Definition 3.5 (State Transition after an Action). Suppose the attacker performs an action $a \in \mathbb{A}(st_i)$ in state $st_i = (Acts_i, (N_i, E_i, compr_i), (SV_i, EV_i))$. This results in the state $tr(st_i, a) = st_{i+1} = (Acts_{i+1}, \mathcal{EN}_{i+1}, G_{i+1})$, where

- if $a = scan(n)$, then $Acts_{i+1} = Acts_i \cup \{scan(n)\}$, $\mathcal{EN}_{i+1} = (N_i, E_i, compr_i)$ and $G_{i+1} = (SV_{i+1} = SV_i \cup \{(n, s, v) | (n, s, v) \in SV\}, \{(sv, sv') | sv, sv' \in SV_{i+1}, (sv, sv') \in EV_i\})$
- if $a = exploit((n, s, v))$, then $Acts_{i+1} = Acts_i \cup \{exploit((n, s, v))\}$, $G_{i+1} = G_i$ and $\mathcal{EN}_{i+1} = (N_i \cup \{n' | (n, n') \in E\}, E_i \cup \{(n, n') | (n, n') \in E\}, compr_i \cup \{n\})$.

Simply put, a scan operation does not change the knowledge the attacker has of the enterprise network, but it allows him or her to augment his or her knowledge of the true SVDG, because he or she can identify potential vulnerabilities in the scanned node. In contrast, an exploit action allows the attacker to uncover new nodes in the enterprise network, as well as new edges.

We assume the existence of a function $costA$ that assigns a cost for each action $a \in \mathbb{A}$. If we assume that the attacker is buying exploits on the market, then $costA$ could be calculated using several existing studies. (i) For instance, on November 18, 2015, the zero-day exploit reseller Zerodium published the cost of various exploits.⁴ They listed the price of an exploit that could compromise Safari browsers at \$50,000, and the price was \$80,000 for Chrome browsers. (ii) Separately, Rand Corporation published a comprehensive book on exploit prices, not just for zero days, but for more mundane attacks (e.g., stolen data) (Ablon et al. 2014). A system security manager who is monitoring the black market for exploits can easily find the cost that an attacker might have to pay for different types of exploits, including potential zero days. We now define valid attacker strategies.

Definition 3.6 (Valid Attacker Strategy). Given an enterprise network $\mathcal{EN} = (N, E, compr)$, an SVDG $G = (SV, EV)$, and a budget (max cost) $\hat{c}^5$, a sequence $as = \langle a_1, \dots, a_m \rangle$ is a valid attacker strategy if there exists an associated sequence $\langle st_0, st_1, \dots, st_m \rangle$ of attacker states s.t.

⁴<https://www.wired.com/2015/11/heres-a-spy-firms-price-list-for-secret-hacker-techniques/>.

⁵Estimating the total budget of an attacker can be a challenge. This requires the defender to first estimate the value of his or her enterprise's data or the value of "taking down" part of an enterprise network. There are many works in the literature that have found ways how to define asset criticality such as Keeney (1996), Keeney (1992), and Dhillon and Torkzadeh (2001). Another method, used by the US Department of Defense (Kim and Kang 2011), defines the criticality of an asset in the DoD's enterprise network by using the hierarchy relations present in DOD. Second, the defender must estimate the attacker: An entity like the DoD has to defend against adversaries with deep pockets, while a small company with credit card data in their network may need to defend against less-sophisticated adversaries. We suggest that the maximal cost that the attacker is willing to bear should be *monotonic* in the value of the data within the enterprise, *anti-monotonic* in the cost of mounting an attack to secure that data, *anti-monotonic* in the size of the adversary, and *monotonic* in the size of attacker. Several such functions can be written down—we note that while this is not a perfect solution, today's system managers are largely not even attempting to estimate adversary costs.

- (i) $\forall i \in \{1, \dots, m\} : a_i \in \mathbb{VA}(st_{i-1})$; (ii) $\forall i \in \{1, \dots, m\} : s_i = tr(st_{i-1}, a_i)$; (iii) $\sum_{i=1}^m \text{costA}(a_i) \leq \hat{c}$; (iv) $\forall a \in \mathbb{VA}(st_m) : \text{costA}(a) + \sum_{i=1}^m \text{costA}(a_i) > \hat{c}$.

While the first three conditions are obvious, the fourth condition may be confusing. We assume that the attacker's budget is tight in the sense that the attacker is willing to carry out a maximal set of attacks subject to the budget. In the event this is not the case, his or her budget can be reduced so this property holds with no loss of generality.

We use $AS(\mathcal{EN}, G, \hat{c})$ to denote the set of all attacker strategies w.r.t. $\mathcal{EN}, G$ and $\hat{c}$. Given an attacker strategy $as = \langle a_1, \dots, a_m \rangle \in AS(\mathcal{EN}, EV, \hat{c})$, the state sequence $ss(as)$ is defined to be the sequence $\langle st_0, st_1, \dots, st_m \rangle$ s.t. $st_i = tr(st_{i-1}, a_i)$ for $i \in \{1, \dots, m\}$. We now define the concept of an attacker strategy probability density function or PDF.

Definition 3.7 (Attacker Strategy Probability (ASP)). Given an enterprise network $\mathcal{EN} = (N, E, \text{compr})$, a system vulnerability dependency graph $G = (SV, EV)$ and budget $\hat{c}$, an attacker strategy probability (ASP for short) is a function $\mathbb{P} : AS(\mathcal{EN}, G, \hat{c}) \rightarrow [0, 1]$ s.t. $\sum_{as \in AS(\mathcal{EN}, EV, \hat{c})} \mathbb{P}(as) = 1$.

Note that this does *not* mean that the attacker will choose the highest probability action. In terms of classical probability theory, one can think of each action as being in a bag, and the attacker picks the action to perform with a certain probability in accordance with the probability distribution. Thus, any of the actions available to the attacker can be taken with a bias based on the utilities of the actions. The attacker, being smart, does not want the defender to guess what he or she might do. To achieve this, the attacker chooses an attack strategy based on his or her ASP.

In general, we can define ASPs based on utilities in many different ways. We proceed under the assumption that attackers will choose ASPs that are monotonic w.r.t. the utility—we call these “monotonic” ASPs. One example of a monotonic ASP is given below.

Example 3.8 (Utility-based ASP). Suppose each action a has a specific utility $\text{utilA}(a)$. For instance, $\text{utilA}(a)$ should be monotonic in the impact measure (of the vulnerability exploited by action a) in NIST's National Vulnerability database and monotonic w.r.t. the value of the assets present on the node targeted by a . Clearly, the system administrator for the network can get the impact measure from NIST's NVD website and can estimate the latter.

Given an attacker strategy $as = \langle a_1, \dots, a_m \rangle \in AS(\mathcal{EN}, G, \hat{c})$ and its state sequence $ss(as) = \langle st_0, st_1, \dots, st_m \rangle$, a *utility based ASP* function is given by $P_u(as) = \prod_{i=1}^m Pr_1(st_{i-1}, a_i)$, where $Pr_1(st, a) = \frac{\text{utilA}(a)}{\sum_{a' \in \mathbb{VA}(st)} \text{utilA}(a')}$.

Basically, the utility-based ASP says that the probability of taking a specific valid action in a given state is proportional to its utility.

However, alternative utility-based ASPs might be based on different versions of the $Pr_1(st, a)$ function, which could just as well be defined via the family of functions that also lead to monotonic ASPs $Pr_1^i(st, a) = \frac{\text{utilA}(a)^i}{\sum_{a' \in \mathbb{VA}(st)} \text{utilA}(a')^i}$ or $Pr_1^{*,b}(st, a) = \frac{b^{\text{utilA}(a)}}{\sum_{a' \in \mathbb{VA}(st)} b^{\text{utilA}(a')}}.$

However, an alternative way of defining the attacker strategy probability could take into account explicit information from the defender. Even though the defender may not precisely know who the adversary is and how he or she might attack the system, he or she would probably have a good idea of how an attack might be launched. Most large companies have extensive penetration testing officials and contractors to probe their system. It is important to note that the use of a specific utility $\text{utilA}(a)$ for each action a that is specific for the machine, where a is applied is a very powerful abstraction. For instance, if the attacker has to choose between exploiting the same vulnerability on a personal computer (action a_1), and exploiting it on a server machine (action

ALGORITHM 1: Attacker Strategy Generator

```

1: procedure GETATTACKERSTRATEGY( $\mathcal{EN} = (N, E, \text{compr})$ ,  $G = (SV, EV)$ ,  $\hat{c}$ )
2:   totalCost = 0;
3:    $st = st_0$ ;
4:    $as = \langle \rangle$ ;
5:   while ( $|\mathbb{VA}(st)| > 0$  and totalCost <  $\hat{c}$ ) do
6:     chose an action in  $a \in \mathbb{VA}(st)$  according to  $\mathbb{P}(a)$ ;
7:      $as = as \cup \{a\}$ ;
8:      $tc = \text{totalCost} + \text{costA}(a)$ ;
9:      $st = tr(st, a)$ ;
10:  end while
11:  return  $as$ ;
12: end procedure

```

a_2), then we can require that $\text{utilA}(a_2) > \text{utilA}(a_1)$, because, in general, a server may be more important for an attacker than a personal computer.

Example 3.9 (Administrator-Defined Utility-based ASP). The administrator, for instance, could decide to associate with each action a a score $\text{sco}(a) = \alpha_1 \cdot \text{impact}(a) + \alpha_2 \cdot \text{exploitability}(a) + \alpha_3 \cdot \text{DI}(a) + \alpha_4 \cdot \text{LEX}(a) + \alpha_5 \cdot \text{TI}(a)$, where $\text{impact}(a)$, $\text{exploitability}(a)$ are the impact and exploitability of the vulnerability used by action a according to NIST's National Vulnerability Database, and BI , DI , and TI are, respectively, the business impact, likelihood of discovery, and technical impact of the vulnerability used by a according to MITRE's Common Weakness Scoring System. The α_i 's are arbitrary constants that the Enterprise Security Officer can set in accordance with his or her specific knowledge of the security risk and the current threats directed at his or her network. Once this is done, $P_u(as)$ is defined as in Example 3.8, but $Pr_1(st, a)$ is defined as $Pr_1(st, a) = \frac{\text{sco}(a)}{\sum_{a' \in \mathbb{VA}(st)} \text{sco}(a')}$.

The GETATTACKERSTRATEGY (see Algorithm 1) function below randomly generates an attacker strategy in accordance with a given ASP.

It is important to note that the greedy strategy does *not* mean that the attacker greedily chooses the action with the highest impact. In Step 6 of Algorithm 1, the attacker's next action is chosen according to the distribution $\mathbb{P}$. So a greedy choice is not made. If there are, for instance, three valid actions A,B,C with probabilities 0.5, 0.3, 0.2, then the attacker chooses *any* one of these three actions by making a choice biased by the distribution. Thus, there is a 50% probability that he or she will choose A, a 30% probability that he or she will choose B, and a 20% probability that he or she will choose C. Note that executing Algorithm 1 many times provides a heuristic as exact computation of the optimal attacker strategy is computationally very hard.

Before concluding this section, we note that Algorithm 1 can use a diversity of ways of defining the Attack Strategy Probability. We have given two *examples*. In one example, the attack strategy probability is based on the impact score of the vulnerability used in NIST's National Vulnerability Database (NVD). In another example, the attack strategy probability is based on a linear combination of five scores that are well published and used in the literature—the impact and exploitability score in NIST's NVD, as well as the Business Impact, Technical Impact, and Exploitability scores from MITRE's Common Weakness Scoring System. Moreover, the enterprise security officer can set the coefficients in the linear combination as he or she sees fit. As a consequence, this provides a very broad and robust framework. Moreover, it is important to note that despite the fact that our Administration Defined ASP is linear, it could just as well be nonlinear. All that would be needed

is to replace the *sco* function in Example 3.9 with a nonlinear function, and the rest of the example would still apply.

4 ANALYSIS OF THE DEFENDER

The defender in our adversarial-theoretic formulation can take two actions: He or she can add an enterprise network and/or an SVDG, and he or she can remove an enterprise network and/or SVDG. The addition actions are used to add honey nodes and honey vulnerabilities, while the remove actions remove nodes (i.e., vulnerabilities) from an SVDG. Removal of an (n, s, v) triple from an SVDG corresponds to fixing a vulnerability.

Definition 4.1 (Add/Remove Block). An *add block* is a pair $ab_i = (\mathcal{EN}^i, G^i)$, where $\mathcal{EN}^i$ is an enterprise network and G^i is an SVDG. A *remove block* is a system vulnerability dependency graph.

Intuitively, an add-block $ab_i = (\mathcal{EN}^i, G^i)$ adds a honey-network $\mathcal{EN}^i$ to the enterprise network and adds a honey-vulnerability network G^i to the associated SVDG. It is also important to note that $\mathcal{EN}^i$ also contains nodes of the original network to which the honeypot network is added. Remove blocks just remove some nodes from an SVDG, i.e., they do not alter the enterprise network. We use AB, RB to denote the set of all add blocks and remove blocks, respectively.

Definition 4.2 (Defender Strategy). A defender strategy δ is a pair (A, R) , where $A \subseteq AB$ and $R \subseteq RB$.

Thus, a defender strategy consists of two things: which honey nodes and/or vulnerabilities to insert into the network and which vulnerabilities to remove from the network (e.g., by patching a vulnerability or uninstalling a vulnerable code base).

Definition 4.3 (Applying Defender Strategy). Suppose $\mathcal{EN} = (N, E, \text{compr})$ is an enterprise network, $G = (SV, EV)$ is an SVDG, and $\delta = (A, R)$ is a defender strategy. The result of applying the defender strategy is a new enterprise network and dependency graph, where $\mathcal{EN}(A, R) = \mathcal{EN} \cup (\bigcup_{(\mathcal{EN}^i, -) \in A} \mathcal{EN}^i)$ and $G(A, R) = (G \setminus \bigcup_{G_i \in R} G_i) \cup (\bigcup_{(_, G^i) \in A} G_i)$. The *honey set* w.r.t. defender strategy $\delta = (A, R)$ is $\text{honey} = \bigcup_{(_, (SV, _, _)) \in A} SV$.

Note that $_$ is a wildcard symbol: Given a tuple $(_, C)$, the blank symbols mean that we do not need the value of the first element of the tuple for the purpose. For instance, $\bigcup_{(\mathcal{EN}^i, -) \in A} \mathcal{EN}^i$ says that for all pairs of the form $(\mathcal{EN}^i, -) \in A$, we compute the union of all such $\mathcal{EN}^i$ irrespective of what the second argument is.

Clearly, defender strategies have several types of cost, three of which we define below. These functions assume that there is a cost the defender has to incur, $\text{ncostD}(n)$, in creating a credible honey node n , a cost $\text{pcostD}(G^i)$ in applying a set of patches, and a cost $\text{prcostD}(G^i)$ in deactivating some vulnerable software. All of these costs can be estimated. For instance, Cavusoglu et al. (2008) estimates patching costs by coalescing the time to identify and test the system after applying the patch. The *negotiated downtime* is the down-time of the system for implementing and testing patch deployment. They define patching cost pcost in terms of the amount of “system downtime” required to implement the patches; that time is correlated with both the level of effort (funds) expended by employees in patching. DARPA also ran a project to estimate patching costs as well (Beattie et al. 2002). Other authors (Mercuri 2003; Cavusoglu et al. 2006) also give methods to estimate patch costs.

The simplest way to evaluate productivity cost (prcost) is by setting it to the cost of buying new software to replace an existing, vulnerable piece of code, plus the cost of man-hours involved in installing it. Thus, if one person can deploy it within a single location of a corporate cloud in an hour, then the installation cost is just 1 hour of that person’s time. On the other hand, if it will take

that person 1 day to do it by going from machine to machine, the installation cost would be 1 day of that person's time. A system manager should be able to estimate these quantities.

Definition 4.4 (Node/Patch/Productivity Cost).

$$\begin{aligned} \text{ncost}(A) &= \sum_{n \in N', (N', -) = \bigcap_{(EN^i, -) \in A} EN^i} \text{ncostD}(n) \\ \text{pcost}(R) &= \sum_{G^i \in R} \text{pcostD}(G^i), \text{prcost}(R) = \sum_{G^i \in R} \text{prcostD}(G^i). \end{aligned}$$

A defender strategy $\delta = (AB, RB)$ makes a conscious attempt to take some actions but not others. Exactly what defender strategy should be used depends on the impact, on the defender, of adopting a specific strategy. We define four types of impact below that use the $\text{impact}(-)$ function described earlier to measure the impact of a vulnerability (e.g., using values from NIST's National Vulnerability Database). We also assume the existence of two functions, $\text{data}(n)$ returning a set of data on node n and a function $\text{impact}(d)$ measuring the impact of losing d (or losing access to d),

$$\begin{aligned} \text{vimpatDA1}(as) &= \max_{\text{exploit}(n, s, v) \in (as - \text{honey})} \text{impact}(v), \text{dimpatDA1}(as) = \max_{d \in \text{data}(n), \text{exploit}(n, s, v) \in (as - \text{honey})} \text{impactData}(d) \\ \text{vimpatDA2}(as) &= \sum_{\text{exploit}(n, s, v) \in (as - \text{honey})} \text{impact}(v), \text{dimpatDA2}(as) = \sum_{d \in \text{data}(n), \text{exploit}(n, s, v) \in (as - \text{honey})} \text{impactData}(d). \end{aligned}$$

What these four measures state is the impact of a vulnerability being exploited by an attacker (the first two measures) and the impact of data being exploited by an attacker if he or she is able to exploit a vulnerability. More specifically:

- vimpatDA1 is the max of the impact of each vulnerability contained in non-honey nodes that are exploited during the attack.
- vimpatDA2 is the sum of the impact of each vulnerability contained in non-honey nodes that are exploited during the attack.
- dimpatDA1 is the max of the impact of the data contained in each non-honey node that is compromised during the attack.
- dimpatDA2 is the sum of the impact of the data contained in each non-honey node that is compromised during the attack.

It is important to note that these four measures may conflict with one another as shown in the example below.

Example 4.5. Consider the network in Figure 1(a) and suppose there are two exploitable vulnerabilities v_1 in n_2 and v_2 in n_4 . However, n_4 has some data d s.t. $\text{impact}(d) = 10$ $\text{impact}(v_1) = 10$, $\text{impact}(v_2) = 1$. If we can patch only one vulnerability, then we have only two choices: patching v_2 that means $\text{vimpatDA1} = 1$ and $\text{dimpatDA1} = 10$. On the other hand, patching v_4 means $\text{vimpatDA1} = 10$ and $\text{dimpatDA1} = 0$. Thus, we can minimize either vimpatDA1 or dimpatDA1 , but not both, suggesting that the two measures conflict.

Definition 4.6 (Expected Impact).

$$\mathbb{E}[\mu, EN, G, \hat{e}] = \sum_{as \in AS(EN, G, \hat{e})} \mathbb{P}(as) \cdot \mu(as).$$

Simply put, what this definition says is that the defender must reason about all possible attacker strategies as , as he or she does not know what the attacker will do (but he or she knows the attacker is smart enough not to necessarily use the highest utility or highest probability strategy). If he or she chooses a defender strategy resulting in an enterprise network EN and an SVDG G , then the above formula specifies the expected impact on the system.

ALGORITHM 2: Monte Carlo Approximation

```

1: procedure SHARE-MC( $\mathcal{EN} = (N, E, \text{compr})$ , honey,  $\mu$ ,  $G = (SV, G)$ ,  $\hat{c}$ )
2:   iteration = 0;
3:   impact = 0;
4:   while (iteration < #maxIteration) do
5:      $as = \text{getAttackerStrategy}(\mathcal{EN}, G, \hat{c})$ 
6:     impact = impact +  $\mu(as)$ ;
7:     iteration = iteration + 1;
8:   end while
9:   return  $\frac{\text{impact}}{\text{iteration}}$ ;
10: end procedure

```

Example 4.7. Consider Figure 1(b). Suppose the set $AS(\mathcal{EN}, G, \hat{c}) = \{as1, as2\}$, where $as1 = \langle \text{scan}(n_2), \text{exploit}(n_2, s_4, v_5), \text{exploit}(n_2, s_3, v_4), \text{scan}(n_4), \text{exploit}(n_4, s_3, v_4) \rangle$ and $as2 = \langle \text{scan}(n_2), \text{exploit}(n_2, s_4, v_5), \text{exploit}(n_2, s_3, v_4), \text{scan}(n_4), \text{exploit}(n_4, s_2, v_5) \rangle$. We can clearly see these two valid attacker strategies in Figure 7 in the appendix. In the same vein as Example 3.8, we can retrieve $\mathbb{P}(as1)$ and $\mathbb{P}(as2)$. For now, suppose each of these probabilities is 0.5. Consider the honey action $\text{exploit}(n_4, s_2, v_5)$, and suppose all exploit actions have the same impact of 3. Therefore, the expected impacts by considering only impactDA1 and impactDA2 are as follows:

$$\begin{aligned} \mathbb{E}[\text{impactDA1}, \mathcal{EN}, G, \hat{c}] &= \mathbb{P}(as1)\text{impactDA1}(as1) + \mathbb{P}(as2)\text{impactDA1}(as2) \\ &= 0.5 \cdot \max(3, 3, 3) + 0.5 \cdot \max(3, 3) = 0.5 \cdot 3 + 0.5 \cdot 3 = 3, \end{aligned}$$

$$\begin{aligned} \mathbb{E}[\text{impactDA2}, \mathcal{EN}, G, \hat{c}] &= \mathbb{P}(as1)\text{impactDA1}(as1) + \mathbb{P}(as2)\text{impactDA1}(as2) \\ &= 0.5 \cdot (3 + 3 + 3) + 0.5 \cdot (3 + 3) = 0.5 \cdot 9 + 0.5 \cdot 6 = 7.5. \end{aligned}$$

Because the space of valid attacker strategies is huge, an exact computation of the expected impact on a network is difficult. We propose the SHARE-MC algorithm (Algorithm 2) that uses Monte Carlo simulation, in conjunction with the GETATTACKERSTRATEGY algorithm defined previously to approximate the expected impact. The algorithm uses a #maxIteration parameter to sample a certain number of attacker strategies. In addition, it is important to notice that defender cannot easily observe the action of the attacker. First, it is very difficult to determine how many attackers are present in the network—there can be one or many. Sometime the attacker is also an insider—in this case, it becomes even more difficult to detect his or her actions. For this reason, a standard game model based on the game tree is not feasible. In this situation, the most popular and effective approach to model the interaction defender-attacker is the Stackelberg game (see Section 5).

5 STACKELBERG PARETO OPTIMAL DEFENDER STRATEGY

To define the optimal defender strategy, we use a Stackelberg game approach. A generic Stackelberg game has two players, a leader and a follower. A leader commits to a strategy first, and then a follower optimizes his or her reward, considering the action chosen by the leader (Kiekintveld et al. 2009a). In our game the leader is the defender who fixes his or her strategy and then prepares the enterprise network for the attacks by the attacker. Once the defender strategy is deployed, the attacker attacks the network by using his or her strategies (driven by his or her utilities and costs) that generates some expected impacts for the defender. As mentioned in the previous section, a defender can choose a defender strategy that minimizes expected impact based on at least four impact metrics articulated in the previous section; other metrics can be easily added. Because these metrics conflict with each other, we formulate the problem of finding the defender's optimal strategy as the multi-objective optimization problem as follows:

$$\begin{aligned} \delta^* = (A^*, R^*) \in \arg \min_{A \subseteq AB, R \subseteq RB} \{ \\ E[\text{vimpactDA1}, \mathcal{EN}(A, R), G(A, R), \hat{c}], E[\text{vimpactDA2}, \mathcal{EN}(A, R), G(A, R), \hat{c}], \\ E[\text{dimapctDA1}, \mathcal{EN}(A, R), G(A, R), \hat{c}], E[\text{dimapctDA2}, \mathcal{EN}(A, R), G(A, R), \hat{c}], \\ \text{ncost}(A), \text{pcost}(R), \text{prcost}(R) \}. \end{aligned}$$

This formulation of the defender optimal strategy problem tries to minimize all four types of expected impact, as well as all three types of cost. It is important to note that our algorithms below are applicable with obvious modifications even if we add additional measures into our set of objective functions in the above Pareto optimization problem. For instance, we could add new measures to be optimized using different ASPs, which would impact the expectation values in the above formulation of the defender's problem. We now define the Pareto optimal set.

Definition 5.1 (Dominance and Pareto Optimal Set). Suppose $\delta_1 = (A^1, R^1)$ and $\delta_2 = (A^2, R^2)$ are two defender strategies. δ_2 dominates δ_1 iff

$$\begin{aligned} E[\text{vimpactDA1}, \mathcal{EN}(A^1, R^1), G(A^1, R^1), \hat{c}] &\geq E[\text{vimpactDA1}, \mathcal{EN}(A^2, R^2), G(A^2, R^2), \hat{c}] \quad \text{and} \\ E[\text{vimpactDA2}, \mathcal{EN}(A^1, R^1), G(A^1, R^1), \hat{c}] &\geq E[\text{vimpactDA2}, \mathcal{EN}(A^2, R^2), G(A^2, R^2), \hat{c}] \quad \text{and} \\ E[\text{dimapctDA1}, \mathcal{EN}(A^1, R^1), G(A^1, R^1), \hat{c}] &\geq E[\text{dimapctDA1}, \mathcal{EN}(A^2, R^2), G(A^2, R^2), \hat{c}] \quad \text{and} \\ E[\text{dimapctDA2}, \mathcal{EN}(A^1, R^1), G(A^1, R^1), \hat{c}] &\geq E[\text{dimapctDA2}, \mathcal{EN}(A^2, R^2), G(A^2, R^2), \hat{c}] \quad \text{and} \\ \text{ncost}(A^1) &\geq \text{ncost}(A^2) \quad \text{and} \quad \text{pcost}(R^1) \geq \text{pcost}(R^2) \quad \text{and} \quad \text{prcost}(R^1) \geq \text{prcost}(R^2). \end{aligned}$$

and

$$\begin{aligned} E[\text{vimpactDA1}, \mathcal{EN}(A^1, R^1), G(A^1, R^1), \hat{c}] &> E[\text{vimpactDA1}, \mathcal{EN}(A^2, R^2), G(A^2, R^2), \hat{c}] \quad \text{or} \\ E[\text{vimpactDA2}, \mathcal{EN}(A^1, R^1), G(A^1, R^1), \hat{c}] &> E[\text{vimpactDA2}, \mathcal{EN}(A^2, R^2), G(A^2, R^2), \hat{c}] \quad \text{or} \\ E[\text{dimapctDA1}, \mathcal{EN}(A^1, R^1), G(A^1, R^1), \hat{c}] &> E[\text{dimapctDA1}, \mathcal{EN}(A^2, R^2), G(A^2, R^2), \hat{c}] \quad \text{or} \\ E[\text{dimapctDA2}, \mathcal{EN}(A^1, R^1), G(A^1, R^1), \hat{c}] &> E[\text{dimapctDA2}, \mathcal{EN}(A^2, R^2), G(A^2, R^2), \hat{c}] \quad \text{or} \\ \text{ncost}(A^1) &> \text{ncost}(A^2) \quad \text{or} \quad \text{pcost}(R^1) > \text{pcost}(R^2) \quad \text{or} \quad \text{prcost}(R^1) > \text{prcost}(R^2). \end{aligned}$$

A point $p = \begin{bmatrix} E[\text{vimpactDA1}, \mathcal{EN}(A, R), G(A, R), \hat{c}] \\ E[\text{vimpactDA2}, \mathcal{EN}(A, R), G(A, R), \hat{c}] \\ E[\text{dimapctDA1}, \mathcal{EN}(A, R), G(A, R), \hat{c}] \\ E[\text{dimapctDA2}, \mathcal{EN}(A, R), G(A, R), \hat{c}] \\ \text{ncost}(A), \text{pcost}(R), \text{prcost}(R) \end{bmatrix}$ associated with the defender strategy $\delta = (A, R)$ is

Pareto optimal if there is no other defender strategy δ' that strictly dominates δ (i.e., that is at least as good as δ on all measures but strictly better on at least one measure).

The first four conditions in the dominance definition say that all costs and impacts w.r.t. δ_1 exceed those w.r.t. δ_2 , while the second four conditions require that δ_2 is strictly better than δ_1 w.r.t. at least one impact measure or cost. The set of all Pareto-optimal points is called the *Pareto frontier*. Our main task is to compute this set.

It is important to note that because we have formulated the defender's problem as a Pareto optimization problem, any existing algorithm for Pareto optimization can be used to solve it. These include Kung et al. (1975) and Godfrey et al. (2007). Thus, we show the existence of an algorithm to find an optimal solution for the defender's problem. However, the following result shows that any such algorithm is likely to be extremely inefficient and, hence, heuristics will need to be used.

THEOREM 5.2. (i) *Computing the Pareto frontier is NP-hard, i.e., if there exists a polynomial algorithm computing the Pareto frontier, then $P = NP$.*

(ii) Unless $P = NP$, the result in (i) holds even if the algorithm works in a polynomial time in its input and output.

PROOF. All proofs are listed in the appendix. $\square$

Since the problem is NP-hard, computing an optimal solution (one that yields a Pareto point) is infeasible. We therefore provide algorithms based on genetic programming that approximate the optimal defender strategy. Section 7.1.4 shows (experimentally) what happens when the attacker is assumed to be sub-rational, i.e., conducts an attack that deviates from the assumed rational behavior.

We propose the HARE-NSGA-II in Algorithm 3, which builds on top of the well-known NSGA-II genetic algorithm (Deb et al. 2000) to find an (approximately) optimal defender strategy (cf. Algorithm 3). Though NSGA-II is a local search algorithm that can get stuck in theory in local minima, HARE-NSGA-II avoids this in practice by iteratively executing our NSGA-II variant multiple times, each with a randomly selected initial state, and then choosing the best result—but this can still be suboptimal. In our adaptation of NSGA-II, a defender strategy $\delta = (A, R)$ is represented by a binary vector of length $|AB| + |RB|$ consisting of either 0 and 1. In the binary vector denoted as v_δ , the h th value is set to 1 if $ab_h \in A$ for $h < |AB|$ or if $rb_{h-|AB|} \in R$ for $h \geq |AB|$.

Thus, the v_δ vector merely captures whether an add block is present in the defender's strategy and likewise for remove blocks.

The NSGA-II in Algorithm 3 initializes N vectors described in the above uniformly at random. Suppose this initial sample is $SA_0 = \{v_{\delta^1}, \dots, v_{\delta^N}\}$. In the sample, each entry $v_{\delta^1}[h]$ is set to 1 with probability 0.5. From each sample set SA_i , we generate a new sample set SA_{i+1} as follows:

- Compute two new sets of samples, M_i , through the mutation operation, and C_i , through the cross-over operation.
- *Filtering Step.* For each $v_\delta \in SA_i \cup M_i \cup C_i$, check to make sure that there is no other $v_{\delta'} \in SA_i \cup M_i \cup C_i$ that dominates v_δ . If this is the case, then insert v_δ in SA_{i+1}^* . Thus, $SA_{i+1}^* = \{v_\delta | v_\delta \in SA_i \cup M_i \cup C_i, \nexists v_{\delta'} \in SA_i \cup M_i \cup C_i : \delta' \text{ dominates } \delta\}$.
- We then select the N samples from SA_{i+1}^* that have high diversity. Diversity, a well-known concept in genetic algorithms, tries to ensure that our samples, which are all members of a binary vector space of length $|AB| + |RB|$, should be chosen randomly from this space and should not be clustered close together.

At the end the procedure, NSGA-II returns a set of non-dominated defender strategies. The SHARE-NSGA-II procedure in Algorithm 3 calls the procedure NSGA-II n times, each with different starting conditions, and, finally, it takes the best from all the n executions (thus avoiding the possibility that NSGA-II gets stuck in a local optima).

In the above algorithm, computing the set F_i requires computing the value of all four impact functions using the Monte-Carlo approximation method presented earlier. This means that this operation is computationally expensive. Further speedups can be obtained by classical island parallelization methods (Whitley et al. 1998).

Note that the computed strategies may depend heavily on how the attacker behavior matches our assumptions. Section 7.1.4 empirically shows that even if the attacker deviates from our initial assumptions, the defender strategies are still effective.

5.1 Bi-optimization with OIE Add Blocks

In this section, we consider the case where only two objective functions are optimized and the blocks used have a particular form that we call Only-Incoming-Edges (OIE) Add Block. The first objective function is the expected impact, which can be any one of the four defined previously

ALGORITHM 3: SHARE-NSGA-II

```

1: procedure SHARE-NSGA-II((n))
2:   Initialize  $SA$ ;
3:   for ( $i = 1$  to  $n$ ) do
4:      $SA = SA \cup \text{SAUNSGA-II}()$ ;
5:   end for
6:   return not dominated points in  $SA$ ;
7: end procedure
8: procedure NSGA-II()
9:   Initialize  $SA_0$ ;
10:   $i = 0$ ;
11:  while ( $i < \#iteration$ ) do
12:     $M_i = \text{mutation}(S_i)$ ;
13:     $C_i = \text{crossover}(S_i)$ ;
14:     $F_i = \{v_\delta \mid v_\delta \in SA_i \cup M_i \cup C_i, \nexists v_{\delta'} \in SA_i \cup M_i \cup C_i : \delta' \text{ dominates } \delta\}$ ;
15:     $SA_{i+1} = \text{DiversityFiltering}(F_i)$ ;
16:     $i = i + 1$ ;
17:  end while
18:  return  $SA_i$ ;
19: end procedure

```

in this article. The second objective function is the cost computed as the number of the blocks used. This can be easily seen as a special case of all the costs function defined thus far.

The definition of the “only incoming edges” add block is the following:

Definition 5.3 (OIE Add Block). An OIE Add Block is a block $ab_i = (\mathcal{EN}^i, G^i)$ s.t. $\mathcal{EN}^i$ and G^i have no outgoing edges from honey nodes to the original network.

The intuitive meaning of an OIE Add Block is that it is possible to add nodes to the original network, but these nodes do not bring the attacker to other nodes in the original network. It means that independently of the block that is used, there will be a reduction in the expected impact by distracting the attacker from the really important targets. We use *OIEAB* to denote the set of all the OIE add blocks.

We first define the concept of *negated expected impact* to be a function $ni : OIEAB \rightarrow R$ defined as follows: $ni(A) = -E[_, \mathcal{EN}(A, \emptyset), G(A, \emptyset), \hat{c}]$.

Note that maximizing the negative expected impact function is the same as minimizing the maximal expected impact function defined earlier, which is what we wish to achieve.

If we use only OIE Add Block, then the following theorem holds.

THEOREM 5.4. *The negated expected impact function is sub-modular and monotonic. Given an impact measure impact, for any two subsets $A_x, A_y \subseteq OIEAB$ s.t. $A_x \subseteq A_y$, and any block $b \in OIEAB$ s.t. $b \notin A_y$, the following inequalities hold: $ni(A_x \cup \{b\}) \geq ni(A_x)$ and $ni(A_x \cup \{b\}) - ni(A_x) \geq ni(A_y \cup \{b\}) - ni(A_y)$.*

Since the negated expected impact is sub-modular and monotonic in *OIEAB* and the goal of this article is to minimize the expected impact (i.e., maximize the negated expected impact) and minimize the cost (i.e., the number of OIE Add Blocks used), the following important theorem holds.

THEOREM 5.5 (HARE-NSGA-II ALGORITHM APPROXIMATION GUARANTEE). *If negated expected impact can be computed in polynomial time, then Algorithm 3 obtains a $(1 - 1/e)$ -approximation of all Pareto points in polynomial time.*

This theorem shows that expected behavior of this algorithm is to produce a guaranteed approximation about the impact value of all the Pareto points discovered by the HARE-NSGA-II algorithm.

6 STOP POLICY FOR THE DEFENDER

Once the defender has chosen a defense strategy $\delta = (A, R)$ that solves the Pareto optimization problem described in the preceding section, he or she still has to determine what to do when an attacker is detected. In this section, we define two stop policies.

- *Stop Upon Detection* stops the attacker immediately on detection, i.e., when the user tries to exploit a honey vulnerability. However, this strategy has a problem: If the attacker is repeatedly launching attacks on the enterprise, he or she can easily learn where the honey vulnerabilities are. Thus, even though the current attack is stopped, the attacker, through his or her enhanced knowledge of the locations of honey vulnerabilities (i.e., through his or her enhanced knowledge of the defender strategy), can now craft better attacks next time.
- The *Stop After Delay* strategy delays the decision to stop, i.e., the attacker is allowed to continue attacking for a while after he or she is detected. The risk is that during this post-detection, but pre-stoppage period, the attacker causes serious damage.

Before going through these two options in detail, we say a few words about the ability of the adversary to learn. Real-world adversaries may not launch just one attack on an enterprise. They may launch an attack, get blocked by the system, and then find another way to launch a future attack (e.g., from a new IP address). Alternatively, they may launch a successful attack and use the knowledge gleaned from that successful attack to launch even more successful attacks. Both of these phenomena fall within the noisy-rich attacker model. For instance, when they are blocked by the enterprise (e.g., their IP is blocked), they are likely to examine what they did—which is another reason to include the attacker's actions within the attacker state—and try to infer what they did wrong. Adversaries may have a wide range of capabilities in making such inferences. Rather than try to model this for each and every adversary, our experiments vary the ability of the adversary to make such inferences by restricting the amount of time that the adversary has to draw inferences. Thus, even a super-smart adversary would be rendered less smart if he or she had a very small amount of time to draw inferences. This is described in more detail in Section 7.1.4.

We now explore both options (Stop Upon Detection, Stop After Delay) below.

6.1 Stop Upon Detection Policy

We introduce the stop operator below to model this situation.

Definition 6.1 (Stop Operator). Given an attacker strategy $as\langle a_1, \dots, a_m \rangle$, the stop operator returns a subsequence stops $(as) = \langle a_1, \dots, a_h \rangle$, where $h = \min_{a_i = \text{exploit}(v) \in as, v \in \text{honey}} i$.

In other words, h is the earliest exploit action by the attacker that attempts to exploit a honey vulnerability. By terminating the attacker after action a_h , we limit his or her impact to the successful actions he or she performed before action a_h . This is defined below.

Definition 6.2 (Stop Expected Impact).

$$E[\text{impactDA}, \mathcal{EN}, G, \hat{c}] = \sum_{as \in AS(\mathcal{EN}, G, \hat{c})} \mathbb{P}(as) \cdot \text{impactDA}(\text{stops}(as)).$$

However, as mentioned earlier, the attacker can learn a lot if he or she is stopped immediately on detection. To estimate his or her reward if this happens, we leverage UCT reinforcement learning algorithm working on Markov decision processes (Kocsis and Szepesvári 2006), where the states

are attacker states and it is possible to move from a state to another by using valid attacker actions—the only difference this time is that the defender is using these structures to simulate what an attacker might do. This is done via a *reward function*.

Given an attacker state st and a valid action a w.r.t. st , a reward function r returns the reward $r(st, a)$ to the attacker when action a is applied in state st . This reward could be any of the four impact functions defined earlier in the article, a combination of them, or something totally different that is defined by the ESO. A generic reinforcement learning algorithm uses the reward function to generate the probability that the attacker will move from one state to another. At the end of this process, the most probable paths in this Markov decision process are the attacker strategies that give the greatest total reward.

Formally, given a valid attacker strategy as the total reward to the attacker is the sum of all rewards for each transition $\sum_{a_i \in as} r(st_{i-1}, tr(st_{i-1}, a))$, where st_i is recursively defined as $st_i = tr(st_{i-1}, a)$.

Of course, the attacker only receives this reward if he or she is not detected and/or stopped. We consider two reward functions, $r1$ and $r2$, along with two associated stopping functions, $stop1$ and $stop2$, respectively.

According to $r1$, which captures the *Stop Upon Detection* policy, the reward to the attacker is split into two cases. If the attacker's action a exploits a honey vulnerability, then the attacker's utility is the negative of the utilities of all previous actions. Otherwise, it is the utility of a . In this case, we want the attacker to stop as soon as the cost of the attack, to the attacker, exceeds the attacker's cost threshold or when he or she hits a honey vulnerability.

$$r1(st, a) = \begin{cases} -\sum_{a' \in Acts, (Acts, -, -)=st} utilA(a') & \text{if } a = \text{exploit}(n, s, v) \wedge v \in \text{honey}, \\ utilA(a) & \text{otherwise} \end{cases}$$

$$stop1(st) = \begin{cases} \text{true} & \text{if } ((Acts, -, -) = st \wedge (\sum_{a' \in Acts} costA(a') + \min_{a' \in VA(st)} costA(a') > \hat{c} \text{ or } \\ & \text{or } \exists \text{exploit}(n, s, v) \in Acts : v \in \text{honey}) \\ \text{false} & \text{otherwise} \end{cases}$$

As stated earlier in the article, the defender estimates the utility of the attacker using linear or non-linear combinations of measures derived from the NIST NVD data (NIST 2017) and/or the MITRE CWSS data (The MITRE Corporation 2011).

The difference between the attacker utility given earlier in the article for an attack sequence and the new notion of reward given above (as well as the one given below) is that the actual utility obtained by the attacker changes if the defender detects him or her, and this latter value is what we are calling reward. Thus, the UCT algorithm optimizes the attacker's reward in this way with a look-ahead ability that also recognizes honeypots used to detect and stop the attacker.

6.2 Attacker Learning and Stop After Delay Policy

In this section, we describe a policy to stop the attacker, but with a delay, so he or she cannot learn too much about the defender's strategy via a set of probes. According to our Stop After Delay policy, we stop the attacker when he or she has previously exploited a honey vulnerability and he or she is visiting a vulnerability that is dangerous for the user, i.e., $\text{impact}(n, s, v) > \tau$ for some threshold τ .

In this case, the reward and stopping function are as follows:

$$r2(st, a) = \begin{cases} -\sum_{a' \in Acts, (Acts, -, -)=st} utilA(a') & \text{if } \begin{matrix} \text{exploit}(n, s, v) = a \wedge \text{impact}(v) > k \wedge \\ (Acts, -, -) = st \wedge \exists \text{exploit}(v) \in Acts : v \in \text{honey} \end{matrix} \\ utilA(a) & \text{otherwise} \end{cases}$$

$$\text{stop2}(st) = \begin{cases} \text{true} & \text{if } (\sum_{a' \in \text{Acts}} \text{costA}(a') + \min_{a' \in \forall A(st)} \text{costA}(a') > \hat{c} \text{ or} \\ & \text{or } \text{exploit}(v) = a \wedge (\text{impact}(v) > k \wedge \exists \text{exploit}(n, s, v) \in \text{Acts} : v \in \text{honey})) \\ \text{false} & \text{otherwise} \end{cases}.$$

The first case says that the attacker's reward is negative of the sum of the utilities of all the actions performed by the attacker if he or she has been previously detected. The second case says that, otherwise, the attacker's reward is the utility of the action. Thus, the attacker gets utility for actions if undetected but loses utility if he or she is detected.

6.3 Extended Defender Strategy Based on Stop Policy Threshold

As discussed in the two previous sections, the delayed stop policy can be effective in stopping the attacker. However, deciding exactly how to fix the stop policy threshold τ can be challenging, because it is dependent on the specific enterprise network and the specific characteristics (i.e., utility and cost) of the attacker.

We estimate it directly with the multi-objective optimization procedure. To this end, we add τ to the definition of defender strategy as follows:

Definition 6.3 (Extended Defender Strategy). An extended defender strategy δ is a pair (A, R, τ) , where $A \subseteq AB$, $R \subseteq RB$, and $\tau \in [0, \infty]$ is the threshold of the delayed stop policy.

Please note that in the extended definition, a defender strategy consists of A , R , and τ . Thus, the proposed HARE-NSGA-II will optimize not only A and R but also τ . This extended defender strategy can be implemented by inserting τ (i.e., one real value) at the last position of HARE-NSGA-II's internal representations of defender strategies. The search space of τ is defined as follows:

- τ equal to 0 means to use the stop policy without any delay
- τ equal to a constant C means to use the stop policy with delay that use stop policy threshold C .
- τ equal to ∞ means not to use the stop policy.

Our optimization algorithm will decide the best option among these options. Section 7.1.6 shows experiment results on computing the Stackelberg Pareto Optimal Defender Extended Strategy.

7 EXPERIMENTS

Our code was implemented in Java with with NSGA2 (Deb et al. 2000) and the UCT reinforcement learning library (Kocsis and Szepesvári 2006) (so the attacker could learn a model of the defender's strategy). All experiments ran on a 64-node cluster with 2.40GHz Intel Xeon CPUs and 24GB RAM.

We used the enterprise network data collected for Jajodia et al. (2011). Vulnerability data came from NIST's National Vulnerability Database (NIST 2017) and included *Exploitability* and *Impact* scores, ranging from 0 to 10. Attackers receive an exploitability value as a utility for each *exploit* action and a certain fixed value f , where $f \in \{1, 3, 5, 7, 9\}$, for *scan* actions.

We considered five different networks with characteristics shown in Table 1. The Small network is from Jajodia et al. (2011). The Medium and Large are created after duplicating subnets of the Small network. NS2-Medium and NS2-Large are well-known network examples provided by NS-2, one of the most popular network simulators around today. System vulnerabilities of a node in the NS2-Medium and NS2-Large networks were copied from a randomly chosen node in the Small network, because NS-2 only provides network topologies.

Algorithm Parameters. The UCT reinforcement learning algorithm (Kocsis and Szepesvári 2006) was used for learning by the attackers—the number of samples used in the learning process was

Table 1. Enterprise Networks and Associated Vulnerability Dependency Graphs Used in Experiments

Network Size	N	E	SV	EV	Vulnerability Data Source
Small	39	475	198	2636	NVD Exploitability and Impact
Medium	118	2343	624	11754	NVD Exploitability and Impact
Large	217	3943	1185	23269	NVD Exploitability and Impact
Synthetic Medium (NS2-Medium)	100	177	1270	30759	NVD Exploitability and Impact
Synthetic Large (NS2-Large)	600	1228	6291	131442	NVD Exploitability and Impact

varied over the set $\{100, 200, 500, 1000, 2000\}$ for Small, $\{500, 1000, 2500, 5000, 10000\}$ for Medium and NS2-Medium and $\{1000, 2000, 5000, 10000, 20000\}$ for Large and NS2-Large networks.⁶

The attacker cost threshold $\hat{c}$ was set to $r \times \sum_{a \in \text{SC}() \cup \text{EXP}(G)} \text{costA}(a)$, where $r \in \{0.3, 0.5, 0.8\}$. Thus, we varied the budget available to the attackers.

Comparison with Greedy Optimization. Many cybersecurity-related problems are modeled as single-objective optimization problems rather than multi-objective ones. For resource allocation problems, however, multi-objective optimization is sometimes preferred. In practice, genetic algorithms effectively solve both single-objective and multi-objective optimizations. Srinivasa (2012), Viegas et al. (2017), and Qian et al. (2015) use genetic algorithms for various security problems. NSGA-II is one of the most popular baseline multi-objective optimization algorithms.

Greedy algorithms are also extensively used for optimization and form another very popular baseline. In certain cases of single-objective optimizations, approximation guarantees exist. There is no guarantee, due to the more complex nature of our problem. Srinivasa (2012) compares a greedy multi-objective optimization method with an exact method and shows excellent performance—one reason we chose a greedy algorithm as a baseline.

Therefore, we want to compare solutions found by the proposed method with other solutions calculated by the simple greedy algorithm that first patches, removes, and protects the most impactful vulnerabilities and nodes for a given budget $\mathcal{B} = (\text{pbudget}, \text{prbudget}, \text{nbudget})$ —each represents a budget for vulnerability patches, software removals, and honey node deployments. The greedy algorithm runs very fast and can sometimes provide high-quality defender strategies. We show how the proposed framework outperforms the greedy algorithm.

Note that a separate budget is defined for each type of defensive actions in $\mathcal{B}$. We designed a greedy algorithm that chooses the biggest gain-to-cost ratio defensive action for each type every iteration until the budget runs out. The gain-to-cost ratio can be defined for each defensive action: (i) $\frac{\text{impact}(v)}{\text{pcostD}(v)}$ for patching a vulnerability v , (ii) $\frac{\sum_{v \in s} \text{impact}(v)}{\text{prcostD}(s)}$ for removing software s , and (iii) $\frac{\sum_{n \in \text{Nei}(h)} \sum_{v \in n} \text{impact}(v)}{\text{ncostD}(h)}$ for deploying a honey node h , where $\text{Nei}(h)$ is a set of neighbor nodes of a honey node h and, thus, the numerator is the sum of the impact scores of all connected vulnerabilities. Because we usually put a honey node in a strategic location that is connected from many other non-honey nodes, this gain-to-cost definition is valid.

We retrieve a patch cost p , productivity cost pr , and node cost nc from a calculated Pareto point (i.e., a defender strategy calculate by our method) and use them as a budget configuration $\mathcal{B} = (p, pr, nc)$ for fair comparison. For each budget configuration, we calculated the best greedy

⁶In our experiments, we used up to 20K attempts not because the attacker would use 20K attacks in the real world but because we wanted to see how many attacks were needed before the attacker's payoff stabilizes. This number is closer to 2K in reality than to 20K. But we needed the experiments going up to 20K to make sure that there would be no unexpected jump. Last but not least, the class of "noisy-rich" attacks described by Lippmann et al. (2012) envisage multiple repeated attacks, although we cannot say exactly how many attacks will occur.

Table 2. Run Time of SHARE-NSGA-II Algorithm

Network Size	Total Runtime	Network Size	Total Runtime	Network Size	Total Runtime
Small	14 hours	Medium	37 hours	Large	74 hours
NS2-Medium	27 hours	NS2-Large	89 hours

(a) Node cost vs. attacker max impact (b) Patch cost vs. attacker max impact (c) Productivity cost vs. sum of impacts

Fig. 2. Results achieved with Monte Carlo Simulations for Medium networks.

solution and compared it with the calculated Pareto point in terms of the max and sum impact value.

7.1 Experimental Results

7.1.1 Runtime.

HYPOTHESIS 1. *Our first hypothesis was that the runtime of SHARE would increase linearly in the size of the enterprise network.*

Table 2 reports the runtime of SHARE-NSGA-II on the three networks we tested with. Though SHARE takes a long time, its runtime increases linearly with the number of nodes. Fortunately, SHARE-NSGA-II is an off-line algorithm and only needs to be run when either the enterprise network changes or the SVDG changes.

7.1.2 Attack Impact without Reinforcement Learning. Many other works assume that the attacker does not learn from his or her past mistakes. Our next experiment proceeds under this naive assumption.

HYPOTHESIS 2. *Our hypothesis was that when attackers are assumed to be dumb (i.e., they do not learn), the Stop Upon Detection strategy can protect the enterprise network, irrespective of whether we measure impact in terms of the sum of the impacts of nodes that they compromised or in terms of the max impact of network nodes compromised.*

METHOD: We ran SHARE-NSGA-II on all four networks to find the Pareto frontiers with the Monte Carlo approximation.

FINDING: Figure 2(a), (b) and (c)⁷ shows the max and the sum of attack impacts when the attacker model described in Section 3 is used with the Monte Carlo approximation. We generated these graphs by first finding all defender strategies in the Pareto frontier. We projected out all cost and impact (max and sum) values across all Pareto-optimal defender strategies for each of the five network sizes (Small, Medium, Large, NS2-Medium, and NS2-Large) and drew a fitted line when the Stop Upon Detection (SUD) and Stop After Delay (SAD) policies were used. We see that if attackers

⁷We listed all charts in the appendix for space reason.

Fig. 3. Results achieved with reinforcement learning for NS2-Medium networks.

do not learn from past attacks, then stopping them immediately on detection (SUD, blue line) is preferable in all five networks. Another blue line marked by stars shows the performance of the greedy optimization. Clearly, greedy defender strategies perform worse than defender strategies in the Pareto frontier. The baseline greedy algorithm does not consider node positions in a network but only impact values. Thus, it fails to patch or remove vulnerabilities or deploy honeys strategically even though impact values are small. However, the difference becomes small if the budget $\mathcal{B}$ is large, which is very natural. Charts for Small and Large networks can be founded in the appendix.

7.1.3 Intelligent Attackers Who Learn from Experience. Attacker learn from experience—as time goes by they will become more sophisticated and potentially recognize honey nodes. Although the exact method of learning may vary from attacker to attacker, we use reinforcement learning to model attackers who learn. (By varying the amount of time that the reinforcement learning algorithm has to run, we vary the quality of learning that attackers might have; Section 7.1.4 describes such experiments.)

HYPOTHESIS 3. *Here, we looked at intelligent attackers who learn from past experiences when they were blocked. We hypothesized that in the case of intelligent attackers, Stop After Delay would be a better strategy than Stop Upon Detection.*

METHOD: We tested the Pareto frontier with attack sequences generated by the UCT reinforcement learning algorithm.

FINDING: The results, shown in Figure 3(a), (b), and (c), show dramatic improvements in attacker efficiency w.r.t. the Monte Carlo approximation with the SUD (blue line). Figure 3(a) and (b) shows the max impact of vulnerabilities exploited by attackers across all Pareto-optimal defender strategies. As we incur more cost by adding honey vulnerabilities and patching existing vulnerabilities, the max impact decreases. Figure 3(c) shows the sum of impact values for various productivity costs. Significantly, when the SUD policy is used, attackers could accurately learn which nodes have honey vulnerabilities—hence, the values depicted with the green lines (RL/SUD) were higher than the purple lines. In fact, the green lines for (RL/SUD) are the biggest upper bounds attackers could get across all experiments, because attackers quickly learn exactly what nodes to avoid. Charts for Small and Large networks can be founded in the appendix.

The SAD policy offers a better-long term defense against intelligent adversaries. We used SAD with two different threshold values: 2 and 4, corresponding to “Low” and “Medium” vulnerability levels as defined by NIST’s National Vulnerability Database. Each was drawn with orange and square-marked (RL/SAD = 2, Threshold = 4) or green and diamond-marked (RL/SAD = 2, Threshold = 2) lines, respectively. With SAD, attackers could not precisely learn where the honey vulnerabilities are, leading to lower impact values. Of course, some honey nodes were known to attackers even with the SAD policy—so, in some cases, the green line is higher than the blue line

Fig. 4. Results achieved when attackers perform sub-rational attacks for an NS2-Medium network.

(MC/Stop = 0). *In general, when defending against intelligent attackers, Stop After Delay is a more effective defense in the long run than Stop Upon Detection.* Some more charts are in the appendix.

We compare, by using a statistical paired t -test, the best performing method (i.e., RL/SAD = 2, Threshold = 2) with the greedy baseline (i.e., G/RL/SAD = 2, Threshold = 2). In this greedy baseline evaluation, the attacker still use reinforcement learning to learn from experience. The greedy method is shown in yellow with circle markers. Because its defensive strategy is not as good as the best case, attackers could exploit more vulnerabilities. We conclude that the average of the max impact difference between the greedy solution and the Pareto point is 0.8 (i.e., in comparison with the greedy solutions, we can decrease the max impact value of exploited vulnerabilities by 0.8 when using the Pareto points). In rare cases, the greedy solution outperformed the Pareto point (i.e., the minimum difference is -0.1). However, the mean value of 0.8 is statistically meaningful from the very low p -value (0.000025). This low p -value says that the Pareto point gives a better performance than the greedy solution in a very stable manner.

7.1.4 Experiment with Random and Bounded Sub-Rational Attacker Behavior. In many real-world scenarios, it is clear that attackers do not always adopt strategies that are the most rational. For instance, they may use a different objective function than the one the defender believes makes sense. Alternatively, they may not be thinking game theoretically at all. In both cases, the attackers achieve different results. We define two ways to model sub-rational attacker behavior. The first is to assume that attackers use utility values that the defender does not know. A second option is to decrease the time allowed for reinforcement learning, making them learn less than they could. We did both.

HYPOTHESIS 4. *As a baseline, we tested how robust SHARE is when an attacker uses a random strategy.*

METHOD: To capture the first method, instead of using exploitability and impact values from the National Vulnerability database, we used random utility values. Under this modification, the orange curves in Figure 4(a), (b), and (c) show that attackers achieve lower max and sum impact values than the ones obtained with our initial assumptions about attacker behavior.

FINDING: Thus, as one would expect, random attackers usually are less effective than attackers that learn imperfectly. However, there can be cases when random attackers could be almost as destructive as attackers that use the initial assumptions we made about attacker behavior (cf. Figure 4(c)). Charts for Small and Large networks can be found in the appendix.

HYPOTHESIS 5. *We hypothesized that as attackers' learning abilities got more and more sophisticated, they would be able to launch more impactful attacks.*

METHOD: Like most researchers in cybersecurity, we did not have data about the learning abilities of real-world attackers. To simulate the sophistication of attackers' learning skills, we varied the

Fig. 5. Results achieved when they are differently trained for a Medium network.

Fig. 6. Results achieved when 20% vulnerabilities are unknown to the defender for an NS2-Medium network.

amount of time that they could run the reinforcement learning algorithm—the longer it is allowed to run, the more sophisticated the attacker’s learning from past experience.

FINDING: As a consequence, this allows us to capture the diversity of learning capabilities of attacks ranging from poor to very sophisticated. Figure 5 and Figure 6 show that the max impact attackers can achieve depends on the quality of learning, and the max impact values saturate after a certain amount of time for learning has elapsed. For example, irrespective of whether the attacker has 1000s or 2000s for learning, the max impact value in Figure 5(a) is unchanged. Charts for Small and Large networks can be founded in the appendix.

7.1.5 Experiment with Incomplete Information. In some cases, the defender does not know all vulnerabilities in his or her enterprise network. This, for instance, would happen with zero-day attacks. It could also happen because the defender has done a sloppy job in identifying known vulnerabilities in his or her network (e.g., if he or she does not diligently check the National Vulnerability Database or does not bother to keep track of alerts from CERT-like organizations). As a consequence, the defender is unable to detect exploits that target such vulnerabilities. However, the attacker might be able to exploit such vulnerabilities, because he or she knows they can be used (after scanning) but the defender does not.

HYPOTHESIS 6. *We hypothesized that if the defender’s knowledge of his or her network is poor, then the impact of the adversary’s attacks would increase.*

METHOD: We randomly choose 20% of system vulnerabilities and excluded from the defender’s detection and re-evaluate the Pareto frontiers.

FINDING: Figures 27, 28, and 29 show that impact values are increased when a randomly chosen 20% of system vulnerabilities are unknown to the defender. However, its difference is not large, because the attacker also does not know which vulnerabilities are unknown to the defender, and

hence he or she must proceed with caution. Charts for Small and Large networks can be founded in the appendix.

7.1.6 Optimization of Stop-After-Delay Threshold. We introduce one more additional experiment for deciding the best SAD threshold. In this experiment, we use the definition of extended defender strategy in Section 6.3.

HYPOTHESIS 7. *We hypothesized that the SAD threshold values that lead to the best defensive performance mostly ranges from 2 to 4.*

FINDING: Because we added τ to δ , each Pareto point has different SAD thresholds. We calculated their frequencies and, in the majority of cases, $\tau \in [2, 4]$. For less than 20% of cases, where many defensive actions are taken so that their defensive performance does not rely on τ , thresholds were out of the range.

One major drawback of optimizing τ with defensive actions (A, R) is that it needs more time to optimize as we more variables to optimize. For large networks, this create the scalability issue. We show other additional experiment results to reveal more detailed characteristics of the proposed evolutionary multi-objective optimization platform in the appendix.

8 RELATED WORK

The problem of defending enterprise networks is well studied in the literature from a wide variety of perspectives.

One body of work focuses on the automatic patching of vulnerabilities, and when this is not possible, the extreme solution is the deactivation of products containing vulnerabilities (see Serra et al. (2015)). Dewri et al. (2007, 2012) and Poolsappasit et al. (2012) consider the problem of finding plans for patching vulnerabilities, that are tradeoffs between cost and risk, by using the Pareto analysis. Dewri et al. (2012) also defines a problem involving a game-theory-based solution, but they do not mix this with the Pareto analysis and assume that the attacker does not know the strategy of the defender (we instead assume this is possible through network scans). Moreover, they do not consider deception via the intelligent placement of honey nodes and honey tokens.

There has been much work on honeypot networks (Campbell et al. 2015; Borkar et al. 2011). A comprehensive report about existing honeypot implementations is given in Pouget and Dacier (2003). This report analyzes and compares the advantages and the weaknesses of several implementations. Many of these are able to emulate the honeypot network on just one machine. Though this solution is less expensive than generating real honey nodes, the attacker who probes the network can easily detect the fake nodes. Moreover, honeypots that look too vulnerable might raise the suspicions of attackers. Therefore, there is a need to strategically deploy a honeypot network that carefully considers attacker behavior.

The use of game theory in cybersecurity problems is not new (Lye and Wing 2005; Serra et al. 2015). For instance, Kiekintveld et al. (2015) and Jajodia et al. (2015) study different game models that address strategies for deploying honeypot networks. The authors define a basic honeypot selection game where the defender choses the properties of the network, an extension in which the attacker can use also probe actions to test the network, and a final version where all the attacker strategies are represented by an attack graph. Unfortunately, these game models make restrictive assumptions, such as the perfect rationality of the attacker. Moreover, the space of operations is limited—they do not consider patching, deactivation, and introduction of honey vulnerabilities that can be used in conjunction with honeypot deployments. Shakarian et al. (2014) highlights the importance of honeypot networks not just for the ability to attract the attacker but also to delay

him or her. They define the concept of distraction chains, i.e., a sequence of decoy systems used to entice an adversary to explore useless information to distract him or her from the real network. They study the problem of creating the distraction chains and embedding them in the network. They too do not consider options such as patching, deactivation, which can be used. Moreover, neither of these works considers the possibility that the attacker can learn from our defensive actions. Cai et al. (2009) define a game-based model to hide the honey nodes from the eyes of the attacker. For more examples of game theory and security not strictly related to our framework, see Alpcan and Baar (2010).

Though not related to the honeypot networks, Bao and Musacchio (2009) study the defender's optimal strategy for expelling (or not expelling) an intruder from a system. By formulating the problem as a Markov Decision Process where both the intruder and the defender can learn about each other, they show that the strategy of always expelling the intruder immediately on detection is not the best one. They show that this improves the learning rate of the intruder and, consequently, increases his or her probability of success. An article (Letchford and Vorobeychik 2013) not related to honeypots adopts Stackelberg games to interdict attack plans. The main feature of this game is to obtain strategies for the defender that reduce the impact of the worst attacker strategy (for the defender). In Lou et al. (2017), the authors investigate a Stackelberg game model, in which multiple defenders are charged with protecting disjoint subsets of targets that might be interdependent.

In contrast, our framework always considers a defender-versus-attacker situation where the defender can do multiple things: strategically deploy honeypot networks, patch vulnerabilities, and deactivate software—and decide what to do when an attacker is detected, such as expelling him or her immediately or letting him or her continue. Moreover, using different measures such as cost, productivity, and risks, we can evaluate each defender strategy. The risk is evaluated as an expected value of the impact of all valid attacker strategies. This enables us to easily evaluate the distraction factor of the honeypot networks deployed in the defender strategy. This is one of the main reasons why we adopt a variant of Stackelberg games—our adversarial model allows the use of the expected risk to compute a distraction factor. Durkota et al. (2015) try to use the Stackelberg game with honeypots, but the real issue is to evaluate the distraction factor, which they define as the risk for the defender to learn about the strategy of the attacker that maximizes his or her expected utility. They require that, first, the attacker is fully rational, using only strategies that maximizes the expected utilities (and the the defender is completely sure about the utilities of the attacker). They do not consider the damage that the attacker can cause using suboptimal strategies. Consequently, even small variations in attacker behaviors can cause catastrophic results. In our approach, instead, we minimize the expected risk and then consider all the attacker strategies weighted against the attacker utilities. It follows that we are less sensitive to the attacker behavior variations, because all the attacker strategies are considered. In addition, unlike past work, we assume that the attacker can learn the strategy of the defender, especially when we use honeypot networks to detect intruders and then expel them.

9 CONCLUSION

In this article, we develop an adversarial-theoretic framework for reasoning about attackers in a cyber-security context that focuses on attackers who follow the noisy-rich model of attackers proposed in Lippmann et al. (2012), in which a target is repeatedly attacked over an extended period of time until the attackers achieve their goals, such as in the recent attacks on the US insurer Anthem and the US Office of Personnel Management (OPM) during the time frame from 2014 to 2015. For instance, the Anthem attackers were within the Anthem network for over

7 months⁸ before detection, while the OPM attackers were inside for over a year⁹ before detection. During this time, the attackers were probing and attacking the network in many ways in a manner consistent with the “noisy-rich” model.

Our work is broken down into two parts. In the first part, we study how an attacker might take advantage of his or her initial knowledge of an enterprise network and an SVDG to further explore the network through a combination of scanning and exploit actions. By scanning the network and carry out exploits, the attacker will be able to learn both the nature of the enterprise network and how he or she may “hop” from one vulnerability to another, building out knowledge of a vulnerability dependency graph. We develop a state-based model where each attacker action allows him or her to cause damage¹⁰ to the defender and learn more information about the network. Our framework is rich enough to allow the attacker to use virtually any known vulnerability—at last count, the National Vulnerability Database (NIST 2017) had over 83K vulnerabilities. As a consequence, our framework is able to consider very powerful attackers.

The defender has four options considered in this article. He or she could add honey nodes, add honey vulnerabilities, patch software, and deactivate risky software. We assume that an attacker who visits a honey node or tries to exploit a honey vulnerability is detected by the defender. Via an adversarial analysis, we identify the optimal strategy for the defender to be a combination of these four types of actions that minimizes the maximal expected damage that the attacker can achieve. Because the defender does not precisely know what the attacker will do, this is an expected damage value. Moreover, because the defender has multiple competing objectives, this problem is a Pareto optimization problem that can be solved by any existing algorithm for computing the Pareto frontier. We show appropriate intractability results and propose the SHARE-NSGA-II algorithm that builds on genetic programming algorithms to identify a good, but suboptimal (due to the intractability results), strategy for the defender to deploy. We prove that HARE-NSGA-II is guaranteed to deliver a $(1 - 1/e)$ -approximation of any Pareto point—and to do so in polynomial time.

Once the defender has deployed his or her strategy, he or she still needs to decide what to do when he or she detects an attack. We propose two strategies: SUD and SAD. In SUD, the attacker’s attack is stopped immediately on detection. Our experiments show that while this strategy has the advantage of mitigating the damage caused by the current attack, a savvy attacker can easily learn a model of the defender’s strategy and launch new attacks with potentially huge damage. In contrast, in the case of the Stop After Delay policy, the attacker’s current attack may be slightly more damaging than in the case of SUD. However, because the attacker has more uncertainty about why he or she was stopped (or because he or she draws incorrect inferences about why he or she was stopped), in the long run, the attacker’s ability to damage the enterprise is smaller than in the case of the Stop Upon Detection policy. As a consequence, SUD in general is preferable to SAD.

Our prototype implementation shows that even though computing the defender’s strategy is expensive (taking a few days, depending on the size of the network), once deployed, the defender’s strategy, together with the Stop After Delay policy, works well in practice.

⁸<http://krebsonsecurity.com/2015/02/anthem-breach-may-have-started-in-april-2014/>.

⁹<https://www.wired.com/2016/10/inside-cyberattack-shocked-us-government/>.

¹⁰Taken broadly to include not only explicit actions to disrupt a network but also to attack the defender via, for instance, exfiltration of data.

APPENDIX: TERMINOLOGIES AND DEFINITIONS

Parameter Symbol	Explanation	Section
$\mathcal{EN}$	Enterprise Network	Section 2
SVDG	Software Vulnerabilities Dependency Graph	Section 2
$\text{impact}(v)$	Impact of a vulnerability v when exploited by the attacker	Section 2
$\text{impactData}(d)$	Organizational impact of data when leaked to the attacker.	Section 4
$\text{ncostD}(n)$	Cost for the defender to deploy a credible honey node n .	Section 4
$\text{pcostD}(G^i)$	Cost for the defender to apply a patch.	Section 4
$\text{prcostD}(G^i)$	Cost for the defender to deactivate software.	Section 4
$\text{costA}(a)$	Cost for the attacker to execute an action a .	Section 3
$\text{utilA}(a)$	Utility for the attacker to execute an action a .	Section 3

Vulnerability Impact

The impact of a vulnerability can be measured in many ways. For instance, NIST's National Vulnerability Database (NIST 2017) has an "impact" subscore that measures the impact of a vulnerability if exploited by an adversary. MITRE's Common Weakness Scoring System (The MITRE Corporation 2011) assigns a quantitative measure to the impact of a vulnerability.

The NVD Common Vulnerability Scoring System (CVSS) defines an open standard framework for sharing characteristics and the impacts of IT system vulnerabilities. It quantifies the overall impact considering many factors, such as whether the attacker can achieve complete or partial access control if successful, how many number of authentication steps the attacker should pass before taking the administrator privilege, whether its patch had been already released by vendors, and so on.

MITRE's Common Weakness Scoring System (CWSS) has a software weakness prioritization mechanism distinct from and complementary to NVD CVSS. CWSS incorporates many metrics of CVSS and some more unique ones. For example, CWSS considers Likelihood of Exploit (LE) and Remediation Effort (RE) when deciding the weakness score. Most importantly, CWSS provides an early assessment method with partial information, whereas CVSS assumes almost complete vulnerability information.

Patch Cost

Cavusoglu et al. (2008) estimate patching costs by coalescing the time to identify and test the system after applying the patch. The *negotiated downtime* is the downtime of the system for implementing and testing patch deployment. They define patching cost (pcost) in terms of the amount of "system downtime" required to implement the patches—that time is correlated with both the level of effort (funds) expended by employees in patching. DARPA also ran a project to estimate patching costs as well (Beattie et al. 2002). In Beattie et al. (2002), one more factor, the recovery cost from bad patch, is considered. In many cases, badly patched software should be removed and re-installed, which incurs additional cost. Other authors (Mercuri 2003; Cavusoglu et al. 2006) also give methods to estimate patch costs.

Productivity Cost

The simplest way to evaluate prcost is by setting it to the cost of buying new software to replace an existing, vulnerable piece of code, plus the cost of man-hours involved in installing it. Thus, if one person can deploy it within a single location of a corporate cloud in an hour, the installation cost is just 1 hour of that person's time. On the other hand, if it takes that person one day to do it by going from machine to machine, then the installation cost would be one day of that person's time. A system manager should be able to estimate these quantities.

Node Cost

The node cost consists of system cost and the operational cost to deploy honey nodes. The system cost, which further can be divided into hardware and software cost, can be easily acquired from system vendors. The operational cost is mostly the human resource cost to monitor and maintain deployed honeys.

Utility of the Attacker

$util_A(a)$ should be monotonic in the impact measure (of the vulnerability exploited by action a) in NIST's National Vulnerability database and monotonic w.r.t. the value of the assets present on the node targeted by a . Clearly, the system administrator for the network can get the impact measure from NIST's NVD website and can estimate the latter.

Cost of the Attacker

If we assume that the attacker is buying exploits on the market, then $cost_A$ could be calculated using several existing studies. (i) For instance, on November 18, 2015, the zero-day exploit reseller Zerodium published the cost of various exploits,¹¹ They listed the price of an exploit that could compromise Safari browsers at \$50,000 and that for Chrome browsers at \$80,000. (ii) Separately, Rand Corporation published a comprehensive book on exploit prices, not just for zero days but also for more mundane attacks (e.g., stolen data) (Ablon et al. 2014). A system security manager who is monitoring the black market for exploits can easily find the cost that an attacker might have to pay for different types of exploits, including potential zero days.

APPENDIX: PROOFS

Proof of Theorem 5.2

PROOF. (i) To show the NP-hardness result we show a reduction from the set cover problem that is NP-hard to our Pareto frontier computation. Given a set of objects U , a set SU of subsets of U , the set cover problem consists in finding a set of H such that $|H| \leq k$ and $\cup_{O \in H} O = U$. In this reduction, the network is composed of only one machine whose vulnerabilities are U plus vk . Each vulnerability has impact 0, except for vk , which is equal to 1, and to exploit the vulnerability vk the attacker has to exploit at least one vulnerability in U . Moreover, the unique valid attacker strategy consists of exploiting all the vulnerabilities available (we do not impose any cost for the attacker in exploiting all the vulnerabilities). Therefore, the expected impacts of the attacker $\mathbb{E}[\text{impactDA1}, \mathcal{EN}, G, \hat{c}]$ ($\mathbb{E}[\text{impactDA2}, \mathcal{EN}, G, \hat{c}]$) will be 0 (0) if all the vulnerabilities are patched and 1 (1) otherwise. We impose that the data impacts are both 0. Moreover, we assume that $AB = \emptyset$ and $RB = \{(L, \emptyset) | L \in SU\}$, where the cost of patching for each remove block is equal to 1. Consequently, we have that a defender strategy represents a subset of SU and the patch cost is its size. Then, the other cost is 0 for each defender strategy; consequently, the unique measure is just the two impacts plus the cost. The size of the Pareto frontier is at most 2, since for each value of the impacts (0 or 1) there exists only one best cost value. Then, we assume now to have an algorithm working in polynomial time in the size of the input and in the output, and, therefore, for our specific case, this algorithm is also polynomial, since the output is almost 2. Once we obtain the Pareto frontier, to answer *yes* to the minimum set cover problem we need only to check if the point with impact 0 has a cost less than or equal to K and *no* otherwise. It follows that if there exists such an algorithm, there exists also a polynomial algorithm solving the minimum set cover that is NP-hard. (ii) We will show the result by contradiction. Suppose the algorithm works in polynomial time in both the input and output. Since our reduction in (i) above generates only two

¹¹<https://www.wired.com/2015/11/heres-a-spy-firms-price-list-for-secret-hacker-techniques/>.

Pareto points (independent of the size of the set cover), this means that if there exists an algorithm that works in polynomial time in the size of the input plus the output (all the Pareto points, only two in this case), then there exists an algorithm that is polynomial only in the size of the input. But this means that this hypothetical algorithm solves an NP problem in polynomial time, which proves our case. $\square$

Proof of Theorem 5.4

PROOF. The function negative impact ni can be modeled as the negated mean of the impact of all the attack paths, where all the paths are distributed with a certain probability. By adding a new OIE add block b , we have a new honeypot node that will only distract the attacker and not will bring him or her to an other real node in the original network. The effect on average is to substitute attacks paths in the network with different ones that have a lower or equal impact. Consequently, we have that the average of the impact will be lower, and thus the negated one will be bigger. This means that the monotone property is proved.

Along the same lines, the number of attack paths that can be substituted with ones with a smaller impact on average of A_x by adding b is bigger w.r.t ones that you can obtain in the case of A_y . This is because A_y w.r.t. A_x may contain more blocks, and, consequently, already some path are substituted with other ones with a smaller impact. Finally, it is possible that the number of the attack paths that b will substitute are already substituted in A_y . Therefore, it means that $ni(A_x \cup \{b\}) - ni(A_x) \geq ni(A_y \cup \{b\}) - ni(A_y)$ and the sub-modular property is proved. $\square$

Proof of Theorem 5.5

PROOF. This proof is based on Theorem 1 provided in Friedrich and Neumann (2014) stating that a generic evolutionary bi-objective (where the first function is sub-modular and the second is only a function that returns the criminality of the objects selected) algorithm finds the not-dominated points such that the value of the sub-modular function is an $(1 - 1/e)$ approximation in an expected polynomial time (expected time because the evolutionary algorithm is stochastic). It is easy to see that this problem perfectly matches the generic problem defined in Friedrich and Neumann (2014), because our evolutionary algorithm (or genetic) is maximizing a sub-modular function (minimizing the impact is equivalent to maximizing the negative impact) and minimizing the number of blocks used. Therefore, we can conclude that if we can evaluate exactly (or with a valid approximation) our impact function in polynomial time, then our algorithm will reach an $(1 - 1/e)$ approximation for all the Pareto point in polynomial time as well. $\square$

APPENDIX: MORE CHARTS AND EXPERIMENT RESULTS

Fig. 7. An example of node scan.

Charts of Attack Impact without Reinforcement Learning

Figure 8 and Figure 9 show the results for Small and Large networks when the attacker is not learning from the experience.

(a) Node cost vs. max impact

(b) Patch cost vs. max impact

(c) Productivity cost vs. sum of impacts

Fig. 8. Results achieved with Monte Carlo simulations for Small networks.

(a) Node cost vs. max impact

(b) Patch cost vs. max impact

(c) Productivity cost vs. sum of impacts

Fig. 9. Results achieved with Monte Carlo simulations for NS2-Large network.

Charts of Intelligent Attackers Who Learn from Experience

Figure 10 to Figure 14 show the results for Small and Large networks when the attacker is learning from the experience.

(a) Node cost vs. max impact

(b) Patch cost vs. max impact

(c) Productivity cost vs. sum of impacts

Fig. 10. Results achieved with reinforcement learning for Small networks.

Charts of Experiment with Random and Bounded Sub-Rational Attacker Behavior

Figure 15 and Figure 16 show the results for Small and Large networks when the attacker is learning from the experience.

Charts of Experiment When They Are Differently Trained

Figure 17 and Figure 18 show the results for Small and Large networks when they are differently trained.

Fig. 11. Results achieved with reinforcement learning for Large network.

Fig. 12. The max impact by node and patch cost.

Experiment with Incomplete Information

Figure 19 and Figure 20 show the results for Small and Large networks when they are differently trained.

Fig. 13. The sum of impacts by node and productivity cost.

Experiment with Attackers Who Prefer to Explore vs. Exploit

Some attackers prefer multiple scans before deciding on a target vulnerability to attack, whereas others may directly attack as soon as a vulnerability is scanned. These two different preferences sometimes lead to significantly different deception strategies. Thus, we test the Pareto frontiers with various attacker models, each with different scanning preferences.

HYPOTHESIS 8. *Here, we hypothesized that the use of too many or too few scanning operations would lead to less-effective attack strategies.*

METHOD: We modeled attacker preference for scanning by assigning varying utilities for the scan operation and re-evaluate the Pareto frontiers.

FINDING: Figures 21, 22, and 23 show the results of experiments where we varied the scan utility value. Interestingly, the results are nuanced. Often, too many scan activities in an attacker strategy lead to a quick increase in the sum of costs (to the attacker) and a quick decrease in max impact. Another interesting observation is that relatively few scans are harmful if many honey

vulnerabilities are deployed and many real vulnerabilities are removed. Thus, additional scanning is useful to attackers in poorly defended networks as opposed to well-defended networks where there are lots of honey nodes and where there are few vulnerabilities.

Fig. 14. The max impact by patch and productivity cost.

Fig. 15. Results achieved when attackers perform sub-rational attacks in Small networks.

Experiment with Different Attacker Budgets

Attackers with different budgets show different attack performance. In this hypothesis testing, we will confirm the performance difference according to the budget difference.

HYPOTHESIS 9. We hypothesized that larger attacker budgets would enable them to have higher impact, irrespective of whether we summed impacts of individual exploit actions or took the max.

Fig. 16. Results achieved when attackers perform sub-rational attacks in Large networks.

Fig. 17. Results achieved when they are differently trained for Small networks.

Fig. 18. Results achieved when they are differently trained for Large networks.

METHOD: We varied the attacker's budget $\hat{c}$ and re-evaluate the Pareto frontiers with various budget values.

FINDING: Figures 24, 25, and 26 vary $\hat{c}$ and show that when $\hat{c}$ is high, max impact values are high as well, which means the attacker can achieve better results.

Experiment with Incomplete Information

In some cases, the defender does not know all vulnerabilities in his or her enterprise network. This, for instance, would happen with zero-day attacks. It could also happen because the defender has done a sloppy job in identifying known vulnerabilities in his or her network (e.g., if he or she does not diligently check the National Vulnerability Database or does not bother to keep track of

alerts from CERT-like organizations). As a consequence, the defender is unable to detect exploits that target such vulnerabilities. However, the attacker might be able to exploit such vulnerabilities, because he or she knows they can be used (after scanning) but the defender does not.

HYPOTHESIS 10. *We hypothesized that if the defender's knowledge of his or her network is poor, then the impact of the adversary's attacks would increase.*

Fig. 19. Results achieved when 20% vulnerabilities are unknown to the defender for Small networks.

Fig. 20. Results achieved when 20% vulnerabilities are unknown to the defender for Large networks.

Fig. 21. The node cost vs. the max impact attackers achieved for various scan utility values.

METHOD: We randomly choose 20% of system vulnerabilities and excluded from the defender's detection and re-evaluate the Pareto frontiers.

FINDING: Figures 27, 28, and 29 show that impact values are increased when a randomly chosen 20% of system vulnerabilities are unknown to the defender. However, its difference is not large, because the attacker also does not know which vulnerabilities are unknown to the defender and hence he or she must proceed with caution.

Optimization of Stop-After-Delay Threshold

In all previous experiments, we fixed the SAD threshold to 2 or 4 based on our preliminary experiments. In fact, this threshold also can be optimized as we optimize the defensive strategy. We introduce one more additional experiment we conducted for deciding the best SAD threshold.

HYPOTHESIS 11. *We hypothesized that the SAD threshold values that lead to the best defensive performance mostly range from 2 to 4.*

Fig. 22. The productivity cost vs. the max impact attackers achieved for various scan utility values.

Fig. 23. The patch cost vs. the max impact attackers achieved for various scan utility values.

Fig. 24. The node cost vs. the max impact attackers achieved when they are differently budgeted.

METHOD: We extended the definition of defensive strategy to $\delta = (A, R, ths)$ and optimized the SAD threshold ths together with defensive actions.

FINDING: Because we added ths to δ , each Pareto point has different SAD thresholds. We calculated their frequencies, and in the majority of cases, $ths \in [2, 4]$. For less than 20% of cases, where many defensive actions are taken so that their defensive performance does not rely on ths , thresholds were out of the range.

One major drawback of optimizing ths with defensive actions (A, R) is that it needs more time to optimize as we more variables to optimize. For large networks, this creates the scalability issue.

Fig. 25. The productivity cost vs. the max impact attackers achieved when they are differently budgeted.

Fig. 26. The patch cost vs. the max impact attackers achieved when they are differently budgeted.

Fig. 27. The node cost vs. the max impact attackers achieved when 20% vulnerabilities are unknown to the defender.

Fig. 28. The productivity cost vs. the max impact attackers achieved when 20% vulnerabilities are unknown to the defender.

Fig. 29. The patch cost vs. the max impact attackers achieved when 20% vulnerabilities are unknown to the defender.

REFERENCES

- L. Ablon, M. C. Libicki, and A. Golay. 2014. *Markets for Cybercrime Tools and Stole Data: Hackers Bazaar*. Technical Report. Retrieved from http://www.rand.org/pubs/research_reports/RR610.html.
- Palvi Aggarwal, Zahid Maqbool, Antra Grover, V. S. Pammi, Saumya Singh, and Varun Dutt. 2015. Cyber security: A game-theoretic analysis of defender and attacker strategies in defacing-website games. In *Proceedings of the International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA'15)*. IEEE, 1–8.
- Tansu Alpcan and Tamer Baar. 2010. *Network Security: A Decision and Game-Theoretic Approach* (1st ed.). Cambridge University Press.
- Bo An, David Kempe, Christopher Kiekintveld, Eric Shieh, Satinder Singh, Milind Tambe, and Yevgeniy Vorobeychik. 2012. Security games with limited surveillance. In *Proceedings of the Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence (AAAI'12)*. 1241–1248.
- Ning Bao and John Musacchio. 2009. Optimizing the decision to expel attackers from an information system. In *Proceedings of the 47th Annual Allerton Conference on Communication, Control, and Computing*. 644–651.
- Tamer Başar and Geert Jan Olsder. 1998. *Dynamic Noncooperative Game Theory*. SIAM.
- Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. 2002. Timing the application of security patches for optimal uptime. In *Proceedings of the Large Installation System Administration Conference (LISA'02)*, Vol. 2. 233–242.
- Maya Bercovitch, Meir Renford, Lior Hasson, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2011. HoneyGen: An automated honeytokens generator. In *Proceedings of the IEEE Conference on Intelligence and Security Informatics (ISI'11)*. IEEE, 131–136.
- Anita Borkar, Akshaya Salunke, Ankita Barabde, and N. P. Karlekar. 2011. Honeypot: A survey of technologies, tools and deployment. In *Proceedings of the International Conference on Web Engineering and Technology (ICWET'11)*. ACM, New York, NY, 1357–1357. DOI: <http://dx.doi.org/10.1145/1980022.1980327>
- Jin-Yi Cai, Vinod Yegneswaran, Chris Alfeld, and others. 2009. *An Attacker-Defender Game for Honeynets*. Springer, Berlin, 7–16.
- R. M. Campbell, K. Padayachee, and T. Masombuka. 2015. A survey of honeypot research: Trends and opportunities. In *Proceedings of the 2015 10th International Conference for Internet Technology and Secured Transactions (ICITST'15)*. 208–212. DOI: <http://dx.doi.org/10.1109/ICITST.2015.7412090>
- Huseyin Cavusoglu, Hasan Cavusoglu, and Jun Zhang. 2006. Economics of security patch management. In *Proceedings of the Workshop on the Economics of Information Security (WEIS'06)*.
- Hasan Cavusoglu, Huseyin Cavusoglu, and Jun Zhang. 2008. Security patch management: Share the burden or share the damage? *Manage. Sci.* 54, 4 (2008), 657–670.
- Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2000. A fast elitist multi-objective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* 6, 2 (2000), 182–197.
- Rinku Dewri, Nayot Poolsappasit, Indrajit Ray, and Darrell Whitley. 2007. Optimal security hardening using multi-objective optimization on attack tree models of networks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*. ACM, 204–213.
- Rinku Dewri, Indrajit Ray, Nayot Poolsappasit, and Darrell Whitley. 2012. Optimal security hardening on attack tree models of networks: A cost-benefit analysis. *Int. J. Inf. Secur.* 11, 3 (2012), 167–188.
- Gurpreet Dhillon and Gholamreza Torkzadeh. 2001. Value-focused assessment of information system security in organizations. In *Proceedings of the International Conference on Information Systems (ICIS'01)*, Veda C. Storey, Sumit Sarkar, and Janice I. DeGross (Eds.). Association for Information Systems, 561–566.

- John P. Dickerson, Gerardo I. Simari, V. S. Subrahmanian, and Sarit Kraus. 2010. A graph-theoretic approach to protect static and moving targets from adversaries. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*, Vol. 1. IFAAMAS, 299–306.
- Karel Durrkötter, Viliam Lisý, Branislav Bošanský, and Christopher Kiekintveld. 2015. Optimal network security hardening using attack graph games. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15)*. AAAI Press, 526–532. <http://dl.acm.org/citation.cfm?id=2832249.2832322>.
- Tobias Friedrich and Frank Neumann. 2014. *Maximizing Submodular Functions under Matroid Constraints by Multi-objective Evolutionary Algorithms*. Springer, 922–931.
- Parke Godfrey, Ryan Shipley, and Jarek Gryz. 2007. Algorithms and analyses for maximal vector computation. *Int. J. VLDB* 16, 1 (2007), 5–28.
- Sergiu Hart. 1992. Games in extensive and strategic forms. In *Handbook of Game Theory with Economic Applications* (1 ed.), R. J. Aumann and S. Hart (Eds.). Vol. 1. Elsevier, 19–40.
- Eric M. Hutchins, Michael J. Cloppert, and Rohan M. Amin. 2011. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. In *Proceedings of the 6th Annual International Conference on Information Warfare and Security* 1 (2011), 80.
- Sushil Jajodia and Steven Noel. 2010. Topological vulnerability analysis. In *Cyber Situational Awareness*, Sushil Jajodia, Peng Liu, Vipin Swarup, and Cliff Wang (Eds.). Advances in Information Security, Vol. 46. Springer, 139–154.
- Sushil Jajodia, Steven Noel, Pramod Kalapa, Massimiliano Albanese, and John Williams. 2011. Cauldron: Mission-centric cyber situational awareness with defense in depth. In *Proceedings of the Military Communications Conference (MILCOM'11)*.
- Sushil Jajodia, Paulo Shakarian, V. S. Subrahmanian, Vipin Swarup, and Cliff Wang (Eds.). 2015. *Cyber Warfare—Building the Scientific Foundation*. Advances in Information Security, Vol. 56. Springer. DOI: <http://dx.doi.org/10.1007/978-3-319-14039-1>
- Ralph L. Keeney. 1992. *Value-Focused Thinking: A Path to Creative Decisionmaking/Ralph L. Keeney*. Harvard University Press, Cambridge, MA, xvi, 416.
- Ralph L. Keeney. 1996. Value-focused thinking: Identifying decision opportunities and creating alternatives. *Eur. J. Operat. Res.* 92, 3 (1996), 537–549. DOI: [http://dx.doi.org/10.1016/0377-2217\(96\)00004-5](http://dx.doi.org/10.1016/0377-2217(96)00004-5)
- Christopher Kiekintveld, Manish Jain, Jason Tsai, James Pita, Fernando Ordóñez, and Milind Tambe. 2009a. Computing optimal randomized resource allocations for massive security games. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*. 689–696. <http://dl.acm.org/citation.cfm?id=1558013.1558108>.
- Christopher Kiekintveld, Manish Jain, Jason Tsai, James Pita, Fernando Ordóñez, and Milind Tambe. 2009b. Computing optimal randomized resource allocations for massive security games. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*. 689–696.
- Christopher Kiekintveld, Viliam Lisý, and Radek Pibil. 2015. Game-theoretic foundations for the strategic use of honeypots in network security. In *Cyber Warfare—Building the Scientific Foundation*. 81–101. DOI: http://dx.doi.org/10.1007/978-3-319-14039-1_5
- A. Kim and M. H. Kang. 2011. Determining Asset Criticality for Cyber Defense. Retrieved from www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA550373.
- Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML'06)*. Springer-Verlag, Berlin, 282–293. DOI: http://dx.doi.org/10.1007/11871842_29
- Dmytro Korzhuk, Zhengyu Yin, Christopher Kiekintveld, Vincent Conitzer, and Milind Tambe. 2011. Stackelberg vs. nash in security games: An extended investigation of interchangeability, equivalence, and uniqueness. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*. 1139–1146.
- Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P. Preparata. 1975. On finding the maxima of a set of vectors. *J. ACM* 22, 4 (1975), 469–476.
- Joshua Letchford and Yevgeniy Vorobeychik. 2013. Optimal interdiction of attack plans. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS'13)*. 199–206. <http://dl.acm.org/citation.cfm?id=2484920.2484955>.
- R. P. Lippmann, J. F. Riordan, T. H. Yu, and K. K. Watson. 2012. *Continuous Security Metrics for Prevalent Network Threats: Introduction and First Four Metrics*. Technical Report. DTIC Document.
- J. Lou, A. M. Smith, and Y. Vorobeychik. 2017. Multidefender security games. *IEEE Intell. Syst.* 32, 1 (Jan 2017), 50–60. DOI: <http://dx.doi.org/10.1109/MIS.2017.11>
- Kong-wei Lye and Jeannette M. Wing. 2005. Game strategies in network security. *Int. J. Inf. Secur.* 4, 1-2 (2005), 71–86. DOI: <http://dx.doi.org/10.1007/s10207-004-0060-x>
- Rebecca T. Mercuri. 2003. Analyzing security costs. *Commun. ACM* 46, 6 (2003), 15–18.
- J. Mirkovic and P. Reiher. 2004. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Comput. Commun. Rev.* 34, 2 (2004), 39–53.

- NIST. 2017. National Vulnerability Database. Retrieved from <http://nvd.nist.gov>.
- Praveen Paruchuri, Jonathan P. Pearce, Janusz Marecki, Milind Tambe, Fernando Ordonez, and Sarit Kraus. 2008. Playing games for security: An efficient exact algorithm for solving bayesian stackelberg games. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*. 895–902.
- James Pita, Manish Jain, Janusz Marecki, Fernando Ordóñez, Christopher Portway, Milind Tambe, Craig Western, Praveen Paruchuri, and Sarit Kraus. 2008. Deployed ARMOR protection: The application of a game theoretic model for security at the los angeles international airport. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*. 125–132.
- Nayot Poolsappasit, Rinku Dewri, and Indrajit Ray. 2012. Dynamic security risk management using bayesian attack graphs. *IEEE Trans. Depend. Secur. Comput.* 9, 1 (2012), 61–74.
- Fabien Pouget and Marc Dacier. 2003. Honeypot, honeynet: A comparative survey. In *Institut EurAl'com*.
- Chao Qian, Yang Yu, and Zhi-Hua Zhou. 2015. On constrained boolean pareto optimization. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI'15)*. AAAI Press, 389–395.
- Edoardo Serra, Sushil Jajodia, Andrea Pugliese, Antonino Rullo, and V. S. Subrahmanian. 2015. Pareto-optimal adversarial defense of enterprise systems. *ACM Trans. Inf. Syst. Secur.* 17, 3, Article 11 (March 2015), 39 pages. DOI : <http://dx.doi.org/10.1145/2699907>
- Asaf Shabtai, Yuval Elovici, and Lior Rokach. 2012. Data leakage detection/prevention solutions. In *A Survey of Data Leakage Detection and Prevention Solutions*. Springer, 17–37.
- Paulo Shakarian, Damon Paulo, Massimiliano Albanese, and Sushil Jajodia. 2014. Keeping intrudors at large: A graph-theoretic approach to reducing the probability of successful network intrusions. In *Proceedings of the International Conference on Security and Cryptography (SECRYPT'14)*. 19–30.
- K. G. Srinivasa. 2012. *Application of Genetic Algorithms for Detecting Anomaly in Network Intrusion Detection Systems*. Springer, Berlin, 582–591.
- Colin Tankard. 2011. Advanced persistent threats and how to monitor and deter them. *Network Security* 2011, 8, 16–19.
- The MITRE Corporation. 2011. Common Weakness Scoring System (CWSS™). Retrieved from <http://cwe.mitre.org/cwss/>.
- Eduardo Viegas, Altair Olivo Santin, Andre Franca, Ricardo Jasinski, Volnei A. Pedroni, and Luiz S. Oliveira. 2017. Towards an energy-efficient anomaly-based intrusion detection engine for embedded systems. *IEEE Trans. Comput.* 66, 1 (2017), 163–177. DOI : <http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/TC.2016.2560839>
- Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. 1998. The island model genetic algorithm: On separability, population size and convergence. *J. Comput. Inf. Technol.* 7 (1998), 33–47.

Received September 2016; revised August 2017; accepted August 2017