A Probabilistic Logic of Cyber Deception

Sushil Jajodia[1], Noseong Park[2], Fabio Pierazzi[3], Andrea Pugliese[4], Edoardo Serra[5], Gerardo I. Simari[6], V.S. Subrahmanian[7]
[1] George Mason University, USA
[2] University of North Carolina, USA
[3] University of Modena and Reggio Emilia, Italy
[4] University of Calabria, Italy
[5] Boise State University, USA
[6] Institute for Computer Science and Engineering (ICIC UNS–CONICET), Department of Computer Science and Engineering, Universidad Nacional del Sur (UNS), San Andres 800, (8000) Bahia Blanca, Argentina, gis@cs.uns.edu.ar
[7] University of Maryland, USA

Malicious attackers often scan nodes in a network in order to identify vulnerabilities that they may exploit as they traverse the network. In this paper, we propose that the system generate a mix of true and false answers in response to scan requests. If the attacker believes that all scan results are true, then he will be on a wrong path. If he believes some scan results are faked, he would have to expend time and effort in order to separate fact from fiction. We propose a Probabilistic Logic of Deception (PLD-Logic) and show that various computations are NP-hard. We model the attacker's state and show the effects of faked scan results. We then show how the defender can generate fake scan results in different states that minimize the damage that the attacker can produce. We develop a Naive-PLD algorithm and a Fast-PLD heuristic algorithm for the defender to use and show experimentally that the latter performs well in a fraction of the run-time of the former. We ran detailed experiments to assess the performance of these algorithms and further show that by running Fast-PLD offline and storing the results, we can very efficiently answer run-time scan requests.

I. Introduction

When targeting a network, cyber-criminals must map it out, by understanding the answers to questions such as: which nodes are connected to which nodes? What operating systems are they running? What other types of software are they running? What are the names of associated subnetworks and users? What do the routing tables look like? What are the IP/MAC addresses of devices on the network? Armed with answers to such questions, attackers are better able to target networks in order to wreak maximal havoc. To achieve this, attackers tend to execute a suite of such requests by using tools such as NMAP—*except for security assessments by administrators, there are few legitimate reasons for network scanning*.

We propose a new solution: returning a mix of true and false results to an attacker's scan on a network so that we both increase the time they need to formulate their attack (allowing us additional time to erect our defenses) and increase costs on them (e.g., to determine what is true vs. what is false), potentially giving us a greater chance of interdicting the attacker.

We use the term "scan query" to denote a wide range of inquiries about the network, such as NMAP commands, TCP connection scans, stealth SYN scans, UDP scans, as well as IP protocol scans to determine the IP protocols used by a node, and more—our proposed methods are therefore very general.

Of course, combinations of true and fake results must be given in a *consistent* manner over time, as the same attacker may make many different scan queries. Moreover, the system must reason about the state of the attacker's knowledge of the network as time proceeds. We propose to develop a probabilistic logic theory by which the defender can provide a realistic fake answer to an attacker's scan query and continue to provide fake answers as the attacker probes the network.

A. Overview of the Approach

When an attacker initially targets a network, they know that the true state of the network is one of a very large number of "possible worlds", each reflecting a different configuration of the network. As they probe the network (e.g., through different types of scan operations, or routing table information requests), they learn more and more about the network, enabling them to prune possible worlds that are inconsistent with the results of the scan queries. By iteratively querying the network through a variety of technical devices, the attacker can quickly zero in on vulnerabilities.

However, this pruning of possible worlds is possible only if the attacker's scan queries are *honestly answered* by the system. If the attacker has to consider the possibility that responses to scan queries include fake results, then his task can potentially become exponentially harder. For instance, consider a node with just two types of software running on it: a web server (either Tomcat or Nginx), and a DBMS (either Oracle 12c or MySQL). Prior to scanning this node and/or having any further intelligence about this node, the adversary has 4 possible worlds to consider, depending on which of the 2x2 combinations of web server and database the system has. If the system honestly answers a scan query about a DBMS, this narrows the space of possible worlds down to just 2. On the other hand, if there is a possibility that the system answers dishonestly, then the adversary has to consider all 4 possible worlds even after making their query.

In general, suppose the system has a set $\{\tau_1, ..., \tau_n\}$ of types of software running on it (e.g., the types might be OSs, web servers, DBMSs, etc.) and each software type τ_i has n_i possible candidate software packages of that type that could be running on a node. If our system honestly answers scan queries and an attacker queries the system about m software types (and suppose $n_i = 2$ for all i), then he can reduce the number of possible worlds by a factor of 2^m. But if the system cannot be assumed to be honest, then the attacker's query does not reduce the number of possible worlds at all. And of course, if the attacker is oblivious as to whether the system is honest or can fake results, then the fake results will lead them astray, increasing the time they need to carry out a successful attack and injecting uncertainty into their strategy.

We develop a formal, *Probabilistic Logic of Deception* (PLD-Logic), in which the adversary's knowledge of the system is captured via a set of formulas in the logic. Each PLD-Logic formula implicitly expresses a set of possible worlds that are intuitively "compatible" with that formula. Using PLD-Logic, we will address the following problems:

- define the structure and software running on the network as well as how the attacker can exploit vulnerabilities in order to compromise the network;
- identify what the attacker may learn from the results of a scan query and how they may use that learned knowledge to continue attacking the network; and
- determine how to answer a scan query from the attacker so that their current query—as well as likely future ones—prevent them from learning the true nature of the network to the maximal possible extent by increasing uncertainty.

Our overall technical approach therefore aims to formalize each of these three problems in order to identify how best to respond to scan queries. Note that as we will discuss in Section V, the bulk of the computational effort is performed *offline*; the result of offline processing is essentially a lookup "scan table" that can be used "online" to answer attackers' scan queries—this table can be accessed in milliseconds. Therefore, *computational costs need only be incurred when there are changes in the network.*

B. Plan of the Paper

The remainder of the paper is organized as follows. In Section II we introduce the syntax and semantics of our proposed PLD-Logic. In Section III we formalize attacker behavior and discuss the problem of minimizing damage by selecting the most appropriate answers to his queries. Then, in Sections IV and V we develop two algorithms and present our experimental assessment. Finally, in Sections VI and VII we discuss related work and outline conclusions.

II. PLD-Logic: Syntax and Semantics

In the following we assume the existence of a finite set of constants $\mathcal{C}$ containing all elements of interest (such as network nodes, software packages, known vulnerabilities, etc.), and an infinite set of variables $\mathcal{V}$. We use lowercase strings to denote constants and uppercase ones to denote variables. We also have a finite set $\mathcal{P}$ of predicates, each with an associated arity.

A. Syntax

We start with a logical model of enterprise networks. As usual, a *term* is either a constant or a variable (note that we do not have function symbols in our language). If $t_1, ..., t_n$ are terms and p is an n-ary predicate, $p(t_1, ..., t_n)$ is an *atom*; if all the t_i's are constants, then it is a *ground atom*. For simplicity—and without loss of generality—we assume that atoms are well-formed, i.e., that constants and predicates have types and that they are always respected.

Both the attacker's and the defender's model of a network are built using the following predicate symbols (where $n, n_1, n_2, t, s, x, v \in \mathcal{C}$):

- $\textit{node}(n)$ declares that n is a node;
- $\textit{edge}(n_1, n_2)$ declares the existence of an edge between nodes n_1, n_2;
- $\textit{runs}(n, t, s, x)$ declares that n runs version x of software s, which is of type t;
- $\textit{vuln}(s, x, v)$ declares that version x of software s has vulnerability v.

Formulas over the above predicates are defined in the usual way using $\forall$, $\exists$, $\wedge$, $\vee$, and $\neg$. We use *Nodes* and *Edges* to denote the set of all nodes and edges in the network, respectively. Moreover, we assume the existence of a function—underlying the *vuln* predicate—that identifies vulnerabilities in software packages; NIST's National Vulnerability Database (NVD) [1] can be considered an encoding of such a function.

A *scan query* over a node allows the attacker to query the node regarding its connectivity in the network, as well as what software it is running; we assume that the former data is automatically gained by the attacker when the query is executed, while the latter can be intercepted by the defender in order to return fake answers to the query. The attacker will also perform *exploits* for a specific software and vulnerability in a node; we assume that the attacker will decide to take this action when he is prepared to do so, and that the defender will immediately realize that this occurred.

Example 1: Figure 1 reports an example of a simple network with two "frontier" nodes, i.e. nodes which may be directly accessed from outside of the network (f_1 and f_2, represented as squares with yellow background), and three "internal" nodes (n_1, n_2 and n_3, represented as circles with green background). Dashed squares report software running on the nodes. The *node* and *edge* ground atoms in Figure 1 are $\textit{node}(n_1)$, $\textit{node}(n_2)$, $\textit{node}(n_3)$, $\textit{edge}(f_1, n_1)$, $\textit{edge}(n_1, n_2)$, $\textit{edge}(n_2, n_3)$, $\textit{edge}(f_2, n_3)$, $\textit{edge}(f_1, n_3)$. The ground atoms that represent the software running on node f_1 are $\textit{runs}(f_1, \mathsf{os}, \mathsf{rhel}, \mathsf{v7})$, $\textit{runs}(f_1, \mathsf{web}, \mathsf{tomcat}, \mathsf{v8.0.30})$, $\textit{runs}(f_1, \mathsf{ssh}, \mathsf{openssh}, \mathsf{v7.2p2})$. In addition, we may know that, according to the NVD, CVE-2016-0763 is a vulnerability in *Tomcat 8.0.30*. Hence, we also have an atom representing this: $\textit{vuln}(\mathsf{tomcat}, \mathsf{v8.0.30}, \textit{cve-2016-0763})$. ■

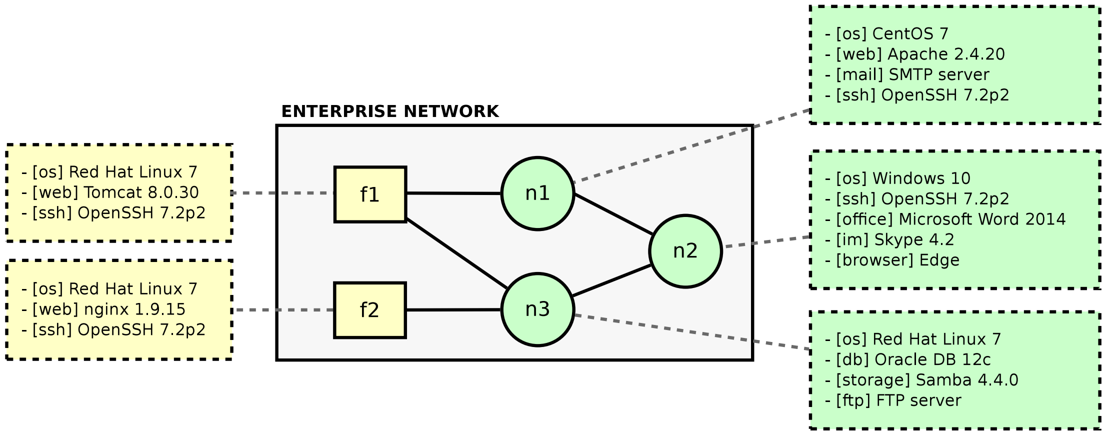


Fig. 1: Simple example of enterprise network.

B. Semantics

We now focus on defining the semantic structures used in our framework. The first structure is a *possible world*, which is simply a set of ground atoms; all atoms in the set are assumed to be true in that possible world, while the rest are false. Satisfaction of formulas in worlds is defined as usual. In order to model extra information about what kinds of situations are possible in the domain, we also assume we have a set of *integrity constraints IC*.

Definition 1 (Integrity Constraint): An *integrity constraint* ic is a statement of the form *head* $\leftarrow$ *body*, where *head* and *body* are logical formulas. Such a constraint is said to be *satisfied* by a set of atoms S, denoted $S \models \{ic\}$, whenever either *body* is not satisfied or *head* is satisfied by S. ■

We use $\mathcal{W}$ to denote the set of all possible worlds and $\mathcal{W}_{IC}$ the subset of $\mathcal{W}$ that satisfies all constraints in *IC*.

Example 2: Suppose that in the enterprise network of Figure 1 each node must have exactly one operating system (e.g., because they are not supposed to run virtual machines on them). To enforce this, the security analyst can define two integrity constraints. The one states that there must exist an OS: $(\forall N)(\exists S)(\exists X)[runs(N, \mathsf{os}, S, X) \leftarrow]$. The second prohibits the existence of two OSs: $(\forall N)(\forall S_1)(\forall S_2)(\forall X_1)(\forall X_2)[S_1 = S_2 \wedge X_1 = X_2 \leftarrow runs(N, \mathsf{os}, S_1, X_1) \wedge runs(N, \mathsf{os}, S_2, X_2)]$. We can also define a set of software types $\mathcal{T}$ (e.g., $\{\mathsf{web}, \mathsf{db}, \mathsf{ssh}\}$) that can appear at most once in each node: $(\forall T \in \mathcal{T})(\forall N)(\forall S_1)(\forall S_2)[S_1 = S_2 \leftarrow runs(N, T, S_1, X) \wedge runs(N, T, S_2, Y)]$. Likewise, the constraint "*number of browsers must be at most 2*" would be expressed as $(\forall N)[\neg(S_1 \neq S_2 \wedge S_2 \neq S_3 \wedge S_1 \neq S_3) \leftarrow \textit{runs}(N, \mathsf{browser}, S_1, X) \wedge \textit{runs}(N, \mathsf{browser}, S_2, Y) \wedge \textit{runs}(N, \mathsf{browser}, S_3, Z)]$. ■

Proposition 1: The problem of determining if there exists a world satisfying a given set of integrity constraints is NP-hard.

Proof sketch. We show a reduction from MAXIMUM INDEPENDENT SET, which is known to be NP-hard, to our problem. The former, given a graph $G = (V_G, E_G)$ and a number k, consists of deciding whether there exists a set $H \subseteq V_G$ with $|H| \geq k$ s.t. for all the edges $(a, b) \in E_G$, it holds that $\{a, b\} \not\subseteq H$. In our reduction we assume that a node in V_G corresponds to a software package that may run on a machine and the set H represents the software packages that are actually running on that machine. By using the integrity constraints IC the same way we did for Example 2, we can impose that there are at least k software packages running on a machine, and for each edge $(a, b) \in E_G$ only one between the software packages a and b is running. It is easy to see that there exists an independent set in G of size k iff there exists a world that satisfies the integrity constraints. ■

Definition 2 (PLD Framework and Scan Query): A *PLD framework* is a pair

$$M = \langle N, \textit{IC} \rangle$$

where N is a set of *node*, *edge*, *runs*, and *vuln* atoms, and *IC* is a set of integrity constraints. Given a PLD framework, we refer to a scan of a specific node (by an alleged attacker) as a *scan query*. ■

Definition 3 (Probabilistic State): A *probabilistic state* σ is a set of annotated atoms of the form

$$A : [\ell, u]$$

where A is a ground atom formed with the *runs* predicate and $[\ell, u] \subseteq [0, 1]$ is a closed probability interval. ■

Intuitively, each annotated atom in a probabilistic state denotes the fact that the probability that a given node runs a specific software of some type is at least ℓ and at most u.

Example 3: Assume that the attacker, after some reconnaissance activities, is in the following initial probabilistic state:

$$\begin{aligned}
\textit{runs}(N, \mathsf{os}, \mathsf{windows}, X) &: [0.4, 0.8] \\
\textit{runs}(N, \mathsf{os}, \mathsf{linux}, X) &: [0.15, 0.5] \\
\textit{runs}(N, \mathsf{web}, \mathsf{tomcat}, X) &: [0.4, 0.75] \\
\textit{runs}(N, \mathsf{web}, \mathsf{nginx}, X) &: [0.5, 0.65] \\
\textit{runs}(f_1, \mathsf{os}, \mathsf{linux}, \mathsf{rhel7}) &: [0.6, 0.85] \\
\textit{runs}(f_1, \mathsf{web}, \mathsf{tomcat}, \mathsf{v8.0.30}) &: [0.6, 0.9] \\
\textit{runs}(f_1, \mathsf{ssh}, \mathsf{openssh}, \mathsf{7.2p2}) &: [0.7, 0.9] \\
\textit{runs}(f_2, \mathsf{os}, \mathsf{linux}, \mathsf{rhel7}) &: [0.6, 0.85] \\
\textit{runs}(f_2, \mathsf{web}, \mathsf{nginx}, \mathsf{v1.9.15}) &: [0.5, 0.8] \\
\textit{runs}(f_2, \mathsf{ssh}, \mathsf{openssh}, \mathsf{v7.2p2}) &: [0.7, 0.9]
\end{aligned}$$

The first four formulas are just a synthetic representation of the a priori knowledge of the attacker on possible software types running in the network. The other formulas contain *runs* atoms for nodes f_1 and f_2 that the attacker can obtain by scanning. We observe that from the defender's perspective there is some uncertainty whether the attacker fully believes what is disclosed to him. For this reason, the probability interval of these *runs* atoms is different from $[1, 1]$. ■

A probabilistic state represents the state of knowledge the attacker might have about the enterprise at a given time. As the attacker issues scan queries to the system, he learns more about the network, causing the state to change. The defender must reason about the attacker's probabilistic state—as well as how it might evolve in the future—in order to reason about the attacker's possible future actions and use them to answer scan queries in a manner that delays his progress or causes maximal uncertainty.

A probabilistic state defines constraints on what probability distribution are valid w.r.t. the space of all possible worlds. In the following, let p_i denote the (unknown) probability of each world $w_i \in \mathcal{W}_{IC}$; for each $A_i : [\ell_i, u_i]$ in a probabilistic state σ, we have the constraint

$$\ell_i \leq \sum_{w_j \in \mathcal{W}_{IC} \wedge A_i \in w_j} p_j \leq u_i.$$

Intuitively, the sum of the probabilities assigned to (constraint-compatible) possible worlds in which A_i is true lies within the specified interval. In addition, we require $\sum_{w_i \in \mathcal{W}_{IC}} p_i = 1$.

Given a probabilistic state σ, we use $LC_M(\sigma)$ to denote the set of linear constraints described above with respect to a PLD framework M; when the framework is clear from context, we write simply $LC(\sigma)$. Moreover, we say that M is *consistent* in state σ if $LC_M(\sigma)$ has a solution.

Example 4: Let us focus on the probability that node f_1 of Example 3 is running windows, linux, tomcat, and/or nginx. Let w_i be the i-th possible world. We have the following list of possible worlds (for the sake of clarity, we use only the initial of each possible software package running on f_1):

$w_0 = \{\emptyset\}$, $w_1 = \{\text{w}\}$, $w_2 = \{\text{l}\}$,
$w_3 = \{\text{t}\}$, $w_4 = \{\text{n}\}$, $w_5 = \{\text{w, l}\}$,
$w_6 = \{\text{w, t}\}$, $w_7 = \{\text{w, n}\}$, $w_8 = \{\text{l, t}\}$,
$w_9 = \{\text{l, n}\}$, $w_{10} = \{\text{t, n}\}$, $w_{11} = \{\text{w, l, n}\}$,
$w_{12} = \{\text{w, l, t}\}$, $w_{13} = \{\text{w, t, n}\}$, $w_{14} = \{\text{l, t, n}\}$,
$w_{15} = \{\text{w, l, t, n}\}$.

Suppose the integrity constraints state that a node has exactly one operating system and at most one web server. Then only the following worlds are possible: $w_1, w_2, w_6, w_7, w_8, w_9$. We recall from Example 3 that for node f_1 the attacker believes that the node is running linux with probability in $[0.6, 0.85]$ and tomcat with probability in $[0.6, 0.9]$. Moreover, the attacker believes that nodes in the network are running windows with probability $[0.4, 0.8]$ and nginx with probability $[0.5, 0.65]$. $LC(\sigma)$ consists of:

$$\begin{aligned} 0.6 \leq p_2 + p_8 + p_9 \leq 0.85 \\ 0.6 \leq p_6 + p_8 \leq 0.9 \\ 0.4 \leq p_1 + p_6 + p_7 \leq 0.8 \\ 0.5 \leq p_7 + p_9 \leq 0.65 \\ p_1 + p_2 + p_6 + p_7 + p_8 + p_9 = 1 \end{aligned}$$

■

III. The Defender's Optimal Actions

We now focus on the problem of choosing the defender's optimal actions.

A. Attacker's Behavior

We begin by formalizing attacker behavior. We assume that the attacker seeks to move around the enterprise network in search of nodes to compromise by exploiting vulnerabilities. For this purpose, we make use of the notion of *utility* of nodes in the network, which we represent via a function.

Definition 4 (Utility of a Node): Let $M = \langle N, IC \rangle$ be a PLD framework. Given possible world $w \in \mathcal{W}_{IC}$, the *utility* of network nodes is given by a function $util : Nodes \rightarrow \mathbb{R}^{\geq 0}$. ■

The following are two examples of utility functions:

$$util_1(n) = \sum_{runs(n,t,s,x) \in w, vuln(s,x,v) \in N} (impact(v) \cdot exploitability(v))$$

$$util_2(n) = \max_{run(n,t,s,x) \in w, vuln(s,x,v) \in N} (impact(v) \cdot exploitability(v))$$

where $impact(v)$ and $exploitability(v)$ are metrics from the NVD that represent the direct consequence of a successful exploit and the ease with which the vulnerability can be exploited, respectively.

In our framework, the attacker scans the neighbors of some known node and receives answers from the defender; by doing this, he builds a *utility array* $[u_1, ..., u_m]$, $u_i = util(n_i)$. We assume that the attacker is smart, i.e. he chooses to attack next, a neighbor with utility greater than some percentile of the utility array multiplied by a given *subrationality factor* in $[0, 1]$—this factor, denoted as *SUB* in the remainder, allows the defender to simulate an attacker making suboptimal decisions which may occur. A completely rational and perfect attacker will be captured in our framework by setting *SUB* to 1.

B. Possible Answers and State Updates

Let $M = \langle N, IC \rangle$ be a PLD framework, Q a scan query over M, and H a set of ground atoms formed with the *runs* predicate. The set of *possible answers* to Q, given that the atoms in H have been provided as answers to past queries, is $possAnswers(Q, H) = \{A_1, ..., A_n\}$ (where each A_i is a set of *runs* atoms) that satisfies the condition that if $H \cup N \models IC$, then $H \cup N \cup A_i \models IC$, for all $A_i \in possAnswers(Q, H)$.

Definition 5 (State Update): Given a probabilistic state σ and a set A of ground atoms formed with the *runs* predicate, the new probabilistic state is obtained by

$$updateState(\sigma, A) = consolidate(\sigma \setminus \sigma' \cup \sigma'')$$

where $\sigma' = \{b : [\ell, u] \mid b : [\ell, u] \in \sigma, b \in A\}$, $\sigma'' = \bigcup_{a \in A}\{a : [1, 1]\}$, and *consolidate*(.) is a function that removes the inconsistencies that can arise in the set of linear constraints after the update of the probabilistic state. ■
Note that the attacker believes everything in the answer.

Our consolidation makes the formulas consistent by widening the probability intervals in $LC(\sigma)$ (which is always possible since in the worst case an interval of $[0, 1]$ is reached). We achieve this via the steps below—recall that for the i-th atom $A_i : [\ell_i, u_i] \in \sigma$, the corresponding constraint in $LC(\sigma)$ has the form $\ell_i \leq \sum_{w_j \in \mathcal{W}_{IC} \wedge A_i \in w_j} p_j \leq u_i$.

1) For every atom $A_i : [\ell_i, u_i] \in \sigma$, (i) replace ℓ_i with a variable ℓ'_i in the i-th constraint and (ii) add a constraint $\ell'_i \leq \ell_i$ to $LC(\sigma)$.
2) For every atom $A_i : [\ell_i, u_i] \in \sigma$, (i) replace u_i with a variable u'_i in the i-th constraint and (ii) add a constraint $u'_i \geq u_i$ to $LC(\sigma)$.
3) Compute $M^* = \mathit{maximize} \sum_i (\beta_i \times (\ell'_i - u'_i))$ *subject to:* $LC(\sigma)$, where β_i is the (user-defined) "believability weight" of the i-th atom.
4) Build the final set of constraints as $LC'(\sigma) = LC(\sigma) \cup \{\sum_i (\beta_i \times (\ell'_i - u'_i)) = M^*\}$.

Example 5: Consider a scenario where σ is

$$\{\mathit{runs}(n_3, \mathsf{os}, \mathsf{linux}, \mathsf{rhel7}) : [0.5, 1], \\ \mathit{runs}(n_3, \mathsf{os}, \mathsf{windows}, \mathsf{v8}) : [0.8, 1]\}$$

If there is an integrity constraint stating that a node can run only one operating system, then the possible worlds are $w_1 = \{\mathsf{linux}\}$ and $w_2 = \{\mathsf{windows}\}$. $LC(\sigma)$ will then include the following constraints:

$$0.5 \leq p_1 \leq 1$$
$$0.8 \leq p_2 \leq 1$$
$$p_1 + p_2 = 1$$

These constraints admit no solution, as the lower bounds of p_1 and p_2 add up to a value greater than 1. Consolidation can make the constraints consistent as follows. Let us assume that $\beta_1 = 1$ is the believability weight for the a priori knowledge of the attacker about node n_3 running linux, whereas $\beta_2 = 50$ is the believability weight about the same node running windows (e.g. if the attacker favors information provided by scan queries over his previous beliefs gained from reconnaissance activities). In this case, $LC(\sigma)$ becomes:

$$\ell'_1 \leq p_1 \leq u'_1$$
$$\ell'_2 \leq p_2 \leq u'_2$$
$$\ell'_1 \leq 0.5, u'_1 \geq 1$$
$$\ell'_2 \leq 0.8, u'_2 \geq 1$$
$$p_1 + p_2 = 1$$

Then, we solve $M^* = maximize\{\ell'_1 - u'_1 + 50 \times (\ell'_2 - u'_2)\}$ *subject to:* $LC(\sigma)$ to obtain a value for M^*, and finally add the constraint $M^* = \ell'_1 - u'_1 + 50 \times (\ell'_2 - u'_2)$ to $LC(\sigma)$. ■

As an alternative, we could widen the probability intervals through an iterative adjustment of the bounds: we start by setting all lower bounds to zero, then we iteratively choose one constraint and do a binary search-style adjustment of its lower bounds to tighten the intervals as much as possible. This can be done for a fixed number of steps, or until some user-defined criterion is satisfied (such as time spent, or the extent to which the bounds are being tightened is small enough).

C. Quantifying Damage

In order to decide among all the possible deceptive answers that the defender could give, we need a way of deciding how certain probabilistic states can be better than others.

First, in order to keep track of the attacker's actions, we store the nodes that become compromised in sequences we call *strategies*, of the form $\lambda = \langle n_1, \ldots, n_k\rangle$, where $n_i \in \mathit{Nodes}$ and the subgraph formed by taking the nodes in λ and their incident edges in *Edges* is connected. We use $\mathcal{A}$ to denote the set of all possible attacker strategies, and sometimes slightly abuse notation and write $n \in \lambda$. Finally, probability distributions over $\mathcal{A}$ in world w are defined as expected: $\Pr_w : \mathcal{A} \to [0, 1]$ s.t. $\sum_{\lambda \in \mathcal{A}} \Pr_w(\lambda) = 1$.

Such probability distributions over strategies can be obtained by a simple procedure that simulates the attacker's behavior (as described before) for a certain number of "future steps" by the attacker (denoted *NSTEPS* in the remainder). This is done multiple times, and the probability of each node being visited is simply derived from the number of times it appears in a simulation (the number of simulations will be denoted with *NSIM*).

Example 6: Consider a scenario where *SUB* $= 0.8$, the attacker strategy has been $\lambda = \langle n_1, n_2\rangle$ so far, and the attacker issues scan queries over all the neighbors of n_1 and n_2, that are n_3, n_4, n_5, and n_6 with $val(n_1) = 10, val(n_2) = 5, val(n_3) = 1, val(n_4) = 5, val(n_5) = 100$, and $val(n_6) = 10$. The defender needs to choose answers to such queries; in particular, he wants to move the attacker away from node n_5 (which has the maximum value). Let us assume that the defender samples an answer that leads to the following utilities for the surrounding nodes: $util(n_3) = 40, util(n_4) = 50, util(n_5) = 10, util(n_6) = 35$. A fully rational attacker would go after node n_4, as it has the highest utility. However, since we are assuming *SUB* $= 0.8$ and the maximum value in the utility array is 50, we then assume that the attacker randomly picks any node with utility higher than $50 \times 0.8 = 40$ at the first attack step (see Section III-A). In a simulation with *NSTEPS* $= 2$, he will therefore randomly choose between n_3 and n_4 at the first step—if for instance he chooses n_4, then at the second step the choice will be between n_3 and n_6, as both have utility greater than $40 \times 0.8 = 32$. We run *NSIM* simulations and get the strategies for those (in this example, one simulation is $\lambda' = \langle n_1, n_2, n_4, n_6\rangle$). After simulating all *NSIM* strategies for a given answer, the defender can get the probability of each strategy and then estimate the associated damages. The process is repeated for all the sampled answers. ■

Given an attacker strategy λ, we can quantify the amount of damage that it represents by computing some aggregate over the *values* assigned by the defender to the nodes affected by the strategy. This value may represent the value of information contained in the node for the enterprise (e.g., sensitive customer data, source code, etc.).

We define $damage(\lambda) = \sum_{n \in \lambda} value(n)$. Another option, as with utility, would be that of using $\max$ instead of summing—of course, further possibilities exist. From here, we can easily compute the expected damage the network will sustain in a given world as $exp\text{-}damage(w) = \sum_{\lambda \in \mathcal{A}} \Pr_w(\lambda) \cdot damage(\lambda)$. Finally, we can compute the damage of a probabilistic state σ from these elements as

$$damage(\sigma) = \sum_{w_i \in \mathcal{W}_{IC}} (p_i \cdot exp\text{-}damage(w_i)) .$$

The probabilties p_i above, each corresponding to a world w_i, must satisfy the set of linear constraints obtained from σ after consolidation (cf. Section III-B). However, there can be many probability assignments that satisfy such constraints. We consider the worst case scenario by choosing the assignment that corresponds to the *worst damage*, i.e. the maximum possible damage obtainable under the given constraints.

Proposition 2: Computing the probability assignment that corresponds to the worst damage is NP-hard.

Proof sketch. The problem amounts to solving a linear program with a polynomial number of constraints and an exponential number of variables, each representing the probability of a specific world under the integrity constraints. It is well known that the complexity of solving a problem with this kind of linear formulation is NP-hard if the corresponding "price problem" is still NP-hard. The price problem for this formulation is that of determining whether there exists a column, i.e. a world, whose associated reduced cost is negative. Since deciding whether there exists a world that satisfies the integrity constraints is NP-hard, the price problem is NP-hard as well. ■

The overall goal of the defender is to keep the attacker away from the most valuable nodes in the network; this is done via a careful selection of answers to the scan queries—the notion of worst damage introduced above directly guides this selection. Starting from an initial probabilistic state of the attacker, the defender answers the attacker's queries with deceptive answers that minimize damage, updates the probabilistic state as shown above, and updates the history of answers given.

Proposition 3: The problems of determining (i) whether there exists an answer to a scan query, and (ii) finding the answer that minimizes the damage, are both NP-hard.

Proof sketch. Answering a scan query amounts to finding at least one set of $runs$ atoms that satisfy the integrity constraints. It is easy to see that the same reduction used for Proposition 1 can be used to prove NP-hardness of determining whether there exists at least one answer. It immediately follows that finding the optimal one is NP-hard as well. ■

IV. Algorithms

In this section we develop two algorithms for solving the problem of selecting answers to the attacker's scan queries.[1]

A. Naive Algorithm

Figure 2 contains the pseudocode of a naive algorithm for solving our problem. Note that, even though the initial probabilistic state can contain non-ground statements, the initial probabilistic state is grounded by the defender, who has knowledge of the entire network.

```
Algorithm Naive-PLD
Input: PLD framework $M$, current probabilistic state $\sigma$,
       scan query $Q$, history $H$ of answers given to past
       scan queries, $NSTEPS, NSIM \in \mathbb{N}$, $SUB \in [0,1]$.
Output: Set of ground runs atoms.
 1. worlds:= compute the set of all possible worlds w.r.t. $M$;
 2. possAns:= possAnswers($Q, H$);
 3. leastDamage:= $+\infty$;
 4. leastDamageAns:= $\emptyset$;
 5. for ($i$:= 0; $i \leq |possAns|$; $i$++) {
 6.     currAns:= possAns.get($i$);
 7.     newState:= updateState($\sigma$, currAns);
 8.     if damage(newState, NSTEPS, NSIM, SUB) $\leq$ leastDamage then {
 9.         leastDamageAns:= currAns;
10.         leastDamage:= damage(newState);
11.     }
12. }
13. return leastDamageAns.
```

Fig. 2: Naive algorithm.

The algorithm cycles through all possible answers that can be provided for the input scan query. For each one, the updated probabilistic state is obtained, and the associated damage is computed; this is accomplished via a simulation of how the attacker would continue to act if they were given that answer (Line 8), which is done *NSIM* times, for *NSTEPS* each time, and with subrationality factor *SUB*. The algorithm finally outputs the answer that has the lowest associated damage.

From now on, *MaxSW* denotes the maximum number of software packages that can run on a network node, and S denotes the number of software packages (of different types and versions) that can occur in the network.

Proposition 4: Naive-PLD correctly computes the answer to a scan query that minimizes the damage of the resulting probabilistic state, and runs in time

$$O(|Nodes| \times \binom{S}{2^{MaxSW}} \times NSIM \times P \times |\mathcal{W}_{IC}|)$$

[1]It should be observed that, if we are in a setting in which all events are *pairwise probabilistically independent*—or this assumption can be made as a reasonable approximation—then a great deal of computational effort can be saved with respect to the algorithms described in this section. In this case, the probability of a conjunction of annotated atoms is the product of the individual probabilities; in interval form, the bounds will be the product of the corresponding bounds of all probabilistic atoms in the conjunction. So, since any atom that does not have an associated annotated atom can be assumed to be annotated with $[0,1]$, we can get the probability of a world by multiplying the bounds of each atom. As a direct consequence, we no longer need to build the linear programs to get probabilities; the complexity of computing the probability of a world is linear in the maximum between the number of atoms in the world and the size of the probabilistic state (i.e., the number of ground instances of annotated atoms in the state). This option was included in the experimental evaluation, but obtained almost no benefit.

where P is the cost of computing probabilities of worlds.

Proof sketch. There are $\binom{S}{2^{MaxSW}}$ different answers that need to be inspected; for each one, the history needs to be inspected to see if the answer is consistent with it (additional $|Nodes|$, as there are $|Nodes|$ possible scan queries), there are *NSIM* simulations run to see the effect on the attacker's behavior, and the computation of the damage that will arise requires time $O(P \times |\mathcal{W}_{IC}|)$. ∎

B. Heuristic Algorithm

As Naive-PLD is expensive, we developed the heuristic algorithm shown in Figure 3 which works by first sampling an answer to a query and then, with this answer fixed, sampling many different sets of possible worlds—the number of answers to be sampled in each iteration is controlled by parameter *NA*, while the number of worlds sampled for each answer is set by *NW*.

```
Algorithm Fast-PLD
Input: PLD framework $M$, current probabilistic state $\sigma$,
       scan query $Q$, history $H$ of answers given to past scan
       queries, $S \in \{1, ..., |\mathcal{W}|\}$, $SUB \in [0,1]$,
       $NA, NW, NSTEPS, NSIM, NIDE \in \mathbb{N}$.
Output: Set of ground runs atoms.
 1. let Ans be a mapping from sets of atoms to real numbers
          (initialized to empty);
 2. leastAvgDamage:= $+\infty$;
 3. prAns:= new uniform prob. dist. over answers;
 4. for ($n$:= 1; $i \leq NIDE$; $n$++) {
 5.     for ($i$:= 1; $i \leq NA$; $i$++) {
 6.         spa:= sample-possAnswer($Q, H, prAns$);
 7.         damage:= 0;
 8.         for ($j$:= 1; $j \leq NW$; $j$++) {
 9.             sw:= sample-worlds($M, \sigma, S, prWorlds$);
10.             newState:= updateState($\sigma, spa$);
11.             avgDamage:= $\Big(avgDamage * (j-1) +$
                    $damage(newState, NSTEPS, NSIM, SUB)\Big)/j$;
12.         } // end for (world sampling)
13.         if avgDamage < leastAvgDamage then
14.             leastAvgDamageAns:= spa;
15.             leastAvgDamage:= avgDamage;
16.         Add (spa, avgDamage) to Ans;
17.     } // end for (answer sampling)
18.     Update prAns according to best samples in Ans;
19.     Reset Ans to an empty mapping;
20. } // end for (I.D.E. iterations)
21. return leastAvgDamageAns.
```

Fig. 3: Heuristic algorithm.

In each case, since all worlds outside the sampled set are left out, their probability has in practice been set to zero, which means that the resulting set of constraints is likely to be inconsistent. This is taken care of in Line 10 along with corrections that may be required to update the state with the consequences of the selected answer. The rest of Fast-PLD works by continuing this sampling-in-tandem process, each time keeping the best answer seen so far and updating the probability distributions used for sampling based on automated learning of how well the algorithm did in the past; this technique is generally known as "*Iterative Density Estimation*" [2], [3] (I.D.E.). The total number of I.D.E. iterations is set by parameter *NIDE*. The sampling procedures in Lines 6 and 9 could be implemented in many different ways—here, it starts out as purely random, and then continues with the importance sampling techniques; the initial sample could also be done using information provided by human experts.

Proposition 5: Fast-PLD computes an answer to a scan query in time

$$O\left(NIDE \times NA \times |Nodes| \times MaxSW \times NSIM \times P \times NW\right)$$

where P is the cost of computing probabilities of worlds.

Proof sketch. The *NA*, *NW*, and *NIDE* parameters are clearly multiplicative factors in the algorithm's computation time, as they are in nested `for` loops. For each sampled answer, the history needs to be inspected to see if the answer is consistent with it (additional $|Nodes|$). In the sampling loop, the computation of the damage that will arise in each world requires time $O(NSIM \times P \times NW)$. ∎

Thus, Fast-PLD is considerably faster than Naive-PLD.

V. Experimental evaluation

We have implemented a Java prototype of the proposed framework. Each experiment was run on an 8-core, 8-processor machine with 20GB of RAM. The optimization problems were solved using the IBM CPLEX library, and the Jung library for handling network graphs.

A. Setting

We considered an enterprise network based on real data. We first ran the Cauldron software [4] in a real network environment consisting of about 60 nodes—Cauldron scans for the software on each node, and can find vulnerabilities that are consistent with the NVD. We then replicated the network into a 600 node network, adding network connections through the NS2 network simulator [5].

In order to assign values to the nodes in the network, we sampled from a Zipf distribution and a Gaussian distribution. The values are normalized so that in both cases their sum is $10,000$. These two alternatives allowed us to verify how a network administrator should "distribute" the resources in the network to reduce the possible damage caused by an attacker.

We compared the Naive-PLD and Fast-PLD algorithms with a TRUTH baseline algorithm where the system answers honestly to scan queries. We evaluated the performance of these algorithms in terms of:

- *expected damage* after a certain number of attacker steps, each of which corresponds to a new node being compromised—the damage of a strategy was defined as the sum of the values of the nodes in the strategy;
- *execution time* required by the algorithms.

In addition, to evaluate the statistical significance of the results we computed p-values through both the student *t-test* [6] and the more challenging *Mann-Whitney u-test* [7].

In the experiments we varied the following parameters:

- *NA* (number of answers evaluated at each I.D.E. iteration—Fast-PLD algorithm);
- *NSTEPS* (number of "future" attacker steps considered in the estimation of the damage);

- *NIDE* (number of I.D.E. iterations—Fast-PLD algorithm);
- *SUB* (subrationality factor of the attacker).

We also introduced an additional parameter *PA* representing the percentage of network nodes for which attacker strategies were considered.

We fixed the number of worlds sampled at each iteration of the Fast-PLD algorithm to $NW = 50$—this value guaranteed a good tradeoff between result quality and execution times. For the sake of fairness, whenever we sample new worlds, we also include the *true world*, i.e. the world corresponding to the actual state of affairs. This allows the defender to choose whether telling the truth might actually bring better long-term benefits. Finally, we fixed the number of attacker simulations done for each answer and each world in order to estimate damage to $NSIM = 10$.

The values of the believability weights β_i in the consolidation function were chosen as follows: if the atom is related to a priori knowledge of the attacker, then we set $\beta_i = 1$; if the atom is contained in an answer of the defender, then $\beta_i = 100$. This way, the replies given by the defender are given higher "priority", but if the defender gives the attacker an inconsistent answer, then his belief automatically decreases.

For the utility function, we used the $util_1$ definition reported in Section III. The a priori knowledge and the "frontier" nodes were initialized randomly.

Finally, we introduced several integrity constraints such as, e.g., (i) each node must have exactly one operating system, (ii) each node can have at most one web server, (iii) each node can have at most one DBMS, (iv) each node can have at most 25 software packages that access the Internet (we therefore had $MaxSW = 25$).

Remark: The Fast-PLD algorithm computes an answer for all the not-yet-scanned neighbors of nodes that have been compromised by the attacker at once (also considering *NSTEPS* future attacker steps in the estimation of the damage). The time needed for this process is called *offline execution time* in the remainder, and is used to assess the efficiency of the algorithms. In the scenario we envision, the network administrator pre-computes a set of deceptive answers for the whole network (or a large fraction of it) initially and after any major modification to the network configuration (which happens with relatively low frequency) and then uses this set to answer scan queries. We will call the time needed to compute the set of answers for the whole network *total offline execution time*, and the time to answer single scan queries *online response time*. It should also be observed that in the results reported in the next section, every curve is averaged over 100 *different* simulations of attacker behaviors, but the set of deceptive answers stays the same.

B. Results

Round 1 – Performance of Naive-PLD

Our first round of experiments was aimed at evaluating the feasibility of the Naive-PLD algorithm, which is expected to require much longer execution times than Fast-PLD given its complexity (Proposition 4). For this round we employed a very small network consisting of just 5 nodes. The results in terms of average damage, obtained with $PA = 1.0$, $NA = 5$, $NIDE = 5$, $NSTEPS = 6$, and $SUB = 0.8$, are reported in Figure 4 (the value of *NIDE* is reported in square brackets in the figure). We obtained similar results with $SUB = 1.0$ that we do not report for space reasons.

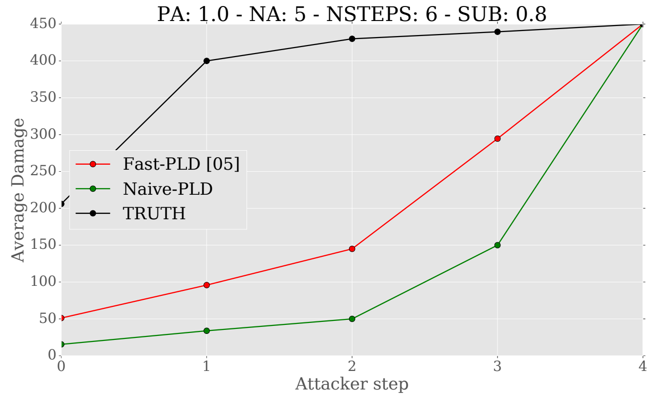


Fig. 4: Average damage on a 5-node network.

The results show that (i) the worst results are obtained with TRUTH—at attacker step 2, the damage is already almost at its maximum, and (ii) the best results are provided by Naive-PLD. Fast-PLD appears to provide satisfactory results, with damage values that are far lower than those obtained with TRUTH. Clearly, at attacker step 5, all of the algorithms reach the same damage, as the whole 5-node network is exploited.

The execution time of Naive-PLD is clearly impractical for real contexts. *The algorithm took around* 10 *hours to obtain a solution*, whereas Fast-PLD just required 3.2 seconds (0.2 seconds for TRUTH). In the remainder of this section we will therefore focus solely on the Fast-PLD algorithm.

Round 2 – Distribution of node value

In a second round of experiments, we compared the results obtained with the two different distributions (Zipf and Gaussian) of the values of the nodes in the enterprise network. Figure 5 reports the results for $PA = 0.5$ (i.e. we assume the attacker compromises 50% of the 600 nodes network), $NA = 10$, $NIDE = 1$, $NSTEPS = 10$, and $SUB \in \{0.8, 1.0\}$.

The results show that, even without I.D.E. learning ($NIDE = 1$), Fast-PLD performs better than TRUTH both for a fully rational ($SUB = 1.0$) and a sub-rational ($SUB = 0.8$) attacker. The performance of Fast-PLD is much better under the Zipf distributions, whereas the improvement under the Gaussian distributions is relatively limited. This clearly suggests that a network administrator should prefer to distribute important resources according to a Zipf distribution, i.e., putting important resources on a small number of highly protected nodes. The low p-values between TRUTH and Fast-PLD (always ≤ 0.01) confirm the statistical significance of the results.

Round 3 – Importance of the number of I.D.E. iterations

The third round of experiments was aimed at evaluating the impact of I.D.E. learning on the performance of Fast-PLD. We expected that a larger number of iterations would increase the probability of sampling "good" atoms, thus increasing the overall performance in terms of expected damage. Figure 6 reports the results under Zipf and Gaussian node

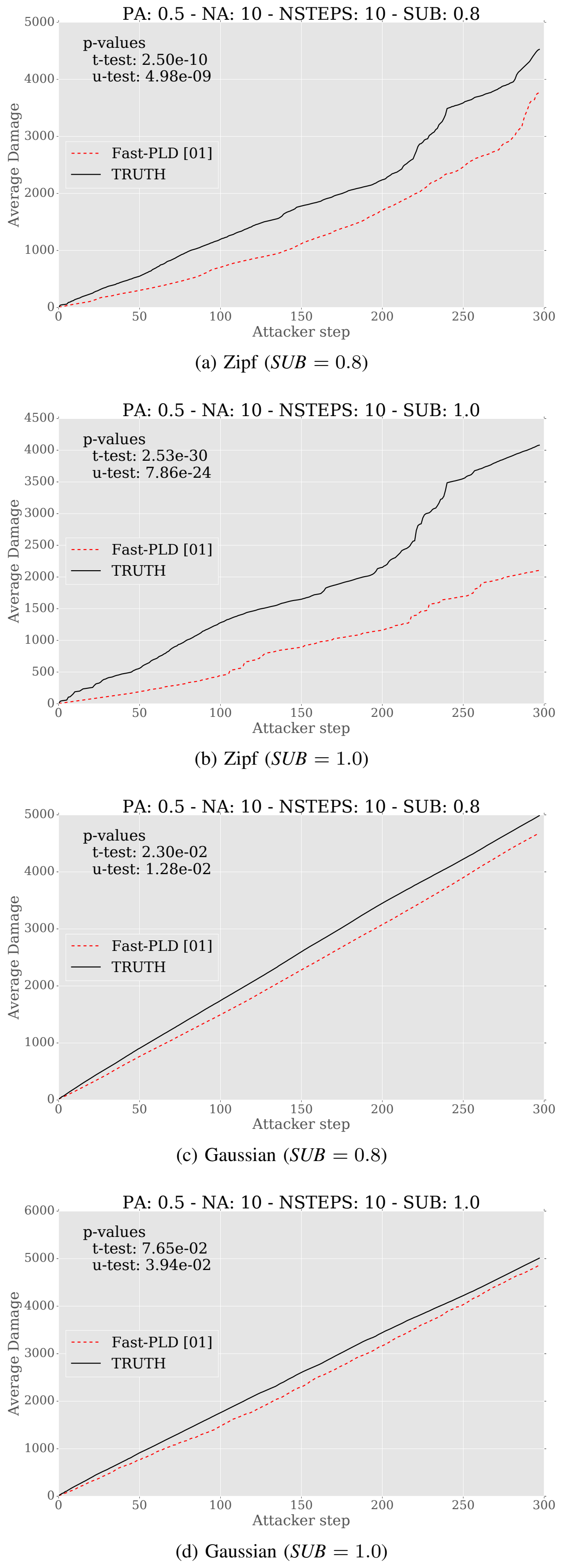


Fig. 5: Average damage under Zipf and Gaussian distributions without I.D.E. learning.

value distributions, with $PA = 0.3$, $NA = 10$, $NIDE \in [1..5]$, $NSTEPS = 10$, and $SUB \in \{0.8, 1.0\}$.

Under the Zipf distribution, increasing the number of I.D.E. iterations provides a significant decrease in the damage caused by the attacker. We also observe that there are some sudden “spikes”; these correspond to cases where the attacker compromises a high-value node despite the deception strategy. However, the chance of such events decreases significantly with $NIDE \geq 4$. Moreover, although on average the results are worse for a fully rational attacker ($SUB = 1.0$), after 100 steps with $NIDE = 5$, we obtain much better performance than the subrational attacker case ($SUB = 0.8$). This confirms that a fully rational attacker that follows exactly the behavior we assume can be better deceived by our strategy under proper configurations. Increasing the number of I.D.E. iterations does not greatly improve the performance under a Gaussian distribution—this can be explained by observing that the node values are more evenly distributed accross the network and hence the specific decisions made by the attacker have a smaller impact on the overall damage produced.

Figure 7 reports boxplots of the average offline execution times when varying *NIDE* for this round of experiments. Again, the results are satisfactory—in particular, the median times increase almost linearly with the increase in *NIDE*.

We also analyzed what happens if we further increase the number of I.D.E. iterations, in order to check whether a good trade-off can be found between expected damage and execution times. Figure 8 reports the results obtained with $PA = 0.1$, $NA = 10$, $NIDE \in \{1, 5, 10, 15, 20, 25, 30\}$, $NSTEPS = 10$, and $SUB = 0.8$.

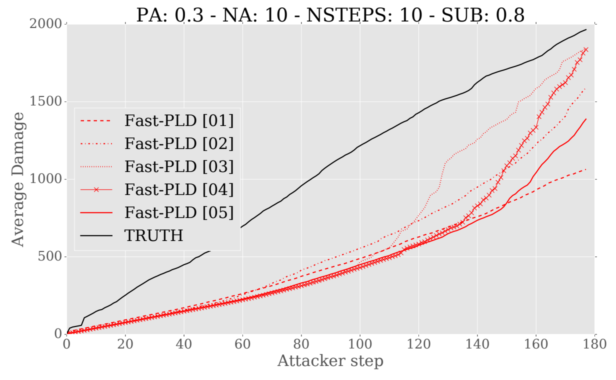
The results show that the benefit obtained with $NIDE > 5$ is almost negligible, as for higher values the expected damage is rather stable. On the other hand, execution times increase significantly—for instance, more than 30 minutes are required when $NIDE > 20$. This suggests that the best trade-off between expected damage and execution times can be obtained with $NIDE = 5$.

Round 4 – Importance of the number of future attacker steps simulated

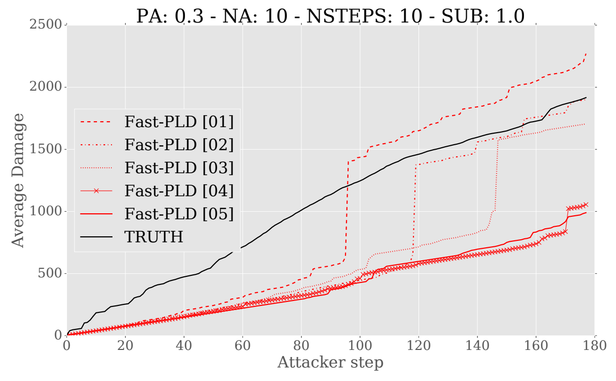
In the fourth round of experiments we assessed the impact of varying the number *NSTEPS* of simulation steps—again, the aim was to possibly identify a good trade-off between expected damage and execution times. Figure 9 reports the results obtained with $PA = 0.3$, $NA = 10$, $NIDE = 5$, $NSTEPS \in \{5, 10, 15, 20\}$, and $SUB = 0.8$.

The results show that low values of *NSTEPS* correspond to cases where we might miss important future decisions made by the attacker and lead him to high-value nodes. However, for $NSTEPS \geq 10$, we observe an important reduction of the expected damage; also, performance is almost the same up to 100 attacker steps. Again, the p-values confirm the statistical significance of the results.

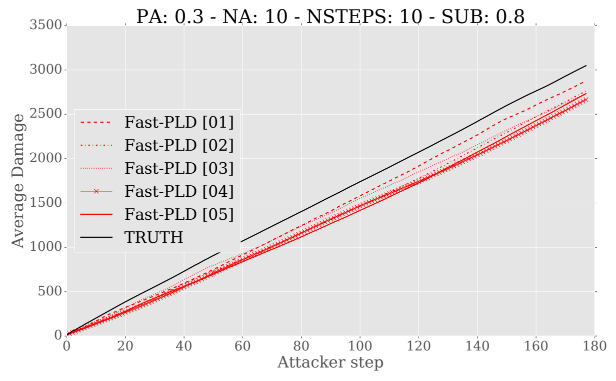
Figure 10 reports the execution times for this round of experiments. For $NSTEPS > 15$, the execution time for each answer goes up to almost 30 minutes—this leads us to identify a good trade-off with $NSTEPS \in \{10, 15\}$.



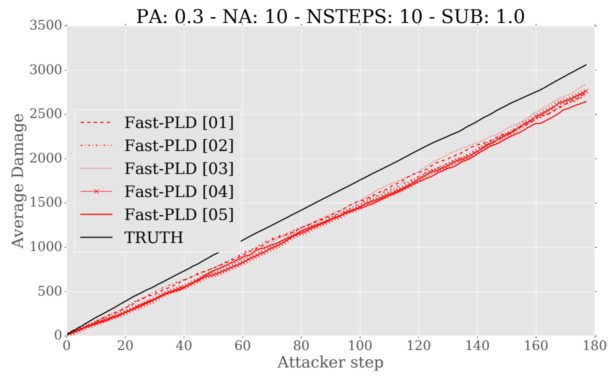
(a) Zipf ($SUB = 0.8$)



(b) Zipf ($SUB = 1.0$)



(c) Gaussian ($SUB = 0.8$)



(d) Gaussian ($SUB = 1.0$)

Fig. 6: Average damage under Zipf and Gaussian distributions when varying *NIDE*.

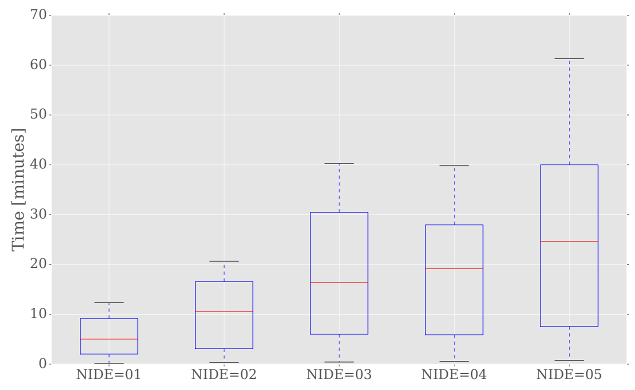
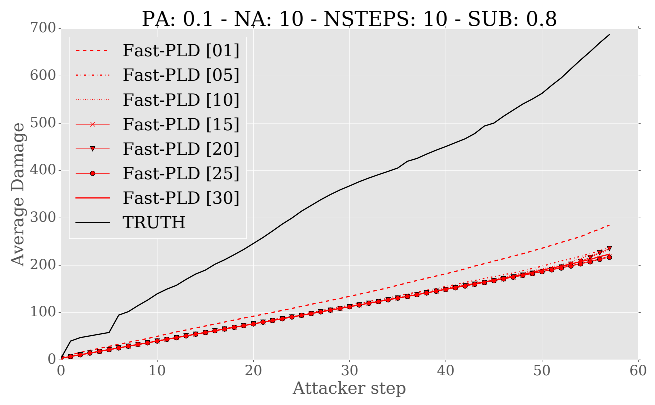
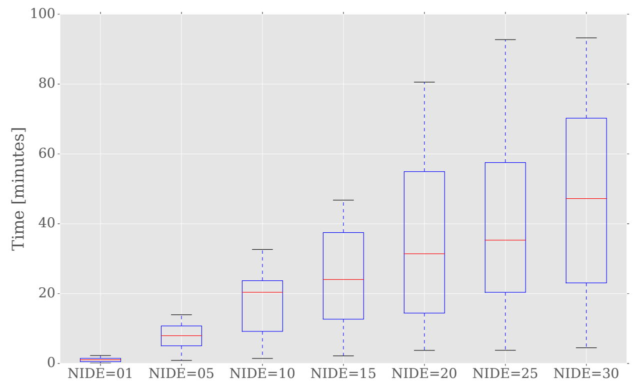


Fig. 7: Offline execution time when varying *NIDE*.



(a) Average damage



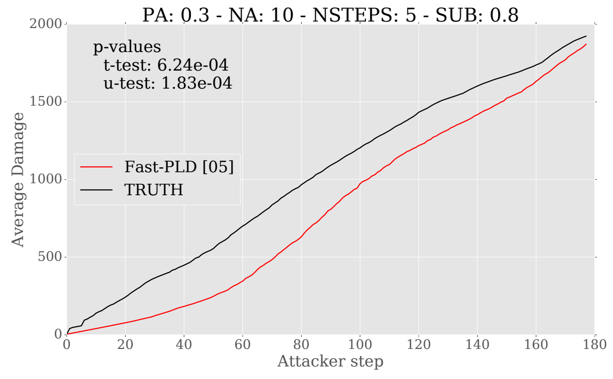
(b) Offline execution time

Fig. 8: Results for $NIDE \in \{1, 5, 10, 15, 20, 25, 30\}$.

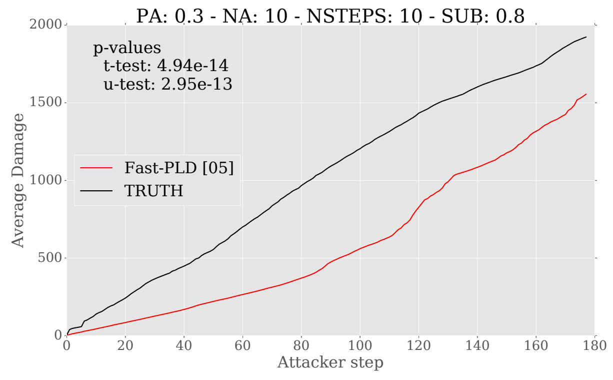
Round 5 – Total offline execution times and online response times

In our final round of experiments we assessed the total offline execution time and the online response time. It is indeed of utmost importance that, after offline computation, the defender is able to reply to the attacker's scan queries in very short time.

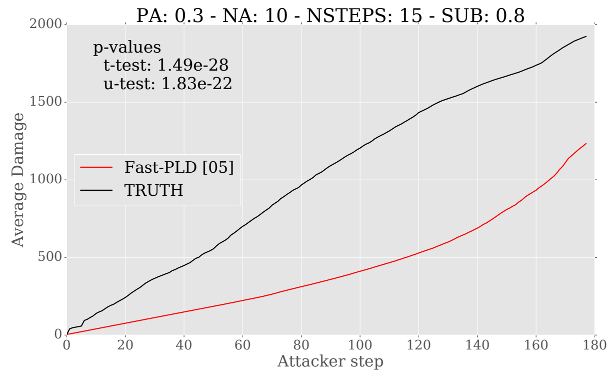
Figure 11 reports the total offline execution times required on two networks containing 60 and 180 nodes. More specifically, Figure 11a reports the results obtained with $PA = 0.1$, $NA = 10$, $NSTEPS = 10$, and $SUB = 0.8$, whereas those in



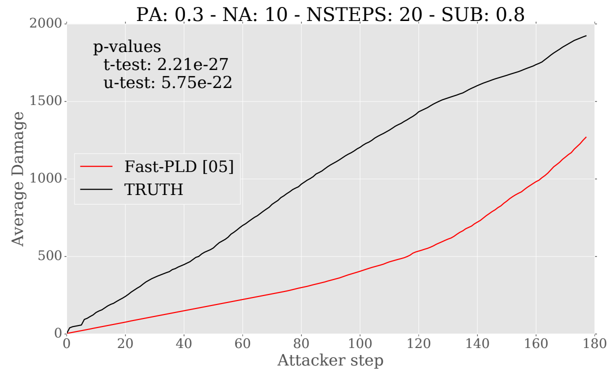
(a) *NSTEPS* = 5



(b) *NSTEPS* = 10



(c) *NSTEPS* = 15



(d) *NSTEPS* = 20

Fig. 9: Average damage when varying *NSTEPS*.

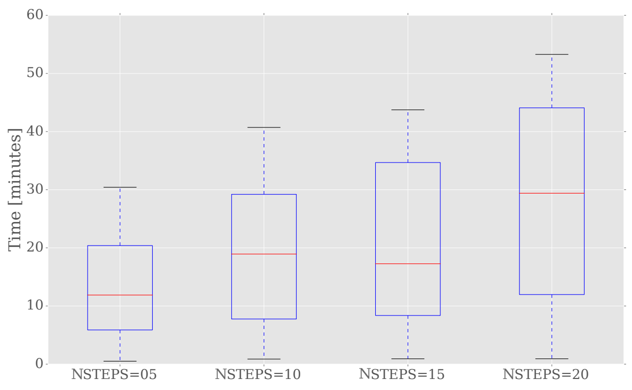
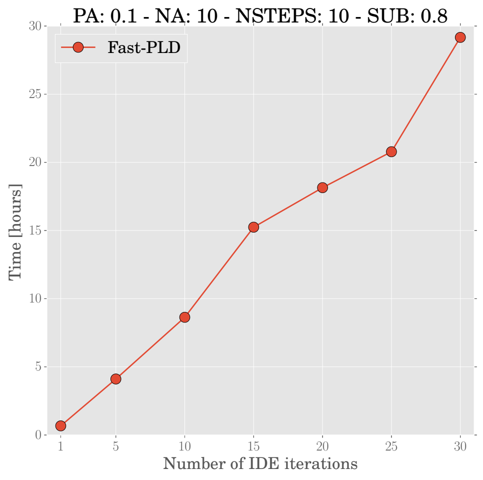
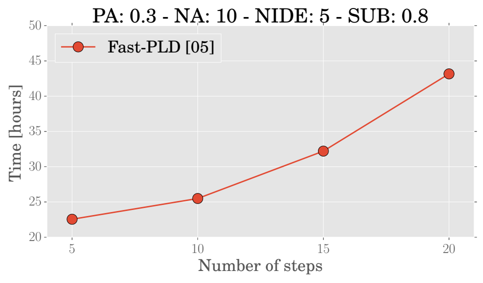


Fig. 10: Offline execution time when varying *NSTEPS*.

Figure 11b were obtained with $PA = 0.3$, $NA = 10$, $NIDE = 5$, and $SUB = 0.8$. Here we observe that with $NIDE = 5$ and $NSTEPS = 10$, that were identified before as good trade-off values, the total offline execution time for about 180 nodes takes just a single day.



(a)



(b)

Fig. 11: Total offline execution time.

A boxplot for the online response time in this round of experiments is reported in Figure 12—we can observe that the median time to reply is around 5-6 milliseconds, which is a realistic and perfectly acceptable latency for a network response to the attacker.

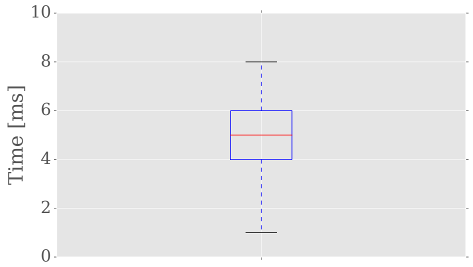


Fig. 12: Online response time.

VI. Related Work

A. Logic, Optimization, and Probabilistic Logic

In this paper we use part of the annotated probabilistic logic introduced in [8]. The computational aspect of this logic is related to linear program formulations that are sometimes very hard to compute. Annotated probabilistic logic is different from its non-probabilistic counterpart due to the presence of probability intervals, describing uncertainty of beliefs. Several approaches to tackle the computational aspects have been proposed [9], [10]; these methods use advanced optimization techniques. The use of the latter for processing logic problems goes back to [11] with a host of successful later work such as [12], [13]. Later, Nerode and his co-workers developed methods to use integer programming for computing structures in non-monotonic logic programming [14], [15]. Constraint logic programming (CLP) [16] embedded numerical constraints within a logic programming framework.

B. Logic for Cybersecurity

Variants of CLP have been used for cybersecurity research [17], [18], [19]. The use of logic in cybersecurity was first introduced in [20], [21], [22]. In particular, [20], [21] looked at the use of deontic logic constructs to capture what is permitted, what is obligatory, and what is forbidden in terms of access control, while [22], [23], [24], [25], [26] looked at the use of logic in order to answer queries to databases without revealing secret information (e.g. letting the user infer data from the answer that he was not authorized to know).

Later efforts looked at the use of logic for expressing security policies, namely who should be authorized to access a body of data, and who should not, and the conditions under which such access should be granted [27], [28], [29], [30]. However, none of these works look at using temporal probabilistic logic in security, nor do they build models of the adversary.

C. Defending Enterprise Networks

The problem of defending enterprise networks is well studied in the literature under 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 [31]). [32], [33], [34] consider the problem of finding plans for patching vulnerabilities, that are tradeoffs between cost and risk, by using Pareto analysis.

Another body of work focuses on the detection of advanced persistent cyberattacks in enterprise networks to identify revealing signals of an attacker. For example, [35] proposes a scalable framework that ranks internal nodes of an enterprise possibly related to “burst” and/or “low-and-slow” data exfiltrations by analyzing suspicious network activities over time. A comprehensive report about existing honeypot implementations is given in [36]. 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. The use of game theory in cybersecurity problems is not new [31], [37]. [38], [39] study different game models that address strategies for deploying honeypot networks. The authors define a basic honeypot selection game where the defender chooses 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.

[40] highlights the importance of honeypot networks not just for the ability to attract the attacker but also to delay him. They define the concept of distraction chains, i.e. a sequence of decoy systems used to entice an adversary to explore useless information. They study the problem of creating the distraction chains and embedding them in the network. [41] defines a game-based model in order to hide honey nodes from the attacker. Further examples of game theory applied to security (not strictly related with our work) can be found in [42]. [43] studies 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 upon detection is not the best one. They show that this improves the learning rate of the intruder, and consequently increases his probability of success. [44] adopts Stackelberg games to interdict attack plans—the main aim of the game is that of obtaining strategies for the defender that reduce the impact of the most successful attacker strategy.

Finally, the two works that are closer in spirit to the framework proposed in this paper are [45], [46]; both use a deceptive approach in order to confuse the attacker. In contrast to these, our framework aims at finding the best way to answer the attacker’s scan queries. Moreover, our proposal takes into account the behavior of the attacker by modeling his beliefs as facts under an annotated probabilistic logic.

VII. Conclusions

Network scanning is used by system administrators in order to identify vulnerabilities in their systems, while attackers do the same to enterprises that they target. In this paper, we propose that for network scans that are not generated by a system administrator, the system should return a mix of real and fake results so that an attacker either is taken in by the fake results or has to expend more resources (time, money) in analyzing the scan results.

Generating such mixes of true and fake results poses many challenges. First, the scan results must be consistent with reasonable expectations of the system configuration—we model these via integrity constraints. Second, the scan results given must be consistent with the knowledge an attacker has accumulated about the system from his past queries. In order to address both of these issues, we propose a probabilistic logic of deception (PLD-Logic) and show how we can generate a mix of real and fake results that result in the least (expected) damage to the system. We show how to use PLD-Logic to model the set of possible worlds that the attacker may infer. We then show how the defender can use this knowlege in order to answer any specific scan query, considering both the attacker's current set of possible worlds, as well as how that set of possible worlds may evolve in the future. We develop two algorithms—Naive-PLD which takes exponential time to run (not surprisingly due to a host of NP-hardness results we show) and Fast-PLD, a fast heuristic algorithm that levereages iterative density estimation methods.

Our experimental results show that Fast-PLD can be deployed in practice by computing the right scan results to return offline and serving them up by a simple lookup of a "scan table" when those scans occur online. The scan table can be periodically updated offline as the network changes.

Acknowledgments

Our research was partly funded by ARO grants W911NF1410358 and W911NF1310421, by ONR grants N000141512007, N000141612896, and N000141512742 (any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ONR), by MIUR grant PON03PE_00032_2, by EU H2020 research and innovation program under the Marie Sklodowska-Curie grant N690974, as well as by funds provided by CONICET, Agencia Nacional de Promoción Científica y Tecnológica, and Universidad Nacional del Sur, Argentina.

References

[1] NIST, "National vulnerability database," 2016, http://nvd.nist.gov.

[2] J. S. D. Bonet, C. Isbell Jr., and P. A. Viola, "MIMIC: Finding optima by estimating probability densities," in *NIPS*, 1996.

[3] M. Pelikan, D. E. Goldberg, and F. G. Lobo, "A survey of optimization by building and using probabilistic models," *Computational Optimization and Applications*, vol. 21, no. 1, pp. 5–20, 2002.

[4] S. Jajodia, S. Noel, P. Kalapa, M. Albanese, and J. Williams, "Cauldron: Mission-centric cyber situational awareness with defense in depth," in *MILCOM*, 2011.

[5] T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*. Springer Publishing Company, Incorporated, 2008.

[6] J. L. Devore, *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, 2015.

[7] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[8] P. Shakarian, A. Parker, G. Simari, and V. Subrahmanian, "Annotated probabilistic temporal logic," *ACM Trans. Comput. Logic*, vol. 12, no. 2, pp. 14:1–14:44, 2011.

[9] P. Shakarian, G. I. Simari, and V. S. Subrahmanian, "Annotated probabilistic temporal logic: Approximate fixpoint implementation," *ACM Trans. Comput. Logic*, vol. 13, no. 2, pp. 13:1–13:33, 2012.

[10] S. Khuller, M. V. Martinez, D. Nau, G. I. Simari, A. Sliva, and V. Subrahmanian, "Computing most probable worlds of action probabilistic logic programs: Scalable estimation for $10^{30,000}$ worlds," *Annals of Mathematics and Artificial Intelligence*, vol. 51, no. 2–4, pp. 295–331, 2007.

[11] R. G. Jeroslow, "Computation-oriented reductions of predicate to propositional logic," *Decision Support Systems*, vol. 4, no. 2, pp. 183–197, 1988.

[12] J. Hooker, *Logic-based methods for optimization: combining optimization and constraint satisfaction*. John Wiley & Sons, 2011.

[13] R. Raman and I. E. Grossmann, "Modelling and computational techniques for logic based integer programming," *Computers & Chemical Engineering*, vol. 18, no. 7, pp. 563–578, 1994.

[14] C. Bell, A. Nerode, R. T. Ng, and V. Subrahmanian, "Implementing deductive databases by linear programming," in *PODS*, 1992.

[15] ——, "Mixed integer programming methods for computing nonmonotonic deductive databases," *Journal of the ACM*, vol. 41, no. 6, pp. 1178–1215, 1994.

[16] J. Jaffar and J.-L. Lassez, "Constraint logic programming," in *POPL*, 1987.

[17] N. Li and J. C. Mitchell, "Datalog with constraints: A foundation for trust management languages," in *Practical Aspects of Declarative Languages*, 2003.

[18] P. Eronen and J. Zitting, "An expert system for analyzing firewall rules," in *NordSec*, 2001.

[19] R. Corin and S. Etalle, "An improved constraint-based system for the verification of security protocols," in *SAS*, 2002.

[20] P. Bieber and F. Cuppens, "Expression of confidentiality policies with deontic logic," in *Deontic logic in computer science*. Wiley and sons, 1992.

[21] J. Glasgow, G. MacEwen, and P. Panangaden, "A logic for reasoning about security," *ACM Transactions on Computer Systems*, vol. 10, no. 3, pp. 226–264, 1992.

[22] P. A. Bonatti, S. Kraus, and V. S. Subrahmanian, "Declarative foundations of secure deductive databases," in *ICDT*, 1992.

[23] M. Winslett, K. Smith, and X. Qian, "Formal query languages for secure relational databases," *ACM Transactions on Database Systems*, vol. 19, no. 4, pp. 626–662, 1994.

[24] P. Stouppa and T. Studer, "Data privacy for knowledge bases," in *LFCS*, 2009.

[25] J. Biskup and P. A. Bonatti, "Lying versus refusal for known potential secrets," *Data & Knowledge Engineering*, vol. 38, no. 2, pp. 199–222, 2001.

[26] H. M. Jamil, "Belief reasoning in mls deductive databases," *ACM SIGMOD Record*, vol. 28, no. 2, pp. 109–120, 1999.

[27] S. Jajodia, P. Samarati, M. L. Sapino, and V. Subrahmanian, "Flexible support for multiple access control policies," *ACM Transactions on Database Systems*, vol. 26, no. 2, pp. 214–260, 2001.

[28] P. Samarati and S. De Capitani di Vimercati, "Access control: Policies, models, and mechanisms," in *Foundations of Security Analysis and Design*. Springer, 2001.

[29] J. Y. Halpern and V. Weissman, "Using first-order logic to reason about policies," *ACM Transactions on Information and System Security*, vol. 11, no. 4, p. 21, 2008.

[30] C. A. Ardagna, M. Cremonini, E. Damiani, S. D. C. di Vimercati, and P. Samarati, "Supporting location-based conditions in access control policies," in *SICCS*, 2006.

[31] E. Serra, S. Jajodia, A. Pugliese, A. Rullo, and V. S. Subrahmanian, "Pareto-optimal adversarial defense of enterprise systems," *ACM Transactions on Information and System Security*, vol. 17, no. 3, pp. 11:1–11:39, 2015.

[32] R. Dewri, N. Poolsappasit, I. Ray, and D. Whitley, "Optimal security hardening using multi-objective optimization on attack tree models of networks," in *CCS*, 2007.

[33] R. Dewri, I. Ray, N. Poolsappasit, and D. Whitley, "Optimal security hardening on attack tree models of networks: a cost-benefit analysis," *International Journal of Information Security*, vol. 11, no. 3, pp. 167–188, 2012.

[34] N. Poolsappasit, R. Dewri, and I. Ray, "Dynamic security risk management using bayesian attack graphs," *IEEE Trans. Dependable Secur. Comput.*, vol. 9, no. 1, pp. 61–74, 2012.

[35] M. Marchetti, F. Pierazzi, M. Colajanni, and A. Guido, "Analysis of high volumes of network traffic for advanced persistent threat detection," *Computer Networks*, vol. 109, pp. 127–141, 2016.

[36] F. Pouget and M. Dacier, "Honeypot, honeynet: A comparative survey," in *Institut Eurcom*, 2003.

[37] K.-w. Lye and J. M. Wing, "Game strategies in network security," *International Journal of Information Security*, vol. 4, no. 1-2, pp. 71–86, 2005.

[38] C. Kiekintveld, V. Lisý, and R. Píbil, "Game-theoretic foundations for the strategic use of honeypots in network security," in *Cyber Warfare - Building the Scientific Foundation*. Springer, 2015.

[39] S. Jajodia, P. Shakarian, V. S. Subrahmanian, V. Swarup, and C. Wang, Eds., *Cyber Warfare - Building the Scientific Foundation*. Springer, 2015.

[40] P. Shakarian, D. Paulo, M. Albanese, and S. Jajodia, "Keeping intrudors at large: A graph-theoretic approach to reducing the probability of successful network intrusions," in *SECRYPT*, 2014.

[41] J.-Y. Cai, V. Yegneswaran, C. Alfeld, and P. Barford, "An attacker-defender game for honeynets," in *COCOON*, 2009.

[42] T. Alpcan and T. Baar, *Network Security: A Decision and Game-Theoretic Approach*. Cambridge University Press, 2010.

[43] N. Bao and J. Musacchio, "Optimizing the decision to expel attackers from an information system," in *ACC*, 2009.

[44] J. Letchford and Y. Vorobeychik, "Optimal interdiction of attack plans," in *AAMAS*, 2013.

[45] M. Albanese, E. Battista, S. Jajodia, and V. Casola, "Manipulating the attacker's view of a system's attack surface," in *CNS*, 2014.

[46] M. Albanese, E. Battista, and S. Jajodia, "A deception based approach for defeating os and service fingerprinting," in *CNS*, 2015.