STAIRS 2014
U. Endriss and J. Leite (Eds.)

doi:10.3233/978-1-61499-421-3-101

Computing Optimal Policies for Attack Graphs with Action Failures and Costs

Karel DURKOTA and Viliam LISY
Agent Technology Center, Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague
{karel.durkota, viliam.lisy}@agents.fel.cvut.cz

Abstract. An attack graph represents all known sequences of actions that compromise a system in form of an and-or graph. We assume that each action in the attack graph has a specified cost and probability of success and propose an algorithm for computing an action selection policy minimizing the expected cost of performing an attack. We model the problem as a finite horizon MDP and use forward search with transposition tables and various pruning techniques based on the structure of the attack graph. We experimentally compare the proposed algorithm to a generic MDP solver and a solver transforming the problem to an Unconstrained Influence Diagram showing a substantial runtime improvement.

Keywords. optimal policy, attack graph, markov decision process, and-or graph

Introduction

Attack graphs (AG) are a popular tool for analysing and improving security of computer networks, but they can be used in any domain, where attacks consist of multiple interdependent attack actions. Attack graphs capture all the known sequences of actions that may lead to compromising a system, and they can contain additional information, such as the cost of individual actions and the probability that the actions will be successfully executed. AGs can be used to evaluate risks and design appropriate countermeasures.

In analysis of attack graphs, it is often of interest to identify the optimal strategy of the attacker (i.e., which actions to execute in what situation) and its expected cost. For example, comparing the expected cost of the attack to the expected reward of successfully compromising the target indicates if a rational attacker would attack the system at all [3]. In penetration testing, following the optimal attack strategy can save a lot of valuable time [7]. Computing the optimal strategy for the attacker is also a building block in solving various game-theoretic models of interaction between the attacker and defender of a system. Furthermore, a problem of computing the optimal attack strategy can also be seen as a complex variant of the generic problem of probabilistic and-or tree resolution analysed in AI research [4].

In this paper, we propose an algorithm for computing the optimal attack strategy for an attack graph with action costs and failure probabilities. Unlike previous works assuming that the attack graph is a tree (e.g, [7]) and/or computing only a bound on the actual value (e.g., [3]), we compute the exact optimum and we do not impose any

restriction on the structure of the attack graph. Specifically, our approach allows the attack graph to contain (even oriented) cycles and to have actions with probabilities and costs as inner nodes of the attack graph.

The drawback of our approach is that even a simplified variant of this problem has been shown to be NP-hard in [4]. As a result, we solve it by a highly optimized search algorithm and experimentally evaluate its scalability limitations. We show that the problem can be mapped to solving a finite horizon Markov decision process (MDP) and how the information about the structure of the attack graph can be used do substantially prune the search space in solving the MDP. We compare the proposed approach to recently published method for solving this problem [6] and to a recent version of a generic MDP solver from the International Planning Competition 2011 [5], showing that the proposed method scales orders of magnitude better.

1. Background and Definition

1.1. Attack Graph

AG is a directed graph consisting of two types of nodes: (i) *fact nodes*, that represent facts that can be either true or false, and (ii) *action nodes*, that represent actions that the attacker can perform. Each action has *preconditions* – a set of facts that must be true before action is performed and *effects* – a set of facts that becomes true if action is successfully performed. Moreover, every action has associated probability $p \in (0,1]$ – which denotes the probability that action succeeds and its effects become true, and with probability $1-p$ action fails and attacker cannot repeat this action anymore. We assume that attacker cannot repeat actions for couple of reasons: (i) if actions are correlated and have static dependencies (installed software version, open port, etc.), another attempts to use the same action would result alike, and (ii) if we allow infinitely many repetitions, optimal attack policy (explained further) would collapse into a linear plan with attempting for each action until action succeeds[3]. Finally, each action has associated cost c; if attacker decides to perform action a, he will pay the cost c, regardless whether the action is successful or not.

Definition Let Attack Graph be a 5-tuple $AG = \langle \mathrm{F}, \mathrm{A}, \mathrm{g}, p, \mathrm{c} \rangle$, where:

- F is a finite set of facts
- A is a finite set of actions, where action $a : pre \rightarrow eff$, where $pre \subseteq \mathrm{F}$ is called *preconditions* (we refer to them as $pre(a)$) and $eff \subseteq \mathrm{F}$ is called *effects we refer to them as* $eff(a)$
- $\mathrm{g} \in \mathrm{F}$ is the goal
- $p : \mathrm{A} \rightarrow (0,1]$ is the probability of action to succeed (we use notation p_a for probability of action a to succeed, and with $p_{\bar{a}} = 1 - p_a$ the probability of action to fail)
- $\mathrm{c} : \mathrm{A} \rightarrow \mathbb{R}^+$ is cost of the action (we use notation c_a for cost of the action a).

We use the following terminology: we say that fact f *depends* on action a if $f \in eff(a)$, and similarly, action a depends on f if $f \in pre(a)$

Example of such Attack Graph is in Fig. 1. Diamonds are the inner fact-nodes, that are initially false, but can be activated performing any action on which the facts depend

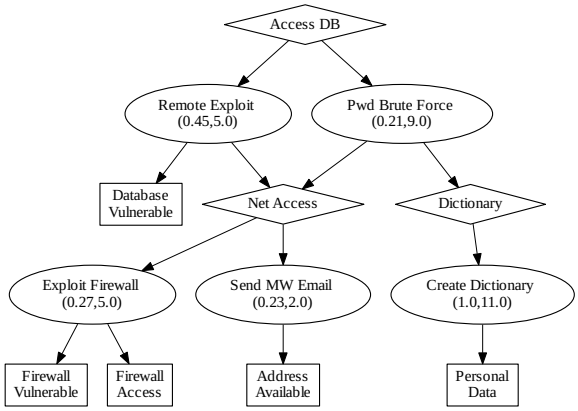

Figure 1. Simple attack graph which shows possible ways how to achieve access to the database (fact node "`Access DB`"). Diamonds are the inner fact-nodes (initially false) that can be turned true, while rectangles are the leaf fact nodes, which are always true. Ellipses depict actions that attacker can perform with success probability p and cost c.

upon. Rectangles represent leaf fact-nodes, that are initially true. Ellipses are the actions that attacker can perform with probability of success p and cost c. In our example we represent action with its name and the couple (p,c). Attacker's goal is to activate fact "`Access DB`" (obtain an access to DB).

The probabilities and costs of the actions can be obtained using Common Vulnerability Scoring System (CVSS)[1] from, i.e., National Vulnerability Database, which scores different properties of vulnerabilities. Probabilities could be computed for example from the access complexities, exploitabilities or availability impacts of the vulnerabilities, whereas costs could be computed from number of required authentication in order to a vulnerability, etc.

1.2. Attack Policy

Solving the AG means to find a policy, that describes what action should attacker perform in every possible evolution of the attack procedure. Fig. 2 depicts optimal policy ξ_{opt} for the problem from Fig. 1, where attacker first attempts to perform action "`Send MW Email`"; if action is successful, he follows the right (sub)policy (solid arc), thus performing action "`Remote Exploit`", otherwise the left (sub)policy (dashed arc), thus action "`Exploit Firewall`", and so on.

Definition An *attack policy* is an oriented binary tree ξ for evaluating attack graph AG. Nodes of ξ are actions, arcs are labeled $+$ or solid line (if parent action was successful) and $-$ or dashed line (if parent action was unsuccessful), and whose leaf-nodes are either $\boxplus$ resp. $\boxminus$ representing successful reps. unsuccessful attack.

[1] www.first.org/cvss

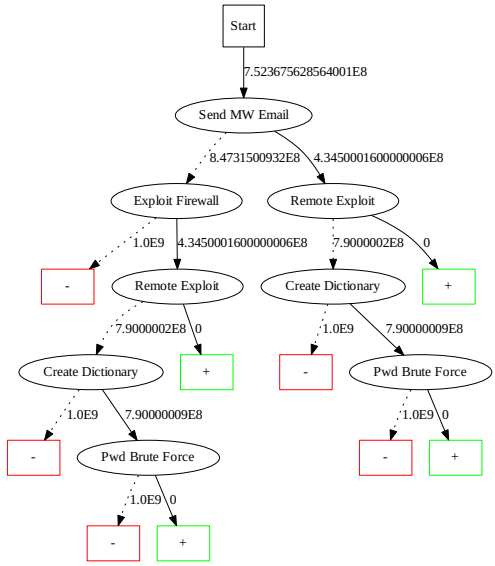

Figure 2. Optimal policy for a simple attack graph from Fig. 1. Attacker should follow solid arcs if previous action was successful, otherwise follow dashed line. Values at arcs represent the expected costs for attacker.

Definition The *expected cost* of an attack policy ξ is a cost over all possible evolutions of the policy. Let ϕ_χ be (sub)tree of ξ rooted at a node χ labeled with an action a, then expected cost of ϕ_χ can be computed recursively as follows:

$$\mathscr{E}(\phi_\chi) = c_a + p_a \times \mathscr{E}(\phi_{\chi^+}) + p_{\bar{a}} \times \mathscr{E}(\phi_{\chi^-})$$

where ϕ_{χ^+} (ϕ_{χ^-}) is the subtree rooted at χ's $+$ branch ($-$ branch) and in the leaf-nodes of the policy is a penalty if attack is unsuccessful $\mathscr{E}(\boxminus) = penalty$ and reward if successful $\mathscr{E}(\boxplus) = reward$.

When we decide which of the two policies, either ϕ_χ rooted at χ labeled with an action a or ϕ_ψ rooted at ψ labeled with action b have lower expected cost, we assume that after performing action a, resp. b, attacker follows an optimal policy. In this case, we override our notation of $\mathscr{E}(\phi_\chi)$ resp. $\mathscr{E}(\phi_\psi)$ to simply $\mathscr{E}(a)$ resp. $\mathscr{E}(b)$.

We assume that our attacker is a *motivated* attacker, that is they continue in attack as long as there are actions that may lead to the goal, regardless of the cost of the attacks. Motivated attacker ceases the attack only when there is no sequence of actions that could result in achieving the goal. Having assumed this type of attacker and the fact that attacks are *monotonic*, meaning that consequence of attack preserves once is achieved [1] (once the fact becomes true, it cannot become false again), it can be shown that every policy, regardless on the order of the action, will have equally the same probability of achieving the goal. Note, that the expected cost of the policy consist of two parts: the probability of achieving the reward or the penalty and the expected cost of the action costs. The probability of successful attack is always the same, thus every policy will have the same

expected cost of the penalty/reward. Thus, it essentially makes no difference whether we choose to reward the attacker for successful attack or penalize for an unsuccessful attack. In fact, distinct policies have different expected costs only because of the different action ordering which imposes different sequences of their costs.

Definition A policy ξ_{opt} is *optimal* if it has the minimal expected cost among all possible policies, thus $\forall \xi \in \Xi : \mathscr{E}(\xi_{opt}) \leq \mathscr{E}(\xi)$, where Ξ is a set of all policies.

Proposition 1.1 *In the optimal policy ξ_{opt} for every (sub)policy ϕ_χ rooted at a node χ labeled with an action a following is true: $\mathscr{E}(\phi_\chi) \leq \mathscr{E}(\phi_{\chi^-})$.*

We will prove it by contradiction. Assume that $\mathscr{E}(\phi_\chi) > \mathscr{E}(\phi_{\chi^-})$ is true. Then due to the monotonicity property the attacker could have followed the (sub)policy ϕ_{χ^-} even before performing action a, which would have saved him the cost of the action c_a. But then this new policy would have had lower expected cost than the policy ϕ_χ, which violates our assumption that ϕ_χ is an optimal policy.

Proposition 1.2 *In the optimal policy ξ_{opt} for every (sub)policy ϕ_χ rooted at a node χ labeled with an action a following is true: $\mathscr{E}(\phi_{\chi^-}) \geq \mathscr{E}(\phi_{\chi^+})$.*

$$\mathscr{E}(\phi_\chi) = c_a + p_a * \mathscr{E}(\phi_{\chi^+}) + p_{\bar{a}} * \mathscr{E}(\phi_{\chi^-}) \tag{1}$$

$$\mathscr{E}(\phi_{\chi^-}) \geq c_a + p_a * \mathscr{E}(\phi_{\chi^+}) + p_{\bar{a}} * \mathscr{E}(\phi_{\chi^-}) \tag{2}$$

$$\mathscr{E}(\phi_{\chi^-}) \geq c_a + p_a * \mathscr{E}(\phi_{\chi^+}) + c_a/p_a \tag{3}$$

1.3. Markov Decision Process

We solve this problem by modeling it as Markov Decision Processes (MDP) [2] which is defined as 4-tuple $\langle S, A, P_\cdot(\cdot,\cdot), R_\cdot(\cdot,\cdot)\rangle$, where:

- S is a finite set states, in our case state is a set of performed actions and label whether the action a was successful (a) or not ($\bar{a}$);
- A is a set of actions, which is equal to the set of actions in the attack graph
- $P_a(s,s')$ is a probability that action a, performed in state s, will lead to state s'; in our case, if action a, with probability p_a is successful, then state $s' = s \cup \{a\}$; if action a is unsuccessful, then state $s' = s \cup \{\bar{a}\}$
- $C_a(s,s')$ is an immediate cost payed after transition to state s' from state s; in our case $C_a(s,s') = c_a$ in all transitions, except when s' is a terminal state, then $C_a(s,s') = c_a - reward$ if goal is achieved in s' and $C_a(s,s') = c_a + penalty$ if goal is not achieved in s'.

Optimal solution is such a policy of MDP, that minimizes an overall expected cost.

2. Algorithm

2.1. Basic Approach

Basic approach is to use MDP with finite horizon, e.g. exhaust every possible action at every decision point and select action having minimal expected cost. In fact,

we use this approach with several pruning techniques which speed up this computation. In Fig. 3 is an example of MDP search of our running example from Fig 1. The root of the MDP is a decision point where we need to decide which of the action among `"Exploit Firewall"`, `"Send MW Email"` and `"Create Dictionary"` is the best to perform. In a naive approach we explore every possible scenario and compute their expected costs $\mathscr{E}$(`"Exploit Firewall"`),$\mathscr{E}$(`"SendMW Email"`) and $\mathscr{E}$(`"Create Dictionary"`). We choose the action with the minimal expected cost.

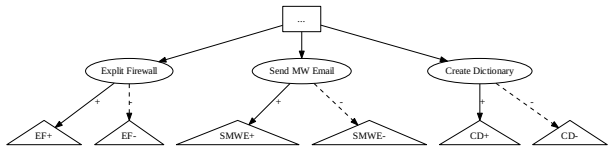

Figure 3. In naive approach we explore every possibility and select action having minimal expected cost.

For performance enhancement, we make use of *transposition tables*, that is: we cache states for which we have computed expected cost and an optimal (sub)policy and reuse these results in future, should we encounter the same state again.

2.2. Sibling-Class Theorem

In [4] authors deal with "probabilistic and-or tree resolution" (PAOTR) problem, mainly for and-or trees with independent tests without preconditions, for which they constructed and proved the Sibling-Class Theorem. Independently, authors in [3] show the same theorem. The Sibling-Class Theorem states, that the leaf-node actions can be grouped into the sibling-classes within which actions' ordering can be determined by simply sorting their R-ratios; hence, no state-search exploration is necessary within sibling-class, only between the sibling classes. Two actions belong to the same sibling class if they have common parent in the and-or tree. As they consider inner nodes to be either AND or OR node, naturally there are two types of sibling classes: the AND-sibling classes and the OR-sibling classes. R-ratios of an action is computed as follows:

$$R(a) = \frac{p_a}{c_a} \text{ if action } a \text{ is in OR-sibling class} \tag{4}$$

$$R(a) = \frac{p_{\bar{a}}}{c_a} \text{ if action } a \text{ is in AND-sibling class} \tag{5}$$

Conjecture 2.1 *Sibling-Class Theorem for and-or trees without preconditions can be applied to an and-or graph with precondition using following rules for creating OR-Sibling Classes:*

- *actions a and b belong to the same OR-Sibling class* iff*:* $pre(a) = pre(b) \wedge |pre(a)| = |pre(b)| = 1$.

and following rules for AND-Sibling Class:

- *action a and b belong to the same AND-Sibling class* iff*:* $pre(a) \neq pre(b) \wedge |pre(a)| = |pre(b)| = 1 \wedge (\exists c \in A : pre(a) \in \mathit{eff}(c) \wedge pre(b) \in \mathit{eff}(c))$.

Action $a \in A$ *cannot be pruned* iff: $|pre(a)| > 1 \vee (\exists c_1, c_2 \in A : c_1 \neq c_2 \wedge pre(a) \in \mathit{eff}(c_1) \wedge pre(a) \in \mathit{eff}(c_2))$.

The Sibling Theorem is proved only for and/or trees, while we empirically checked and use it for and/or graphs.

Example Assume we have the same problem as in Fig. 3 and we come to the same decision point as previously. But now we computed $R(\texttt{"Exploit Firewall"}) = \frac{0.27}{5.0} = 0.054$, $R(\texttt{"Send MW Email"}) = \frac{0.23}{2.0} = 0.115$ and $R(\texttt{"Create Dictionary"}) = \frac{1.0}{11.0} = 0.091$ and we know that actions `"Exploit Firewall"` and `"Send MW Email"` have the same parent node `"Net Access"`, thus belong to the same OR-sibling class, while action `"Create Dictionary"` belongs to a separate sibling class. Now we explore only actions that have maximum R-ratios in each sibling class, thus only actions `"Send MW Email"` and `"Create Dictionary"`, and action `"Exploit Firewall"` surely will not be the first action in the optimal policy. Fig. 4 depicts nodes that must be explored (white nodes), and nodes that are pruned (grey nodes).

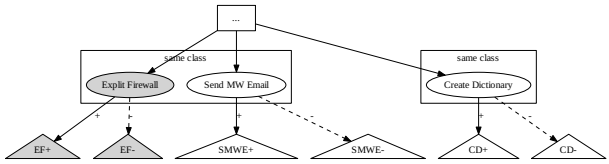

Figure 4. This figure presents nodes that are explored (white) and nodes that can be pruned (grey) in the MDP search if we know that actions `"Exploit Firewall"` and `"Send MW Email"` are in the same sibling class and action R-ratios.

2.3. Branch and Bounds

As another pruning technique we use branch and bounds. For this technique we reuse previously computed expected costs of the subtrees of the MDP to prune future subtrees if know that an optimal solution cannot exist there. Specifically, when we face the decision either utilize (sub)policy ϕ_a – starting with an action a – or (sub)policy ϕ_b – starting with action b – we choose policy ϕ_b only if $\mathscr{E}(\phi_b) < \mathscr{E}(\phi_a)$, implying:

$$\mathscr{E}(\phi_b) < \mathscr{E}(\phi_a) \tag{6}$$

$$c_b + p_b * \mathscr{E}(\phi_{b^+}) + p_{\bar{b}} * \mathscr{E}(\phi_{b^-}) < \mathscr{E}(\phi_a) \tag{7}$$

$$c_b + p_b * \mathscr{E}(\phi_{b^+}) + p_{\bar{b}} * \mathscr{E}(\phi_{b^+}) < \mathscr{E}(\phi_a) \tag{8}$$

$$c_b + \mathscr{E}(\phi_{b^+}) < \mathscr{E}(\phi_a) \tag{9}$$

$$\mathscr{E}(\phi_{b^+}) < \mathscr{E}(\phi_a) - c_b \tag{10}$$

where from (8) to (9) we used property of the optimal policy that $\mathscr{E}(\phi_{b^+}) \leq \mathscr{E}(\phi_{b^-})$, and then fact that $p_b + p_{\bar{b}} = 1$. Thus, $\mathscr{E}(\phi_a) - c_b$ is an upper-bounds for $\mathscr{E}(\phi_{b^+})$. If anytime

during the computation it exceeds this bound, we can immediately stop the computation of the b^+ branch.

Similarly, having computed the $\mathscr{E}(\phi_b)$, we can bound again the branch $\mathscr{E}(\phi_{b^-})$ as follows

$$\mathscr{E}(\phi_b) < \mathscr{E}(\phi_a) \tag{11}$$

$$c_b + p_b * \mathscr{E}(\phi_{b^+}) + p_{\bar{b}} * \mathscr{E}(\phi_{b^-}) < \mathscr{E}(\phi_a) \tag{12}$$

$$p_{\bar{b}} * \mathscr{E}(\phi_{b^-}) < \mathscr{E}(\phi_a) - c_b - p_b * \mathscr{E}(\phi_b) \tag{13}$$

$$\mathscr{E}(\phi_{b^-}) < (\mathscr{E}(\phi_a) - c_b - p_b * \mathscr{E}(\phi_b))/p_{\bar{b}} \tag{14}$$

Example Assume different example, where we face the problem of choosing the best (sub)policy ϕ_a, ϕ_b, ϕ_c and ϕ_d. Branches $\mathscr{E}(\phi_{a^+})$ and $\mathscr{E}(\phi_{a^-})$ we must compute to obtain $\mathscr{E}(\phi_a)$. Next, we compute expected cost $\mathscr{E}(\phi_{b^+})$ and assume that it turns out to be higher than $\mathscr{E}(\phi_a)$, hence, we can prune the computation of the branch b^-, as action b will never have less expected cost then action a. Next, let's say $\mathscr{E}(\phi_{c^+})$ in the branch c^+ turned out to be lower than $\mathscr{E}(\phi_a)$. It means that we can upper bound the $\mathscr{E}(\phi_{c^-})$ by $\frac{\mathscr{E}(\phi_a)-c_c-p_c*\mathscr{E}(\phi_c)}{p_{\bar{c}}}$. Let's say that $\mathscr{E}(\phi_{c^-})$ obeyed the bound, thus, action c is the best action that attacker can perform so far. Note, that it is unnecessary to compute total expected cost of $\mathscr{E}(\phi_c)$ to determine that it is lower then $\mathscr{E}(\phi_a)$, it is direct implication from the fact that it satisfied both bound conditions. Finally, during the computation of branch d^+ it turned out to violate new upper bound $\mathscr{E}(\phi_c) - c_d$, thus its computation was terminated and branch d^- was pruned as well.

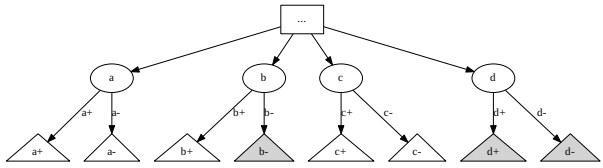

Figure 5. This figure presents nodes that are explored (white) and nodes that can be pruned (grey) due to the branch and bound technique.

2.4. Heuristics

Finally, we designed heuristic approach to compute the lower bound of the expected cost of an attack graph by setting costs of the actions to zero and taking into account only the actions' probabilities. This relaxation gives us freedom in action ordering as any (valid) ordering will produce exactly the same probability of success of the policy and thus the overall expected cost. We use this heuristics in two ways: (i) if computed heuristics exceeds the given upper bound, this branch of computation can be pruned, and (ii) according the heuristics we order the action in which we compute remaining actions'

expected costs. If we start with the most promising action, more of the future branches might get pruned.

Nevertheless, we came across an issue in this approach due to the fact that our attack representation is not a tree but a directed cyclic graph. Which means that performing an action can be beneficial in several possible branches at once if action has more then one root-node paths in the attack graph, which results, that the probability of the action will be count multiple times into the overall probability of success is increased. This causes expected cost, computed as $(1 - probability) * penalty$ to decrease. Since we minimize the expected cost this issue keeps the heuristics still admissible.

3. Experiments

We experimentally compared our algorithm with two other approaches, namely Unconstrained Influence Diagrams, and using probabilistic planner from International Planning Competition.

3.1. Guido approach

This approach, described in [6], converts an attack graph into an Unconstrained Influence Diagrams (UID) — a graphical representation of a decision situations using probabilistic interference — upon which existing solvers can be run. We ran a Guido solver as described in the article. This approach showed to be insufficiently scalable for the problems with large (>20 actions) attack graphs.

3.2. Probabilistic planning

As an other approach, we decided to use a domain independent probabilistic planner SPUDD that competed in International Plannign Competition (IPC) in 2011. SPUDD is based on iterative value computation of MDP and uses own specification language. Since it computes MDP, it needs to have set either discount factor $\gamma = [0,1)$, or $\gamma = 1$ and the horizon set to an integer. For our purposes, discount factor γ must be set to 1, hence horizon had to be chosen appropriately. To ensure that SPUDD finds an optimal solution, we chose to set the horizon to number of actions in the attack graph.

3.3. Experiment settings

We experimentally ran and compared our algorithm DynProg, Guido and SPUDD approaches on the three different realNetwork frameworks with different configurations. We ran experiments on Intel 3.5GHz with memory resource up to 10GB. In DynProg we set the $penalty = 10^9$ and $reward = 0$. In Tab. 1 we present running times of each approach.

4. Conclusion and Future Works

Our algorithm showed to outperform other two approaches in time complexity and scalability. Unfortunately, it ofter runs out of the memory due to the transposition tables and

Problem	DynProg [ms]	Guido [ms]	SPUDD [ms]
Local+2	**51**	85	1000
Local+3	**155**	546	11000
Local+4	**443**	76327	70000
Local+5	**5389**	(OoM)	656000
Local+6	(OoM)	(OoM)	**6152000**
Cross2	**4**	408	1000
Cross3	**38**	23796	9000
Cross4	**504**	(OoM)	287000
Cross5	**3587**	(OoM)	8373000
Cross6	**60351**	(OoM)	(OoT)
LocalChain3-3	**0**	9	**0**
LocalChain4-4	**0**	70	1000
LocalChain5-5	**0**	1169	3000
LocalChain6-6	**0**	17133	23000

Table 1. Time comparison of DynProg, Guido and SPUDD approaches over three types of problems with different complexities. Shortcuts: (OoM) - Out of Memory, (OoT) - Out of Time ($> 10^7$ ms).

very large search state-space anyway. Other optimizations can be proposed, as better representation of a state or more accurate heuristics for better pruning. This algorithm can be used in game theoretic manner in couple of ways. Here we present two directions: (i) determine what honeypot configurations maximize the probability that an attacker would be detected during their attacks on the realNetwork and (ii) for security hardening determining which subset of vulnerabilities should administrator fix in order to secure the realNetwork, that is, that for the attacker it is not worth to attack to begin with.

Acknowledgement

This research was supported by the Office of Naval Research Global (grant no. N62909-13-1-N256) and Czech Ministry of Interior grant number VG20122014079.

References

[1] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based realnetwork vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224. ACM, 2002.

[2] Richard Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 42(10):767, 1956.

[3] Ahto Buldas and Roman Stepanenko. Upper bounds for adversaries utility in attack trees. In Jens Grossklags and Jean Walrand, editors, *Decision and Game Theory for Security*, volume 7638 of *Lecture Notes in Computer Science*, pages 98–117. Springer Berlin Heidelberg, 2012.

[4] Russell Greiner, Ryan Hayward, Magdalena Jankowska, and Michael Molloy. Finding optimal satisficing strategies for and-or trees. *Artificial Intelligence*, 170(1):19–58, 2006.

[5] Jesse Hoey, Robert St-Aubin, Alan J Hu, and Craig Boutilier. Spudd: Stochastic planning using decision diagrams, 1999.

[6] Viliam Lisý and Radek Píbil. Computing optimal attack strategies using unconstrained influence diagrams. In *Intelligence and Security Informatics*, pages 38–46. Springer, 2013.

[7] Carlos Sarraute, Gerardo Richarte, and Jorge Lucángeli Obes. An algorithm to find optimal attack paths in nondeterministic scenarios. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, AISec '11, pages 71–80, New York, NY, USA, 2011. ACM.