Scalable Edge Blocking Algorithms for Defending Active Directory Style Attack Graphs

Mingyu Guo[1], **Max Ward**[2,3], **Aneta Neumann**[1], **Frank Neumann**[1], **Hung Nguyen**[1]

[1] School of Computer and Mathematical Sciences, University of Adelaide, Australia
[2] School of Physics, Maths and Computing, Computer Science and Software Engineering, University of Western Australia
[3] Department of Molecular and Cellular Biology, Harvard University, Cambridge, Massachusetts, USA
{mingyu.guo, aneta.neumann, frank.neumann, hung.nguyen}@adelaide.edu.au, max.ward@uwa.edu.au

Abstract

Active Directory (AD) is the default security management system for Windows domain networks. An AD environment naturally describes an attack graph where nodes represent computers/accounts/security groups, and edges represent existing accesses/known exploits that allow the attacker to gain access from one node to another. Motivated by practical AD use cases, we study a Stackelberg game between one attacker and one defender. There are multiple entry nodes for the attacker to choose from and there is a single target (Domain Admin). Every edge has a failure rate. The attacker chooses the attack path with the maximum success rate. The defender can block a limited number of edges (i.e., revoke accesses) from a set of blockable edges, limited by budget. The defender's aim is to minimize the attacker's success rate.

We exploit the tree-likeness of practical AD graphs to design scalable algorithms. We propose two novel methods that combine theoretical fixed parameter analysis and practical optimisation techniques.

For graphs with small tree widths, we propose a tree decomposition based dynamic program. We then propose a general method for converting tree decomposition based dynamic programs to reinforcement learning environments, which leads to an anytime algorithm that scales better, but loses the optimality guarantee.

For graphs with small numbers of non-splitting paths (a parameter we invent specifically for AD graphs), we propose a kernelization technique that significantly downsizes the model, which is then solved via mixed-integer programming.

Experimentally, our algorithms scale to handle synthetic AD graphs with tens of thousands of nodes.

Introduction

Active Directory (AD) is the *default* system for managing access and security in Windows domain networks. Given its prevalence among large and small organisations worldwide, Active Directory has become a major target by cyber attackers.[1] An AD environment naturally describes a *cyber attack graph* — a conceptual model for describing the causal relationship of cyber events. In an AD graph, the nodes are computers/user accounts/security groups. An edge from node A to node B represents that an attacker can gain access from A to B via an existing access or a known exploit. Unlike many attack graph models that are of theoretical interest only,[2] AD attack graphs are actively being used by real attackers and IT admins. Several software tools (including both open source and commercial software) have been developed for scanning, visualizing and analyzing AD graphs. Among them, one prominent tool is called BLOODHOUND, which models the *identity snowball attack*. In such an attack, the attacker starts from a low-privilege account, which is called the attacker's entry node (i.e., obtained via *phishing* emails). The attacker then travels from one node to another, where the end goal is to reach the highest-privilege account called the DOMAIN ADMIN (DA).

Given an entry node, BLOODHOUND generates a *shortest attack path* to DA, where a path's distance is equal to the number of hops (fewer hops implies less chance of failure/being detected). Before the invention of BLOODHOUND, attackers used personal experience and heuristics to explore the attack graph, hoping to reach DA by chance. BLOODHOUND makes it easier to attack Active Directory.

Besides the attackers, defenders also study Active Directory attack graphs. The original paper that motivated BLOODHOUND (Dunagan, Zheng, and Simon 2009) proposed a heuristic for blocking edges of Active Directory attack graphs. The goal is to cut the attack graph into multiple disconnected regions, which would prevent the attacker from reaching DA. In an AD environment, edge blocking is achieved by revoking accesses or introducing monitoring. *Not all edges are blockable*. Some accesses are required for the organisation's normal operations. *Blocking is also costly* (i.e., auditing may be needed before blocking an edge).

We study how to *optimally* block a limited number of edges in order to minimize a strategic attacker's success rate (chance of success) for reaching DA. In our model, we assume that different edges have different *failure rates*. The defender can block a *blockable* edge to increase the failure rate of that edge (from its original failure rate) to 100%. The defender can block at most b edges, where b is the defensive budget. For example, if it takes 1 hour for an IT admin to block an edge (auditing, reporting, implementation) and one

[1]Enterprise Management Associates (2021) found that 50% of organizations surveyed had experienced an AD attack since 2019.

[2](Lallie, Debattista, and Bal 2020) surveyed over 180 attack graphs/trees from academic literatures on cyber security.

eight-hour day is dedicated to AD cleanup, then $b = 8$.

We study both *pure* and *mixed* strategy blocking. A pure strategy blocks b edges deterministically. A mixed strategy specifies multiple sets of b edges, and a distribution over the sets. We follow the standard Stackelberg game model by assuming that the attacker can observe the defender's strategy and play a best response. For mixed strategy blocking, the attacker can only observe the probabilities, not the actual realizations. There is a set of entry nodes for the attacker to choose from. The attacker's strategy specifies an entry node and from it an attack path to DA. The attacker's goal is to maximize the success rate by choosing the best path.

The pure strategy version of our model can be reduced to the *single-source single-destination shortest path edge interdiction* problem, which is known to be NP-hard (Barnoy, Khuller, and Schieber 1995). However, NP-hardness on general graphs does not rule out efficient algorithms for practical AD graphs. *Active Directory style attack graphs exhibit special graph structures and we can exploit these structures to derive scalable algorithms.* We adopt *fixed-parameter analysis*. Formally, given an NP-hard problem with problem size n, let the easy-to-solve instances be characterized by special parameters $k_1, \ldots, k_c$. If we are able to derive an algorithm that solves these instances in $O(f(k_1, \ldots, k_c)\text{POLY}(n))$, then we claim that the problem is *fixed-parameter tractable (FPT)* with respect to the k_is. Here, f is an arbitrary function that is allowed to be exponential. We do require that the running time be polynomial in the input size n. That is, an FPT problem is practically solvable for large inputs, as long as the special parameters are small (i.e., they indeed describe *easy* instances).

It should be noted that this paper's focus is not to push the frontier of theoretical fixed-parameter analysis. Instead, fixed-parameter analysis is our **means** *to design scalable algorithms for our specific application on AD graphs.* As a matter of fact, our approaches combine theoretical fixed-parameter analysis and practical optimisation techniques.

We observe that practical AD graphs have two noticeable structural features.[3] We first observe that *the attack paths tend to be short*. Note that we are not claiming that long paths do not exist (i.e., there are cycles in AD graphs). The phrase "attack paths" refer to shortest paths that the attacker would actually use. The BLOODHOUND team uses the phrase "six degrees of domain admin" to draw the analogy to the famous "six degree of separation" idea from the small-world problem (Milgram 1967) (i.e., all people in this world are on average six or fewer social connections away from each other). That is, in an organisation, it is expected that it takes only a few hops to travel from an intern's account to the CEO's account. Similar to the "small-world" hypothesis, attack paths being short is an *unproven observation* that we expect to hold true for practical purposes.

The second structural feature is that AD graphs are very similar to trees. The tree-like structure comes from the fact that it is considered a best practise for the AD environment to follow the organisation chart. For example, human resources would form one tree branch while marketing would form another tree branch. However, an Active Directory attack graph is almost never exactly a tree, because there could be valid reasons for an account in human resources to access data on a computer that belongs to marketing. We could interpret Active Directory attack graphs as *trees with extra* non-tree edges *that represent security exceptions*.

Our aim is to design *practically scalable* algorithms for optimal pure and mixed strategy edge blocking. For organisations with thousands of computers in internal networks, the AD graphs generally involve *tens of thousands of nodes*. We manage to scale to such magnitude by exploiting the aforementioned structural features of practical AD graphs.

We first show that having short attack paths alone is not enough to derive efficient algorithms. Even if the maximum attack path length is a constant, both pure and mixed strategy blocking are NP-hard. We then focus on exploring the tree-likeness of practical AD graphs.

Our first approach focuses on pure strategy blocking only. For graphs with small tree widths, we propose a tree decomposition based dynamic program, which scales better than existing algorithms from (Guo et al. 2022). We then propose a general method for converting tree decomposition based dynamic programs to reinforcement learning environments. When tree widths are small, the derived reinforcement learning environments' observation and action spaces are both small. This leads to an anytime algorithm that scales better than dynamic program, but loses the optimality guarantee.

Our second approach handles both pure and mixed strategy blocking. We invent a non-standard fixed parameter specifically for our application on AD graphs. A typical attack path describes a *privilege escalation* pathway. It is rare for a node to have more than one *privilege escalating* out-going edges (as such edges often represent security exceptions or misconfigurations). We observe that practical AD graphs consist of *non-splitting paths* (paths where every node has one out-going edge). For graphs with small numbers of non-splitting paths, we propose a kernelization technique that significantly downsizes the model, which is then solved via mixed-integer programming. We experimentally verify that this approach scales *exceptionally well* on synthetic AD graphs generated by two open source AD graph generators (DBCREATOR and ADSIMULATOR).

Related Research

(Guo et al. 2022) studied edge interdiction for AD graphs, where the attacker is given an entry node by nature. In (Guo et al. 2022), the defensive goal is to maximize the attacker's *expected* attack path length (i.e., number of hops). In this paper, the defensive goal is to minimize the attacker's *worst-case* success rate, based on the assumption that different edges may have different failure rates. The authors proposed a tree decomposition based dynamic program, but it only applies to acyclic AD graphs. Practical AD graphs do contain cycles so this algorithm does not apply. The authors resorted to graph convolutional neural network as a heuristic to handle large AD graphs with cycles. *Our proposed algorithms*

[3]The AD environment of an organisation is considered sensitive. In this paper, we only reference synthetic AD graphs generated using two tools: BLOODHOUND team's DBCREATOR and another open source tool called ADSIMULATOR.

can handle cycles, scale better, and produce the optimal results (instead of being mere heuristics). Furthermore, (Guo et al. 2022) only studied pure strategy blocking.

(Goel et al. 2022) studied a different model on edge interdiction for AD graphs. Under the authors' model, both the attacker's and the defender's problem are #P-hard, and the authors proposed a defensive heuristic based on combining neural networks and diversity evolutionary computation.

The model studied in this paper is similar to the *bounded length cut* problem studied in (Golovach and Thilikos 2011) and (Dvořák and Knop 2018), where the goal is to remove some edges so that the minimum path length between a source and a destination meets a minimum threshold. (Dvořák and Knop 2018) proposed a tree decomposition based dynamic program for the bounded length cut problem. The authors' algorithms require that all source and destination nodes be added to every bag in the tree decomposition. *This is fine for showing the theoretical existence of FPT algorithms, but it is practically not scalable.* Furthermore, for bounded length cut, if a path is shorter than the threshold, then it *must be cut* and if a path is longer than the threshold, then it *can be safely ignored.* This is a much clearer picture than our model where we need to judge the relative importance of edges and spend the budget on the most vital ones.

(Jain and Korzhyk 2011) proposed a double-oracle algorithm for equilibrium calculation on attack graphs, whose model is defined differently. Their approach is designed for general graphs so it only scales to a few hundred nodes and therefore is not suitable for practical AD graphs. (Aziz et al. 2018; Aziz, Gaspers, and Najeebullah 2017) studied node interdiction for minimizing inverse geodesic length. (Durkota et al. 2019) and (Milani et al. 2020) studied deception based defense on cyber attack graphs.

Formal Model Description

We use a directed graph $G = (V, E)$ to describe the Active Directory environment. Every edge e has a failure rate $f(e)$. There is one destination node DA (Domain Admin). There are s entry nodes. The attacker can start from any entry node and take any route. The attacker's goal is to maximize the success rate to reach DA, by picking an optimal entry node and an optimal attack path. The defender picks b edges to block from a set of blockable edges $E_b \subseteq E$, where b is the defensive budget. The aim of the defender is to minimize the attacker's success rate.

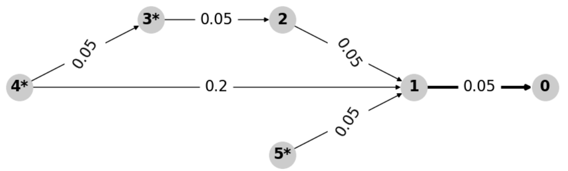

Figure 1: Example attack graph. Node 0 is DA. Node $4, 3, 5$ are entry nodes (marked using $*$). Edge labels represent the edges' failure rates. Thick edges (i.e., $1 \to 0$) are not blockable.

We study both pure and mixed strategy blocking. We use $B(e)$ to describe the probability that a blockable edge $e \in E_b$ is blocked. For pure strategy blocking, $B(e)$ equals either 0 or 1. For mixed strategy blocking, $B(e)$ is in $[0, 1]$. The budget constraint is $\sum_{e \in E_b} B(e) \leq b$. We adopt the standard Stackelberg game model by assuming that the attacker can observe the defensive strategy and then plays a best response. For a mixed strategy defense, we assume that the attacker can observe $B(e)$'s probabilistic values, but not the realisations.

Given B, the attacker's optimal attack path can be found via $\max_{p \in P} \left\{ \prod_{e \in p} (1 - f(e))(1 - B(e)) \right\}$, where P is the set of all attack paths from all entry nodes. This maximization problem is equivalent to $\min_{p \in P} \left\{ \sum_{e \in p} (-\ln(1 - f(e)) - \ln(1 - B(e))) \right\}$. By applying natural log to convert from product to sum, we treat an edge's "distance" as $-\ln(1 - f(e)) - \ln(1 - B(e))$ (nonnegative). The attacker's optimal attack path can be solved using Dijkstra's shortest path algorithm (Dijkstra 1959). Let $\mathrm{SR}(B)$ be the success rate of the attacker facing blocking policy B. The defender's problem is $\min_B \mathrm{SR}(B)$.

Earlier we mentioned that our plan is to exploit the special structural features of practical AD graphs. Our first result is a negative result, which shows that having short attack paths alone is not enough to derive efficient algorithms. That is, we do need to consider the tree-like features.

Theorem 1. *Both pure and mixed strategy blocking are NP-hard for constant maximum attack path length.*

Proof is omitted due to space constraint.

Tree Decomposition Based Dynamic Program for Pure Strategy Blocking

Tree decomposition is a process that converts a general graph to a tree, where every tree node is a bag (set) of graph vertices. The maximum bag size minus one is called the *tree width*. A small tree width indicates that the graph is close to a tree. Many NP-hard combinatorial problems for general graphs become tractable if we focus on graphs with small tree widths. We show that this is also true for our model. For the rest of this section, we assume the readers are already familiar with tree decomposition related terminologies.

Throughout the discussion, we use *nodes* to refer to tree nodes in the tree decomposition and *vertices* to refer to vertices in AD graphs.

Besides assuming a small tree width, another key assumption of our dynamic program is that we assume a path's success rate is from a small set of at most H values. A path's success rate is the attacker's success rate for going through it without any blocking. If all edges have the same failure rate, then H is just the *maximum attack path length* l plus 2 (0 to l hops, plus "no path"). In general, if the number of edge types is a small constant k, then $H \in O(l^k)$. In our experiments, we assume that there are two types of edges (high-failure-rate and low-failure-rate edges), which corresponds to $H \in O(l^2)$. It should be noted that H is only used for *worst-case* complexity analysis. In experiments, given a specific graph, a path's number of possible success rates is often significantly less (i.e., if the path is not blockable altogether, then there is only one possible success rate).

We call our DP TDCYCLE (tree decomposition with cycles[4]). The first step is to treat the attack graph as an undirected graph and then generate a tree decomposition. It should be noted that the optimal tree decomposition with the minimum tree width is NP-hard to compute (Arnborg, Corneil, and Proskurowski 1987). In our experiments, we adopt the *vertex elimination* heuristic for generating tree decomposition (Bodlaender et al. 2006). We then convert the resulting tree decomposition into a *nice tree decomposition* (Cygan et al. 2015), where the root node is a bag containing DA only and all the leaf nodes are bags of size one. We use TD to denote the resulting nice tree decomposition. TD has $O(wn)$ nodes where w is the tree width.

Lemma 1. *Let (u, v) be an arbitrary edge from the original AD graph. Under* TD*, there exists one and only one* forget node X*, whose child is denoted as X', where $\{u, v\} \subseteq X'$ and $X' \setminus X$ is either $\{u\}$ or $\{v\}$.*

The above lemma basically says that every edge can be "assigned" to exactly one forget node. For forget node $(x_2, x_3, \ldots, x_k)$ with child $(x_1, x_2, \ldots, x_k)$ (i.e., x_1 is forgotten), we assign all edges between x_1 and one of $x_2, \ldots, x_k$ to this forget node. The high-level process of our dynamic program is that we first remove all edges from the graph. We then go through TD **bottom up**. At forget node X, we examine all the edges assigned to X. If an edge is not blockable or we decide not to block it, then we put it back into the graph. Otherwise, we do not put it back. After we finish the whole tree (while ensuring that the budget spent is at most $b \iff$ we have put back at least $|E| - b$ edges), we end up with a complete blocking policy.

Let $X = (x_1, x_2, \ldots, x_k)$ be a tree node ($k \leq w + 1$). Let $St(X)$ be the subtree of TD rooted at X. Let $Ch(X)$ be the set of all graph vertices referenced in $St(X)$. Let $Ch(X)' = Ch(X) \setminus X$. $Ch(X)'$ is then the set of vertices already forgotten after we process $St(X)$ in the bottom-up fashion. A known property of tree decomposition is that the vertices in $Ch(X)'$ cannot directly reach any vertex in $V \setminus Ch(X)$. That is, any attack path from an entry vertex in $Ch(X)'$ to DA must pass through some x_i in X. Also, any attack path (not necessarily originating from $Ch(X)'$) may involve vertices in $Ch(X)'$ by entering the graph region form by $Ch(X)'$ via a certain x_i and then exit the region via a different node x_j. An attack path may "enter and exit" the region multiple times but all entries and exists must be via the vertices in X.

Suppose we have spent b' units of budget on (forget nodes of) $St(X)$. We do not need to keep track of the specifics of which edges have been blocked. We only need to track the total budget spending and the following "distance" matrix:

$$M = \begin{bmatrix} d_{11} & d_{12} & \ldots & d_{1k} \\ \ldots & & & \\ d_{k1} & d_{k2} & \ldots & d_{kk} \end{bmatrix}$$

d_{ij} represents the minimum path distance[5] between x_i and x_j, where the intermediate edges used are the edges we have already put back after processing $St(X)$. Diagonal element d_{ii} represents the minimum path distance from any entry vertex (among $Ch(X)$) to x_i. We say the tuple (M, b') is *possible* at X if and only if it is possible to spent b' ($b' \leq b$) on $St(X)$ to achieve the distance matrix M.

Every tree node of TD corresponds to a DP subproblem and there are $O(wn)$ subproblems. The subproblem corresponding to X is denoted as $DP(X)$. $DP(X)$ simply returns the collection of all possible tuples at node X.

Base cases: For a leaf node $X = \{x\}$, if x is an entry vertex, then $DP(X)$ contains one tuple, which is $([0], 0)$. Otherwise, the only possible tuple in $DP(X)$ is $([\infty], 0)$.

Original problem: The root of TD is $\{\text{DA}\}$. The original problem is then $DP(\{\text{DA}\})$, which returns the collection of all possible tuples at the root. Every tuple from the collection has the form $([d_{\text{DA,DA}}], b')$, which represents that it is possible to spend b' to ensure that the attacker's distance from DA is $d_{\text{DA,DA}}$. The maximum $d_{\text{DA,DA}}$ in $DP(\{\text{DA}\})$ corresponds to the attacker's success rate facing optimal blocking.

We then present the recursive relationship for our DP:

Introduce node: Let $X = (x_1, \ldots, x_k, y)$ be an introduce node, whose child is $X' = (x_1, \ldots, x_k)$. Given a possible tuple in $DP(X')$, we generate a new tuple as follows, which should belong to $DP(X)$. d_{yy} is 0 if y is an entry vertex and it is ∞ otherwise (when y is introduced, all its edges have not been put back yet so it is disconnected from the x_i).

$$\left(\begin{bmatrix} d_{11} & \ldots & d_{1k} \\ \ldots & & \\ d_{k1} & \ldots & d_{kk} \end{bmatrix}, b' \right) \rightarrow \left(\begin{bmatrix} d_{11} & \ldots & d_{1k} & \infty \\ \ldots & & & \\ d_{k1} & \ldots & d_{kk} & \infty \\ \infty & \ldots & \infty & d_{yy} \end{bmatrix}, b' \right)$$

Forget node: Let $X = (x_2, \ldots, x_k)$ be a forget node, whose child is $X' = (x_1, \ldots, x_k)$. At X, we need to determine how to block edges connecting x_1 and the rest $x_2, \ldots, x_k$. There are at most $k - 1$ edges to block so we simply go over at most 2^{k-1} blocking options. For each specific blocking option (corresponding to a spending of b''), we convert a tuple in $DP(X')$ to a tuple in $DP(X)$ as follows (the new tuple is discarded if $b' + b'' > b$):

$$\left(\begin{bmatrix} d_{11} & \ldots & d_{1k} \\ \ldots & & \\ d_{k1} & \ldots & d_{kk} \end{bmatrix}, b' \right) \rightarrow \left(\begin{bmatrix} d'_{22} & \ldots & d'_{2k} \\ \ldots & & \\ d'_{k2} & \ldots & d'_{kk} \end{bmatrix}, b' + b'' \right)$$

The d'_{ij} are updated distances considering the newly put back edges. We need to run an *all-pair shortest path* algorithm with complexity $O(k^3)$ for this update.

Join node: Let X be a join node with two children X_1 and X_2. For $(M_1, b_1) \in DP(X_1)$ and $(M_2, b_2) \in DP(X_2)$, we label $(M', b_1 + b_2)$ as a possible tuple in $DP(X)$ if $b_1 + b_2 \leq b$. M' is the element-wise minimum between M_1 and M_2.

[4]This is to differentiate from the dynamic program proposed in (Guo et al. 2022), which cannot handle cycles and does not guarantee correctness for practical AD graphs as they do contain cycles.

[5]Recall that given an edge with failure rate $f(e)$, we treat the edge's "distance" as $-\ln(1 - f(e))$ when it is not blocked.

Theorem 2. TDCYCLE*'s complexity is* $O(H^{2w^2}b^2w^2n)$.

Proof is omitted due to space constraint.

To summarize our dynamic program, we follow a bottom-up order (from leaf nodes of nice tree decomposition TD to the root). We propagate the set of all possible tuples (M, b') as we process the nodes. At introduce/join nodes, we follow a pre-determined propagation rule and do not make any blocking decisions. At forget nodes, we decide which edges to block from at most w edges. Given a specific AD graph and its corresponding tree decomposition TD, we can convert our dynamic program to a reinforcement learning environment as follows, which leads to an anytime algorithm that scales better than dynamic program (i.e., RL can always produce a solution, which may or may not be optimal, and generally improves over time; on the other hand, dynamic program cannot scale to handle slightly larger tree widths). *Our conversion technique can potentially be applied to tree decomposition based dynamic programs for other combinatorial optimisation problems.*

- We use post-order traversal on TD to create an ordering of the nodes (children are processed before parents).
- Instead of propagating *all possible* tuples (M, b'), we only propagate the *best tuple* (we have found so far during RL training). For example, consider a forget node X with child X'. The best tuple at node X' is passed on to X as *observation*. The *action* for this observation is then to decide which edges to block at X.
- Specifically to our model, for forget node X, if there are k ($k \leq w$) edges to block, then we treat it as k separate steps in our reinforcement learning environment. That is, every *step* makes a blocking decision on a single edge and the action space is always binary.
- For introduce/join nodes, since we do not need to make any decisions, our reinforcement learning environment automatically processes these nodes (between *steps*).
- We set a final *reward* that is equal to the solution quality.

After we convert a specific AD graph into a reinforcement learning environment, we can then apply standard RL algorithms to search for the optimal blocking policy for the AD graph under discussion.

Following our conversion, both the observation and the action spaces are small when tree widths are small. Unfortunately, one downside of the above conversion technique is that there is no guarantee on having a small *episode length*. We could argue that it is impossible to guarantee small observation space, small action space, and small episode length at the same time, unless the AD graph is relatively small in scale. *After all, we are solving NP-hard problems.* Experimentally, for smaller AD graphs, we are able to achieve near-optimal performances as the episode lengths are manageable. For larger AD graphs, the episode lengths are too long. We introduce the following heuristic for limiting the episode length. Let T be the target episode length. Our idea is to hand-pick T *relatively important* edges and set the unpicked edges not blockable. In our experiments, we first calculate the *min cut* that separates the entry nodes from DA (unblockable edges' capacities are set to be large). Let the number of blockable edges in the min cut be C. If $C \geq T$, then we simply treat these C edges as important and set the episode length to C. If $C < T$, then we add in $T - C$ blockable edges that are closest to DA.

It should be noted that our RL-based approach is not merely performing "random searching" via exploration. It is indeed capable of "learning" to react to the given observation. Under our original approach, the observation contains the distance matrix M, the budget spent b', and also the current step index. If we replace M by the zero matrix or by a random matrix, then the training results significantly downgrade in experiments.

Kernelization

Kernelization is a commonly used fixed-parameter analysis technique that preprocesses a given problem instance and converts it to a much smaller *equivalent* problem, called the *kernel*. We require that the kernel's size be bounded by the special parameters (and not depend on n).

As mentioned in the introduction, for practical AD graphs, most nodes have at most one out-going edge. If an edge is not useful for the attacker, then we can remove it without loss of generality. If an edge is useful for the attacker, then generally, it is *privilege escalating*. It is rare for a node to have two separate *privilege escalating* out-going edges (as they often correspond to security exceptions or misconfigurations). We use SPLIT to denote the set of all splitting nodes (nodes with multiple out-going edges). We use SPLIT+DA to denote SPLIT with DA added. We use ENTRY to denote the set of all entry nodes. We invent a new parameter called the number of *non-splitting paths*. Experimentally, this parameter leads to algorithms that scale exceptionally well for synthetic AD graphs generated using two different open source AD graph generators.

Definition 1 (Non-splitting path)**.** Given node u, let v be one of u's successors. The non-splitting path $\text{NSP}(u, v)$ is defined recursively:

- If $v \in$ SPLIT+DA, then $\text{NSP}(u, v)$ is $u \to v$.
- Otherwise, v must have a unique successor v'. $\text{NSP}(u, v)$ is the path that combines $u \to v$ and $\text{NSP}(v, v')$.

In words, $\text{NSP}(u, v)$ is the path that goes from u to v, then repeatedly moves onto the only successor of v if v has only one successor, until we reach either a splitting node or DA. We use $\text{DEST}(u, v)$ to denote the ending node of $\text{NSP}(u, v)$. We have $\text{DEST}(u, v) \in$ SPLIT+DA. A non-splitting path is called *blockable* if at least one of its edges is blockable.

An AD graph can be viewed as the **joint** of the following set of non-splitting paths. Our parameter (the number of non-splitting paths #NSP) is the size of this set.

$$\{\text{NSP}(u, v) | u \in \text{SPLIT} \cup \text{ENTRY}, v \in \text{SUCCESSORS}(u)\}$$

The blockable edge furthest away from u on the path $\text{NSP}(u, v)$ is denoted as $\text{BW}(u, v)$. BW stands for *blockworthy* due to the following lemma.

Lemma 2. *For both pure and mixed strategy blocking, we never need to spend more than one unit of budget on a non-splitting path. There exists an optimal defense that blocks* *only edges from the following set* BW*:*

$$\{\text{BW}(u,v) | u \in \text{SPLIT} \cup \text{ENTRY}, v \in \text{SUCCESSORS}(u)\}$$

We present how to formulate our model as a nonlinear program, based on the aforementioned non-splitting path interpretation. The nonlinear program can then be converted to MIPs and be efficiently solved using state-of-the-art MIP solvers. We use B_e to denote the unit of budget spent on edge e. B_e is binary for pure strategy blocking and is between 0 and 1 for mixed strategy blocking. As mentioned earlier, $B_e \geq 0$ for $e \in \text{BW}$ and $B_e = 0$ for $e \notin \text{BW}$. We use r_u to denote the success rate of node u. The success rate of a node is the success rate of the optimal attack path starting from this node, under the current defense (i.e., the B_e). We use $c_{u,v}$ to denote the success rate of the non-splitting path $\text{NSP}(u,v)$ when no blocking is applied. $c_{u,v} = \prod_{e \in \text{NSP}(u,v)}(1 - f(e))$. The $c_{u,v}$ are constants. We use r^* to represent the attacker's optimal success rate.

$$C = \{(u,v) | u \in \text{SPLIT} \cup \text{ENTRY}, v \in \text{SUCCESSORS}(u)\}$$

$$C^+ = \{(u,v) | (u,v) \in C, \text{NSP}(u,v) \text{ blockable}\}$$

$$C^- = \{(u,v) | (u,v) \in C, \text{NSP}(u,v) \text{ not blockable}\}$$

We have the following nonlinear program:

$$\begin{array}{rcll}
\min & r^* & & \\
r^* & \geq & r_u & \forall u \in \text{ENTRY} \\
r_u & \geq & r_{\text{DEST}(u,v)} c_{u,v} (1 - B_{\text{BW}(u,v)}) & \forall (u,v) \in C^+ \\
r_u & \geq & r_{\text{DEST}(u,v)} c_{u,v} & \forall (u,v) \in C^- \\
b & \geq & \sum_{e \in \text{BW}} B_e & \\
r_{\text{DA}} & = & 1 & \\
r_u, r^* & \in & [0,1] & \\
B_e & \in & \{0,1\} \text{ or } [0,1] &
\end{array}$$

The above program has at most $O(\#\text{NSP})$ variables and at most $O(\#\text{NSP})$ constraints (both do not depend on n).

Integer program for pure strategy blocking: For pure strategy blocking, the above program can be converted to an IP by rewriting $r_u \geq r_{\text{DEST}(u,v)} \cdot c_{u,v} \cdot (1 - B_{\text{BW}(u,v)})$ into an equivalent linear form $r_u \geq r_{\text{DEST}(u,v)} \cdot c_{u,v} - B_{\text{BW}(u,v)}$.

Mixed Strategy Blocking

We can convert the nonlinear program from the previous section into a program that is *almost* linear. We first observe that if under the optimal mixed strategy defense, the attacker's success rate is at least ϵ, then that means we never want to block any edge with a probability that is *strictly* more than $1 - \epsilon$. Due to this observation, we artificially enforce that we do not block an edge with more than $1 - \epsilon$ probability and solve for the optimal defense under this restriction. If in the end result, the attacker's success rate is at least ϵ, then our restriction is not actually a restriction at all. If our solution says that the attacker's success rate is less than ϵ, then we have an almost optimal defense anyway. In this paper, we set $\epsilon = 0.01$. That is, for $e \in \text{BW}$, we require that $0 \leq B_e \leq 0.99$ instead of $0 \leq B_e \leq 1$.

With the above observation, we can convert the nonlinear program, which involves multiplication, to an *almost* linear program as follows. Our trick is to replace r_u by $r'_u = -\ln(r_u)$, replace r^* by $r'^* = -\ln(r^*)$, replace B_e by $B'_e = -\ln(1 - B_e)$, and finally replace $c_{u,v}$ by $c'_{u,v} = -\ln(c_{u,v})$. Our variables are now r'^*, the r'_u and the B'_e. Due to monotonicity of natural log, we can rewrite the earlier nonlinear program as:

$$\begin{array}{rcll}
\max & r'^* & & \\
r'^* & \leq & r'_u & \forall u \in \text{ENTRY} \\
r'_u & \leq & r'_{\text{DEST}(u,v)} + c'_{u,v} + B'_{\text{BW}(u,v)} & \forall (u,v) \in C^+ \\
r'_u & \leq & r'_{\text{DEST}(u,v)} + c'_{u,v} & \forall (u,v) \in C^- \\
b & \geq & \sum_{e \in \text{BW}} (1 - e^{-B'_e}) & \\
r'_{\text{DA}} & = & 0 & \\
r'_u, r'^* & \in & [0, \infty) & \\
B'_e & \in & [0, -\ln(0.01)] &
\end{array}$$

Unfortunately, the above program is not linear as the budget constraint is not linear. Furthermore, the above program is not even convex: Since $1 - e^{-x}$ is concave, the average of two feasible solutions may violate the budget constraint.

Iterative LP Based Approximation Heuristic

We note that $1 - e^{-x}$ is increasing. That is, as long as we push down the total of B'_e, we eventually will reach a situation where the budget constraint is satisfied. That is, we rewrite the budget constraint as $b' \geq \sum_{e \in \text{BW}} B'_e$, which is a linear constraint. We **guess** a value for b' and then solve the corresponding LP. We verify whether the LP solution also satisfies the original nonlinear budget constraint. If it does, then that means we could increase our guess of b' (increasing b' means spending more budget). If our LP solution violates the original nonlinear budget constraint, then it means we should decrease our guess of b'. A good guess of b' can be obtained via a binary search from 0 to $-|\text{BW}| \ln(0.01)$.

MIP Based Approximation

Another way to address the nonlinear budget constraint is to add the B_e back into the model ($0 \leq B_e \leq 0.99$ for $e \in \text{BW}$). The budget constraint $\sum_{e \in \text{BW}} B_e \leq b$ is now back to linear. Of course, this cannot be the end of the story, since we also need to link the B_e and the B'_e together. We essentially have introduced a new set of nonlinear constraints, which are $B'_e = -\ln(1 - B_e)$ for $e \in \text{BW}$.

The function $-\ln(1 - x)$ is close to a straight line if we focus on a small interval. If $x \in [a, b]$, the straight line connecting $(a, -\ln(1 - a))$ and $(b, -\ln(1 - b))$ is an upper bound of $-\ln(1 - x)$. A straight line representing a lower bound of $-\ln(1 - x)$ is the tangent line at the interval's mid point $\frac{a+b}{2}$.

Given a specific B_e, we could divide B_e's region $[0, 0.99]$ into multiple smaller intervals. For each region, we have two straight lines that represent the lower and upper bounds on $-\ln(1 - x)$. We could join the regions and compose $g_L(x)$ and $g_U(x)$, which are the *piece-wise linear* lower and upper bounds on $-\ln(1 - x)$. We replace $B'_e = -\ln(1 - B_e)$ by $B'_e = g_L(B_e)$ or $B'_e = g_U(B_e)$, respectively, to create two different MIP programs. We use the *multiple choice model* presented in (Croxton, Gendron, and Magnanti 2003) to implement piece-wise linear constraints with the help of auxiliary binary variables. The program with $B'_e = g_L(B_e)$ underestimates the B'_e, which results in an overestimation of the budget spending and therefore an overestimation of the attacker's success rate. This program results in **an achieved feasible defense**. The program with $B'_e = g_U(B_e)$ on the other hand results in **a lower bound on the attacker's success rate**. When the intervals are fine enough, experimentally, the achieved feasible defense is close to the lower bound (therefore close to optimality).

Experiments

All our experiments are carried out on a desktop with i7-12700 CPU and NVIDIA GeForce RTX 3070 GPU. Our MIP solver is Gurobi 9.5.1. For pure strategy blocking, we proposed two algorithms: TDCycle and IP (integer program based on kernelization). We also include a third algorithm Greedy to serve as a baseline. Greedy spends one unit of budget in each round for a total of b rounds. In each round, it greedily blocks one edge to maximally decrease the attacker's success rate. For mixed strategy blocking, we have an iterative LP based heuristic IterLP, a mixed integer program MIP-F(easible) for generating a feasible defense and a mixed integer program MIP-LB for generating a lower bound on the attacker's success rate.

We evaluate our algorithms using two attack graphs generated using BloodHound team's synthetic graph generator DBCreator. We call the attack graph R2000 and R4000, which are obtained by setting the number of computers in the AD environment to 2000 and 4000. R2000 contains 5997 nodes and 18795 edges and R4000 contains 12001 nodes and 45780 edges. We also generate a third attack graph using a different open source synthetic graph generator adsimulator. adsimulator by default generates a trivially small graph. We increase all its default parameters by a factor of 10 and create an attack graph called ADS10. ADS10 contains 3015 nodes and 12775 edges. Even though ADS10 contains less nodes, experimentally it is actually more expensive to work with compared to R2000 and R4000, as it is further away from a tree.

We only consider three edge types: AdminTo, MemberOf, and HasSession. These are a representative sample of edges types used in BloodHound. We set the failure rates of all edges of type HasSession to 0.2 (requiring a session between an account and a computer, therefore more likely to fail) and set the failure rates of all edges of the other two types to 0.05. We set the number of entry nodes to 20. We select 40 nodes that are furthest away from DA (in terms of the number of hops to reach DA) and randomly draw 20 nodes among them to be the entry nodes. We define $\text{Hop}(e)$ to be the minimum number of hops between an edge e and DA. We set MaxHop to be the maximum value for $\text{Hop}(e)$. An edge is set to be blockable with probability $\frac{\text{Hop}(e)}{\text{MaxHop}}$. That is, edges further away from DA are set to be more likely to be blockable. Generally speaking, edges further away from DA tend to be about individual employees' accesses instead of accesses between servers and admins. We set the budget to 5 and 10. All experiments are repeated 10 times, with different random draws of the entry nodes and the blockable edges. The numbers in the table are the attacker's average success rates over 10 trials. The numbers in the parenthesis are the average running time.

budget=5	R2000	R4000	ADS10
Greedy	0.521 (0.04s)	0.376 (0.34s)	0.448 (4.57s)
TDCycle	0.480 (0.10s)	0.373 (15366s)	-
IP	0.480 (0.01s)	0.373 (0.06s)	0.409 (0.09s)
IterLP	0.337 (0.37s)	0.180 (0.57s)	0.300 (1.90s)
MIP-F	0.335 (0.11s)	0.179 (3.54s)	0.303 (4.75s)
MIP-LB	0.333 (0.11s)	0.176 (2.61s)	0.297 (4.98s)

budget=10	R2000	R4000	ADS10
Greedy	0.499 (0.07s)	0.376 (0.65s)	0.448 (8.97s)
TDCycle	0.263 (0.18s)	-	-
IP	0.263 (0.01s)	0.117 (0.03s)	0.315 (0.10s)
IterLP	0.266 (0.39s)	0.033 (0.60s)	0.190 (1.92s)
MIP-F	0.270 (0.02s)	0.023 (0.54s)	0.191 (11.27s)
MIP-LB	0.263 (0.02s)	0.014 (0.24s)	0.183 (3.62s)

Interpretation of Results: For pure strategy blocking, TDCycle and IP are both expected to produce the optimal results. As expected, they perform better than Greedy. TDCycle doesn't scale for 3 out of 6 settings. On the other hand, IP scales exceptionally well. As mentioned earlier, IP scales better since it is based on a parameter that we invent specifically for describing AD graphs. For graphs with large number of non-splitting paths and small tree widths, we expect TDCycle to scale better, *but such graphs may not be AD graphs*. For mixed strategy blocking, the attacker's success rates under both IterLP and MIP-F(easible) are close to MIP-LB (lower bound on the attacker's success rate), which indicates that both heuristics are near-optimal.

Results on scaling TDCycle via reinforcement learning: We present results on two settings: R2000 with $b = 5$ and ADS10 with $b = 10$. These two are the *cheapest* and the *most expensive* among our six experimental settings. All our experimental setups are the same as before. We directly apply *Proximal Policy Optimization* PPO (Schulman et al. 2017). For each environment, we use training seed 0 to 9 and record the best result.

	Opt	Greedy	RL	=Opt	Time
R2000, $b = 5$	0.480	0.521	0.480	10/10	1hr
ADS10, $b = 10$	0.315	0.448	0.319	9/10	4hr

Opt, Greedy, RL each represents the average performance of these three different approaches (average over 10 trials with random draws of the entry nodes and the blockable edges). "Time" refers to training time. For R2000 with $b = 5$, we obtain the optimal result in 10 out of 10 trials without limiting the episode length. The maximum episode length is 27 over 10 trials. For ADS10 with $b = 10$, we set an episode length of 15 and obtain the optimal result in 9 out of 10 trials. We recall that, for this setting, TDCycle doesn't scale at all, but we manage to achieve near-optimal results via reinforcement learning.

Acknowledgements

Frank Neumann has been supported by the Australian Research Council through grant FT200100536. Hung Nguyen is supported by the Next Generation Technology Fund (NGTF-Cyber) grant MyIP 10614 and the Australian Research Council (ARC-NISDRG) grant NI210100139. This work was supported with supercomputing resources provided by the Phoenix HPC service at the University of Adelaide.

References

Arnborg, S.; Corneil, D. G.; and Proskurowski, A. 1987. Complexity of Finding Embeddings in a K-Tree. *Siam Journal of Discrete Mathematics*, 8(2): 277–284.

Aziz, H.; Gaspers, S.; Lee, E. J.; and Najeebullah, K. 2018. Defender Stackelberg Game with Inverse Geodesic Length as Utility Metric. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, 694–702. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Aziz, H.; Gaspers, S.; and Najeebullah, K. 2017. Weakening Covert Networks by Minimizing Inverse Geodesic Length. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, 779–785. Melbourne, Australia: International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-0-3.

Baier, G.; Erlebach, T.; Hall, A.; Köhler, E.; and Schilling, H. 2006. Length-Bounded Cuts and Flows. In *In Proc. 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, 679–690.

Bar-noy, A.; Khuller, S.; and Schieber, B. 1995. The Complexity of Finding Most Vital Arcs and Nodes. Technical report, University of Maryland.

Bodlaender, H. L.; Fomin, F. V.; Koster, A. M. C. A.; Kratsch, D.; and Thilikos, D. M. 2006. On Exact Algorithms for Treewidth. In Azar, Y.; and Erlebach, T., eds., *Algorithms – ESA 2006*, Lecture Notes in Computer Science, 672–683. Berlin, Heidelberg: Springer. ISBN 978-3-540-38876-0.

Croxton, K. L.; Gendron, B.; and Magnanti, T. L. 2003. A Comparison of Mixed-Integer Programming Models for Nonconvex Piecewise Linear Cost Minimization Problems. *Management Science*, 49(9): 1268–1273.

Cygan, M.; Fomin, F. V.; Kowalik, Ł.; Lokshtanov, D.; Marx, D.; Pilipczuk, M.; Pilipczuk, M.; and Saurabh, S. 2015. *Parameterized Algorithms*. Springer International Publishing. ISBN 978-3-319-21274-6.

Dijkstra, E. W. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische mathematik*, 1: 269–271.

Dinur, I.; and Safra, S. 2005. On the Hardness of Approximating Minimum Vertex Cover. *Annals of Mathematics*, 162(1): 439–485.

Dunagan, J.; Zheng, A. X.; and Simon, D. R. 2009. Heat-Ray: Combating Identity Snowball Attacks Using Machine-learning, Combinatorial Optimization and Attack Graphs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09*, 305. Big Sky, Montana, USA: ACM Press. ISBN 978-1-60558-752-3.

Durkota, K.; Lisý, V.; Bošanský, B.; Kiekintveld, C.; and Pěchouček, M. 2019. Hardening Networks against Strategic Attackers Using Attack Graph Games. *Computers & Security*, 87: 101578.

Dvořák, P.; and Knop, D. 2018. Parameterized Complexity of Length-bounded Cuts and Multicuts. *Algorithmica*, 80(12): 3597–3617.

Enterprise Management Associates 2021. The Rise of Active Directory Exploits: Is It Time to Sound the Alarm? Technical report *https://www.enterprisemanagement.com/*.

Goel, D.; Ward-Graham, M. H.; Neumann, A.; Neumann, F.; Nguyen, H.; and Guo, M. 2022. Defending Active Directory by Combining Neural Network based Dynamic Program and Evolutionary Diversity Optimisation. In *GECCO '22: Genetic and Evolutionary Computation Conference, 2022*.

Golovach, P. A.; and Thilikos, D. M. 2011. Paths of Bounded Length and Their Cuts: Parameterized Complexity and Algorithms. *Discrete Optimization*, 8(1): 72–86.

Guo, M.; Li, J.; Neumann, A.; Neumann, F.; and Nguyen, H. 2022. Practical Fixed-Parameter Algorithms for Defending Active Directory Style Attack Graphs. In *AAAI 2022*.

Jain, M.; and Korzhyk, D. 2011. A Double Oracle Algorithm for Zero-Sum Security Games on Graphs. In *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, May 2-6, 2011, Volume 1-3*, 327–334.

Lallie, H. S.; Debattista, K.; and Bal, J. 2020. A Review of Attack Graph and Attack Tree Visual Syntax in Cyber Security. *Computer Science Review*, 35: 100219.

Milani, S.; Shen, W.; Chan, K. S.; Venkatesan, S.; Leslie, N. O.; Kamhoua, C.; and Fang, F. 2020. Harnessing the Power of Deception in Attack Graph-Based Security Games. In Zhu, Q.; Baras, J. S.; Poovendran, R.; and Chen, J., eds., *Decision and Game Theory for Security*, 147–167. Cham: Springer International Publishing. ISBN 978-3-030-64793-3.

Milgram, S. 1967. The Small-World Problem. *Psychology Today*, 1: 61–67.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347.