Practical Fixed-Parameter Algorithms for Defending Active Directory Style Attack Graphs

Mingyu Guo, Jialiang Li, Aneta Neumann, Frank Neumann, Hung Nguyen

School of Computer Science
University of Adelaide, Australia
{mingyu.guo, j.li, aneta.neumann, frank.neumann, hung.nguyen}@adelaide.edu.au

Abstract

Active Directory is the default security management system for Windows domain networks. We study the shortest path edge interdiction problem for defending Active Directory style attack graphs. The problem is formulated as a Stackelberg game between one defender and one attacker. The attack graph contains one destination node and multiple entry nodes. The attacker's entry node is chosen by nature. The defender chooses to block a set of edges limited by his budget. The attacker then picks the shortest unblocked attack path. The defender aims to maximize the expected shortest path length for the attacker, where the expectation is taken over entry nodes.

We observe that practical Active Directory attack graphs have small maximum attack path lengths and are structurally close to trees. We first show that even if the maximum attack path length is a constant, the problem is still $W[1]$-hard with respect to the defender's budget. Having a small maximum attack path length and a small budget is not enough to design fixed-parameter algorithms. If we further assume that the number of entry nodes is small, then we derive a fixed-parameter tractable algorithm.

We then propose two other fixed-parameter algorithms by exploiting the tree-like features. One is based on tree decomposition and requires a small tree width. The other assumes a small number of splitting nodes (nodes with multiple outgoing edges). Finally, the last algorithm is converted into a graph convolutional neural network based heuristic, which scales to larger graphs with more splitting nodes.

Introduction

Cyber attack graphs model the chain of events (conceptual or physical) that lead to successful cyber attacks. Despite its popularity in both academia and industry, there is not a *canonical* definition of cyber attack graphs. Lallie, Debattista, and Bal (2020) surveyed over 180 attack graphs/trees studied in literature, and discovered over 90 different *self-nominated* definitions of attack graphs/trees.

For industry practitioners, there is one attack graph model that stands out and finds its place in many practitioners' toolkit, which is the Active Directory attack graph. Microsoft Active Directory is the *default* security management system for Windows domain networks, which has a dominant market share among large organisations worldwide. Due to its popularity, Active Directory has been a focused cyber attack target. An Active Directory environment naturally describes an attack graph, where the nodes represent accounts, computers, security groups, etc. A directed edge from node A to B represents that an attacker can reach from A to B via existing accesses or known exploits. There are a number of software tools for analysing/visualising Active Directory attack graphs. Among these tools, BLOODHOUND[1] is the most influential. Motivated by (Dunagan, Zheng, and Simon 2009), BLOODHOUND models the *identity snowball attack* under Active Directory. Typically, an identity snowball attack starts when an attacker gains initial access to the internal network. The attacker starts from a low-privilege user account (often obtained via phishing emails). The attacker then moves from low-privilege nodes to high-privilege nodes (*i.e.*, account A $\xrightarrow[\text{AdminTo}]{\text{admin access}}$ computer B $\xrightarrow[\text{HasSession}]{\text{scan memory}}$ account C). The goal of the attacker is to reach the highest-privilege account, called the *Domain Admin* DA. The core functionality of BLOODHOUND is to automatically generate the *shortest attack path* from the attacker's entry node to DA, where the distance is defined as the number of *hops*. Following the shortest attack path implies less time spent on the attack and less chance of failure. Before the invention of BLOODHOUND, attackers used to explore aimlessly in the internal network hoping to discover a privilege escalation pathway. Dunagan, Zheng, and Simon (2009) briefly described a heuristic edge blocking algorithm. The aim is to block a small number of edges to cut the attack graph into multiple disconnected regions, which essentially removes the attack paths from most entry nodes.

We derive **optimal** edge blocking policies. The defender can assign different utilities on attack paths of different lengths. That is, maximizing the number of attack paths cut is a special case of our model. We adopt a two-player *Bayesian Stackelberg game* (Paruchuri et al. 2008) setup with *pure strategies only*. In our game, the defender (leader) has a limited **budget** b for blocking edges. Not all edges are blockable. The attacker's type is characterized by his entry node. In practise, the entry account is often from a phishing attack victim. Therefore, we assume that the attacker's entry node is drawn randomly by nature from a set of entry

[1] https://github.com/BloodHoundAD/BloodHound

nodes (*i.e.*, users whose emails are listed on the organisation's website). The attacker is the follower in the Stackelberg game. That is, the attacker is aware of which edges have been blocked.[2] We assume that the attacker follows BLOODHOUND's advice and attacks via a shortest attack path. The defender aims to maximize the attacker's expected shortest path length, where expectation is taken over all entry nodes.

When there is only one entry node, our model reduces to the well-studied shortest path *edge interdiction* problem (also called the *most vital edges* problem). Bar-noy, Khuller, and Schieber (1995) already showed that the problem is NP-hard. Nevertheless, this negative result does not rule out scalable algorithms for *practical* Active Directory attack graphs. We adopt *parameterized complexity analysis* and design *fixed-parameter tractable (FPT)* algorithms. Fixed-parameter algorithms allow us to solve some NP-hard problem **instances** *efficiently* and *optimally*, under the assumption that the problem instances are characterized by a few parameters that are small. For example, vertex cover is known to be NP-complete, but the question "whether there exists a vertex cover of size k" can be solved in $O(1.2738^k + kn)$, where n is the number of nodes (Chen, Kanj, and Xia 2006). That is, if one's goal is to solve vertex cover, and in one's practical scenario k is never too large, then the NP-completeness of vertex cover is irrelevant. Formally speaking, given problem instances with c parameters $k_1, k_2, \ldots, k_c$, the computational problem is fixed-parameter tractable if we can design an algorithm with complexity $O(f(k_1, \ldots, k_c)poly(n))$. That is, the complexity is allowed to be exponential in the parameters, but it needs to be polynomial in the input size n. This allows us to scale to large input sizes, as long as the parameters are small.

A natural question to ask is what should be the appropriate parameters for describing Active Directory attack graph instances. To answer this, let us consider the following attack graph. Figure 1 is a synthetic attack graph generated using BLOODHOUND team's synthetic database generator DBCREATOR.[3] We make two observations. First, the maximum attack path length is small. Second, the graph looks like a tree. Our observation is not an artifact based upon DBCreator. It is considered a best practise for Active Directory to follow the organisation chart. For example, marketing and human resources tend to form two separated tree branches. The graph will not be exactly a tree because some computers from human resources may need to access data from marketing. That is, an Active Directory attack graph can be thought of as a *tree with additional security exceptions* (non-tree edges connecting tree nodes to non-parents, also called the *feedback edges*). The maximum attack path length tends to be small for even the largest organisations. *I.e.*, it most likely only takes a few hops to go from an intern's account to the CEO's account. This is similar to the "six degree of separation" idea: all people are six or fewer social connections from each other (Milgram 1967).

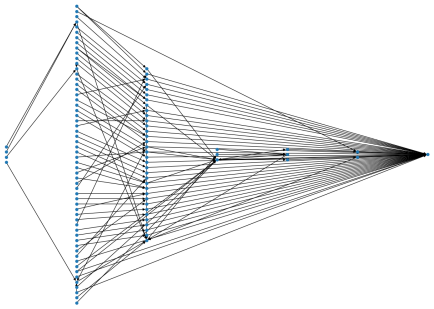

Figure 1: Synthetic attack graph generated using DBCREATOR (500 computers). Only nodes reachable to DA (rightmost node) are shown.

We use n and m to denote the number of nodes and the number of edges. We adopt the following list of parameters. In experiments, we show that some of these parameters stay tiny and our largest experiment involves graphs with 5997 nodes (among which 2000 are computers). That is, our algorithms scale well and are able to handle large organisations.

l	maximum attack path length
w	*tree width*
h	number of *feedback edges*
b	defensive budget
s	number of entry nodes

Tree width is a standard measure in graph theory for describing how close a graph is to a tree. The optimal tree width is NP-hard to compute (Arnborg, Corneil, and Proskurowski 1987). In this paper, tree width refers to the achieved tree width using our own heuristic. A connected graph can be interpreted as a tree with h additional feedback edges. Note that $h = m - (n - 1)$ as a tree has $n - 1$ edges.

We first show that having a small l does not make the problem easy. We prove that our model is $W[1]$-hard with respect to b even if l is a constant. We then propose three different fixed-parameter tractable algorithms. Our algorithms' complexities are summarized in the table below. It should be noted that these complexities only describe the *worst-case* running time. When it comes to specific problem instances, the running time often is significantly faster than what the table suggests. We do not use m in our complexity notation because we assume our graphs are similar to trees.

BUDGETFPT	$O\left(l^b\binom{b+s-1}{b}n\right)$	requires small l, b, s
DP	$O((l+2)^{w+1}b^2n)$	requires small l, w acyclic graphs only
SPLITFPT	$O(3^h(h^2 + hl + bn))$	requires small h

Finally, we convert SPLITFPT into a graph convolutional neural network based heuristic. SPLITFPT enumerates which route the attacker would choose at splitting nodes (nodes with multiple out-going edges) when facing the *optimal* defence. When h is too large, we cannot afford to enumerate all scenarios. Instead, we use a graph convolutional

[2] In practise, the attacker can use a tool called SHARPHOUND to scan the environment to obtain information on all edges.

[3] It should be noted that an organisation's Active Directory attack graph is considered sensitive information. Our paper only references synthetic graphs generated using DBCREATOR. DBCREATOR generates a lot of details, such as a node's operating system, department, real names. We only extract the topology.

neural network (GCN) to estimate the attacker's decision. Our neural network is not trained based on real attacker data. Instead, our network is purely used as an optimisation tool.

In summary, we propose 3 *fixed-parameter algorithms that scale in practise and a GCN-based approach. These will help IT admins to identify high-risk edges (accesses/-exploits) in practical Active Directory environments.*

Related Works

Bazgan et al. (2019) studied single-source single-destination shortest path edge interdiction. The authors studied a long range of parameters. Unfortunately, most of the parameters are irrelevant to practical Active Directory attack graphs (*i.e.*, distance to clique). Nevertheless, one of the parameters considered is the number of feedback edges, which is also used by our SPLITFPT algorithm. The authors proposed a kernelization technique that converts an arbitrary graph into a graph with $6h$ edges, and then exhaustively search over all 2^{6h} combinations of edges.

Another similar but different problem is the bounded length cut problem, which studies how to block b edges in order to ensure that the shortest path is greater than a parameter l'. Golovach and Thilikos (2011) proposed an elegant FPT algorithm for solving the single-source single-destination bounded length cut problem. Our BUDGETFPT builds upon the core idea behind the authors' algorithm. The authors showed that if the maximum path length and the budget are both small, then the problem is fixed-parameter tractable. We prove that this is not the case for our model. Dvořák and Knop (2018) also studied bounded length cut. The authors proposed a FPT algorithm in tree width w and maximum cut length l' with complexity $O(l'^{12w^2}n)$ for the single-source single-destination model.

There have been existing works on Stackelberg games on attack graphs (Aziz et al. 2018; Aziz, Gaspers, and Najeebullah 2017; Durkota et al. 2019; Milani et al. 2020). Besides models being different, all the above only discuss tiny attack graphs with at most 50 nodes. Our setting and approaches are different and we deal with realistic Active Directory attack graphs with thousands of nodes. Wang et al. (2019) applied graph neural networks to network interdiction games. The authors trained GCN using real attacker data to simulate Markov chain based boundedly rational attackers. We use graph neural networks purely as an optimization tool. Graph neural networks have been shown to be effective for combinatorial graph problems (Dai et al. 2017).

Model Description

An Active Directory attack graph is denoted by $G(V, E)$. There are $n = |V|$ nodes and $m = |E|$ directed edges. There is one attacker who enters the graph via one of the *entry nodes*. Let s be the total number of entry nodes. We assume that the entry node is selected by nature. For simplicity, we assume uniform chances for all entry nodes. The attacker's goal is to reach a single *destination* node called Domain Admin or DA. We consider only one destination.[4] There is a set of edges that are *blockable*, denoted by $E_b \subseteq E$. Figure 2 is an example graph. The entry nodes are 10, 11, and 12. Node 0 is the destination. The thin edges are blockable (only $1 \to 0$ and $2 \to 0$ are not blockable).

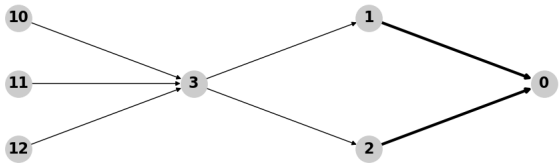

Figure 2: Example attack graph. DA is the rightmost node.

The defender selects the best b edges to block, where b is the defensive budget. We use B to denote the set of edges blocked. We consider only pure actions. That is, we do not study mixed-strategy probabilistic blocking. The graph after blocking is denoted as $G - B$. We assume the attacker can observe the defensive action and then perform a best-response attack. In the context of Active Directory attack graphs, the attacker performs the *shortest path attack* on $G - B$. We use $SP(s_i, G - B)$ to denote the shortest path from entry node s_i to the destination, on graph $G - B$.

We assume that an attack path with x hops has a success probability of $f(x)$, where f could be any decreasing function in x. If no attack paths exist, then $f(+\infty) = 0$. f can essentially be any performance evaluation function. For presentation purposes, we call it the "success rate", and we set $f(x) = 0.95^x$ (*i.e.*, every action has a 5% failure rate).

The defender's task is to pick the best b edges to block, in order to minimize the attacker's expected success rate. Expectation is taken by averaging over all s entry nodes. Formally, our model is the following:

$$\min_{B \subseteq E_b, |B| \leq b} \quad \frac{1}{s} \sum_{i=1}^{s} f(|SP(s_i, G - B)|)$$

A baseline algorithm is the greedy algorithm (GREEDY). We present it here to serve as an example to help readers understand our model. The greedy algorithm will also be used as an algorithm component in later sections.

Algorithm 1 (GREEDY)**.** Given budget b, we pick the single best edge to block in terms of reducing the attacker's expected success rate.[5] We repeat b times.

GREEDY is actually the optimal algorithm if our graph is exactly a tree. This is straightforward to see. Blocking actions happening in different tree branches are independent from each other. Given edge e_1 and e_2, if the path from e_1 to DA passes through e_2, then GREEDY will always block e_2.

On the other hand, GREEDY works poorly if there are *substitutable block-worthy* edges. For Figure 2, if our budget is 2, under GREEDY, the edges blocked are any two of $\{10 \to 3, 11 \to 3, 12 \to 3\}$. Neither $3 \to 1$ nor $3 \to 2$ is blocked. There is still one entry node that can reach DA.

[4] In real-life Active Directory attack graphs, there are often multiple admin nodes. We simply merge all admin nodes into a single node and call it our DA.

[5] For example, we could use the almost linear algorithm proposed in (Nardelli, Proietti, and Widmayer 2001) for each step.

The attacker's success rate is $\frac{1}{3}f(3)$. It is easy to see that the optimal defence should be blocking both $3 \to 1$ and $3 \to 2$.

$W[1]$ Hardness and BUDGETFPT

We mentioned that Active Directory attack graphs tend to have short attack paths. We first prove that having a small maximum attack path length alone is not enough to derive efficient algorithms.

Theorem 1. *Let b be the budget. Let l be the maximum attack path length. Our problem is $W[1]$-hard with respect to b even for constant l.*

Theorem 1 implies that having a small maximum attack path length and a small budget is not enough to derive fixed-parameter algorithms, which makes it different from the single source single destination bounded length cut problem studied in (Golovach and Thilikos 2011). Nevertheless, the authors' core idea still applies to our setting. The core idea goes like this: For single source and single destination, we pick an arbitrary shortest path. We must block somewhere along this path. Otherwise, it is as if no blocking happens. Our fixed-parameter algorithm builds on the above idea.

Algorithm 2 (BUDGETFPT)**.** We go over all combinations to find the blocking setup that minimizes the expected success rate for the attacker. We pick an arbitrary entry node s_1 and pick an arbitrary shortest path from s_1 to DA. We either block at least one edge along this shortest path, or s_1 can be removed from our consideration (as its shortest path is unaffected). There are at most l options for picking an edge to block. After blocking an edge, the budget is reduced by 1. There is an option for ignoring s_1. After ignoring s_1, the number of entry nodes is reduced by 1.

We use $c_{b,s}$ to denote the number of combinations we need to go over when the remaining budget is b and the remaining number of entry nodes is s. We have the following recursive relationship and the base cases:

$$c_{b,s} = lc_{b-1,s} + c_{b,s-1}; \quad c_{0,s} = 1; \quad c_{b,1} = l^b$$

For each combination, the success rate evaluation takes linear time. We derive the analytical form of $c_{b,s}$, which gives the following complexity result.

Proposition 1. BUDGETFPT *has a complexity of*

$$O\left(l^b \binom{b+s-1}{b} n\right)$$

Tree Decomposition based Dynamic Program

In this section, we propose a dynamic programming algorithm based on tree decomposition. This algorithm scales the best in experiments, but one restriction is that it requires the attack graph be acyclic. The synthetic attack graph in Figure 1 does contain cycles. However, all cycles involve one common user. So if we remove that one user, the graph becomes acyclic. In other graphs generated using DBCREATOR, we always only need to remove a few nodes to make the graph acyclic. We argue that the network admin has the flexibility to convert existing graphs into acyclic with minimum changes. Our algorithm can also serve as a heuristic.

The *tree width* of a graph is a commonly used index to measure how close a graph is to a tree. Many computationally difficult problems on general graphs are easy if the tree width is small. This is also the case for our problem.

Our attack graphs are directed and all nodes have paths to DA (nodes that cannot reach DA are ignored). We further assume that the attack graphs are acyclic in this section. Under the above assumptions, we could use topological sorting to divide our attack graphs into several *security levels*. DA belongs to the highest security level. An edge only goes from a lower security level node to a *strictly* higher security level node. Take Figure 2 as an example, we could interpret it as that there are four security levels. 10 to 12 are in the lowest security level while DA (node 0) has the highest security level. With this security-level interpretation, there is no ambiguity on the direction of an edge. If an edge involves node a and b, then the direction must be from a to b if a has a lower security level. We will directly reuse the tree decomposition and tree width definition for undirected graphs.

As an example, Figure 3 shows a tree decomposition of the graph in Figure 2. Every tree node contains a subset of the original graph vertices. The union of all tree nodes contains all graph vertices. Every edge is covered by a tree node. For example, tree node $(3, 1, 2)$ covers all out-going edges of graph vertex 3. Finally, for every graph vertex, the tree nodes containing it form a subtree. For example, the tree nodes containing 3 are $(3, 1, 2)$, $(10, 3)$, $(11, 3)$ and $(12, 3)$.

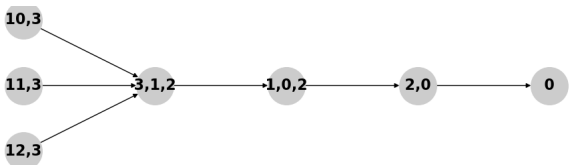

Figure 3: Tree decomposition of Figure 2 using our heuristic

Tree decompositions are not unique. The optimal tree decomposition that minimizes the tree width is NP-hard to compute (Arnborg, Corneil, and Proskurowski 1987). For our dynamic program, we require a specific style of tree decompositions that satisfy extra properties. We call such tree decompositions *desired* tree decompositions. Figure 3 is a desired tree decomposition. We first explain how we can convert a desired tree decomposition into a dynamic program via an example. During the process, we will explain what it means for a tree decomposition to be desired. Finally, we propose a heuristic that guarantees to generate a desired tree decomposition.

In Figure 3, the number of tree nodes is the same as the number of graph vertices. The first coordinate values are unique. A dynamic program subproblem involves a tree node and a remaining budget b'. For example, $((3, 1, 2), b')$ corresponds to the following subproblem: given the remaining budget b' and the distances (to DA) for 1 and 2, what should be the distance between 3 and DA? To answer this question, we essentially need to determine the budget investment on 3 (budget spent on 3 is used to block 3's out-going edges). 3 has two successors (1 and 2). Both successors are blockable, so we could spend 0 to 2 units of budget on 3. If

x units of budget is spent on a node (to block its out-going edges), it will always be blocking the shortest x paths. Once the budget investment on 3 is determined, 3's final distance is also determined. To summarize, $((x, y_1, y_2, \ldots), b')$ is the subproblem where the defender needs to decide how many units of budget (at most b') are invested on x, considering the *context information* on x's potential successors (the y_i). Here, context information refers to the distances to DA.

Let us run through a solution on the basis of Figure 3. Let us recall that the actual graph is in Figure 2. We still assume the total budget is 2. We start from $((0), 2)$. The budget investment on 0 will be 0 since its distance to itself is always 0. This information is then passed down to $((2, 0), 2)$. 2 has no blockable out-going edges, so we cannot spend any budget anyway. The distance (to DA) for 2 is then 1. This information is passed down to $((1, 0, 2), 2)$. Again, we cannot invest any budget because 1's out-going edge is not blockable. We get that 1's distance is 1. This context is passed down to $((3, 1, 2), 2)$. At this point, we are deciding how many units of budget to spend on 3. Based on the context information, we know that 1 and 2's distances are both 1. The best decision here is to invest 2 units of budget. After that, 3's distance becomes infinity. This information is then passed down to all three leaf nodes. For example, one subproblem at the leaf level is $((10, 3), 0)$. The context information says that 10's successor 3's distance to DA is infinity, and there is no budget left. So we get 10's distance to DA is also infinity. Since 10 is one of three entry nodes, we add $\frac{1}{3}f(+\infty) = 0$ to the expected success rate. Essentially, the original problem is $((0), b)$. Our dynamic program makes decisions on the budgets spent on each node. Context information and remaining budgets are passed down to future subproblems. Through out the process, one property we need is that as we move from the root to the leaves, when a vertex first enters into our consideration (*i.e.*, vertex 3 enters at tree node $(3, 1, 2)$), the context information (*i.e.*, 1 and 2's distances) must be enough for us to decide the new vertex' distance right away. That is, for any vertex i, we consider the subtree containing i. The root of this subtree must contain all out-going edges of i. Also, the root of the whole tree should be a node containing DA only. The above properties are not held by all tree decompositions. In this paper, we use the vertex elimination technique to generate tree decomposition (Bodlaender et al. 2006). This technique maps an arbitrary permutation of the vertices into a tree decomposition. It should be noted that the vertex elimination technique is only a heuristic framework because it is NP-hard to figure out the best permutation that results in the smallest tree width. Our heuristic is simple: eliminate vertices based on the security level ranking, with the lowest security level eliminated first. We of course cannot guarantee that the generated tree width is the smallest, but it is provable that it generates a tree decomposition that is desired.

Lastly, actually our example in Figure 3 does not capture one complication, which is that at node $(3, 1, 2)$, we also need to divide the remaining budget among three child branches. We need to divide it four ways (one for $(3, 1, 2)$ itself, for blocking 3's out-going edges; and three for three child branches). This has a complexity of b^4. If there are too many branches, then the decision process gets too expensive. We resolve this using the *nice tree decomposition* idea from (Cygan et al. 2015). The main idea is that we can always clone a node into two. For example, we can insert another $(3, 1, 2)$ in between $(3, 1, 2)$ and $(1, 0, 2)$. The copy closer to DA only makes the decision on 3 itself (how many units of budget to spend on 3). The copy further away from DA deals with the splitting problem for splitting the remaining budget for three children. However, this node still needs to split three ways. The nice tree decomposition idea can resolve this with ease as well. We simply need to add another clone of $(3, 1, 2)$ to handle the second round of splitting. One node splits between $(10, 3)$ and the rest $\{(11, 3), (12, 3)\}$. The second node splits between $(11, 3)$ and $(12, 3)$. For our setting, the number of clone nodes needed is at most $2n$, which does not affect the complexity. The gain is that for every node, the decision is always one dimensional, so the number of decisions at a node is at most $b + 1$.

Proposition 2. DP *has a complexity of*

$$O((l + 2)^{w+1} b^2 n)$$

w *is the* achieved *tree width using our desired tree decomposition heuristic.*

Classification-based SPLITFPT

Another parameter to describe a graph's tree-likeness is the number of feedback edges h. A related parameter is the number of splitting nodes t (nodes with multiple out-going edges). If the maximum out degree is d, then we have $h \leq t(d-1)$. For the synthetic graph in Figure 1, the number of splitting nodes is only 12, and the maximum out degree d is only 3. Having small t and d essentially means that the attacker's strategy space is tiny. If the attacker starts from a non-splitting node, then the attacker has no choice but to move on to the only successor, and keeps moving on until 1) the attacker is facing a blocked edge, which means the attacker's entry node cannot reach DA; 2) the attacker has reached DA; or finally 3) the attacker has reached another splitting node. A path where every intermediate node has only one out-going edge is called a *simple path*. At a splitting node, an attacker faces at most d simple paths (one for each successor). Every simple path leads to either another splitting node or to DA. Essentially, the attacker's strategy is characterized by his route choices at the splitting nodes. At each splitting node, there are at most $d+1$ options, including at most d simple paths to choose from, and not choosing at all, which happens if none of the simple paths leads to DA. The attacker's strategy space has a size of $(d + 1)^t$.

Typically for Stackelberg games, we optimize over the defender's strategy space. For each defender's strategy, we figure out the attacker's best-response, and then check how good this best-response attack is in terms of the defender's utility. In this paper, because the attacker's strategy space is tiny, we instead *optimize over the attacker's strategy space*. We **guess** what the attacker would do when facing the **optimal** defence. That is, we guess the attacker's route choices at the splitting nodes. When both t and d are tiny, we can simply go over all strategies of the attacker. If we go over all

combinations, then at least one is indeed the best response to the optimal defence. Given a guessed attacker's strategy, we can derive the distances (to DA) from all splitting nodes. For example, given split node t_1, we know what route the attacker would choose at this node, which has to be the shortest unblocked route, since we assume it is a best response. We can follow along this simple path to another splitting node (or to DA). At the next splitting node, we also know what route the attacker would choose. This way we can generate the shortest paths from all splitting nodes to DA, except for splitting nodes at which the attacker chooses not to take any route (the distances are set to infinity for these nodes). For all generated shortest paths, we mark all blockable edges covered by them as *not blockable*. The defender cannot block these edges, otherwise it is contradictory to our assumption that the attacker's strategy is a best response. Then, at each splitting node, there are routes not taken by the attacker (either all but one are not taken, or simply all are not taken). Let us consider a splitting node a, and one of its successors b, where route $a \to b$ is not taken. We follow the simple path starts from $a \to b$, and continue on until we either reach DA or another splitting node. We always end up with a node whose distance to DA is known. We calculate whether the simple path under discussion is a better choice for the attacker or not. If not, we do not need to do anything. If it is a better choice for the attacker, then we must block this simple path. Again, otherwise it is contradictory to our assumption that the attacker's strategy is the best response to the optimal defence. For a simple path, it is without loss of generality to block the blockable edge that is closest to DA. There is no reason to block further away edges or multiple edges. After the above actions (marking some edges as not blockable and blocking some edges), we end up with an attack graph that is exactly a tree (only one rational decision at any splitting node). When the attack graph is a tree, we simply run GREEDY using the remaining budget.

Proposition 3. SPLITFPT *has a complexity of*

$$O(3^h(h^2 + hl + bn))$$

Graph Convolutional Neural Networks

SPLITFPT exhaustively goes over the attacker's $d + 1$ options at all t splitting nodes. This is exactly a *node classification* problem. We use graph convolutional neural network to perform node classification so that it scales to larger graphs.

We perform unsupervised learning as follows. A node classification neural network takes one splitting node as input, and outputs its classification (*i.e.*, which route the attacker would take when facing the optimal defence; the attacker may take no routes, which is just another classification category). Given a classification on the splitting nodes, we can evaluate the corresponding defence that would induce this classification using the same procedure in SPLITFPT. That is, we can map a node classification to an expected success rate for the attacker.

A short description of our neural network approach is that, given the current neural network (the current node classification rule), we randomly flip some nodes' classifications to check whether the perturbed classification leads to better result (worse success rate for the attacker). If so, we instruct the neural network to learn toward the better classification (treat it as *true labels*). We obviously do not have the computational resources to exhaustively go over all node classification combinations (otherwise we should simply run SPLITFPT). The goal is to get a small number of nodes' classifications correct, and hope the neural network is able to pick up the underlying rules, and can help generalize to produce correct classifications on all nodes.

Below are the details of our neural network.

Node and edge features: Because the original graph is too large to handle, we construct a *condensed graph* containing only the splitting nodes. The splitting nodes are connected via simple paths in the original graph. In the condensed graph, a simple path is interpreted as a single edge. An example node feature is its *out degree*. An example edge feature is *the length of the corresponding simple path*. Due to space constraint, we omit our list of manually derived features (7 node features, 6 edge features).

Network structure: Both the node features and the edge features are expanded to 64 dimensions using linear encoders. The features then go through 10 layers of *crystal graph convolutional layer* with MAX as the aggregator and batch normalization turned on (Xie and Grossman 2018). Between each layer, we have a dropout layer that drops an edge with 0.1 probability. The last layer is a linear layer that converts the output to $d + 1$ dimensions. Given an output, the coordinate with the highest value is taken as the neural network's classification for the input splitting node.

Training: We use a batch size of 16 (splitting nodes). For splitting nodes not in the batch, their classifications follow the current network's decision. For a node in the batch, with 0.9 probability, we follow the current network's classification, but we do not just pick the coordinate with the highest value. Instead, we use SOFTMAX to get each dimension's probability and we draw a classification accordingly. With 0.1 probability, we disregard the network and draw a classification uniformly at random. Essentially, we slightly perturb the in-batch nodes' classifications. We then go over the in-batch nodes one by one based on a random permutation order. For each node, we exhaustively go over all $d+1$ classifications and check whether unilateral change leads to a better-performing classification. We end up with a new classification. With 0.5 probability, we use this new classification as true labels. With the other 0.5 chance, we use the historically best-performing classification as true labels. We stop after 50 epochs. Loss is based on cross entropy and the optimizer is Adam with a learning rate of 0.01. We also always train 5 times with random seed 0 to 4.

Experiments

Our hardware specs are I7-6700 3.4GHZ and double TURBO-GTX1080-8G GPUs. We conduct all experiments using a synthetic Active Directory attack graph generated by DBCREATOR. We set the number of computers to 2000. We only consider three types of edges: ADMINTO, MEMBEROF, HASSESSION. These three edge types are the only three *default* edge types in BLOODHOUND. The final

	Success Rate	Time[s]	#Opt
BUDGETFPT	0.308	5.517	
GCN	0.324	34.166	9
GREEDY	0.483	0.117	1

Table 1: R2000 with $b = 10$ and $s = 5$

	Success Rate	Time[s]	#Opt
DP	0.449	0.007	
GCN	0.449	2.340	10
GREEDY	0.449	0.017	10

Table 2: R2000-DAG with $b = 10$ and $s = 5$

graph contains 5997 nodes (computers + user accounts + security groups, etc.) and 18795 edges.[6] We call this graph R2000. We can make R2000 acyclic by removing another 4 nodes from it. We call the acyclic version R2000-DAG.

We randomly set some edges to be blockable. For a non-splitting node, its out-going edge is blockable with p_b probability. For a splitting node, with p_b probability, all of its out-going edges are blockable. We set p_b to be 0.2. All of our experiments are repeated 10 times.

Table 1: Low budget setting. We randomly pick 5 nodes to be the entry nodes. We set the budget to 10. With a small budget and a small number of entry nodes, BUDGETFPT is scalable. We compare BUDGETFPT against GREEDY and GCN (the graph convolutional neural network approach). Note that BUDGETFPT is optimal. Success Rate is short for the expected success rate for the attacker (lower is better). Time[s] is seconds per trial. #Opt shows how many times the algorithm under discussion produces a result that is within 0.000001 of the optimal result (among 10 trials).

Table 2: Acyclic setting. We use R2000-DAG instead. We compare DP against GREEDY and GCN. It turns out that both GCN and GREEDY always achieve the optimal results. GREEDY being optimal is not entirely surprising, because we have shown that GREEDY is optimal if the graph is exactly a tree, and R2000-DAG is very close to a tree.

Table 3: Acyclic setting with substitutable edges. We could artificially modify R2000-DAG and create an acyclic graph that is not close to a tree. We recall that GREEDY fails when there are substitutable blockable-worthy edges, as shown in Figure 2. We introduce substitutable edges into R2000-DAG. Given a blockable edge $a \to b$, we duplicate b. We create two paths $a \xrightarrow{blockable} b \to c$ and $a \xrightarrow{blockable} b' \to c$, where c is a successor of b. Essentially, we are recreating the structure of Figure 2. This setup is actually simulating a practical scenario. If there are two different exploits that allow the attacker to travel from a to b, then we need to block both exploits. Under this artificially created acyclic graph, GCN is still always getting the optimal result, and GREEDY fails 4 out of 10 trials.

[6]There are 7 admin nodes. We merge them into one node DA. Out of 5997 nodes, only 339 can reach DA. So we can preprocess our graphs to make it much smaller.

	Success Rate	Time[s]	#Opt
DP	0.465	0.008	
GCN	0.465	8.000	10
GREEDY	0.566	0.030	6

Table 3: R2000-DAG with additional substitutable edges, $b = 10$ and $s = 5$

	Success Rate	Time[s]	#Win
GCN ($p_b = 0.2$)	0.450	37.987	1
GREEDY ($p_b = 0.2$)	0.438	0.245	3
GCN ($p_b = 0.4$)	0.226	24.121	5
GREEDY ($p_b = 0.4$)	0.287	0.600	0

Table 4: R2000 with $b = 20$ and $s = 10$

Table 4: Setting without optimal solutions. We go back to R2000. We double the budget to 20 and double the number of entry nodes to 10. We can no longer afford to run BUDGETFPT. We also cannot run DP because R2000 contains cycles. In this setting, we do not have optimal solutions. We compare GREEDY against GCN. #Win refers to how many times an algorithm beats the other one (ties are not counted as wins). It turns out that GREEDY performs better than GCN in this setting. In R2000, there are naturally occurring substitutable edges. With double budget, GREEDY will block the substitutable edges despite they are perceived as low in priority (*i.e.* no place to spend the budget, so might as well spend on substitutable edges). If we double p_b, then GREEDY again has plenty of places to spend budget on, so it will not spend budget on substitutable edges. As a result, GCN constantly wins again.

Conclusion

We studied edge blocking for defending Active Directory style attack graphs. We proposed 3 fixed-parameter algorithms based on the observation that practical Active Directory attack graphs have small maximum attack path lengths and are similar to trees. BUDGETFPT can be applied when both the budget and the number of entry nodes are small. DP scales the best experimentally, but it is only applicable to acyclic graphs. SPLITFPT is based on performing fixed-parameter analysis on the attacker's strategy space, which happens to be much smaller compared to the defensive strategy space in our model. For each attacking strategy, we analysed what kind of defense would make this attacking strategy a valid best response. This FPT technique is potentially useful for other Stackelberg games. Lastly, we *scaled up* SPLITFPT by converting it to a graph convolutional neural network based heuristic. A typical FPT approach is to exhaustively search among a set of solutions characterized by the fixed parameters. Instead of exhaustive search, we trained a neural network to guess the best solution. Our algorithms will help IT admins to identify high-risk edges (accesses/exploits) in practical Active Directory environments.

Acknowledgements

Frank Neumann has been supported by the Australian Research Council through grant FT200100536. Hung Nguyen is partially supported by the "Cyber NGT – Provable Network Security" grant.

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.

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

Bazgan, C.; Fluschnik, T.; Nichterlein, A.; Niedermeier, R.; and Stahlberg, M. 2019. A More Fine-Grained Complexity Analysis of Finding the Most Vital Edges for Undirected Shortest Paths. *Networks*, 73(1): 23–37.

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.

Chen, J.; Kanj, I. A.; and Xia, G. 2006. Improved Parameterized Upper Bounds for Vertex Cover. In Královič, R.; and Urzyczyn, P., eds., *Mathematical Foundations of Computer Science 2006*, Lecture Notes in Computer Science, 238–249. Berlin, Heidelberg: Springer. ISBN 978-3-540-37793-1.

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.

Dai, H.; Khalil, E. B.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning Combinatorial Optimization Algorithms over Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 6351–6361. Red Hook, NY, USA: Curran Associates Inc. ISBN 978-1-5108-6096-4.

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.

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.

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*, Lecture Notes in Computer Science, 147–167. Cham: Springer International Publishing. ISBN 978-3-030-64793-3.

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

Nardelli, E.; Proietti, G.; and Widmayer, P. 2001. A Faster Computation of the Most Vital Edge of a Shortest Path. *Information Processing Letters*, 79(2): 81–85.

Paruchuri, P.; Pearce, J. P.; Marecki, J.; Tambe, M.; Ordonez, F.; and Kraus, S. 2008. Efficient Algorithms to Solve Bayesian Stackelberg Games for Security Applications. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, 1559–1562. Chicago, Illinois: AAAI Press. ISBN 978-1-57735-368-3.

Wang, K.; Mate, A.; Wilder, B.; Perrault, A.; and Tambe, M. 2019. Using Graph Convolutional Networks to Learn Interdiction Games. In *AI for Social Good Workshop, International Joint Conference on Artificial Intelligence (IJCAI)*.

Xie, T.; and Grossman, J. C. 2018. Crystal Graph Convolutional Neural Networks for an Accurate and Interpretable Prediction of Material Properties. *Physical Review Letters*, 120(14): 145301.