SAND2021-5474C

NetProtect: Network Perturbations to Protect Nodes against Entry-Point Attack

Ricky Laishram
Syracuse University
Syracuse, NY, USA
rlaishra@syr.edu

Pegah Hozhabrierdi
Syracuse University
Syracuse, NY, USA
phozhabr@syr.edu

Jeremy D. Wendt
Sandia National Laboratories
Albuquerque, NM, USA
jdwendt@sandia.gov

Sucheta Soundarajan
Syracuse University
Syracuse, NY, USA
susounda@syr.edu

ABSTRACT

In many network applications, it may be desirable to conceal certain target nodes from detection by a *data collector*, who is using a crawling algorithm to explore a network. For example, in a computer network, the network administrator may wish to protect those computers (target nodes) with sensitive information from discovery by a hacker who has exploited vulnerable machines and entered the network. These networks are often protected by hiding the machines (nodes) from external access, and allow only fixed entry points into the system (protection against external attacks). However, in this protection scheme, once one of the entry points is breached, the safety of all internal machines is jeopardized (i.e., the external attack turns into an internal attack). In this paper, we view this problem from the perspective of the *data protector*. We propose the *Node Protection Problem*: given a network with known entry points, which edges should be removed/added so as to protect as many target nodes from the data collector as possible? A trivial way to solve this problem would be to simply disconnect either the entry points or the target nodes – but that would make the network non-functional. Accordingly, we impose certain constraints: for each node, only $(1-r)$ fraction of its edges can be removed, and the resulting network must not be disconnected. We propose two novel scoring mechanisms - the `Frequent Path Score` and the `Shortest Path Score`. Using these scores, we propose `NetProtect`, an algorithm that selects edges to be removed or added so as to best impede the progress of the data collector. We show experimentally that `NetProtect` outperforms baseline node protection algorithms across several real-world networks. In some datasets, With 1% of the edges removed by `NetProtect`, we found that the *data collector* requires up to 6 (4) times the budget compared to the next best baseline in order to discover 5 (50) nodes.

KEYWORDS

graph, network, adversary, perturbation

ACM Reference Format:
Ricky Laishram, Pegah Hozhabrierdi, Jeremy D. Wendt, and Sucheta Soundarajan. 2021. NetProtect: Network Perturbations to Protect Nodes against Entry-Point Attack. In *Proceedings of WebSci '21: International ACM Conference on Web Science (WebSci '21).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In many network applications, an agent may wish to prevent certain nodes in the network from being detected in a network crawl. For example, a computer network administrator may want to reduce the probability of hackers locating certain machines that contain confidential or sensitive information [11]. A popular protection scheme in such networks is to restrict the entry points to the system through jump servers (or jump boxes) [15]. This type of protection provides two separate security zones (external and internal) and jump servers act as intermediaries between these zones. Although these jump servers do not store sensitive data, they store credentials that allow access to the machines inside the protected zone. Accordingly, these entry points are popular targets for attackers breaching into the network [17]. In this scenario, it is of crucial importance to protect nodes that are vital to the stability of the network against the internal attackers (attackers from breached entry points) [16]. In this paper, we ask the question: *How can one best modify the network so as to preserve its functionality while also protecting target nodes from detection in a crawl-based attack?* This general problem can be viewed from two perspectives: that of the *data collector*, who wishes to crawl a network and locate so-called *target nodes*; and that of the *data protector*, who can perform small modifications to the network to lower the chances of target nodes' discovery by the *data collector*.

While the network crawling (data collection) and vulnerable nodes identification problems are well-studied [2, 5, 10–13], to the best of our knowledge, the existing literature has not considered the perspective of the *data protector* in considering the attacks on vulnerable nodes. Thus, in this paper, we assume the role of the *data protector*, and consider the following problem: *Given a network with a set of known target nodes, a set of entry points to the system, and a protection budget b_p, which b_p edges can we add or remove to most* *hinder the data collector's access to the target nodes?* Note that this problem could be trivially solved by removing all edges adjacent to the target nodes or entry points. But in practice, the *data protector* would wish to ensure the modified network is still functional as a network. We thus impose connectivity-related constraints: (1) The network must remain connected after the modification; and (2) The number of neighbors of a node that are removed/added is at most $r \cdot d(u, G)$, where $d(u, G)$ is the original degree of the node in graph G, for some specified r.

We propose two node-level scores which are intended to identify those nodes that are most important to the *data collector* in reaching its target(s): (1) the `Frequent Path Score` (`FPS`) which measures the likelihood that a node will appear in an absorbing random walk, and is thus intended to protect against random-walk-based crawlers; (2) the `Shortest Path Score` (`SPS`), which incorporates the number and length of shortest paths passing through a node with the length of those paths, and is suited for protection against expansion-type crawlers like breadth first search. Our proposed algorithm, `NetProtect`, uses these scores to make modifications to the network. We perform a variety of experiments on real-world networks with different budgets, pitted against popular network crawling algorithms. Our results show if a *data protector* removes 1% of edges based on `NetProtect`, the *data collector* must increase its budget up to 6 (4) times than the next best baseline in order to discover 5 (50) nodes respectively. (e.g., see `NetProtect-`, *Degree based Target* for `musae-facebook` in Table 1). The contributions of this paper are:

- We consider the *Node Protection Problem* of protecting target nodes in a network from being discovered in an entry-point attack. We formalize this problem with respect to the goals of the *data collector* and the *data protector*.
- We propose the `FPS` and `SPS` for nodes based on their importance in random walk and expansion-based searches originating from a set of entry points before hitting the targets. We use these scores to propose `NetProtect`, an algorithm for determining which edges to add or remove to best protect the target nodes.
- We compare the performance of `NetProtect` to a variety of baseline protection schemes on networks, including ROAM which is the state of the art node protection algorithm [18]. Our experiments show that, in comparison with baselines, the deletion or addition of edges by `NetProtect` makes it considerably harder for the *data collector* to find the target nodes.

In Section 2, we describe previous work that addresses similar problems. Next, in Section 3, we describe the problem statement in more detail, including roles of the *data collector* and *data protector*. Our proposed scoring measures are discussed in Section 4. Finally, we present our experimental evaluations on real-world networks in Section 5.

2 RELATED WORK

Our work mainly relates to two bodies of research: (1) node protection in social networks, and (2) graph crawling algorithms.

2.1 Node Protection in Social Networks

The main idea behind hiding (protecting) certain nodes in the graph is to decrease the importance of the target node through local graph manipulations. These methods do not consider entry-point attacks, nor the different crawling algorithms. The most well-known method in locally manipulating graph structure is ROAM, the algorithm proposed by [18]. ROAM (Remove One, Add Many) decreases the degree centrality of the target node by removing a neighbor and connecting it to other immediate neighbors of the target node. [1] expands the same idea to decrease the eigenvector centrality of the target node. The problem with this approach is the costly computation of the eigenvector centrality after each iteration. Moreover, although ROAM decreases the degree centrality of the target node, it increases the centrality of the final immediate neighbors of the target. A degree-based crawling algorithm can easily access this immediate neighbor which is only 1-hop away from the target. As the number of iterations increases, ROAM is also biased towards creating star-shaped subgraphs that can be detected by an anomaly-detection-based algorithm. In a different approach, [8] uses greedy edge removal to decrease the closeness centrality of a target node. Their approach is prone to transforming the target into an isolated node and is not effective in practical setting. Our proposed method preserves the connectivity of the graph (no isolated nodes created) and is not biased towards creating anomalous subgraphs. As ROAM shows better performance than eigenvector-based ROAM and does not produce isolated nodes, we benchmark our proposed algorithm against ROAM in Section 5.

These studies often focus on minimizing some centrality of the node without considering the perspective of the data collector. In our study, we define scoring schemes that consider not only the centrality of the target node, but also the centrality of the nodes that contribute the most to guiding the crawler to the target. The current methods, as explained above, do not disturb the crawler's path to the neighborhood of the target node. As such, a crawler can still find its way easily to the vicinity of the target node, and depending on the density of the target's neighborhood, they can find the target through a few trial and error steps. Our approach, on the other hand, helps us to diverge the crawler's path from the target in early stages of the crawling by manipulating the paths that lead to the target, rather than the target node itself.

2.2 Graph Crawling Algorithms

There have been numerous studies on graph crawling algorithms. Common graph crawling algorithms include classical techniques such as random walk (RW), breadth first search (BFS), depth first search (DFS), and their variants such as selective BFS/DFS [14] and Metropolis-Hasting RW [6]. We encourage the reader to refer to [3, 9, 19] for a more comprehensive review of various graph crawling algorithms and their applications. Our problem specifically deals with aggressive crawling in contrast to innocent sampling attempts, as discussed in [14]. According to this study, aggressive crawlers generally use expansion-based methods that allows them to travel as far as possible from the starting node (e.g., DFS), whereas innocent crawlers stay in the vicinity of the starting node gathering as many neighbors as possible (e.g., BFS). As expansion-based and random-walk-based methods are the core of all these methods, we choose BFS, DFS, and RW in this study.

3 PROBLEM

To formalize the *Node Protection Problem*, consider a graph $G = \langle V, E \rangle$, where V and E denote the nodes and edges respectively. In this study, we focus on undirected graphs; however, our proposed method can trivially be extended to directed graphs as well (see Section 5.1). We assume that G has limited entry points to reduce external accessibility (e.g., it uses jump servers to monitor the ingoing traffic and build a controlled security zone). As such, only $V_s \in V$ nodes have connections to outside of G and are vulnerable to external attacks. Among the internal nodes in G, there are $V_t \in V \setminus V_s$ sensitive nodes (target nodes) that are to be protected against internal attacks initiated from either of nodes in V_s. There are two agents that operate on the graph: (1) the *data collector* who seeks to observe the network and find sensitive information; and (2) the *data protector* who tries to hide nodes containing sensitive information from the *data collector*. In this paper, we address the problem from the *data protector*'s perspective: that is, *which edges from the network should the data protector add or remove to protect the target nodes?* Next, we discuss each of these agents in details.

3.1 Data Collector

Initially, the *data collector* knows the identity of the nodes in set $V_s \subset V$ and can explore (i.e., crawl) the graph G beginning from any of the nodes in V_s. To explore the graph, the *data collector* queries for the neighbors of a node it has seen, and adds the neighbors and edges found to its observed subgraph. We assume that a query accurately returns all neighbors of the queried node, and once a node is discovered by the collector, all its information is accessible (i.e., the collector knows it has found a target node). There are many algorithms a *data collector* could use to crawl the network, such as a random walk, snowball sampling, or more sophisticated methods (see Section 2.2). In most cases, there is a limit to the number of unique nodes that can be queried (due to time or budget resources). We will use $\tilde{G} = \langle \tilde{V}, \tilde{E} \rangle$ to denote the subgraph observed by the *data collector* after b_c unique nodes have been queried, where b_c is the *collector budget*. Given a collector budget b_c and an initial set entry points V_s, the task of the *data collector* is to (1) collect the subgraph $\tilde{G}$, and (2) identify the *target nodes* ($V_t \subset V \setminus V_s$) in $\tilde{V}$. The goal is to identify as many target nodes as possible. More formally, let $\pi_C(v, G, b_c, V_s)$ be the probability of node $v \in V$ being included in $\tilde{V}$ through some data collection algorithm C on graph G. When the context is clear, we will abbreviate this term with $\pi_C(v, G)$. The objective of the *data collector* is to find a crawling algorithm that maximizes this probability for nodes in the target set, i.e., $C^* = \underset{C}{\arg\max} \sum_{v \in V_t} \pi_C(v, G)$.

3.2 Data Protector

The data protector has global knowledge of the graph (e.g., as the admin of the network). In particular, it is aware of (1) external gateways to the network (V_s) where intruders can potentially access internal network, and (2) target nodes that have to be protected. Given a *protector budget* b_p, the goal of the *data protector* is to add or remove b_p edges from the network such that the performance of the *data collector* is degraded by the greatest amount possible while still respecting network connectivity-related constraints. It is worth noting that we consider a *data protector* that either deletes or adds edges, but not both. This is for two reasons: (1) Our experiments showed that edge deletion alone is far more effective than combining the two operations, and (2) The relative costs of edge addition and deletion are very application dependent, and there is no obvious relationship between the two. As such, we only include the separate edge addition and deletion analysis in our experiments (Section 5). Formally, the *data protector* wishes to find the set of edges for graph $G' = \langle V, E' \rangle$ such that $G^* = \underset{G'}{\arg\min} \sum_{v \in V_t} \pi_C(v, G')$, subject to the following constraints:

- Because adding/removing an edge may have an associated cost (for example if too many edges are added/removed, it may affect other properties of the network), the *data protector* may only add/remove a total of b_p edges ($|E' \setminus E| = b_p$ for edge addition and $|E \setminus E'| = b_p$ for edge deletion).
- There is a limit to edge modifications on a single node (otherwise there may be undesirable side effects such as isolated nodes or shifted centralities). So, we require that $\forall v \in V$, $\frac{|d(v,G') - d(v,G)|}{d(v,G)} \leq r$, where $d(v, G)$ is the degree of node v in G.
- The number of connected components in G^* must be the same as that in G.

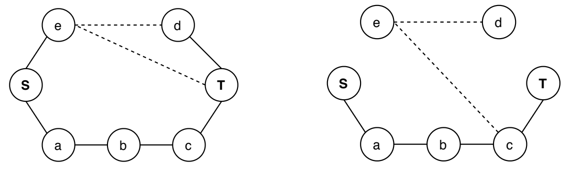
These constraints exist because in real applications, the *data protector* would likely wish to ensure that the network as a whole demonstrates the same functionality (e.g., the nodes should still be able to communicate effectively with one another and resources should still be able to flow efficiently between nodes).

4 METHOD

As discussed in Section 3, the problem we are addressing is to add/remove edges from G to obtain a graph G' so that $\sum_{v \in V_t} \pi_C(v, G')$ is minimized. To achieve this, there are a number of challenges that the *data protector* faces, including:

Unknown data collection strategy. The *data protector* does not know, *a priori*, the collector budget b_c or the precise *data collector* algorithm C. Although the *data protector* can observe the *data collector* to get a sense of the kind of strategy it is using – e.g., expansion-type sampling or random-walk-based – the exact workings of the *data collector* are not known. There are many crawling algorithms that a *data protector* can use [2], including techniques built in-house and tailored for a particular domain. However, in the literature, most real-world crawling techniques are based on random walk or expansion (e.g., breadth first search). We thus consider scores based on these two algorithms for the *data collector*. Separately, to account for the unknown collector budget, we maximize the walk length required to find the target nodes.

Computational efficiency. The *data protector* has $\binom{|E|}{b_p}$ possible combinations of edges that can be deleted, and, in a sparse graph, nearly $\binom{|V|^2}{b_p}$ possible edges that can be added. It is clearly not computationally efficient to measure the effect of adding/deleting each



(a) edge deletion (b) edge addition

Figure 1: Edge deletion/addition for protecting T from a crawler C with budget b_c starting at node S is not submodular. (a) and (b) are counterexamples for the edge deletion and edge addition scenarios. The budgets b_c for (a) and (b) are 3 and 4, respectively. The crawling algorithm used in both examples is *Pick Maximum Degree Next.* The dashed edges in (a) are originally included in the graph, whereas in (b) they initially are not in the graph.

of these sets. Moreover, this problem is not submodular: for example, it may be the case that there are two edges whose individual removal is not very useful in protecting the target nodes, but whose joint removal might seriously hinder data collection. Formally,
Claim: The problem of deleting/adding edges from/to a graph for increasing the *data collector's* path-length from starting node S to target nodes T, using crawling algorithm C with budget b_c, is not submodular.
Proof: Consider the toy graphs in Figure 1. In both figures, we try to hinder the discovery of target node T by a crawler algorithm C that starts at node S and has a limited budget b_c (i.e., the crawler can only visit b_c unique nodes). Suppose that the budgets b_c for 1a and 1b are 3 and 4, respectively. The crawling algorithm used in both examples is *Pick Maximum Degree Next* (i.e., at each node, the crawler picks the neighboring node with the highest degree as the next node to visit). In Figure 1a, consider the dashed edges eT and ed originally existing in the graph. Deleting any of these two edges alone does not hinder the crawler to get from S to T. However, deleting both eT and ed together, makes the shortest path from S to T longer than $b_C = 3$ and, thus, T is protected. Similarly in Figure 1b in which the two dashed edges ed and ec are not part of the graph, if we add any of the two edges separately, the chances of S reaching to T is high. However, if we add the two edges simultaneously, crawler will not be able to reach T (note that in this scenario, crawler in node c will pick node e as the next target due to *Pick Maximum Degree Next* strategy that it follows). ■

For these reasons, we take the approach of assigning each edge a score corresponding to how much its deletion/addition would hinder the *data collector.*

Connectivity-related constraints. If the only goal of the *data protector* was to prevent the *data collector* from reaching the target nodes, the protector could simply remove all edges adjacent to those nodes. However, the *data protector* must obey network connectivity-related constraints in order to ensure that the network is still functional.

4.1 Frequent Path Score

We propose the `Frequent Path Score` (FPS) to specifically handle the case of random walk based *data collector.* Note that we are not assuming any knowledge on the crawler that *collector* would use. Indeed, as shown in Section 5 (Figure 3), we provide evidence that this score gives satisfactory results for expansion-based crawling as well. To elaborate on FPS, let the mean walk length to node $v \in V_t$, given equal probability $p_0 = 1/|V_s|$ of starting at any of the nodes in V_s, be $\mathcal{L}(v, p_0, G)$. We will denote this with $\mathcal{L}(v)$ where p_0 and G are clear from the context. Then, we have,

$$\mathcal{L}(v) = \sum_{i=1}^{\infty} i \cdot \hat{\pi}_C(v, G, i) = \sum_{i=1}^{j} i \cdot \hat{\pi}_C(v, G, i) + \sum_{i=j+1}^{\infty} i \cdot \hat{\pi}_C(v, G, i), \tag{1}$$

where $\hat{\pi}_C(v, G, i)$ is the probability of landing on v with a walk of length i (and not before), and $0 \leq j < \infty$. For any j,

$$\sum_{i=0}^{j} \hat{\pi}_C(v, G, i) + \sum_{i=j+1}^{\infty} \hat{\pi}_C(v, G, i) = 1, \tag{2}$$

because we assume that the graph is connected. That is, if we minimize $\sum_{i=0}^{j} \hat{\pi}_C(v, G, i)$, $\sum_{i=j+1}^{\infty} \hat{\pi}_C(v, G, i)$ will increase by the same amount.

Theorem 1. *If we have a subgraph of* G, G', *such that* $\sum_{i=0}^{j} \hat{\pi}_C(v, G, i) > \sum_{i=0}^{j} \hat{\pi}_C(v, G', i)$, *then,* $\mathcal{L}(v, G) < \mathcal{L}(v, G')$.

Proof. Since the graph is finite, for a sufficiently large k, we have $\hat{\pi}(v, G, l) = 0$ and $\hat{\pi}(v, G', l) = 0$, where $l \geq k$. Then, we can write $\mathcal{L}(v, G)$ and $\mathcal{L}(v, G')$ as,

$$\begin{aligned}\mathcal{L}(v, G) &= \sum_{i=1}^{j} i \cdot \hat{\pi}(v, G, i) + \sum_{i=j+1}^{k} i \cdot \hat{\pi}(v, G, i) \\ &\leq j \cdot \sum_{i=1}^{j} \hat{\pi}(v, G, i) + k \cdot \sum_{i=j+1}^{k} \hat{\pi}(v, G, i) \end{aligned} \tag{3}$$

$$\mathcal{L}(v, G') \leq j \cdot \sum_{i=1}^{j} \hat{\pi}(v, G', i) + k \cdot \sum_{i=j+1}^{k} \hat{\pi}(v, G', i) \tag{4}$$

$$\begin{aligned}&\mathcal{L}(v, G) - \mathcal{L}(v, G') \leq j \cdot \left(\sum_{i=1}^{j} \hat{\pi}(v, G, i) - \sum_{i=1}^{j} \hat{\pi}(v, G', i)\right) + \\ &k \cdot \left(\sum_{i=j+1}^{k} \hat{\pi}(v, G, i) - \sum_{i=j+1}^{k} \hat{\pi}(v, G', i)\right) \leq j \cdot \left(\sum_{i=1}^{j} \hat{\pi}(v, G, i) - \sum_{i=1}^{j} \hat{\pi}(v, G', i)\right) \\ &+ k \cdot \left(1 - \sum_{i=1}^{j} \hat{\pi}(v, G, i) - 1 + \sum_{i=1}^{j} \hat{\pi}(v, G', i)\right) \leq \\ &(j - k) \cdot \left(\sum_{i=1}^{j} \hat{\pi}(v, G, i) - \sum_{i=1}^{j} \hat{\pi}(v, G', i)\right)\end{aligned} \tag{5}$$

We are given that $\sum_{i=0}^{j} \hat{\pi}_C(v, G, i) > \sum_{i=0}^{j} \hat{\pi}_C(v, G', i)$, and by construction $k > j$. Therefore, $\mathcal{L}(v, G) < \mathcal{L}(v, G')$. □

From Theorem 1, we can see that the problem of maximizing $\mathcal{L}(v, G)$ is equivalent to minimizing $\pi_C(v, G, j) = \sum_{i=1}^{j} \hat{\pi}_C(v, G, i)$. When we have multiple target nodes, the paths to discovering them may not be independent. So, we restate the goal of the *data protector* to find G' such that $\arg\max_{G'} \min_{v \in V_t} \mathcal{L}(v, G')$.

We model the random walk using an absorbing Markov chain. The first step is to merge all the nodes in $V_t \cup N(V_t, G')$ into one node, and consider this node as the absorption state. The rest of the nodes are the transient states (including the source nodes). Let Q be the sub-matrices representing the transition between the transient states. Then the fundamental matrix is given by $F = (I - Q)^{-1}$. The expected number of steps before hitting an absorption state starting from v is given by the t_v, where $\mathbf{t} = F\mathbf{1}$. Then, we can show that $\min_{v \in V_t} E(v, G') = \sum_{u \in V_s} \pi_0(u) \cdot t_u$. If F_d is the diagonal matrix of F, transient probabilities are given by $H = (F - I) \cdot F_d^{-1}$. That is, $H_{i,j}$ is the probability of visiting node j in a walk starting from node i before being absorbed. Our goal is to assign scores to nodes based on their importance in a random walk starting from V_s before hitting a node in V_t. So, we define the `Frequent Path Score (FPS)` of node u as,

$$\mathcal{S}_f(u) = \sum_{v \in V_s} p_0(v) \cdot \frac{H_{v,u}}{d(u, G')}. \tag{6}$$

Running Time. Calculating the FPS for all the candidate edges requires inverting the matrix $(I - Q)$. This can be done in $O(|V|^3)$. To scale up, we developed an efficient approximation that is described in Section 4.5.

4.2 Shortest Path Score

In an expansion-based crawling algorithm, the length of the shortest path from the source node determines if a node $v \in V_t$ will be reached. That is, in contrast to the *frequent* path score used above, what matters here is the *shortest* path score: if l is the shortest path length of $v \in V_t$ from the source nodes, $\pi_C(v, G', b_c)$ decreases with increasing l and for $l > b_c$, $\pi_C(v, G', b_c) = 0$. Let $P(u, v)$ be the set of all shortest paths between nodes u and v. For a path $p \in P(u, v)$, let $\delta(w, p)$ be such that $\delta(w, p) = 1$ if $w \in p$ and 0 otherwise. Then, we define the *Shortest Path Score* (SPS) for node u as,

$$\mathcal{S}_p(u) = \frac{1}{d(u, G')} \sum_{v \in V_s} p_0(v) \sum_{v \in V_t} \frac{1}{|P(v, u)|} \sum_{p \in P(v,u)} \frac{\delta(u, p)}{|p|} \tag{7}$$

The idea behind the `Shortest Path Score (SPS)` is that if an edge appears in multiple shortest paths then it is important. It is weighted by: (1) $\frac{1}{|p|}$: The inverse of the shortest path length since nodes on longer paths are less important; (2) $\frac{1}{|P(u,v)|}$: The inverse number of possible paths between the source and target because, if there are a lot of possible paths, it dilutes the importance of the paths; (3) $\frac{1}{d(u,G')}$. The inverse of the node's degree because, if a node has a lot of neighbors, it is more likely that the crawler will follow other paths. SPS is different from existing measures such as betweenness centrality because it considers the importance with respect to source nodes V_s and target nodes V_t. The length and number of shortest paths is also important in SPS, unlike in betweenness centrality. As with FPS, we show that SPS can be used for random-walk-based crawling as well.

Running Time. Computing the SPS for all the candidate edges can be done in $O(|V_s||E|)$. In general $|V_s| \ll |V|$. So, the running time is $O(|V|)$.

4.3 Edge Importance

SPS and FPS give us measures of the importance for the nodes in the network with respect to finding the target nodes. Since our goal is to remove/add edges from/to the network, we need to assign scores to existing and non-existing edges. That is, in the case of edge removal, we need to identify the important edges and remove them (while obeying the constraints), and for edge addition, we need to assign scores to node pairs that do not have an edge between them.

Algorithm 1: `NetProtect-()`

```
Input: $G$, $V_t$, $V_s$, $b_p$
$\forall u \in V$, calculate $\mathcal{S}_*(u)$;
$\forall (u, v) \in E$, calculate $\mathcal{S}_*^-(u, v)$;
$R \leftarrow$ Sort $(u, v) \in E$ by decreasing order of $\mathcal{S}_*^-(u, v)$;
$counter \leftarrow 0$;
while $couter < b_p$ do
    $(u, v) \leftarrow R.pop(0)$;
    Remove $(u, v)$ from $G$;
    if ConstraintsSatisfied(G) then
        $counter \leftarrow counter + 1$;
    else
        Add $(u, v)$ to $G$;
    end
    if IsEmpty(R) then
        return $G$
    end
end
return $G$
```

Algorithm 2: `NetProtect+()`

```
Input: $G$, $V_t$, $V_s$, $b_p$
$\forall u \in V$, calculate $\mathcal{S}_*(u)$;
$X \leftarrow$ Sort $u \in V$ by decreasing order of $\mathcal{S}_*(u)$;
$counter \leftarrow 0$;
$i \leftarrow 0$;
$j \leftarrow |V| - 1$;
while $i < |V| - 1$ do
    while $j > 0$ do
        if $(X[i], X[j]) \notin E$ then
            Add $(X[i], X[j])$ to $G$;
            if ConstraintsSatisfied(G) then
                $counter \leftarrow counter + 1$;
                if $counter \geq b_p$ then
                    return $G$
                end
            else
                Remove $(X[i], X[j])$ from $G$;
            end
        end
    end
end
return $G$
```

Edge Deletion The candidates for edge deletion are all the edges that exist in the network and are allowed by the constraints. Let us consider SPS. For an edge $(u, v) \in E$, the *Edge Deletion Shortest Path Score* (ED-SPS) is defined as $\mathcal{S}_s^-(u, v) = \mathcal{S}_s(u) + \mathcal{S}_s(v)$. That is, edges with important nodes as endpoints are more important. Similarly, we can define *Edge Deletion Frequent Path Score* (ED-FPS).

Edge Addition For edge addition, the goal is to add enough edges to the important nodes so that we 'mislead' the data crawler, and send it to a less-important part of the network. This means we have to connect important nodes with unimportant ones. So, we define the *Edge Addition Shortest Path Score* (EA-SPS) of $(u, v) \notin E$, $u, v \in V$ as $\mathcal{S}_s^+(u, v) = |\mathcal{S}_s(u) - \mathcal{S}_s(v)|$. We can define the *Edge Addition Frequent Path Score* (EA-FPS) similarly.

4.4 NetProtect: Data Protector Algorithm

In this section, we describe the NetProtect algorithm. Depending on whether we are dealing with edge deletion or addition, we can pick one of the two algorithms NetProtect- and NetProtect+ respectively. If *protector* has the knowledge on the type of crawler used by *data collector*, we can use either FPS or SPS. However, in a general setting, we can use either of these scores as they both surpass benchmarks, regardless of the crawling algorithm. The first step in NetProtect is to calculate the node importance (FPS or SPS) for all nodes in the network. Then, the edge scores are calculated depending on whether we are dealing with edge deletion or addition. Finally, the top b_p edges with the highest scores are removed/added from/to the network. Algorithm 1 and 2 describe the NetProtect- and NetProtect+ algorithms in detail. Although recomputing the scores after each edge perturbation yields a more accurate result, it also increases the computation cost and does not scale for larger graphs. Our experiments showed that using the initial scores as the estimated score in each step yields a satisfactory trade-off between the performance and time.

4.5 Speeding Up the Computation of FPS

As described, the time required to compute the FPS for all nodes is $O(|V|^3)$. Thus, in many applications, it may not be feasible to compute FPS. In this section, we describe a sampling method to calculate approximate FPS values. Recall that the idea behind FPS is that nodes that are traversed frequently during a random walk from V_s to V_t should be given more importance. This means that nodes that are very far away from V_s and V_t will not be important. Before we begin, the *data protector* sets a sample size for approximating the scores, denoted by s. A larger sample size results in a more accurate approximation of FPS but, obviously, increases the running time. The first step is to find all the shortest paths from all nodes in V_s to all nodes in V_t, as is done during the SPS computation. Let V' be the set of all nodes that lie in at least one shortest path that does not begin or end at that node. The set of all nodes in the original graph that have a neighbor in V' but are not themselves in V' is given by $N(V', G) \setminus v'$. A random node is selected from this set of nodes and added to V'. This process repeats until $|V'| = s$. Then, FPS is calculated for the nodes in V' using the induced subgraph. The FPS for the nodes that are not included in the sample is assigned as 0. The time complexity of this approach is $O(|V|^2)$.

5 EXPERIMENTS & RESULTS

In this section, we perform an experimental evaluation of the proposed versions of NetProtect algorithm across several real-world networks. In these experiments, we consider various types of target nodes, data collection strategies, and baseline data protection algorithms.[1]

Datasets. We consider five real-world networks: lastfm-asia, musae-twitch, deezer-europe, musae-facebook, and musae-github. All datasets are available from SNAP repository[2]. Table 1 shows basic statistics of these networks.

Baseline Algorithms. As mentioned in Section 2.1, we benchmark NetProtect against the state-of-the-art node hiding algorithm, ROAM, which uses both edge deletion and addition. We also benchmark against two protection algorithms with the objectives of minimizing betweenness centrality and Personalized PageRank of the target nodes, respectively. Finally, we include Random edge deletion/addition as a naïve baseline. In

- **ROAM:** We use the same algorithm as in [18] and iteratively remove the neighbor with highest degree from a target node and add b new edges from this node to neighbors of the target (we set $b = 3$ as in the original paper). To adapt this algorithm to our experimental setup, we enforce the same connectivity constraint that we have considered for NetProtect: we can decrease the degree of a node by at most $r.d(v, G)$. We also treat the cost and budget of node deletion and addition to be the same, and choose a total budget equal to the protector budget in NetProtect. In each iteration, we choose one target node uniformly at random and perform ROAM (remove one, add many) if the connectivity constraint holds. The iteration is continued until either the budget is exhausted, or no target nodes can be manipulated without violating the connectivity constraint.
- **Random Edge deletion/addition:** A random edge satisfying the connectivity constraint is selected for deletion. Edges are deleted until the required number is reached. Similarly, for random edge addition, a random pair of nodes that are not already connected is selected and the corresponding edge is added as long as the constraints are not violated.
- **Betweenness Centrality deletion/addition:** As explained in Section 4.2, SPS is different from Betweenness centrality, despite both grounding their intuition on the shortest paths. To empirically emphasize this difference, we assign each node a score based on their betweenness centrality [4]. Then, the edges to delete/add are selected as described in Section 4.3.
- **Personalized PageRank deletion/addition**: It is similar to Betweenness Centrality, but uses the average Personalized PageRank [7] calculated from the target nodes.

Data Collection Algorithms. As described in Section 2.2, the literature contains numerous graph crawling algorithms. Many of these algorithms have been designed for specific data collection goals and domains, but a large number are based on random walk or expansion-based strategies. Thus, we consider Random Walk, BFS, and DFS crawlers as they are the foundations for many other, possibly more sophisticated, methods. Note that, in our experiment results in Table 1, DFS results are not reported due to both the lack of space and the similarity of results to that of BFS.

Experimental Setup. For each network, we consider two types of target nodes (V_t): random target nodes and degree-based target nodes. As the name suggests, in the first case, the target nodes are selected uniformly at random. The degree-based target nodes are selected randomly with the selection probability proportional to the node degree. For each network, we consider five sets of

[1]Our code is publicly available at https://github.com/rlaishra/NetProtect.

[2]https://snap.stanford.edu/data/index.html

Table 1: The fraction of nodes queried (data collector budget) to find 5 and 50 target nodes of various types after perturbations implemented by different data protector algorithms. Higher numbers indicate better performance by the data protector.

Networks	Data Protector		Random Target				Degree based Target			
			BFS		RW		BFS		RW	
			5	50	5	50	5	50	5	50
lastfm-asia $\lvert V\rvert = 7,624$ $\lvert E\rvert = 27,806$	Addition	NetProtect	**0.15**	**0.56**	**0.14**	**0.79**	**0.04**	**0.08**	**0.08**	**0.12**
		Betweenness	0.12	0.51	0.09	0.61	0.02	0.05	0.04	0.09
		P PageRank	0.10	0.52	0.10	0.63	0.02	0.08	0.05	0.08
		Random	0.05	0.41	0.08	0.53	0.02	0.04	0.04	0.06
	Deletion	NetProtect	**0.18**	**0.78**	**0.19**	**0.74**	**0.24**	**0.40**	**0.17**	**0.27**
		Betweenness	0.10	0.60	0.12	0.65	0.08	0.12	0.07	0.13
		P PageRank	0.07	0.65	0.10	0.62	0.06	0.14	0.09	0.15
		Random	0.07	0.58	0.08	0.52	0.01	0.04	0.02	0.05
	Both	ROAM	0.05	0.55	0.05	0.53	0.03	0.31	0.02	0.22
musae-twitch $\lvert V\rvert = 7,126$ $\lvert E\rvert = 35,324$	Addition	NetProtect	**0.09**	**0.56**	**0.12**	**0.68**	**0.07**	**0.10**	**0.11**	**0.16**
		Betweenness	0.03	0.42	0.07	0.45	0.02	0.05	0.03	0.05
		P PageRank	0.03	0.41	0.07	0.57	0.02	0.05	0.02	0.05
		Random	0.04	0.47	0.04	0.51	0.01	0.05	0.02	0.04
	Deletion	NetProtect	**0.13**	**0.80**	**0.17**	**0.74**	**0.14**	**0.20**	**0.12**	**0.18**
		Betweenness	0.08	0.60	0.08	0.62	0.01	0.04	0.02	0.05
		P PageRank	0.07	0.65	0.07	0.59	0.02	0.03	0.02	0.04
		Random	0.06	0.55	0.10	0.53	0.01	0.03	0.02	0.03
	Both	ROAM	0.07	0.59	0.05	0.53	0.01	0.10	0.00	0.06
deezer-europe $\lvert V\rvert = 28,281$ $\lvert E\rvert = 92,752$	Addition	NetProtect	**0.12**	**0.32**	0.11	**0.38**	**0.02**	**0.05**	**0.04**	**0.09**
		Betweenness	0.10	0.25	**0.13**	0.25	0.01	0.04	0.01	0.06
		P PageRank	0.10	0.24	0.12	0.20	0.01	0.04	0.02	0.07
		Random	0.07	0.22	0.12	0.18	0.01	0.02	0.02	0.06
	Deletion	NetProtect	**0.20**	**0.66**	**0.20**	**0.81**	**0.12**	**0.24**	**0.09**	**0.19**
		Betweenness	0.10	0.60	0.08	0.52	0.02	0.05	0.04	0.09
		P PageRank	0.07	0.61	0.12	0.59	0.02	0.04	0.05	0.10
		Random	0.07	0.58	0.08	0.52	0.01	0.03	0.02	0.04
	Both	ROAM	0.10	0.61	0.07	0.60	0.03	**0.40**	0.02	**0.30**
musae-facebook $\lvert V\rvert = 22,470$ $\lvert E\rvert = 171,002$	Addition	NetProtect	**0.18**	**0.59**	**0.11**	**0.66**	**0.09**	**0.12**	**0.08**	**0.10**
		Betweenness	0.07	0.52	0.06	0.51	0.02	0.07	0.01	0.08
		P PageRank	0.06	0.51	0.10	0.53	0.02	0.07	0.06	0.09
		Random	0.04	0.47	0.05	0.48	0.01	0.05	0.01	0.06
	Deletion	NetProtect	**0.16**	**0.62**	**0.17**	**0.64**	**0.23**	**0.33**	**0.07**	**0.11**
		Betweenness	0.06	0.59	0.07	0.53	0.02	0.05	0.01	0.06
		P PageRank	0.08	0.55	0.10	0.53	0.04	0.05	0.04	0.08
		Random	0.05	0.51	0.05	0.50	0.01	0.05	0.01	0.04
	Both	ROAM	0.10	0.68	0.07	0.61	0.01	0.08	0.00	0.04
musae-github $\lvert V\rvert = 37,700$ $\lvert E\rvert = 289,003$	Addition	NetProtect	**0.14**	**0.55**	**0.15**	**0.61**	**0.11**	**0.17**	**0.03**	0.05
		Betweenness	0.06	0.51	0.05	0.49	0.01	0.08	0.02	0.06
		P PageRank	0.08	0.47	0.12	0.52	0.02	0.06	0.02	**0.07**
		Random	0.05	0.45	0.04	0.42	0.01	0.04	0.01	0.06
	Deletion	NetProtect	**0.17**	**0.70**	**0.11**	**0.67**	**0.03**	**0.17**	**0.05**	**0.14**
		Betweenness	0.08	0.52	0.07	0.56	0.02	0.07	0.02	0.06
		P PageRank	0.06	0.49	0.06	0.49	0.02	0.06	0.03	0.06
		Random	0.05	0.45	0.05	0.49	0.01	0.05	0.01	0.05
	Both	ROAM	**0.39**	**0.79**	**0.11**	0.64	0.02	0.16	0.00	0.01

source (V_s) and target nodes (V_t). In all cases, the source nodes are selected randomly from the set of non-target nodes. We set $|V_s| = 10$, $|V_t| = 100$, and $p_0(v) = 0.1$ for all $v \in V_s$. For each (V_s, V_t) pair, we perform 30 trials, and set $r = 0.5$. We select protector budgets (number of edges that can be deleted/added) ranging from $0.2\% - 1.00\%$ of the edges in the entire network.

Performance Metric. To measure the performance of a data protector algorithm, we compute the number of queries the data collector has to make to find a fixed number of target nodes. If the data protector is effective, the data collector will have to perform a large number of queries before finding target nodes. As the goal is to maximize the number of queries required by the data collector to find the target nodes, higher values indicate better performance.

Figure 2: NetProtect- performance for different values of r. For smaller set of target nodes, the performance is almost independent from r. As the target set grows and protector budget shrinks, increasing r improves the performance.

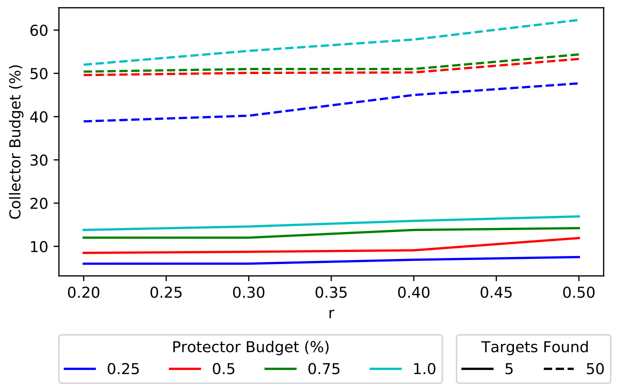
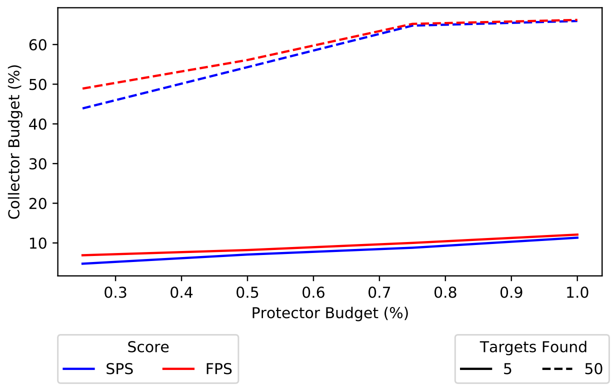


Figure 3: Using SPS and FPS for random walk crawler. The similar performance of the two scores against the same crawling algorithm shows that they can be used interchangeably, regardless of the crawling algorithm.



5.1 Performance Comparison against Baseline Methods

Table 1 shows the amount of budget the collector has to invest in order to find 5 and 50 target nodes (the best values in each batch are indicated in bold). The protector budget in these experiments is set to 1% of the edges. We have run the experiments with different budgets as well (see Figure 3) and chose this value as it resulted in a decent trade-off between performance and computation cost. We also performed experiments using a DFS crawler, which gave results similar to BFS which are not shown here to reduce space. In almost all cases, `NetProtect` offers better performance than Betweenness, Personalized PageRank, and Random edge perturbations. Note that Personalized PageRank and Betweenness Centrality are skewed towards low degree nodes that are acting as bridges. However, tampering with the connectivity of these bridge nodes leads to changing the connectivity of the graph. As such, the nodes detected by these two centrality measures are often useless and result in a performance close to Random edge perturbation, as seen in Table 1. Indeed, the superiority of `Netprotect` in comparison is in finding the nodes that are not crucial in maintaining the connectivity of the graph, but crucial in reaching the target nodes.

`NetProtect` generally outperforms ROAM for all datasets. In fact, in many cases, ROAM does not even surpass the three other benchmarks in hindering the collector. The only instances in which ROAM offers a better performance are for random target nodes in `musae-github` and degree-based targets in `deezer-europe`. This interesting observation hints at the fact that ROAM for certain sets of target nodes can offer a competitive performance. Consider a randomly chosen target set that contains one target node with high degree and the rest of the targets with degree 1. In this case, our implementation of ROAM focuses on that one high degree node, as pruning the other target nodes would violate the connectivity constraint (creating isolated nodes). As a result, the high degree node will use up all the budget and lose its degree centrality rapidly. The final network will have target nodes that are in the fringe of the network, i.e., accessible through only one or two nodes. In this case, a BFS crawler will have a hard time finding the targets. However, once we use random walk crawler or change the structure of target nodes (for example, degree-based targets), ROAM immediately loses this advantage and lags behind `NetProtect`, as seen in `muse-github` results. Now consider a similar setup with many target nodes in fringe of the network, but this time with more than one high-degree target node. In this scenario, due to budget constraint, ROAM cannot turn all target nodes to fringe nodes. In this scenario, ROAM gives a poor performance for protecting a handful of target nodes (e.g., 5), but does better as the tolerance for number of discovered target nodes increases (see degree-based target for `deezer-europe`).

Another interesting observation from Table 1 is the superiority of edge deletion over edge addition for `NetProtect` and other benchmarks, which is consistent with previous observations [8]. Note that we also tested a variation of `NetProtect` that combines edge addition and removal by deleting from high score region and adding to low score regions (not shown due to space constraint). We found the edge deletion to be still superior.

5.2 Tuning r

The main connectivity constraint that we imposed in all of our experiments depends on parameter r (the allowed fraction of change in a node's degree). To study the impact of r on our results, we repeated our experiments with different values of r. Figure 2 shows the result for `NetProtect-` and a random-walk-based collector on `musae-facebook`. The different colors represent different protector budgets. As we see, for discovering 5 target nodes, the performance is almost independent from r. As the number of target nodes and protector budget increase, higher value of r makes the task harder for data collector.

5.3 Scores and Different Crawling Algorithms

We initially proposed `FPS` and `SPS` to target random-walk-based and expansion-based crawlers, respectively. However, as we assumed that the protector does not have prior knowledge on the

Table 2: Average time (in seconds) to find SPS and FPS for all the nodes

Network	SPS	FPS (Approximate)	FPS (Exact)
musae-twitch	3.9×10^1	4.1×10^2	8.7×10^4
musae-facebook	2.5×10^2	6.5×10^3	9.1×10^5
musae-github	4.1×10^2	1.1×10^4	-

crawling algorithm that collector uses, we need to confirm that both of these scores are indeed successful with respect to both random walk or expansion-based crawling. To achieve this, we repeated our experiments using SPS for random walk and FPS for expansion-based crawling. The result for NetProtect- with random walk crawler on musae-facebook is shown in Figure 3. As we see, their performances are similar and we can use these two scores interchangeably. In general, SPS has a lower computation cost (our experiments for SPS were 10 times faster than FPS, see Table 2). However, for random walk crawlers, as we reduce the protector budget, FPS gives a better performance than SPS.

5.4 Running Times

Table 2 shows the time it takes to calculate FPS and SPS for all nodes in three graphs. In the case of FPS, we use the approximate method described in Section 4.5. The values presented are the average over 50 trials (except for FPS exact). Although computation of FPS without the approximation is not feasible for some of the networks we consider, it is much faster with the approximation.

5.5 Limitation

The focus of this study has been on increasing the robustness of the network against external attacks while maintaining the functionality of the network through connectivity criterion. However, there are other criteria that might be of interest depending on the application of the network. For example, if latency is of crucial importance, increasing the shortest path length between targets and certain nodes might not be desirable. In this case, we need to define a tolerance threshold for the maximum latency that the network can handle and include it in ConstraintsSatisfied in Algorithms 1 and 2. The overall latency of the network is tune-able by using the right value for parameter r (see Section 5.2).

6 CONCLUSION

In this study, we formulated the problem of modifying a network so as to best hide target nodes from an entry-point attack. We proposed two node-level scores: FPS and SPS. FPS assigns importance to nodes to prevent a random-walk-based crawler from discovering the target nodes within their budget limit; and SPS is designed for expansion-type crawlers. We proposed the NetProtect algorithm to remove or add edges to hinder a data collector from finding the target nodes. NetProtect uses FPS and SPS to identify the candidate edges for removal or addition. Our experiments on multiple real-world networks show that NetProtect outperforms all considered baseline algorithms, including the state-of-the-art protection algorithm, ROAM. In some networks, with 1% of edges removed by NetProtect, the data collector, compared to the best baseline method, requires up to 6 times the query budget in order to find the same number of target nodes. We also show that our proposed scores can maintain their superior performance even for crawlers that they are not optimized for (i.e., SPS and FPS can be used for random walk and expansion-based crawlers as well).

7 ACKNOWLEDGMENTS

Wendt was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. Soundarajan and Laishram were supported by Army Research Office award W911NF-18-1-0047. Soundarajan was additionally supported by NSF award 1908048.

REFERENCES

[1] Olle Abrahamsson. [n.d.]. Master's thesis. Linköping University, Sweden.

[2] Katchaguy Areekijseree, Ricky Laishram, and Sucheta Soundarajan. 2018. Guidelines for online network crawling: A study of data collection approaches and network properties. In *Proceedings of the 10th ACM Conference on Web Science*. 57–66.

[3] Katchaguy Areekijseree and Sucheta Soundarajan. 2019. Crawling Complex Networks: An Experimental Evaluation of Data Collection Algorithms and Network Structural Properties. *The Journal of Web Science* 6 (2019).

[4] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* 25, 2 (2001), 163–177.

[5] Hale Cetinay, Karel Devriendt, and Piet Van Mieghem. 2018. Nodal vulnerability to targeted attacks in power grids. *Applied network science* 3, 1 (2018), 34.

[6] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. 2010. Walking in Facebook: A case study of unbiased sampling of osns. In *IEEE Conference on Computer Communications*. 1–9.

[7] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *Proceedings of the 12th International Conference on World Wide Web*. 271–279.

[8] Jie Ji, Guohua Wu, Chenjian Duan, Yizhi Ren, and Zhen Wang. 2019. Greedily Remove k Links to Hide Important Individuals in Social Network. In *International Symposium on Security and Privacy in Social Networks and Big Data*. Springer, 223–237.

[9] Manish Kumar, Rajesh Bhatia, and Dhavleesh Rattan. 2017. A survey of Web crawlers for information retrieval. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 6 (2017), e1218.

[10] Ricky Laishram, Katchaguy Areekijseree, and Sucheta Soundarajan. 2017. Predicted max degree sampling: Sampling in directed networks to maximize node coverage through crawling. In *IEEE Conference on Computer Communications Workshops*. IEEE, 940–945.

[11] Yali Liu, Cherita Corbett, Ken Chiang, Rennie Archibald, Biswanath Mukherjee, and Dipak Ghosal. 2008. Detecting sensitive data exfiltration by an insider attack. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*. 1–3.

[12] Arun S Maiya and Tanya Y Berger-Wolf. 2010. Online sampling of high centrality individuals in social networks. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 91–98.

[13] Arun S Maiya and Tanya Y Berger-Wolf. 2010. Sampling community structure. In *Proceedings of the 19th international conference on World wide web*. 701–710.

[14] Mainack Mondal, Bimal Viswanath, Allen Clement, Peter Druschel, Krishna P Gummadi, Alan Mislove, and Ansley Post. 2012. Defending against large-scale crawls in online social networks. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 325–336.

[15] Anthony Piltzecker. 2011. *The Best Damn Windows Server 2008 Book Period*. Elsevier.

[16] Yilin Shen, Nam P Nguyen, Ying Xuan, and My T Thai. 2012. On the discovery of critical links and nodes for assessing network vulnerability. *IEEE/ACM Transactions on Networking* 21, 3 (2012), 963–973.

[17] Chris Steffen. 2017. Should jump box servers be consigned to history? *Network Security* 2017, 11 (2017), 5–6.

[18] Marcin Waniek, Tomasz P Michalak, Michael J Wooldridge, and Talal Rahwan. 2018. Hiding individuals and communities in a social network. *Nature Human Behaviour* 2, 2 (2018), 139–147.

[19] Shaozhi Ye, Juan Lang, and Felix Wu. 2010. Crawling online social graphs. In *2010 12th International Asia-Pacific Web Conference*. IEEE, 236–242.