Optimal Sensor Placement in Network Intrusion Detection Systems

Joshua Rinaldi, Student, Air Force Institute of Technology

***Abstract*—This paper proposes an algorithmic approach to determining where on a network to place packet capture devices used in a Network Intrusion Detection System in order to be most likely to catch malicious actors moving laterally through the network. Designed are three algorithmic implementations, a deterministic, stochastic and local search approach are all used.**

***Index Terms*—Intrusion Detection, Algorithm Design, Genetic Algorithm, Tabu Search**

I. Introduction

INEVITABLY one issue that Cyber Defense teams encounter is determining where to place sensors on a network when setting up monitoring tools. Most of these teams typically have a limited number of packet capture devices available to place on the network, and need to ensure that they are monitoring the most likely route that an attacker might take from an edge node to a target machine on the network. This is the problem I aim to solve with the algorithm proposed below; given a knowledge of network topology, along with vulnerabilities in the machines on the network, producing a list of connections on the network to monitor that maximized the likelihood of detecting an adversary moving laterally through the network.

For this paper, three algorithm implementations, a Deterministic, Stochastic and Local Search, are designed and implemented in code. All code used for experimentation, along with data sets and pseudocode can be found at the associated GitHub repository: https://github.com/jarnm04/Sensor-Placement.

II. Deterministic Approach

A. Problem Domain Requirements Specification

1) Input Domain D_I:

$$(D_I) \rightarrow (G, n) \tag{1}$$

For this problem, we will need to supply network information. This will be represented as a graph:

$$G(V, E, N_E, N_T) \tag{2}$$

Where:

- V is the set of all vertices in the graph (e.g., the machines in the network).
- E is the set of edges connecting those vertices (e.g., the communications pathways flowing between machines).
- N_E is the set of edge nodes in the network, which are most likely to provide an entry point into the network for an attacker.
- N_T is the set of target nodes in the network, the machines which are critical to the network and are most in need of protecting from the outside world.
- Every edge, $e \in E$ will have some weight w such that $w(a, b)$ will be equal to the number of vulnerabilities present on machine b.
  - If $w(*, b) = 1$, it means that the node $b \in V$ does not have any known vulnerabilities, however it is still given a weight of 1 to account for the potential of a zero day exploit.

Additionally, we will need a number n of the available sensors that we can place on the network.

2) Input Conditions:

- $(N_E, N_T) \in V$
  - the target and edge nodes should exist in the list of vertices in the graph.
- $\forall e \in E \ \exists \ w$
  - all edges in the graph should have a weight value.

3) Output Domain D_O:

$$S \rightarrow (D_O) \tag{3}$$

The output of this problem should be a set S of edges which should be monitored in order to maximize the likelihood of detecting an intrusion in the network.

4) Output Conditions:

- $||S|| = n$
  - The number of edges in the solution set S should be equal to the number of sensors, n, that are available.
- $(\forall e \in S) \subset (\forall e \in P)$
  - For some set of paths P which could be taken from any N_E to any N_T, all edges e in the solution set S should appear in the paths contained in P.
- Edges $e \in S$ should come from the most likely paths $p \in P$.

5) Discussion on Complexity: This problem is one that can be viewed as a combination of two other problems: the K-Cut problem, which is NP-Hard under the correct circumstances, and the Stable Paths Problem, which is NP-Complete.

Per [3] the Multi-Cut Problem is a problem that is able to be solved in polynomial time. However, certain variations of the problem are considered to be in NP-Hard. One such variation is the K-Cut problem, where the aim is to split the graph into K separate components. Ordinarily this problem is also solvable in polynomial time, however, when the number of cuts, k, that need to be made is included as a part of the input, the problem becomes NP-Hard.

The other portion of the problem presented in this paper is that of finding the most likely path an attacker would take to travel from $v \in N_E$ to $v \in N_T$. One thing to keep in mind is that an attacker would likely be moving to those machines that have a greater number of known vulnerabilities, and will not likely be moving into those machines that do not have any known vulnerabilities, e.g. $(v \in V)\,\exists w\,(*, v) = 1$. Per [5], a common exploitation technique employed when moving laterally through a network is to use machine as a "pivot", where the attacker will use a machine on the network that has already been exploited to forward malicious traffic to other machines. Because the machines that the hacker has exploited might normally talk to one another, this can help any malicious traffic to be hidden amongst traffic that is considered legitimate.

Because each machine that a hacker uses as a pivot can be viewed as a sort of router, this bring us around to the Stable Paths Problem, which is a problem proposed in [2] to determine the best path through which to route data across the internet. Per [4], the Stable Paths Problem, and its variants, will always be in NP-Complete.

Therefore, because of the combined complexity of these two problems, we defer to the higher of the two complexity class and say that the problem presented in this paper exists in NP-Hard space.

B. Algorithm Domain Specification

The algorithm to solve this problem will be based around a Depth First Search approach, utilizing backtracking. All Algorithm Domain Specifications for DFS derived from [8].

- Next-state-generator: $(D_I) \to x \in X; I(x)$
- Selection: $D_I \to x; x \in X$
- Feasibility: $(x, D_P) \to$ boolean (if true $\cup\,(x, S)$)
- Solution: argmax $(O\,(x, z))\,; (D_P) \to$ boolean; $z = D_P$
- Objective: $(D_P) \to D_O$

C. Problem Domain/Algorithm Domain Integration Specification

1) Initialization: After the network graph has been ingested, it will be stored as a list of structs, the specifics of which will be discussed in further detail later in this paper. The vertices contained within N_E and N_T will be placed inside of a separate data structure.

2) Next State Generator: The next vertex that will be visited will come from the input domain. This vertex will specifically be chosen from the neighbors of the current vertex.

3) Selection Function: When looking at which neighboring vertex to pivot to next, we will consider a few factors:

- The vulnerability score of that particular neighbor, v.
- The number of unexplored vertices v has as neighbors.
- The vulnerability scores of the unexplored neighbors of v.
- Whether the neighboring vertex is in the path to the current vertex.

These will be fed as input into a likelihood function, and the neighboring vertices will be explored in the order of their likelihood.

4) Feasibility Function: A neighboring vertex will only be considered to be a feasible next hop if it has a non-zero likelihood.

5) Objective Function: This function will check to see if the current path ends on a target node. If it does, the overall probability of an adversary traversing the path will be determined and the path will be added to the set of viable paths for the current starting vertex of the path (which will be an edge node). This updated set of viable paths will then be used to update the set of edges that will reliably provide visibility on traffic coming from an edge node to a target node.

6) Delay Termination: Termination of the algorithm will be delayed in order to ensure that all potential paths from all edge nodes to any of the target nodes are considered. Only after all of these potential paths have been explored, and the solution set of edges to monitor has been fully updated, will the algorithm be terminated and a solution returned.

D. Algorithm Domain Design Specification Refinement

1) Initialization and Data Structures: When the network graph is ingested, each vertex will be stored in a struct. These structs will be stored in a list V and contain:

- v: Vertex name
- λ: Vulnerability score
  - This value is actually the weight of all edges $e\,(*, v)$ where v is the current vertex. Because $w\,(*, v)$ will be the same for all edges terminating at v, these edge weights will be stored as a part of v for the sake of space efficiency.
- A_v: Neighboring vertices and their vulnerability scores
  - This will be stored key-value pairs, and used to help determine the most stable paths

2) Next State Generator:

$$(D_I) \to v_{next} \in A_{v_{cur}} \tag{4}$$

From our input domain we will pull some vertex v_{next} to pivot to next from the list of neighbors, A, of current vertex v_{cur}.

3) Selection Function: When looking at which neighbors of the current vertex v_{cur} to pivot to next, the following likelihood value, which is calculated for each neighbor, is considered:

$$L\,(v_{next} \in A_{v_{cur}}) = \varphi_{v_{next}} \lambda_{v_{next}} \left(\frac{\sum_{i=1}^{A_{v_{next}}} \lambda_i \varphi_i}{\sum_{i=1}^{A_{v_{next}}} i \varphi_i} \right) \tag{5}$$

Where:

- v_{next} is the neighboring vertex being considered.
- $A_{v_{cur}}$ is the matrix of neighbors of v_{cur}.
- $A_{v_{next}}$ is the matrix of neighbors of v_{next}.
- $\varphi = \begin{cases} 1 & \text{if } v \text{ is in current path} \\ 0 & \text{if } v \text{ is not in current path} \end{cases}$
- λ is the vulnerability score of the vertex in question.

Neighbors of v_{cur} will be ranked in order of the largest to smallest likelihood, $L\left(v_{next} \in A_{v_{cur}}\right)$, and then popped from that list in order for consideration by the feasibility function.

4) Feasibility Function: Whether a particular neighboring vertex, v_{next} is a feasible next hop will be determined by:

$$\text{bool}\left(L\left(v_{next}\right) > 0\right) \tag{6}$$

If true, then v_{next} will be appended to the current path P. P will then be passed to the Objective Function for further evaluation.

5) Objective Function: Whether a path P is feasible will be determined by:

$$\text{bool}\left(P_{last} \in N_T\right) \tag{7}$$

Where P_{last} is the final vertex in path P. If true, then P fulfills the objection function, P is then evaluated according to:

$$\Lambda_P = \frac{\sum_{i=1}^{P} \lambda_i}{||P||^3} \tag{8}$$

Where:

- Δ_P is the path score of some path P.
- $\sum_{i=1}^{P} \lambda_i$ is the sum of vulnerability scores of all vertices $v \in P$.
- $||P||$ is the magnitude of (number of vertices in) P. This value is cubed to add increased emphasis on shorter paths.

Finally, each edge $e \in P$ is added to the solution set F according to:

$$(\forall e \in P) \rightarrow \begin{cases} \Delta_e + \Lambda_P & \text{if } e \in F \\ F = F \cup e, \Delta_e = \Lambda_P & \text{if } e \notin F \end{cases} \tag{9}$$

Where:

- Δ_e is the final worthiness of an edge.

F is then sorted based on Δ_e, with the larger values being placed at the front of the list.

6) Delay Termination: Once all paths have been explored, return the first n edges from F to D_O:

$$\left(F_{0..n}\right) = S \rightarrow D_O \tag{10}$$

E. Deterministic Algorithm Pseudocode

See Algorithm 1 for the pseudocode implementation of the deterministic algorithm.

F. Deterministic Search Tree Diagram

See figure 1 for a tree diagram derived from the *verySmall-Graph.txt* file

Algorithm 1 Deterministic Implementation

```
###SET OF CANDIDATES
INPUT G: Graph;
###INTERMEDIATE SETS AND VALUES
Verts: set of objects; NumSens: number;
VertsEdge: set objects; VertsTarget: set of objects;
P: set of objects; F: set of objects;
###OUTPUT
OUTPUT S: set of objects
    ###INITIALIZATION
    F = empty; S = empty
    Main Loop
        P = empty
        ###DELAY TERMINATION
        Choose object v from VertsEdge, append to P
        Find_Paths(P, Verts, VertsTarget, F)
    Endloop
    i in range (1..NumSens)
    S.append(F[i])
    Endfor
    Return S

    Func Find_Paths(P, Verts, VertsTarget, F)
        v_cur = P.last
        ###OBJECTIVE FUNCTION
        bool($P_{last} \in N_T$)
            find $\Delta_P$
            add edges in P to F, per bool($\forall e \in P$)
        Else
            ###SELECTION FUNCTION
            likelihood $L(v_{next} \in A_{v_{cur}})$
            sort neighbors by likelihood
            For v in neighbors
                ###FEASIBILITY FUNCTION
                bool($P_{last} > 0$)
                    ###NEXT STATE GENERATOR
                    Find_Paths(P+v, Verts, VertsTarget, F)
                Endif
            Endfor
        Endif Endif
    Endfunc
```

III. Stochastic Algorithm Specification

The stochastic implementation for this problem will be a population based Genetic Algorithm based on Karger's Algorithm [1]. In Karger's Algorithm, edges are chosen at random, and the end vertices of that edge are combined. The chosen edge is removed, thereby ensuring that no self edges exist, however, multi-edges are still permitted. This process is repeated until there are only two vertices left in the graph, and the edges connecting those two are considered to be the min cut. The strength of Karger's algorithm comes from the fact that it is statistically likely to produce the minimum edge cut of the graph, it does not, however, provide any guarantees that the resulting set of edges is in fact the true minimum cut of the graph.

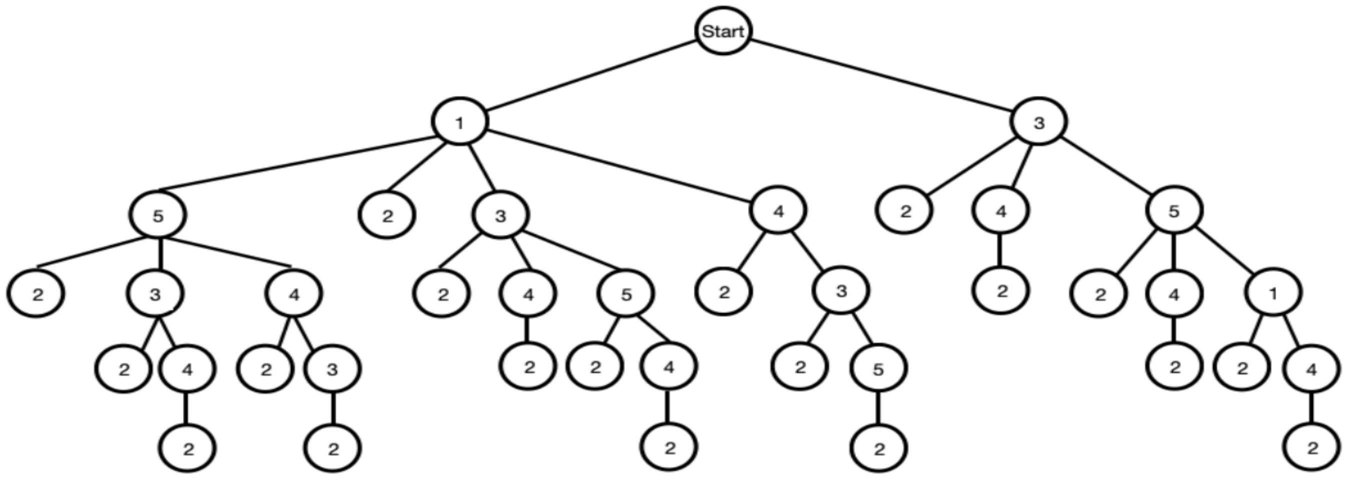

Fig. 1. Tree diagram of *verySmallGraph.txt*

The stochastic algorithm for this problem will use this concept of randomly chosen edges. Each candidate will contain a binary gene string representing the edges to cut. Whether a candidate is feasible will be dependent upon whether it successfully divides the graph into two groups. Because it is possible that with the sensor placements, that vertices in N_T could end up on the same side of the cut as vertices in N_E, it will not be required that $\forall v \in N_T$ be completely cut off from $\forall v \in N_E$, however, the strength of the divide between these two sets will be considered in the fitness function. The fitness function will also take into consideration the weight of the edges that are cut. Finally, the candidate population will be kept at a steady size, and which members of the candidate population are allowed to remain in the population will be determined by an elitism based selection function.

A. Initialization

After the graph is ingested into the algorithm, edges will be stored as structs, containing the following information:

- id: numerical value
- source: source vertex
- dest: destination vertex
- weight: vulnerability score of destination vertex

These edges will then be placed into set E, where they will be able to be referenced by their ID. Next, the candidate population, C, will be randomly seeded with some number of candidates, c. Each candidate will contain the following information:

- gene: binary string, each position corresponding to one of the graph edges
  - $\text{gene} = \begin{cases} 1 & \text{if } e \in c \\ 0 & \text{if } e \notin c \end{cases}$
- fitness: number, the overall fitness score of candidate feasible: bool, whether the candidate is a feasible solution
  - for all $c \in C$, this value should be true, it will be used to determine if candidates are in need of repair
- reduction: the reduced graph according to the edges to be included
  - This will be generated using the same process outlined by Karger's algorithm [1]. Edges to remove will be determined by the gene of the candidate, not random choice

B. Evaluation Function

This function will be used to determine the fitness of a particular candidate. It is known that for a particular candidate, $c \in C$, the network graph will be split into two separate halves. Each half of the graph may contain and arbitrary number of $v \in N_E$ as well as an arbitrary number of $v \in N_T$. For one of the two sides of the graph, we find:

$$\delta_E = \frac{v \in N_E}{||N_E||} \tag{11}$$

Which is the percentage of edge vertices on that side of the graph. We also find:

$$\delta_T = \frac{v \in N_T}{||N_T||} \tag{12}$$

Which is the percentage of target vertices on that side of the graph.

Given that these values are percentages, it can then be said that:

$$(\delta_E, \delta_T) = \begin{cases} 1 & \text{if all edge/target vertices present} \\ 0 & \text{if no edge/target vertices present} \\ 0 < \delta < 1 & \text{if some other number present} \end{cases} \tag{13}$$

Therefore, we wish to maximize the value:

$$\vartheta = |\delta_E - \delta_T| \tag{14}$$

As this value will be closer to 1 if the percentages of edge and target vertices present on one side of the graph differ greatly (which is desired), and will be closer to zero if these percentages are closer together. Additionally, we only need to calculate this value for one side of the graph, as the respective δ are percentages, it is true that:

$$\delta_{E_L} = 1 - \delta_{E_R} \text{ and } \delta_{T_L} = 1 - \delta_{T_R} \tag{15}$$

Where 'L' and 'R' represent the left and right side of the graph respectively. Through algebra, it is shown that:

$$|\delta_{E_R} - \delta_{T_R}| = |\delta_{T_L} - \delta_{E_L}| \tag{16}$$

It is also true that for any value x and y, $|x - y| = |y - x|$, so therefore, we need only look at one side of the graph to determine the value of ϑ for that candidate. Finally, this gives the fitness function:

$$\text{fitness}(c \in C) = \frac{\vartheta}{||c||} = \frac{|\delta_E - \delta_T|}{||c||} \tag{17}$$

Where $||c||$ is the magnitude of the candidate, or in other words, the number of edges included in the candidate. This value is important, because due to the limited number of sensors available on the network, those candidates that effectively split N_E and N_T using a smaller number of edges are more desirable.

C. Selection Function

The number of candidates, $||c||$, will be maintained at a constant value of 50. Therefore, if bool $(||C|| > 50) =$ true, then C will be reduced by removing the $||C|| - 50$ least fit candidates from the population.

D. Crossover Function

In order to generate new members of the population, at each iteration of the algorithm, 10 $c \in C$ will be chosen at random to be bred, generating 5 offspring. For the generation of the offspring, it will be assumed that present between the two parents would still be present after the performance of the Karger algorithm in a deterministic approach, therefore, the base gene for the child will be created from a bitwise OR operation over the parent genes:

$$c_{child} = c_{p1} \text{ OR } c_{p2} \tag{18}$$

From here, the child gene will be passed to the mutation function to introduce some amount of randomness into the population.

E. Mutation Function

Whether a certain allele in a gene is mutated will be up to a probability. This probability, $\mu(a)$, will be based on the difference in the average magnitude of the parent genes, and the magnitude of the child gene. This probability will be given by:

$$\mu(a) = 1 - \frac{0.5 * (||c_{p1}|| + ||c_{p2}||)}{||c_{child}||} \tag{19}$$

From here, the gene string will be iterated through, and for each allele, a random number, x, will be generated, and this and the probability of mutation will be passed to the mutation function:

$$\text{if bool}\,(x < \mu(a)) : \text{flip allele } a \tag{20}$$

After the child's gene has been mutated, it should be passed to the feasibility function for validation.

F. Feasibility Function

This function is intended to determine whether a certain candidate $c \in C$ successfully splits the graph in half. It is determined with:

$$\text{bool}\,(||v|| \in G_R = 2) \tag{21}$$

Where:

- G_R is the reduction graph for c
- $||v|| \in G_R$ is the number of vertices in the graph

If this value is true, then c is considered feasible and added to the population. If the result of this function is false, it means that c is in need of repair.

G. Repair Function

If some $c \in C$ is in need of repair, it means that it contains too many edges to effectively split the graph in two. Therefore, for this function we will generate a random number, and if the allele in the position of the gene is a 1, it will be flipped to a 0, otherwise, the number will be incremented until the first non-zero allele is found, and then it will be flipped to a zero.

Once this has been performed, c will again be checked for feasibility.

H. Solution Function

Once a predetermined number of generations has been created, the fittest $c \in C$ will be chosen as the solution. From there, the edges included in the solution set will be evaluated for their respective likelihoods. This function will be similar to the likelihood function defined in the solution function of the deterministic approach:

$$L\,(e \in c) = \lambda_{v_{dest}} \left(\frac{\sum_{i=1}^{A_{v_{next}}} \lambda_i}{\sum_{i=1}^{A_{v_{next}}} i} \right) \tag{22}$$

From these likelihood values, the n most likely edges will be chosen to have a sensor placed on them.

I. Stochastic Algorithm Pseudocode

See Algorithm 2 for the pseudocode implementation of the stochastic algorithm.

IV. Local Search Algorithm

Tabu search is used for the local search algorithm. Similar to the approach described above, this search method will be based off of Karger's algorithm [1].

A. Input Domain D_I

$$G\,(V, E, N_E, N_T) \to (D_I) \tag{23}$$

Where:

- V is the set of vertices
- E is the set of edges
- N_E is the set of edge vertices
- N_T is the set of target vertices

Additionally, some numerical value n which is the number of sensors will also be provided.

Algorithm 2 Stochastic Implementation

```
###SET OF CANDIDATES
Input G: Graph, NumSens: number; Verts: set of objects;
VertsEdge: set objects; VertsTarget: set of objects;
###OTHER INTERMEDIATE AND OUTPUT SETS
C: set of objects; Output S: set of objects
  ###INITIALIZATION
  S = empty; i = 0
  Initialize C
  Main Loop, while i ¡ 50
    Find and remove duplicate candidates in C
    ###EVALUATION
    Evaluate all members of C
    ###SELECTION
    Remove bool$(||c|| - 50)$ worst candidates
    ###CROSSOVER
    Crossover 10 randomly chosen members of C
    i = i + 1
  Endloop
  ###SOLUTION
  Select fittest $(c \in C)$ by $L(e \in C)$
  Determine likelihood of each edge in c
  return n most likely edges in c as S

  Func Initialize
    Randomly create 50 candidates and add them to C
    For each candidate c in C
      ###FEASIBILITY
      While bool$(||v|| \in G_R > 2)$
        ###REPAIR
        repair c
      Endwhile
      evaluate fitness$(c \in C) = \frac{\vartheta}{||c||}$
      Generate reduction graph of c
    Endfor
  Endfunc

  Func Repair
    Randomly choose allele in gene string of c
    If allele = 1
      change allele to 0
    Else if allele = 0
      move to first 1 value, change to 0
    Endif Endif
  Endfunc

  ###CROSSOVER
  Func Crossover
    new gene $= c_{p1}$ OR $c_{p2}$
    ###MUTATION
    mutate gene according to bool$(x < \mu(a))$
    generate reduction graph of c
    ###FEASIBILITY
    While bool$(||v|| \in G_R > 2)$
      ###REPAIR
      repair c
      recreate reduction graph
    Endwhile
    evaluate fitness$(c \in C) = \frac{\vartheta}{||c||}$
    add c to C
  Endfunc
```

B. Output Domain D_O

$$(E_s \subset E) \rightarrow (D_o) \tag{24}$$

The output will be some subset of edges in E such that they are the most likely (based upon the likelihood function defined below), of the edges that connect the vertices remaining in the reduced graph produced when stopping criteria are met.

C. Initialization

Upon ingestion of the graph file, several data structures are created:

- a hashmap of vertices
- a hashmap of edges
- a bit string representing whether an edge is in the graph

D. Neighborhood Function $N(S)$

The neighborhood will be generated by either adding or removing a certain edge from the current graph. Essentially, “flipping the bit” for that edge in the bit string. The action of adding or removing an edge is then assigned a value which considers the following items:

- Does the action merge or add nodes to the graph?
- How does the action affect the value of ϑ for each vertex in the graph?

Therefore, the value of a given neighbor is:

$$v(n) = \omega + \xi \tag{25}$$

Where:

- $\omega = \begin{cases} 1 & \text{if vertex is removed from graph} \\ 0 & \text{if no change in vertexes} \\ -1 & \text{if vertex is added to graph} \end{cases}$
- ξ is the average ϑ for all vertices in the graph

Whether a specific edge has been flipped recently will be stored in short-term memory. Once flipped, that edge will be required to persist in short-term memory for some number of iterations before it is allowed to be considered in the neighborhood again.

E. Feasibility Function

Check that a given neighbor does not exist in the tabu list.

F. Selection Function

Choose the neighbor with the highest value and set as the current solution.

G. Stopping Criteria

There will be two stopping criteria for this algorithm:

$$\text{bool}\left(||V \in G_R|| = 2\right) \tag{26}$$

- The number of vertices in the reduced graph G_R is 2

$$\text{bool}\left(\xi \text{ has not changed in 10 iterations}\right) \tag{27}$$

- This would indicate that despite changing the edges being included in the solution set, we are not achieving a greater amount of separation between N_E and N_T.

If either of these conditions are true, then the algorithm is stopped and the current set of edges remaining in G_R is returned for evaluation.

H. Evaluation Function

Once a solution set has been produced, the edges contained in that set need to have their respective likelihoods calculated. For this function, the same likelihood function from the stochastic implementation above will be used again:

$$L\left(e \in c\right) = \lambda_{v_{dest}} \left(\frac{\sum_{i=1}^{A_{v_{next}}} \lambda_i}{\sum_{i=1}^{A_{v_{next}}} i} \right) \tag{28}$$

After these likelihoods have been calculated, the n most likely edges are returned to the output domain.

I. Local Search Pseudocode

See Algorithm 3 for the pseudocode implementation of the local search algorithm.

V. Mapping to Chosen Computer Language

All algorithms described above will be implemented in Python. If not found with this document, implementations, along with pseudocode text files and sample graph files may be found at the associated GitHub repository: https://github.com/jarnm04/Sensor-Placement. Software was implemented and tested using a 2015 MacBook Pro with the following specifications:

- Model ID: MacBookPro12,1
- Processor: 2.9 GHz Dual-Core Intel Core i5
- Number of cores: 2 (only 1 used for these executions)
- Memory: 8GB
- Operating System: macOS Big Sur, Version 11.4
- Kernel Version: Darwin 20.5.0

VI. Test and Evaluation of Software Execution

For the sake of expedience, and also to contend with hardware limitations, during the initial testing of these algorithms, the sizes of the three test graphs that were used were kept relatively small. Table 1 shows the relevant information about each of the graphs that were used. Additionally, for each graph, it was assumed that only two Packet Capture Devices were available, thus, each algorithm should return 2 different edges that should be monitored. The results of each execution can be found in Table 2. Table 3 also contains some more relevant information about each of the graphs.

The most common trend between all algorithms, is that on all three graphs the algorithms indicated that an edge node or target node should be monitored. Every single edge selected for monitoring included at least one of the target or edges nodes in the graph. Across all three algorithms and all three graphs, there were a total of 18 edges selected for monitoring. Of those 18 edges, 8 were edges between two edge nodes or an edge node and a target node. This tendency to pick edge and target nodes seems to outweigh even the vulnerability scores of the individual nodes in the graph. It should be noted that in all three graphs, the only times the most vulnerable node in the graph was chosen for monitoring was also when it happens to be an edge or target. In fact, in Test Graph 3, the most vulnerable node 3, which is still a target node, is not chosen to be monitored by any of the three algorithms.

Algorithm 3 Local Search Implementation

```
###SET OF CANDIDATES
Input G: Graph; Verts: set of objects; NumSens: number;
VertsEdge: set objects; VertsTarget: set of objects;
###OTHER INTERMEDIATE AND OUTPUT SETS
curFittest: tuple(bit string, value); Output S: set of objects
###STOPPING CRITERIA
Split: bool; noImprovement: bool;
  ###INITIALIZATION
  S = empty; Split = false; noImprovement = false;
  S = initial solution
  Loop while (Split = false AND noImprovement = false)
    ###NEIGHBORHOOD FUNCTION
    neighborhood = empty set
    evaluate neighbors and add them to neighborhood
    ###SELECTION
    SNew = choose the neighbor with highest value
    ###FEASIBILITY
    $S_{new} \notin$ tabu
      If $v(S_{new}) > v($curFittest$)$
        update curFittest
      Endif
      S = SNew
    Else if SNew is in tabu list
      return to selection and find new SNew
    Endif Endif
    ###STOPPING CRITERIA
    check:
    bool($||V \in G_R|| = 2$)
    bool($\xi$has not change in 10 iterations)
  Endloop
  ###EVALUATION
  Determine $\forall e \in c, L(e)$
  add n most likely edges in curFittest to S
  return S
```

Execution time of the algorithms also tended to vary greatly. With execution times at least two full orders of magnitude below the other two algorithms for each of the graphs, it is clear that the deterministic approach is the fastest algorithm, at least when it comes to graphs of such a small size. It is very likely though that the observed trends in edge selection as well as execution time will vary greatly when using larger graphs, as well as by changing the number of available packet capture devices.

All code, pseudocode, and test graph files can be found on the GitHub repository for this project. Additionally, there are other graph files of varying sizes included in the repository, as well as a script to randomly generate new graphs in the

Graph	Num Nodes	Edge Nodes	Target Nodes
Graph 1	5	2	1
Graph 2	10	5	2
Graph 3	12	6	3

TABLE I
TEST GRAPH INFORMATION

Algorithm	Graph	Execution Time	Result
Deterministic Algorithm	Test 1	0.00064 sec	$(1 \rightarrow 5), (3 \rightarrow 5)$
	Test 2	0.00538 sec	$(1 \rightarrow 2), (2 \rightarrow 4)$
	Test 3	0.03780 sec	$(1 \rightarrow 12), (6 \rightarrow 7)$
Stochastic Algorithm	Test 1	0.18787 sec	$(2 \rightarrow 5), (3 \rightarrow 5)$
	Test 2	0.83509 sec	$(2 \rightarrow 3), (3 \rightarrow 4)$
	Test 3	2.40299 sec	$(2 \rightarrow 4), (2 \rightarrow 5)$
Local Search	Test 1	0.05869 sec	$(2 \rightarrow 5), (2 \rightarrow 3)$
	Test 2	0.84793 sec	$(2 \rightarrow 3), (3 \rightarrow 4)$
	Test 3	3.32476 sec	$(1 \rightarrow 2), (2 \rightarrow 4)$

TABLE II
TRIAL ALGORITHM EXECUTIONS

proper format for the scripts to ingest and a script to evaluate the run time execution of each of the algorithms.

VII. Conclusion, Comments and Future Work

The issue of determining where to place packet capture devices on a network is one that still plagues Cyber Defense Teams today. Presently, this task is done by the intuition of the members of the team, however, by using the known network data, the optimal sensor placement should be able to be derived algorithmically. The algorithms proposed in this paper are a step toward that, however, there is quite a bit of work that can and should be done on the algorithms herein to make them actually usable in a real world scenario.

The first thing that can be done would be to refine the coded implementation of the algorithms as they are. There are likely more efficient executions of these processes that can be attained, and by writing these algorithms in a compiled programming language, such as C++ or Rust, there could be huge leaps in execution time over the scripted Python implementation tested here. Additionally, there are likely inefficiencies in the actual algorithms themselves that can be addressed through the use of more sophisticated techniques and heuristics.

Finally, to make this approach applicable in the real world, it will be necessary to better model a computer network. The graphs used here were randomly generated, and did not contain a structure similar to that used by actual networks. In the future, actual network packet captures should be attained and analyzed to generate a much more realistic network graph. Additionally, the vulnerability score of each node on the network could be refined quite a bit further. In this example I only considered the number of known vulnerabilities in the score, whereas other factors such as Operating System, system uptime, system usage and many others all play an important role in determining the likelihood of a machine actually being exploited by malicious actors.

VIII. Acknowledgements

Pseudocode format derived from [6]. Standard search elements, algorithm and reporting structure derived from [7].

Graph	Targs	Edges	Most Vuln	Least Vuln
Test 1	3	4, 5	3	2
Test 2	3, 6	1, 2, 5, 8, 9	5, 6, 7	9
Test 3	3, 4, 9	2, 6, 7, 10, 11, 12	3	12

TABLE III
TEST GRAPH INFORMATION

References

[1] Arora, S. (2013). Karger's Min Cut Algorithm. https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf
[2] Griffin, T., Shepherd, F. B., & Wilfon, G. (2002). The Stable Paths Problem and Interdomain Routing. IEEE/ACM Transactions on Networking (TON), 10(2), 232–243. https://dl.acm.org/citation.cfm?id=508332
[3] Chekuri, C., & Blatti, C. (2009). The Multiway Cut Problem (Vol. 3111). https://doi.org/10.1007/978-3-540-27810-8_24
[4] Donnelly, K., Kfoury, A., & Lapets, A. (2010). The complexity of restricted variants of the stable paths problem. Fundamenta Informaticae, 103(1–4), 69–87. https://doi.org/10.3233/FI-2010-319
[5] Sabharwal, S. (2020). Pivoting - Moving Inside a Network (Cyber Security). GeeksforGeeks. https://www.geeksforgeeks.org/pivoting-moving-inside-a-network/
[6] Talbi, E.-G. (2009). Metaheuristics, From Design to Implementation. Wiley.
[7] Barr, R. S., Golden, B. L., Kelly, J. P., Resende, M. G. C., & Stewart, W. R. (1995). Designing and reporting on computational experiments with heuristic methods. Journal of Heuristics, 1(1), 9–32. https://doi.org/10.1007/BF02430363
[8] Christofides, N. (1975). Graph Theory: An Algorithmic Approach. Academic Press Inc.