Check for updates

ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3638529.3654035

RESEARCH-ARTICLE

Optimizing Cyber Response Time on Temporal Active Directory Networks Using Decoys

HUY NGO, The University of Adelaide, Adelaide, SA, Australia

MINGYU GUO, The University of Adelaide, Adelaide, SA, Australia

HUNG NGUYEN, The University of Adelaide, Adelaide, SA, Australia

Open Access Support provided by:

The University of Adelaide

PDF Download
3638529.3654035.pdf
01 March 2026
Total Citations: 4
Total Downloads: 569

Published: 14 July 2024

Citation in BibTeX format

GECCO '24: Genetic and Evolutionary Computation Conference
July 14 - 18, 2024
VIC, Melbourne, Australia

Conference Sponsors:
SIGEVO

https://doi.org/10.1145/3638529.3654035
ISBN: 9798400704949

Optimizing Cyber Response Time on Temporal Active Directory Networks Using Decoys

Huy Q. Ngo
The University of Adelaide
Adelaide, Australia
quanghuy.ngo@adelaide.edu.au

Mingyu Guo
The University of Adelaide
Adelaide, Australia
mingyu.guo@adelaide.edu.au

Hung X. Nguyen
The University of Adelaide
Adelaide, Australia
hung.nguyen@adelaide.edu.au

ABSTRACT

Microsoft Active Directory (AD) is the default security management system for Window domain network. We study the problem of placing decoys in AD network to detect potential attacks. We model the problem as a Stackelberg game between an attacker and a defender on AD attack graphs where the defender employs a set of decoys to detect the attacker on their way to Domain Admin (DA). Contrary to previous works, we consider time-varying (temporal) attack graphs. We proposed a novel metric called response time, to measure the effectiveness of our decoy placement in temporal attack graphs. Response time is defined as the duration from the moment attackers trigger the first decoy to when they compromise the DA. Our goal is to maximize the defender's response time to the worst-case attack paths. We establish the NP-hard nature of the defender's optimization problem, leading us to develop Evolutionary Diversity Optimization (EDO) algorithms. EDO algorithms identify diverse sets of high-quality solutions for the optimization problem. Despite the polynomial nature of the fitness function, it proves experimentally slow for larger graphs. To enhance scalability, we proposed an algorithm that exploits the static nature of AD infrastructure in the temporal setting. Then, we introduce problem-tailored repair operations, ensuring the convergence to better results while maintaining scalability for larger graphs.

CCS CONCEPTS

• **Security and privacy → Network security**; • **Computing methodologies → Randomized search**.

KEYWORDS

Active Directory, Network Security, Decoy Placement, Evolutionary Diversity Optimization, Stackelberg Game

ACM Reference Format:
Huy Q. Ngo, Mingyu Guo, and Hung X. Nguyen. 2024. Optimizing Cyber Response Time on Temporal Active Directory Networks Using Decoys. In *Proceedings of The Genetic and Evolutionary Computation Conference 2024 (GECCO '24).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3638529.3654035



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.

GECCO '24, July 14–18, 2024, Melbourne, Australia
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0494-9/24/07…$15.00
https://doi.org/10.1145/3638529.3654035

1 INTRODUCTION

Active Directory is Microsoft's identity and access management system designed for Windows domain networks. It's widely adopted and plays a critical role in the networks of many enterprises and government bodies. However, its popularity has also made it a prime target for many cyber adversaries over the years. According to a report from Microsoft [15], there has been an alarming surge in attacks targeting AD users, with 30 billion attempted password attacks on AD accounts reported each month in 2023.

An Active Directory network naturally describes an attack graph, with **nodes** representing both physical and virtual entities such as users, computers, security groups, etc., and **directed edge** (i, j) representing the vulnerability and accesses that an attacker can exploit to gain access from node i to node j. BloodHound [20] is one of the most influential tools for analysing/visualising the AD attack graph. BloodHound models the *identity snowball attack*, a concept initially introduced by Dunagan et al. [8]. The identity snowball attack models the sequence of attack in the network allowing them to gain access of higher privileges nodes from a low privilege node (ex. Account A $\xrightarrow{\text{AdminTo}}$ Computer B $\xrightarrow{\text{HasSession}}$ Account C). However, Dunagan et al. [8] and several other works on defending Active Directory network [9–11, 13, 25] over-simply the problem with the assumption that AD network is static. In practice, the AD graph is very dynamic which will effect the security landscape overtime. One of the major sources of changes in the AD graphs is caused by users' activities. In Windows systems, user authentication leaves behind credential material, typically in the form of a hash or clear-text password in the computer's memory. Adversaries can exploit this vulnerability, harvesting credentials for lateral movement. In the BloodHound, this vulnerability is presented as 'HasSession'. HasSession edges will stay online until being removed from the graph when the user signs off from the computer after a period of time. In this work, we formally model the dynamics of the AD graph using the **temporal attack graph**, wherein attackers gain access to nodes in the AD graph through the *identity snowball attack*, presented as *temporal paths*. For example, in Figure 1, the identity snowball attack in temporal graph for gaining access of account U_3 from compromised node s_2 can be the following temporal path: $s_2 \xrightarrow{1} Cp_1 \xrightarrow{2} Cp_3 \xrightarrow{4} U_3$ where number on each arrow is time the attacker exploit the edge to gain the access to next node. The static attack graph can not capture this attack path if generated at time steps $t \in [1, 4) \cup (6, 10]$

In this paper, we study a method for defending temporal AD attack graph by using active defense with cyber decoys. Decoys or honeypots [6, 14] are fake assets such as fake users, and fake hosts that trigger an alert when attackers engage them. They are designed

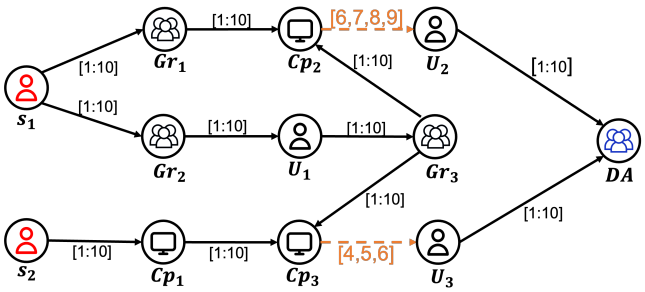


Figure 1: Example of an Active Directory graph sampled over a period of 10 time units. The timestamps on each edge indicates its appearance time. Black labels represent static edges, while orange labels denote dynamic edges (HasSession).

to lure attackers to them by mirroring authentic assets. By allocating decoys in strategic locations, they can serve as the sentinel for the early detection of the cyber threat. An early detection of a threat can increase the effectiveness of incident response process of IT admin and reduce the further damage caused by the attack [2]. Motivated by the early detection use case of the network decoy, we proposed a problem for decoy allocation in temporal network called *maxRT*. In this problem, defender aims to optimize the **response time** of the decoy to any attack path. The response time is defined as the duration from the moment attacker triggers the first decoy to when they can reach DA. Defender want to maximize the response time to ensure the early detection, providing IT admin with sufficient time to respond to the incident before the attacker reaches the DA.

We model our problem as a two-player Stackelberg game model with a pure strategy. In our game, the defender (leader) wants to allocate at most b decoys on a set of *blockable* nodes. The defender's allocation intercepts the attacker's temporal attack path while maximizing the response time of the allocation to ensure early detection. The attacker (follower) has access to a set of compromised entry nodes. The attacker can also observe the entire temporal AD graph and the defensive strategy. This assumption is based on the practicality of attackers employing reconnaissance tools similar to Sharphound [22] to collect data from domain controllers and build an AD attack graph. The attacker's strategy specifies an entry node, and from it, a temporal attack path to DA. The attacker's best response is to choose a temporal attack path that has the minimum response time. We will later show that the attacker's optimal plan can be found in polynomial time.

Our Contribution. This paper aims to propose a solution for the decoys allocation in Active Directory problem. We first prove $\mathcal{NP}$-hard nature of the defender's combinatorial optimization problem. We then introduce the Evolutionary Diversity algorithm as a heuristic solver. However, the vanilla EDO algorithm does not scale well for our problem when it fails to converge to any feasible solution (response time > 0) in some graphs. When we mention "vanilla" EDO algorithm, we refer to directly applying the EDO implementation from Goel et al. [9, 10] to our problem. In an attempt to run the vanilla EDO algorithm on the ADX10 graph in our experiment, the response time of the solution remains 0 even after 2 million iterations (equivalent to almost 3 days of computational effort). To enhance our algorithm, we propose several improvements. Firstly, the computation of the attacker's optimal path relies on the earliest-arrival path, which is computationally slow in AD graphs. We observe an contradiction that the state-of-the-art algorithm for computing the earliest-arrival path becomes highly inefficient in temporal graphs with a large number of static edges. Despite the dynamic characteristics of the AD graph, a significant portion of the AD infrastructure remains static. To address this problem in AD-specific graphs, we present a novel Dijkstra-based algorithm for computing the earliest-arrival path which significantly improves the run-time of the fitness function. Secondly, we introduce two constraint-handling techniques to tackle the difficulty of finding feasible solutions in the vanilla EDO. The first method introduces a repair mechanism using Integer Linear Program (ILP) to directly patch the infeasible solution every round. The second approach introduces the surrogate/penalty fitness function. The surrogate function is a lightweight fitness function that replaces the computationally expensive real fitness function, allowing the evaluation of individuals at a lower cost during each iteration. The surrogate function is designed to evaluate a solution on a set of "important" attack paths instead of the whole graph and penalize the infeasible individuals. We experimentally verify that our proposal effectively improves the scalability of the EDO algorithms on our problem.

2 MODEL DESCRIPTION

2.1 Background

Temporal directed graph define as $G = (V, E_1, \cdots, E_{t_{max}}) = (V, E = (E_i)_{i\in[t_{max}]})$ where V is a set of vertices in the graph and E_i is the set of edges at time i. We denote the tuple $(u, v, t) \in E_t$ the edge from u to v appears at time t. For the sake of model simplicity, we assume that every edge has a duration of 1. I.e, if an attacker traverse edge (u, v, t), they will start from node u at time t and arrive v at time $t + 1$. Despite this simplification, all our algorithms remain effective in more general settings (traverse duration > 1). We call t_{max} the lifetime of the graph. We also define the **underlying graph** of graph G as $G_\downarrow = (V, E_\downarrow)$ where $E_\downarrow = \bigcup_{t=1}^{t_{max}} E_t$. For the ease of demonstration in the paper, we also denote $time_label(u, v) = (t_i)_{i=1}^{k}$ as a (ascending) sorted list of time units that edge (u, v) appears (or is on) if $(u, v, t_i) \in E_i$ where $t_i \in time_label(u, v)$. Otherwise, we say edge (u, v) disappears or is off at time step t_i if $t_i \notin time_label(u, v)$. We denote a set of **static edges** as E_s, we say an edge $(u, v) \in E_s$ is static if they appear in every time step throughout the graph's lifetime or $|time_label(u, v)| = t_{max}$. Similarly, we denote set of **dynamic edges** as E_d, we say an edge $(u, v) \in E_d$ is dynamic if they disappear from the graph for some of the time units or $|time_label(u, v)| < t_{max}$.

Temporal (s, d)**-path** is defined as a sequence of edges in graph G exhibiting a monotonic increase in edge labels. For any two distinct nodes $s, d \in V$, the temporal path between s and d is presented by: $\pi = \pi(s, d) = \langle(s = v_0, v_1, t_1), (v_1, v_2, t_2), \ldots, (v_{k-1}, v_k = d, t_k)\rangle = \langle(v_{i-1}, v_i, t_i)\rangle_{i=1}^{k}$ where $v_i \neq v_j$ and $t_i < t_j$ for all $i, j \in \{0, \ldots, k\}$ with $i \neq j$. We denote $start(\pi) = t_1$ and $end(\pi) = t_k + 1$ as the **starting time** and **ending time** of a path $\pi(s, d)$. We further

denote by $dur(\pi(s,d)) = end(\pi(s,d)) - start(\pi(s,d))$ the duration of travelling from the starting vertex to the ending vertex of the path $\pi(s,d)$. Next, we define a set of every possible temporal path from s to d between interval $[t_\alpha, t_\omega]$ as $\Pi(s,d,[t_\alpha,t_\omega]) = \{\pi : \pi$ is a (s, d)-temporal path such that $start(\pi) \geq t_\alpha, end(\pi) \leq t_\omega\}$. Then, a path $p \in \Pi(s,d,[t_\alpha,t_\omega])$ is an **earliest-arrival path** if $end(\pi) = min\{end(\pi') : \pi' \in \Pi(s,d,[t_\alpha,t_\omega])\}$.

Temporal (s,d)**-cut**, or temporal (s,d)-separator, refers to the set of nodes $C(s,d)$ in the graph G that the removal of every node in set $C(s,d)$ will disconnects all temporal paths from s to d. *It is essential to note that in this paper, the terms "cut" or "separator" specifically refers to the allocation of decoy on vertices.* When we employ a temporal (s,d)-cut the graph, we guarantee every (s, d) temporal path has a contact with the cut C. Given a path $\pi(v_0,v_k) = \langle(v_{i-1},v_i,t_i)\rangle_{i=1}^{k}$ in graph G and a cut $C(v_0,v_k)$, we define a node v as the **first point of contact** between the path $\pi(v_0,v_k)$ and the cut $C(v_0,v_k)$ if $v \in I : \forall u \in I, \text{dur}(v_0,v) \leq \text{dur}(v_0,u)$ where $I = V(\pi(v_0,v_k)) \cap C(v_0,v_k)$. In short, the first point of contact represents the first honeypot encountered when following the path.

Response time denoted as RT is a key parameter introduced in this paper for our specific problem. The response time of a path π is defined as the duration between the moment when the attacker encounters or triggers the first honeypot and the time when the attacker compromises the Domain Admin while following path π. Let's us consider the temporal path $\pi(s,DA) = \langle(s = v_0,v_1,t_1),\ldots,(v_{k-1},v_k = DA,t_k)\rangle = \langle(v_{i-1},v_i,t_i)\rangle_{i=1}^{k}$ with v_x where $1 < x < k$ is the first point of contact of $\pi(s,DA)$ and the defense solution $C(s,DA)$. The response time of path $\pi(s,DA)$ is defined as $RT(\pi,C) = dur(\pi(s,DA)) - dur(\pi(s,v_x)) = t_k - t_x$ As a defender, we want to maximize the response time of every temporal path in the attack graph to let IT admin have enough time to react to the incident.

Example 2.1. Figure 1 illustrates a temporal Active Directory graph. The graph includes two compromised users, denoted as s_1 and s_2. The graph consists of two sets of edges: static edges, allowing the attacker to move between nodes at every time step, and dynamic edges, which appear for a limited time. In this example, we assume the defender allocates honeypots to a set of nodes $C = \{Cp_2, Cp_3\}$. Consider the following temporal path $\pi = \langle(s_1,Gr_1,2),(Gr_1,Cp_2,4),(Cp_2,U_2,6),(U_2,DA,7)\rangle$ from S to DA. Assuming the attacker from s_1 chooses this path, the honeypot on node Cp_2 is triggered at time 4 (as the attacker steps on it), alerting the IT admin to the attacker's presence. In this context, node Cp_2 is considered as the first point of contact for the attacker. The response time, defined as the time from honeypot alert to the attacker compromising the DA, is $RT = dur(\pi(s_1,DA)) - dur(\pi(s_1,Cp_2)) = (7+1-2) - (4+1-2) = 3$ units (plus 1 due to the assumption that traversing each edge takes 1 time unit). During this window, the IT admin has 3 time units to isolate compromised systems and terminate the attacker's unauthorized session.

2.2 Problem formulation

Temporal directed attack graph We define an AD attack graph in our model as a Temporal directed graph $G = (V, E_1, \cdots, E_{t_{max}})$. Set of vertices V represents all physical and virtual entities such as user, computer, security group, etc. The set of edge E_i denotes the link modelling the security dependency and relationships between entities which represent vulnerabilities for attacker to make lateral movements. There is a set $S \subseteq V$ of initial footholds called entry vertices, and the attacker has already compromised these vertices at the start of the attack. The attack goal is to compromise the Domain Admin (DA), the attacker can laterally move through the network using any of the **temporal (s, DA)-path**.

Formulation with game theory The problem of defending a temporal AD network with honeypots can be modelled as a Stackelberg game. In our proposed model, the defender can deploy a set of honeypots on a set of vertices C (a cut) such that form a temporal (S,DA)-cut. In our model, each honeypot will "monitor" any malicious activities on their allocated vertices. The honeypots will set an alert to IT admin once the attacker steps on one of these vertices. Defender can only allocate honeypot on a set of blockable vertices, denoted by $N_b \subseteq V$. In consideration of a worst-case scenario, we assume the attacker has full visibility into the temporal graph and the honeypot placements. The attacker can bypass these honeypots if the honeypot's placement does not form a temporal (s,DA)-cut, which in this case, the response time is 0. Consequently, when the budget of the defender problem is exactly the size of the minimum temporal cut, our problem's solution is also the solution for the minimum temporal (s,d)-separator problem [26] which is known to be a $\mathcal{NP}$-complete problem. However, our problem goes beyond this by also maximizing the response time of the temporal cut which tends to "push" the solution further away from the DA. Generally speaking, nodes further away from the DA tend to be lower privilege nodes instead of servers or admin. Therefore, our solution incurs lesser disruption to the network. We say C is a defender's **feasible solution** if C is strictly a temporal (S,DA)-cut, otherwise, it is a **infeasible solution**. Strategically, when facing a defence solution C, the attacker selects a path that minimizes the response time. The **attacker optimal attack** path can be found via $\min_{\pi\in\Pi} RT(\pi,C)$ where Π is the set of every possible temporal path between each vertex $s \in S$ to DA. In contrast, the defender aims to find a cut C that maximize the response time. The **defender's objective** is formulated as

$$\max_{C\subseteq N_b, |C|<b} \{\min_{\pi\in\Pi} RT(\pi,C)\}. \tag{1}$$

Theorem 1. Defender's problem is $\mathcal{NP}$-hard.

Due to space constraints, detailed proofs of every theorems in this paper is provided in the extended version [17].

3 RELATED WORK

Identity Snowball Attack in dynamic environment. Ngo et al. [16] also study the honeypot/decoys allocation on Active Directory network with the consideration of the dynamic setting. However, their approach to modelling dynamism is somewhat simplistic. They capture the dynamic nature by taking independent static snapshots of the attack graph at each time step, treating each snapshot as an attacker's scenario in a static graph. Their allocation strategy jointly optimizes the number of attack paths in each snapshot. Ngo et al. [16] fail to model the identity snowball attack in the temporal graph. In practical scenarios, attackers can patiently "lurk" in a node until a more opportune path emerges. This characteristic makes

our model more sophisticated and practical than theirs. Albanese et al. [1] attempted to model the credential hopping attacks/identity snowball attacks on the time-varying user-computer graph. They assume that attacker does no observation on the network topology and employ a heuristic algorithm to find the upper-bound of the attacker attack effort. In contrast, our work considers the worst scenario where attacker have the observation on the attack graph and we can derive the optimal attack response. Pope et al. [19] also consider a similar model to Albanse et al. except they employ genetic programming to predict the attacker success rate/effort. We highlight that none of these works considers the temporal graph for modelling the dynamic of AD graph.

Active Directory. The seminal work by Dunagan et al. [8] was the first to study the defense problem in Active Directory through edge-blocking by introducing the heuristic edge-blocking algorithm. Follow-up researches on the edge-blocking optimization problem includes Guo et al. [11] proposed an optimal edge-blocking strategy using Fixed-Parameter Tractable algorithms; [13, 25] improved scalability through Mixed-Integer Programming and the Double Oracle algorithm; Goel et al. [9, 10] proposed the Evolutionary Diversity Optimization (EDO) algorithm to defend against attackers in a configurable environment; and Guo et al.[12] studied optimal edge-blocking problem with minimal human input. Another approach for defending Active Directory found in the literature involves node-blocking, which abstracts the concept to decoy allocation. Ngo et al. [16] are the first to study the honeypot allocation problem for defending Active directory where they proposed MIP algorithm to solve the problem.

Evolutionary Diversity Optimization [21] is a recent branch of Evolutionary Computation. EDO is designed to identify a set of solutions that is both high-quality and structurally diverse. There have been considerable efforts exploring the EDO algorithm for various combinatorial problems, including the travelling salesperson problem [3, 7, 18], minimum spanning tree problem [5], knapsack problems [4], and more. Among these studies, the work of Goel et al. [9, 10] is particularly relevant to our research. Goel et al. consider the edge-blocking problem against attacker in AD graph where edges are associated with a failure rate and detection rate. They deploy a neural network/reinforcement learning to approximate the attacker's strategy and apply EDO algorithm to solve the defender problem. In our study, our EDO algorithm draws inspiration from Goel et al. [10], including the design of the mutation/crossover operator and diversity measure strategy. However, our experimental findings reveal that the vanilla EDO algorithm performs poorly when directly applied to our specific problem.

4 PROPOSED METHODOLOGY

4.1 Game-theoretical rational attacker

In our model, the game-theoretical rational/optimal attacker will choose the attack path that has the minimal response time. We illustrate such paths using the following example from Figure 1. We assume the defender allocates honeypots on node set $C = \{Cp_2, Cp_3\}$. Starting from the entry node s_1, let's examine two potential attack paths: $\pi_1 = \langle (s_1, Gr_1, 1), (Gr_1, Cp_2, 2), (Cp_2, U_2, 6), (U_2, DA, 7)\rangle$ and $\pi_2 = \langle (s_1, Gr_1, 1), (Gr_1, Cp_2, 5), (Cp_2, U_2, 6), (U_2, DA, 7)\rangle$. The difference between these 2 paths lies in the departure time of exploiting the second edge (Gr_1, Cp_2). After exploiting the first edge (s_1, Gr_1) at time 1, the attacker has 2 options: either immediately exploit the next edge at time 2 (π_1) or wait until time 5 to continue (π_2). Despite both paths leading to the attacker reaching DA at time 7, the attacker is more "troublesome" if they opt for π_1. This is because the decoy only identifies them at time 5 ($RT = 2$) for path π_1, whereas for path π_2, the attacker is detected at time 2 ($RT = 5$), providing the defender with significantly more time to respond to the incident. π_1 in this example is actually the worst-case/optimal attack path.

Algorithm 1 for finding such paths can be described as follows. Let's consider an attack graph G and a defender's honeypot allocation $C \in V$. We define a tuple (π_1, c, t_c), where π_1 represents a temporal path, c is a node in C, and t_c is a time. Firstly, for each node $c \in C$, we verify if it is reachable from any of the entry nodes $s \in S$ at time t_c in a graph $G' = ((V \setminus C) \cup c, E)$ (line 4) —here, we remove all nodes in C except node c (line 2). The condition in line 4 ensures the *worst-case* condition of the optimal attack path. If we can reach node c from S at time t_c using path π_1, we then find the earliest-arrival path π_2 from c to DA within the interval $[t_c, t_\omega]$ (line 5-6). We add the tuple (π_1, π_2) to Ψ (line 7). Next, for every tuple $(\pi_1, \pi_2) \in \Psi$, we merge 2 path to form a temporal (s, DA)-path $\pi = \pi_1 + \pi_2$. We identify the tuple with the smallest duration $dur(\pi_2)$, *the duration of the earliest-arrival path π_2 is actually the response time for the attack path $\pi = \pi_1 + \pi_2$* (line 8). Therefore, the optimal attack path π_{OPT} is the one where the π_2 sub-path has the smallest duration. The fitness function giving a defender solution C can be defined as:

$$f(C) = \begin{cases} \min_{\pi \in \Pi} RT(\pi, C), & \text{if } C \text{ is feasible (temporal cut).} \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

Algorithm 1 Algorithm for Computing Optimal Attack Path

Input: Temporal graph G, set of source nodes S, set of honeypot C, destination node DA, time interval $[t_\alpha, t_\omega]$
Output: Optimal attack path π

```
1: foreach c ∈ C do
2:    Remove nodeset C \ c from graph G
3:    foreach t ∈ [t_α, t_ω] do
4:       if c can be reached from any s ∈ S at time t do
5:          Store the path used to reach c by time t to π_1
6:          π_2 ← compute_earliest_arrvl_path(G, c, DA, [t, t_ω])
7:          Add (π_1, π_2) to Ψ
8: π = arg min_{(π_1,π_2)∈Ψ} dur(π_2)
9: return π
```

For computing earliest-arrival path subroutine (line 6) we can use the state-of-the-art algorithm proposed by Wu et al. [23] which has been proven to be time-polynomial. This makes computing attacker optimal attack plan time polynomial. Despite this, Wu's algorithm is inefficient when running on AD-specific graph, slowing down the computation of the optimal attack plan. We will discuss this

issue in the next section and propose a more efficient approach for calculating the earliest-arrival path.

4.2 Faster computation for earliest-arrival path

As outlined in Section 4.1, the response time of an attack path is determined by the duration of the earliest-arrival path from an initial point of contact to the DA. Therefore, the computation of optimal attack for fitness function required the call of computing the earliest arrival path subroutine. The first candidate algorithm that we use for computing the earliest arrival path in our implementation is Wu's algorithm [23]. In [23], the author explored the computation of minimal paths in temporal graphs, including the earliest-arrival path. Wu et al. introduced a one-pass algorithm for computing the earliest-arrival path, which stands as one of the state-of-the-art algorithms for this task. Wu's algorithm generates a set of edge streams, a chronological sequence of all edges E ordered by the time at which the edge is collected. The algorithm scans through the edge stream, greedily updating the earliest arrival time at each node that satisfies the arrival condition. This process required the duplication of every static edge to correctly update the earliest arrival time which explain the contradictory of the inefficiencies of Wu's algorithm in AD-specific temporal graph. In general, Wu's algorithm poses inefficiencies when applied to graphs with a substantial number of static edges as Wu's algorithm requires the scan of every edge in E. In practice, while the AD graph exhibits dynamic characteristics, a significant portion of the AD infrastructure remains static. For instance, in a snapshot taken from the University of Anonymous on 13/10/2021 at 02:00 pm, a total of 1,151,962 relationships (edges) were identified as online at that time while only 4,039 of these edges were the HasSession edges, which are deemed as the primary source contributing to the dynamism of the AD graph.

Our proposed approach utilises the Dijkstra's edge scanning strategy which allows us to perform the scan only on the underlying edges $E_{\downarrow}$. The intuition behind this algorithm lies in using the Dijkstra greedy scanning strategy, which scans through each underlying edge only once to expand the earliest-arrival paths. The pseudocode is given in Algorithm 2. The idea of using Dijkstra for finding earliest-arrival path has been proposed in [24]. However, we further enhance the runtime on graphs with numerous static edges by introducing a conditional statement between lines 11-14 in Algorithm 2

The correctness of the Dijkstra Greedy Strategy for computing the earliest-arrival path is provided in Theorem 2. In the general case, the time complexity of Wu et al.'s algorithm can be expressed in our notation as $O((\varepsilon_s+\varepsilon_d)\cdot t_{max})$, whereas the time complexity of our proposed algorithm is $O((\varepsilon_s+\varepsilon_d \cdot t_{max}) \log(|V|))$. In scenarios where the number of static edges $\varepsilon_s = |E_s|$ outweighs the number of dynamic edges $\varepsilon_d = |E_d|$, our algorithm demonstrates more efficient runtime, as theoretically presented in Theorem 3.

Experimentally, when we use these algorithms to find the earliest path from every source to every node in graph $ADX10$ (section), while Wu's algorithm takes 18.370 seconds to complete the task, Dijkstra Greedy's runtime is only about 3.389 seconds (5x faster).

Theorem 2. Algorithm 2 correctly compute the earliest-arrival path from a source vertex x to every vertex $v \in V$ within a given interval $[t_\alpha, t_\omega]$ with the complexity of $O((\varepsilon_s + \varepsilon_d \cdot t_{max}) \cdot \log(|V|))$

Theorem 3. When $\varepsilon_s \gg \varepsilon_d$, the complexity of Dijkstra-based algorithm become $O(\varepsilon_s \cdot \log(|V|)$ while complexity of Wu's algorithm become $O(\varepsilon_s \cdot t_{max})$

Algorithm 2 Dijkstra-based algorithm for Computing Earliest-Arrival Time

Input: Temporal Graph G, source nodes S, time interval $[t_\alpha, t_\omega]$
Output: The earliest-arrival time from every source nodes $s \in S$ to every vertex

1: $PQ = Priority_Queue$
2: $INSERT_{PQ}(t_i, s), \forall s \in S$
3: $seen[s] = t_i, \forall s \in S$
4: $arrvl_time \leftarrow empty\ dictionary$
5: **while** $PQ \neq \emptyset$ **do**
6: $(t_u, u) \leftarrow POP_MIN_{PQ}()$
7: **if** u in $arrvl_time$ **do**
8: continue
9: $arrvl_time[u] = t_u$
10: **for** successor v of u **do**
11: **if** (u, v) is a *static edge* **then**
12: $v_arrvl \leftarrow t_u + 1$
13: **else then**
14: $v_arrvl \leftarrow min\{t : t \in time_labels(u, v), and\ t > t_u\}$
15: **if** v in $arrvl_time$ **then**
16: **continue**
17: **elseif** v not in $seen$ or $v_arrvl < seen[v]$ **do**
18: $seen[v] \leftarrow v_arrival$
19: $INSERT_{PQ}(v_arrvl, v)$
20: **return** $arrvl_time$

4.3 EDO Algorithm for max-*RT*

In this section, we discuss the application of the Evolutionary Diversity Optimization (EDO) algorithm within our problem context. The pioneering work of Goel et al. [9, 10] introduced the EDO technique for addressing the edge-blocking problem in AD attack graphs. We initially applied Goel's EDO algorithm to our scenario. In our problem, the defender employs EDO to acquire a diverse set of defensive plans denoted as C, where the fitness function $f(C)$ can be obtained by computing the optimal attack plan. Let's define P as the population of defensive solutions. An individual $p \in P$ is defined as the binarization of solution C where each individual has a length of $|N_b|$, with 1 signifying the decision to block the corresponding node and 0 implying no blocking.

We initiate the process by generating a random population P of defensive solutions. An individual p is randomly selected from P to undergo either mutation or crossover, each with a probability of 0.5. The number x of mutated bits in the offspring is chosen randomly based on a Poisson distribution. For **mutation**, we randomly select an individual p' from P and flip x random bits, changing 0s to 1s and 1s to 0s. For example, if we choose $p' = \langle 1, 0, 1, 1, 0, 1\rangle$ from P and $x = 2$, the resulting offspring could be $p = \langle 0, 1, 0, 1, 1, 1\rangle$. For **crossover**, we again randomly select two parents p' and p'' from

P. We identify x coordinates where p' has 0s and p'' has 1s, and flip the bits at those coordinates on both p' and p''. Similarly, we identify x coordinates where p' has 1s and p'' has 0s, and flip the bits at those coordinates. After having the offspring using mutation and crossover operation, we add the new offspring to the population only if their fitness score is close to the best fitness score of the population and reject the individuals that contribute the least to the diversity of the population. We follow the **diversity measure** of population implementation of [9, 10]. But to summarise our diversity measure aims to maximise the diversity of "unique" nodes in the population. Let $Cnt_P(v_i)$ be the function that counts the number of individuals in population P that contain v_i. We say that v_i is more "unique" to the population if they have a lower $Cnt_P(v_i)$ score.

Our preliminary investigation of the EDO algorithm revealed that most of generated offspring solution are infeasible. This challenge arises due to the expansive nature of the defender solution space ($\binom{|N_b|}{b}$ combinations), which makes it difficult to generate feasible solutions using conventional evolution operators alone. Another challenge with the vanilla EDO algorithm is its requirement to execute the full fitness function. Although we have presented that the fitness function can be computed in polynomial time and pushed the runtime frontier by proposing a modification of Dijkstra-based for computing earliest-arrival paths. The algorithm execution time remains slow for larger graphs. In the following section, we will explore two constraint-handling techniques that we propose to enhance convergence to feasible solutions and improve the algorithm's runtime efficiency.

4.4 Constraint-Handling Evolutionary Algorithm

In this section, we shall introduce two constraint-handling approaches for our EDO algorithm.

4.4.1 Integer Programming repair operator. In this proposal, we aim to address the issue of infeasible offspring directly by employing a problem-specific *repair* operator. The repair mechanism involves solving an Integer-Programming (IP) to "patch" the cutting solution. The complete algorithm can be describe as following. Suppose we encounter an infeasible offspring, denoted as p after the mutation or crossover. For each blocked node, excluding those that have undergone a state change during the mutation or crossover, we probabilistically unblock them (i.e., change 1s to 0s) with a probability of 1/2. The purpose of this unblocking operation is to reserve additional space for the subsequent repair process and fulfill the cardinality condition. Finally, we solve our problem-specific IP repair operator. *Due to space constraints, we provide the detailed ILP formulation in the extended version [17].* The ILP is formulated on the idea that if any node i is connected to j via edge (i, j, t), and if j can reach DA at any time before t, then i can also reach DA at every time after $t + 1$.

While the repair operator ensures convergence to a feasible solution in each iteration, it is worth noting that this approach is very memory costly. The IP requires $O(|V| \cdot t_{max})$ variable and upto $O(\varepsilon \cdot t_{max} + |V|)$ constraints, which can become exponentially large for certain graphs. Additionally, solving the IP itself is known to be a $\mathcal{NP}$-hard problem. Experimentally, the bottleneck of this approach is the IP-solving process, and it is difficult to run on large graphs due to out-of-memory issues.

4.4.2 Surrogate-assisted and penalty-based repair operator. From our experiment, we observed that employing the full fitness function on the entire graph in each iteration proves to be very costly, especially for large graphs. Additionally, when using the vanilla EDO algorithm, we encountered difficulties as the solution failed to converge towards feasibility.

To tackle the challenges mentioned earlier, we propose Algorithm 3. The core concept behind Algorithm 3 is to evaluate the population on a lightweight surrogate fitness function in every iteration instead of the inefficient complete fitness function (2). The complete fitness function required to run Algorithm 1 on the whole graph. Our idea for design is that we only need to focus on a set of "important" paths that are likely to have the most impact on the evaluation, instead of inefficiently spending time on the entire graph. We will have two separate sets of populations in our algorithm namely global population P_{global} and local population P_{local}. The local population is evaluated every iteration by the surrogate fitness, while the global population is only evaluated by the complete fitness function when a specific condition is met. Let Φ be the set of "important" temporal (s, DA)-path for the surrogate function. We initialize the set Φ by adding a random set of temporal paths in graph. Then we iteratively improve the function by adding to Φ the most up-to-date optimal attack path by the attacker when facing the current population. Our experimental results demonstrate that the surrogate function eventually becomes as effective as the complete fitness function. The proposed algorithm is designed to guide the solution towards convergence of the feasible solution. The pseudocode of the algorithm is presented in Algorithm 3. It involved the call of 3 other subroutines:

Local Search (line 5): In the local search, the algorithm performs the standard mutation or crossover, diversity measure and rejection. The key difference is that instead of using a resource-intensive fitness function, we employ a lightweight surrogate fitness function for evaluation. We say an individual p is a **locally feasible** solution if p can intercept all paths in Φ. Individuals failed to block any paths in Φ will be penalized. The penalty score is determined by the number of paths in Φ that an individual p cannot block. The **surrogate fitness function** can be presented as follows:

$$f_s^{\phi}(C) = \begin{cases} \min_{\pi \in \Phi} RT(\pi, C), & \text{if } C \text{ is locally feasible.} \\ -|\{\pi \in \Phi : \pi \cap C = \emptyset\}|, & \text{otherwise.} \end{cases} \tag{3}$$

Global Search (line 7): We define that global search starts only when there are no locally infeasible individuals in the local population, and a specified number of local iterations have been completed. In the Global Search, the algorithm adds each "candidate" individual from the local population to the global population and employs diversity measures and rejection on the global population. We use the complete fitness function to evaluate each individual. *It's important to note that a solution C is locally feasible may not necessarily be globally feasible.* This concern arises because the local search evaluates only a fraction of the graph (Φ), which may not provide enough samples to form a cut in the graph. However, as stated in

Theorem 4, we establish that eventually, the locally feasible solution yields the globally feasible solution after a certain number of iterations.

Update the Surrogate Fitness Function (line 8 - 12): Following every global search, we improve the surrogate function by updating the important path set Φ. The update is based on the performance of each individual in the local population. For every $p \in P_{local}$ that is globally infeasible, we add some random temporal (s, DA)-path to Ψ in graph $G' = (V \setminus p, E)$ after removing nodes in cut set p. Those are the paths that make the individual p globally infeasible. We use the modification of the Depth First Search algorithm for temporal graphs to find the random paths. For every $p \in P_{local}$ that is globally feasible, we improve the surrogate function by adding the optimal attack path when facing the defense solution p to Φ.

Theorem 4. In Algorithm 3, the number of iterations of Global Search until feasible solution C on local evaluation function $f_s^{\phi}(C)$ yield feasible solution on the global evaluation function is $O(|V|)$ iterations at worst.

Algorithm 3 EDO with surrogate-assisted/penalty-based fitness function

Input: Temporal Graph G, honeypot budget b
Output: Blocking population P

```
Initialize local population P_local
Initialize global population P_global
Initialise set of paths Φ
while A termination criterion is met do
    P_local ← LocalSearch(P_local, φ)
    if ∏_{p∈P_local, π∈Φ} |p ∪ π| ≠ 0 and global criterion is met do
        P_global ← GlobalSearch(P_global, P_local)
        foreach p ∈ P_local do
            if p is a (S, DA) − cut in G do
                Compute π^p_opt(G) and add to Φ
            else
                Add a random paths from s ∈ S to DA in graph G' = (V\p, E) to Φ
return P_global
```

5 EXPERIMENT RESULT

5.1 Experiment Set Up

All of the experiments are carried out on a high-performance computing cluster with 1 CPU and 24GB of RAM allocated to each trial. As the real-world AD graph is sensitive, we will conduct experiments on synthetic graph generated by DBCreator [1] and Adsimulator [2] - two state of the art tools for creating AD graphs. Every graph starting with R ("Rxxx") is generated by DBCreator while the one starting with label AD ("ADxxx") is generated by the ADsimulator. DBCreator only allows us to fine-tune the number of computers and users. In contrast, Adsimulator provides greater flexibility by enabling adjustments to various entities in the AD graph, including Security Groups, Organizational Units (OUs), Group Policy Objects (GPOs), and more. Consequently, we have two types of graphs generated by Adsimulator: 'ADX**x**', where default parameters are increased by a factor of '**x**' (e.g., ADX10 is 10 times the default setting), and 'ADU**y**', mimicking '**y**' fractional proportions of the structure of the real AD network at the University of Anonymized (e.g., ADU05 represents 5 % of the mimicked network). Due to space constraints, detailed information about the size of each graph will be provided in the technical report. However, for a quick estimate, here are the sizes of the largest graph for each type: R4000 (12001 nodes and 45780 edges), ADX20 (6013 nodes and 26671 edges), and ADU (6875 nodes and 37292 edges).

[1] https://github.com/BloodHoundAD/BloodHound-Tools/tree/master/DBCreator
[2] https://github.com/nicolas-carolo/adsimulator

Table 1: Comparison of all algorithms with DBCreator's graph. The results show the average response time (higher is better) and the average last improvement time (lower is better) of each setting. The numbers in the parenthesis are the average last improvement time.

	R2000+C	R4000+C	R2000+L	R4000+L
VAN-V	2.10 (53177s)	3.60 (42445s)	0	0
VAN-D	2.03 (68268s)	4.09 (28514s)	0	0
ILP-V	4.07 (16715s)	4.49 (17037s)	2.62 (40999s)	3.18 (41826s)
ILP-D	4.09 (25177s)	4.58 (19067s)	2.90 (43272s)	2.90 (33305s)
EST-V	4.17 (302s)	4.70 (706s)	4.50 (11862s)	3.70 (20536s)
EST-D	4.17 (473s)	4.70 (302s)	4.60 (10257s)	3.70 (20536s)

However, these tools only generate static snapshots of the graph. To generate a temporal AD attack graph, we will merge a "mould" of static AD graph with authentication data which simulates the characteristic of Hassession edge. The first source is the authentication data from The Comprehensive, Multi-Source Cyber-Security Events dataset [7], referred to as **LANL**. The second source is from an anonymous organization, labelled as **COMP**. We will provide the details of each dataset in the appendix. Combining these datasets involved the following process. First, in each static mould AD graph, we removed all HasSession edges. Next, we randomly mapped users, computers and authentication events from the authentication data to the mould graph to create an instance of the temporal graph. For clarity in denoting the generated instances, we referred to a temporal graph in the format $\{graph\} + \{auth_source\}$. For instance, **R2000+C** indicates a temporal graph derived from the mould static graph **R2000**, with Hassession edge data sourced from the **C**OMP authentication dataset. In this notation, L refers to the LANL dataset, and C refers to the COMP dataset.

We use Gurobi 9.0.2 solver for solving the ILP module. For each experiment instance, we ran 10 trials. In each trial, we randomly choose a set of 10 starting nodes and randomly re-map the authentication data to the mould graph. To define the defensive budget for our problem, we have to determine the size of the minimum temporal cut $|minC|$. We will discuss how we determine $minC$ in our appendix [17]. Given that the condition $b \geq |minC|$ has to be met to ensure our problem is feasible, we define the budget for our problem as $b = b_f * |minC|$ where $b_f > 1$ is the budget factor. We set the budget factor to $b_f = 1.5$ for every experiment. We define that only 90 percent of nodes in the graph is blockable. To construct the HasSession edges, we captured snapshots of the authentication dataset every 1 hour. In the experiment, we considered

Table 2: Comparison all algorithms with ADsimulator's graph. No feasible result found by VIN so we did not include it here. OOM is stand for Out-of-Memory. All notion in Table 1 will be also applied here.

	ADX5+C	ADX10+C	ADX20+C	ADU5+C	ADX5+L	ADX10+L	ADX20+L	ADU5+L
ILP-V	3.21 (31362s)	OOM	OOM	OOM	3.50 (26796s)	0.83 (78836s)	OOM	OOM
ILP-D	3.30 (25250s)	OOM	OOM	OOM	3.40 (25228s)	0.60 (73151s)	OOM	OOM
EST-V	3.30 (10517s)	3.50 (20498s)	2.50 (34762s)	1.60 (67432s)	5.27 (1965s)	1.05 (38057s)	1.4 (68776s)	1.8 (70165s)
EST-D	3.40 (2415s)	3.30 (14839s)	2.50 (34365s)	1.70 (65532s)	4.77 (3175s)	1.75 (23709s)	1.90 (74284s)	1.40 (73482s)

a total of 1000 snapshots in each setting (about 40 days). To avoid confusion in metrics, we will use "time unit" as the metric for the response time. We generate a population of 10 defensive blocking plans. The termination condition for the evolution algorithms was set at 2,000,000 iterations or 24 hours, whichever came first.

In our experiment, we adopt specific denotation for clarity: the Integer Linear Programming approach is denoted as ILP, the surrogate-assisted approach as EST, and the vanilla EDO algorithm as VIN. Additionally, we introduce a **Value-based Evolutionary Computation (VEC)** which greedily rejects the worst individual from the population instead of rejecting individuals based on diversity measure. In total, we will have 6 sets of algorithm includes: Vanilla EDO algorithm (**VAN-D**), Vanilla VEC algorithm (**VAN-V**), ILP-repair approach with EDO framework (**ILP-D**), ILP-repair approach with VEC framework (**ILP-V**), Surrogate-assisted approach with EDO framework (**EST-D**), Surrogate-assisted approach with VEC framework (**EST-V**). Note that our vanilla EDO algorithm's fitness function is implemented with our Dijkstra-based algorithm for computing the earliest-arrival path. Results would significantly degrade if Wu's algorithm were employed.

5.2 Result Interpretation

From Figure 2, the EST approach significantly improves the convergence speed of the Evolutionary Algorithm, allowing it to reach the best result much faster than ILP and VAN. Notably, ILP approach can find feasible solution from the early iteration since the repair operator will guarantee the mutation/crossover yields a feasible defensive solution. However, solving Integer Linear Programming itself is highly resource-intensive and is the bottleneck for this technique. Unfortunately, ILP failed to run in 5 out of 12 graphs due to Out-of-Memory errors.

Among the graphs, R2000+C and R4000+C are the only two where VAN can find any feasible solution. When we record the time to find the feasible solution (R2000+C and R4000+C), while vanilla takes about 21,728 seconds to find the feasible solution, EST takes on average 201 seconds which is about 108 times faster. For our setting, the EST method performs, on average, about 23% better than the ILP. The convergence speed of EST is also superior to ILP.

To compare the performance of EDO-based algorithms (ended with D) with VEC-based algorithms (ended with V), we conducted a head-to-head comparison between these two approaches. Out of 21 comparable settings (excluding those with OOM errors and infeasible solutions), EDO outperformed VEC in 12 settings, while VEC performed better in only 9 cases (in instances where the response times were equal, we compared the average last improvement time). Overall, EDO outperformed VEC when applied to our problem.

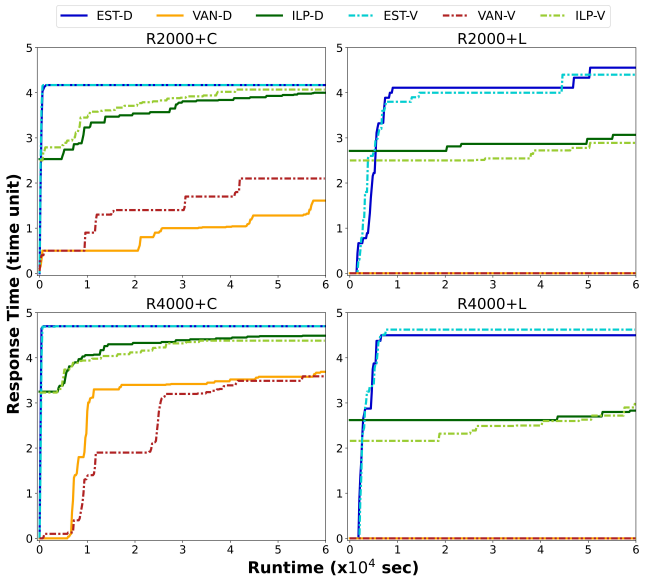


Figure 2: Performance comparison of all 6 algorithms. The EST approaches exhibit significantly faster convergence to the best result compared to the other methods.

6 CONCLUSION

This paper investigated a Stackelberg game model between an attacker and a defender in temporal Active Directory attack graphs. We propose the use of Evolutionary Diversity Optimization algorithms to address this problem. However, the vanilla EDO encounters challenges when scaling to larger graphs and struggling to find feasible solutions. To improve our solution, we first improve the computation of the attacker's optimal path (fitness function) by refining the calculation of the earliest-arrival path. Our novel Dijkstra-based algorithm for computing the earliest-arrival path, based on the observation that a significant portion of the AD infrastructure remains static. Experimentally, our algorithm is approximately 5 times faster than the SOTA algorithm when running on AD-specific graphs. Next, we introduce two constraint-handling techniques: a repair mechanism using Integer Linear Program (ILP) and a surrogate-assisted model with a penalty fitness function (EST). While ILP guarantees to find a feasible solution in early iterations, the EST method achieves this approximately 108 times faster than the vanilla approach. Moreover, EST outperforms ILP, demonstrating approximately a 23% improvement in our specific setting.

ACKNOWLEDGMENTS

REFERENCES

[1] Massimiliano Albanese, Karin L Johnsgard, and Vipin Swarup. 2022. A Formal Model for Credential Hopping Attacks. In *European Symposium on Research in Computer Security*. Springer, 367–386.

[2] Evgeny Bogokovsky and Andrey Karpovsky. 2022. Detecting malicious key extractions by compromised identities for Azure Cosmos DB. https://www.microsoft.com/en-us/security/blog/2022/06/23/detecting-malicious-key-extractions-by-compromised-identities-for-azure-cosmos-db/.

[3] Jakob Bossek, Pascal Kerschke, Aneta Neumann, Markus Wagner, Frank Neumann, and Heike Trautmann. 2019. Evolving diverse TSP instances by means of novel and creative mutation operators. In *Proceedings of the 15th ACM/SIGEVO conference on foundations of genetic algorithms*. 58–71.

[4] Jakob Bossek, Aneta Neumann, and Frank Neumann. 2021. Breeding diverse packings for the knapsack problem by means of diversity-tailored evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 556–564.

[5] Jakob Bossek and Frank Neumann. 2021. Evolutionary diversity optimization and the minimum spanning tree problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 198–206.

[6] Yoav Daniely. 2021. What's new: Microsoft Sentinel Deception Solution. https://techcommunity.microsoft.com/t5/microsoft-sentinel-blog/what-s-new-microsoft-sentinel-deception-solution/ba-p/2904945.

[7] Anh Do, Mingyu Guo, Aneta Neumann, and Frank Neumann. 2022. Analysis of evolutionary diversity optimization for permutation problems. *ACM Transactions on Evolutionary Learning* 2, 3 (2022), 1–27.

[8] John Dunagan, Alice X Zheng, and Daniel R Simon. 2009. Heat-ray: combating identity snowball attacks using machinelearning, combinatorial optimization and attack graphs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 305–320.

[9] Diksha Goel, Aneta Neumann, Frank Neumann, Hung Nguyen, and Mingyu Guo. 2023. Evolving Reinforcement Learning Environment to Minimize Learner's Achievable Reward: An Application on Hardening Active Directory Systems. *GECCO '23: Genetic and Evolutionary Computation Conference, 2023, 2023* (2023).

[10] Diksha Goel, Max Hector Ward-Graham, Aneta Neumann, Frank Neumann, Hung Nguyen, and Mingyu Guo. 2022. Defending active directory by combining neural network based dynamic program and evolutionary diversity optimisation. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1191–1199.

[11] Mingyu Guo, Jialiang Li, Aneta Neumann, Frank Neumann, and Hung Nguyen. 2022. Practical fixed-parameter algorithms for defending active directory style attack graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 9360–9367.

[12] Mingyu Guo, Jialiang Li, Aneta Neumann, Frank Neumann, and Hung Nguyen. 2024. Limited Query Graph Connectivity Test. *Proceedings of the AAAI Conference on Artificial Intelligence* (2024).

[13] Mingyu Guo, Max Ward, Aneta Neumann, Frank Neumann, and Hung Nguyen. 2023. Scalable edge blocking algorithms for defending active directory style attack graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5649–5656.

[14] Evald Markinzon. 2023. Ignite News: Augment your EDR with deception tactics to catch adversaries early. https://techcommunity.microsoft.com/t5/microsoft-defender-for-endpoint/ignite-news-augment-your-edr-with-deception-tactics-to-catch/ba-p/3982253.

[15] Microsoft. 2023. Microsoft Digital Defense Report. https://www.microsoft.com/en/security/security-insider/microsoft-digital-defense-report-2023/.

[16] Huy Quang Ngo, Mingyu Guo, and Hung Nguyen. 2024. Catch Me if You Can: Effective Honeypot Placement in Dynamic AD Attack Graphs. *IEEE International Conference on Computer Communications (IEEE INFOCOM)* (2024).

[17] Huy Quang Ngo, Mingyu Guo, and Hung Nguyen. 2024. Optimizing Cyber Response Time on Temporal Active Directory Networks Using Decoys (Extended Version). https://arxiv.org/abs/2403.18162.

[18] Adel Nikfarjam, Jakob Bossek, Aneta Neumann, and Frank Neumann. 2021. Entropy-based evolutionary diversity optimisation for the traveling salesperson problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 600–608.

[19] Aaron Scott Pope, Robert Morning, Daniel R Tauritz, and Alexander D Kent. 2018. Automated design of network security metrics. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1680–1687.

[20] Andy Robbins. 2023. "Bloodhound: Six Degrees of domain admin. https://github.com/BloodHoundAD/BloodHound. Accessed: 2022-08-02.

[21] Tamara Ulrich, Johannes Bader, and Eckart Zitzler. 2010. Integrating decision space diversity into hypervolume-based multiobjective search. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. 455–462.

[22] Rohan Vazarkar. [n. d.]. SharpHound - C# Rewrite of the BloodHound Ingestor. https://github.com/BloodHoundAD/SharpHound3. Accessed: 2022-08-02.

[23] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.

[24] B Bui Xuan, Afonso Ferreira, and Aubin Jarry. 2003. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science* 14, 02 (2003), 267–285.

[25] Yumeng Zhang, Max Ward, Mingyu Guo, and Hung Nguyen. 2023. A Scalable Double Oracle Algorithm for Hardening Large Active Directory Systems. *The 18th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS)* (2023).

[26] Philipp Zschoche, Till Fluschnik, Hendrik Molter, and Rolf Niedermeier. 2020. The complexity of finding small separators in temporal graphs. *J. Comput. System Sci.* 107 (2020), 72–92.