Identification of Device Dependencies Using Link Prediction

Lukáš Sadlek*†, Martin Husák*, Pavel Čeleda†
†Faculty of Informatics, Masaryk University, Brno, Czech Republic
*Institute of Computer Science, Masaryk University, Brno, Czech Republic
sadlek@mail.muni.cz, husakm@ics.muni.cz, celeda@fi.muni.cz

***Abstract*—Devices in computer networks cannot work without essential network services provided by a limited count of devices. Identification of device dependencies determines whether a pair of IP addresses is a dependency, i.e., the host with the first IP address is dependent on the second one. These dependencies cannot be identified manually in large and dynamically changing networks. Nevertheless, they are important due to possible unexpected failures, performance issues, and cascading effects. We address the identification of dependencies using a new approach based on graph-based machine learning. The approach belongs to link prediction based on a latent representation of the computer network's communication graph. It samples random walks over IP addresses that fulfill time conditions imposed on network dependencies. The constrained random walks are used by a neural network to construct IP address embedding, which is a space that contains IP addresses that often appear close together in the same communication chain (i.e., random walk). Dependency embedding is constructed by combining values for IP addresses from their embedding and used for training the resulting dependency classifier. We evaluated the approach using IP flow datasets from a controlled environment and university campus network that contain evidence about dependencies. Evaluation concerning the correctness and relationship to other approaches shows that the approach achieves acceptable performance. It can simultaneously consider all types of dependencies and is applicable for batch processing in operational conditions.**

***Keywords*—device dependency, link prediction, dependency embedding, network traffic analysis, graph-based analysis, random walk**

I. INTRODUCTION

Each network contains devices that provide essential services, e.g., domain name service that translates domain names to IP addresses or domain controller that enforces active directory policy for Windows devices. Network communication reveals that essential services are often targeted with requests upon which other devices rely. The mapping of asset dependencies has a supportive function for other tasks in network management (e.g., reveals performance issues in practice) and is usable for risk analysis of critical systems [1], [2].

Automated dependency detection has been studied for a long time since manually determining these dependencies is infeasible [3]. The motivation is often network configuration management, e.g., analysis of potential impacts in case of failures, performance issues, and malicious attacks [4], [5], [1]. The previous research revealed dependencies using passively monitored network traffic [6], active approach [7], system logs instead of network traces [8], and applying time series or graph mining [9], [10]. Even though it provided valuable results, it focused on specific input data, had some limitations, or revealed only specific types of dependencies.

The current research results from graph-based machine learning can improve the methods since machine learning is recommended for complex network topologies [11]. Link prediction was shown to be useful for recommender systems in social networks, spam detection, and network routing [12]. Moreover, latent graph representation learning (alternatively node embedding) reveals the hidden structure of graphs. In other words, it transforms nodes to their low-dimensional embedding representation and can identify even relationships not captured in the input data. Hence, algorithms can be more efficient with it than with the original graph [13].

We propose an approach that uses latent graph representation learning (inspired by the Node2Vec approach [14]) for a new use case, which is to compute dependency embedding for the training of a dependency classifier. We create its novel core and most complex part – the custom exploration of communication chains present in data. We introduce conditions for timestamps of IP flows, which must be fulfilled by communication chains, discuss how to prepare input IP flows, and accomplish other design adjustments of graph representation learning. Our contribution also includes the measurement of the approach's properties on data from a controlled environment and campus network and its comparison with local similarity indices using correlation coefficients.

In this paper, we focus on two research questions:

1) *Can we identify device dependencies using graph-based machine learning for the link prediction problem?*
2) *What correctness, time aspects, dependency types, and amount of processed data of the link prediction approach for device dependency identification can we obtain?*

We also focus on passively collected network traffic in the form of IP flows as input data for our approach.

This paper is organized as follows. Section II describes the related work from graph-based network analysis and defines device dependencies. Section III proposes a novel method for device dependency identification using link prediction. Implementation of the method is introduced in Section IV, including conditions related to types of dependencies. Evaluation concerning the method's correctness, time aspects, dependency types, and amount of data is explained in Section V. The last Section VI concludes the paper.

II. Related Work

Graph-based approaches to data analysis are not uncommon in the network security domain. The attack graphs, for example, are the most well-known application of graphs in this domain. Their construction and usage were exhaustively covered by Kaynar et al. [15]. Akoglu et al. [16] surveyed graph-based techniques for anomaly detection in diverse domains, including network traffic analysis. The application of graphs in network-wide situational awareness was covered by Noel et al. [17] or recently by Husák et al. [18]. Bowman and Huang [19] reviewed the challenges of the application of Graph AI in cyber security. Atzmüller and Kanawati [20] provided an overview of explainability for complex network analysis in cyber security. Lagraa et al. [21] surveyed the application of graphs in intrusion and botnet detection.

A prime example of the application of advanced graph-based techniques in network security management is criticality and dependency detection, i.e., finding which devices in the network are the most important or how they depend on each other [6], [3]. In this work, we recognize three essential dependency classes. *Direct dependency* (DD) is a network connection between a source and destination IP address that repeats more than threshold times with the same IP flow parameters except for timestamps. The *local-remote* (LR) dependency from the view of a requesting server (e.g., the webserver in Figure 1) consists of communication with another (remote) server (e.g., the database server) to answer the request from the user device [6]. The *remote-remote* (RR) dependency describes that one server is indirectly dependent on another to provide functionality for user devices [4]. Both servers provide remote services for the user device. For example, the user device in Figure 1 cannot access the webserver without obtaining its IP address from the DNS server.

Correct estimation of criticality and dependencies is vital for securing and hardening the networks but is hard to achieve in large networks due to the lack of detailed situational awareness and local knowledge [22], [18]. Passive network measurement, e.g., NetFlow [23], is the most widely used for dependency detection and is briefly described later in this section. Natarajan et al. [6] proposed the NSDMiner suite that determines LR dependencies based on passive network monitoring techniques. Zand et al. [3] proposed the automatic detection of critical services in the network based on finding cliques in the graphs of correlated services, i.e., services active at similar times. Laštovička and Čeleda [22] proposed graph centrality-based approaches. Lange et al. [9] used the time series-based analysis of network traffic to detect dependencies of network services. The topic became of utmost importance in cloud management, and thus, the SCoRMiner was proposed by Slimani et al. [10] to detect dependencies between cloud services using both network traces and application-level data and graph mining.

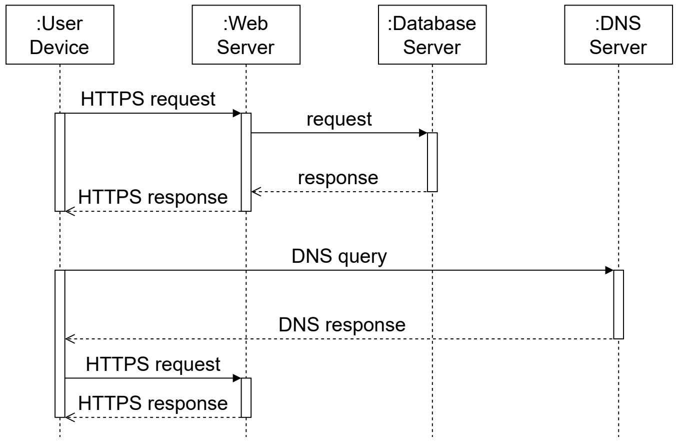


Fig. 1. A sequence diagram containing local-remote dependency of the web server on the database server (the first activation of the user device) and remote-remote dependency of the web server on the DNS server (the second activation of the user device). Activations (vertical rectangles) denote participation of lifelines. Time passes from top to bottom in the diagram.

Nevertheless, passive network-based approaches are not the sole approach to dependency detection. Zand et al. [7] proposed an active approach based on delay injection and implemented it in the Ripper tool. Lan et al. [8] proposed an approach based on system log analysis rather than network traffic, aiming at cloud environment and dependency discovery in microservices. Aksoy et al. [24] presented the most elaborated method of important IP address selection based on Laplacian centrality in communication graphs. Recently, Husák et al. [18] summarized the approaches to criticality estimation, connected them with graph-theoretical background, and briefly commented on their usability in network security management. An example of a practical take on dependency detection is the Application Discovery and Dependency Mapping (ADDM) from the Server & Application Monitor (SAM) by SolarWinds [1], which combines manual data entry and automated discovery.

Link prediction [12] is one of the most widely-used graph-theoretical approaches in data analysis, with a vast application potential in cybersecurity as well. The goal is to predict or disclose future or missing links between entities. Pope et al. [25] proposed the use of tailored link prediction heuristics in this domain, while Noel and Swarup [26] used dependency-based link prediction for learning microsegmentation policies. It is worth noting that we are not aware of any work that would use this particular combination of the problem of device dependency detection and the approach of link prediction.

The rise of machine learning (ML) also covered the graph-based data. Since traditional ML methods mostly take low-dimensional vectors as the inputs, there is a need to embed the graphs or nodes in the graphs into vectors. A well-known approach inspiring this work is the Node2Vec [14]. An alternative node representation for ML is the DeepWalk [27]. Readers are kindly referred to the survey by Zhang et al. [13] for a comprehensive background and comparison. Nevertheless, there are classes of ML that allow the processing of graph structures directly. For example, graph neural networks (GNNs) are becoming increasingly popular even in network security management, such as in intrusion detection [28], malware detection [29], or network slicing in digital twins [30].

Since this and most of the related work relies on network flows or similar data, we briefly introduce them here as well. Flow monitoring is a prevalent network traffic monitoring in large-scale and high-speed networks [23]. Network flow is defined as a unidirectional sequence of packets that share the source and destination IP address, IP protocol number, and TCP or UDP port or ICMP code. The flows are exported in NetFlow or IPFIX formats and accompanied by further information, such as timestamp, duration, and number of transferred packets and bytes. A typical bidirectional network connection exports as two flows but can be later paired into biflows, special flow records describing network traffic in both directions. Readers are kindly referred to an exhaustive tutorial by Hofstede et al. [23].

III. Method for Identification of Dependencies

The proposed method for the identification of device dependencies creates a dependency classifier obtained via multiple steps expressed in Figure 2. Even though the approach processes bidirectional network communication, input IP flows must be unidirectional since we sample communication chains from a directed graph. Bidirectional IP flows must be converted to unidirectional using either distinct or the same start and end timestamps for both unidirectional flows.

The input IP flows can contain more IP addresses than can be processed by a neural network in a feasible time. It is necessary to focus only on IP addresses, among which we attempt to determine dependencies. We assume that the most important IP addresses have the highest number of IP flows except for attacks that produce a lot of communication, e.g., network scanning. When these IP addresses become vertices in a graph sample, we need a realistic communication neighborhood (i.e., communicants) for each vertex during random walk exploration. For this purpose, we should process

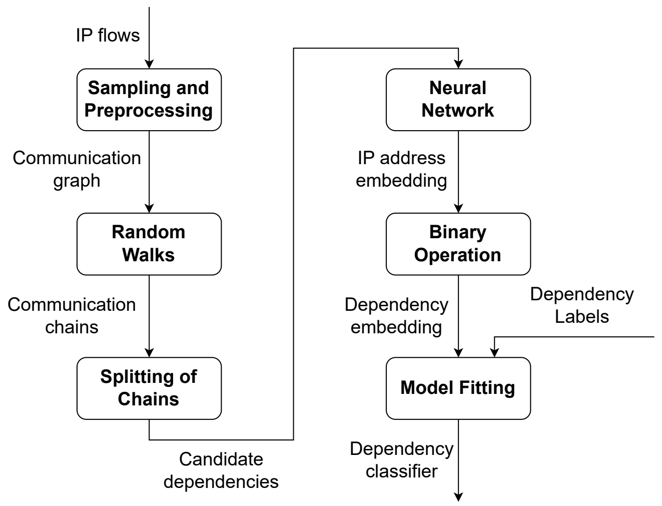


Fig. 2. Steps of the proposed approach from processing of input data to obtaining dependency classifier.

enough edges, and sampling must be fair, i.e., all sampled edges must be chosen with equal probability.

The requirement can be achieved by sampling with a reservoir of length n [31]. Any incoming k-th edge (i.e., IP flow for a specific IP address) receives a random number from 0 to $k-1$. If the number represents an index from the reservoir (i.e., $k < n$), the item is stored at such a position. As a result, we should obtain a directed communication multigraph that contains the source port, destination port, protocol, and start and end timestamps for each edge.

The next step is the core and the most complex one. It creates directed random walks representing communication chains of IP addresses. The chains are constrained by conditions imposed on timestamps of two subsequent IP flows to choose the next vertex of the random walk according to the current and previous vertex. Imposing conditions on more than two consequent IP flows in a communication chain could result in the trying of all possible sequences of higher length and drastically impact performance.

The first condition for creating communication chains is called the *opening of LR dependency*, expressed as Condition 1 in Section IV. It corresponds to a sequence of communication from the user device to the database server via the web server in Figure 1. The second one is a *return from LR dependency* related to a sequence from the database server to the user device via the web server in Figure 1. It is mathematically described as Condition 2 in Section IV.

Opening of RR dependency expresses a communication chain from the user device to the DNS server, from the DNS server to the user device, and then from the user device to the web server in Figure 1. However, we omit redundant repeating user device from the communication chain because the DNS server and the web server would still be very close to it in the communication chain. Therefore, the final chain is the user device, the DNS server, and the web server. The opening of RR dependency is formally described by Condition 3 in Section IV. The last condition is the *return to the previous IP address* expressed as Condition 4 that is fulfilled, e.g., by a communication chain of the web server, the database server, and the web server in Figure 1.

The four conditions represent the building blocks of the communication chains. Chains that fulfill them can contain device dependencies in a form suitable for splitting in the next step. As a result, the approach also reveals transitive dependencies (TDs) composed of at least two DDs fulfilling conditions on LR dependency but with communication from solely one user device. An example of such a transitive dependency could be the dependency of the user device on the database server in Figure 1, where also LR dependency exists between the web server and the database server.

The random walk exploration uses these conditions to create the inputted number of random walks starting in each vertex. The length of the random walk is also specified as input and should typically range from three to seven to capture dependencies. The second vertex of the random walk is chosen randomly from the vertices fulfilling the input threshold

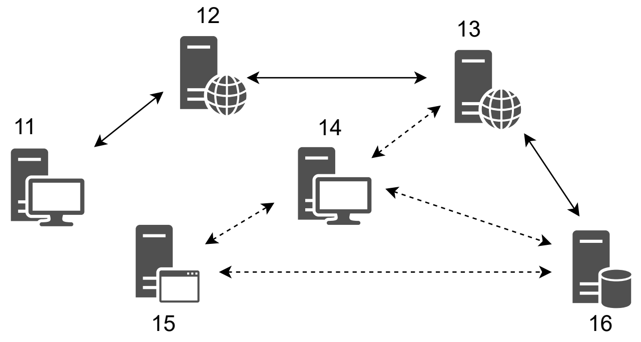


Fig. 3. Example of network communication between workstations and servers. Solid lines represent a communication chain. All edges consist of forward and reverse IP flows. Numbers are the last octets from IPv4 addresses.

imposed on the number of IP flows, in which the first and second vertices appear as source and destination IP addresses (threshold n_t in Table I). The third and consequent vertices are randomly chosen out of vertices that fulfill at least one condition. If no vertex fulfills a condition, then a vertex that fulfills the IP flow threshold is randomly chosen. If no vertex has enough appearances, then any target of directed edges from the source vertex is randomly chosen so that the random walk continues using graph edges.

The algorithm creates communication chains where directed edges are transformed from the captured communication in concordance with the order of communication steps required by the dependency types. If an IP address appears commonly in the same time-constrained communication chain with another IP address, then it is probable that one of them could be dependent on the other one. The longer the distance between the IP addresses in the chain, the lower the chance that dependency exists. Thus, it is beneficial to process the close pairs as candidate dependencies.

Candidate dependencies are determined from the communication chain of a specific length using a sliding window (context), which we explain based on Figure 3. For simplicity, assume that a communication chain with length four consisting of 11, 12, 13, and 16 fulfills conditions. If the context size is three, then we obtain two contexts from this chain. The first is 11, 12, and 13, and the second is 12, 13, and 16. The candidate dependencies are pairs of addresses from one context where the first address is the initial address from the context, and the second one can be any of the following. We obtain candidate dependencies $(11, 12)$ and $(11, 13)$ from the first context and $(12, 13)$ and $(12, 16)$ from the second one.

The neural network uses the candidate dependencies to estimate features of the hidden layer so that IP addresses from candidate dependencies would be close together in the IP address embedding space, as is usual for embedding [13]. IP address embedding contains a vector with a number of values equal to the number of features in the hidden layer for each IP address and must be transformed into dependency embedding using vectors for individual IP addresses.

Binary operations such as average and L1 distance can transform node embedding to edge embedding [14]. Since the main purpose of embedding is to have similar nodes close together and distance is commutative, these operations should be commutative. In our case, we use a scalar product of two vectors to combine representation from node embedding to dependency embedding. It causes the range of values appearing in vectors to increase when we apply the product. In other words, it extends the space where candidate dependencies are mapped, and this space is then used for classification. As a result, the dependency embedding contains a vector of values for each candidate pair.

As a last step, each possible dependency in the embedding is given its label that denotes whether it is a dependency. Training the model with dependency embedding and labels (see Figure 2) creates a dependency classifier that can determine whether a pair of IP addresses can be a dependency. This approach reveals the existence of the dependency, but it cannot automatically distinguish the direction of dependency because it uses embedding. Moreover, the revealed dependencies are influenced by the position of the IP flow collector. If we capture communication only at the edge of the network, then all dependencies containing one communication pair will be between one internal and one external IP address since the communication graph, in this case, will be bipartite between internal and external IP addresses.

The method can provide as good results as its input IP flows. Tunneling approaches and anonymization protocols can replace original IP headers, and the approach will process the replaced IP addresses. However, a dependency on a Virtual Private Network (VPN) server that could be identified is valid when the VPN is needed to access internal network resources. Using anonymization protocols in the long term for communication with critical devices (such as DNS servers and cloud storage) is a rare scenario. The method also focuses only on the most essential IP flow properties from packet headers. Therefore, privacy-preserving protocols (e.g., HTTPS) that encrypt packet data do not hinder its use.

IV. Implementation of the Method

The implementation consists of six steps, as depicted in Figure 2. The first three are discussed separately, while the remaining are described in Subsection IV-D. Our implementation reuses existing supportive functionality of the Node2Vec approach in Python (mainly related to neural network training) [32]. However, we provide the method's most complex step of exploring valid random walks and implement the remaining parts to process IP flows. The implementation is available in supplementary materials [33].

A. Sampling and Data Preprocessing

The first step obtains a representative sample with n internal and m external IPv4 addresses based on the highest number of IP flows from a batch of data. Consequently, we sample k edges for the selected internal and external IP addresses. For this purpose, we implemented reservoir sampling that selects each edge with equal probability [31]. Data preprocessing also includes removing communication that does not use TCP and UDP protocols since, for them, the dependencies are defined in related work [6] used for ground truth comparison in Section V.

B. Random Walks

A directed random walk is a sequence $v_1, v_2, \dots, v_n$ where vertex v_{i+2} for $i \in \{1, \dots, n-2\}$ is determined based on vertices v_{i+1} and v_i. Since general node embedding approaches cannot directly support computer network graphs, it considers time constraints and IP flow properties, e.g., transport ports.

Let $t_1(v_i, v_{i+1})$ denote the start timestamp and $t_2(v_i, v_{i+1})$ the end timestamp of IP flow between vertices (i.e., IP addresses) v_i and v_{i+1}. At least one of four conditions must hold for three subsequent vertices v_i, v_{i+1}, and v_{i+2}. The condition expressing opening of LR dependency:

$$\begin{aligned} t_1(v_i, v_{i+1}) \le t_1(v_{i+1}, v_{i+2}) \le t_2(v_{i+1}, v_{i+2}) \le \\ \le t_2(v_i, v_{i+1}), i \in \{1, \dots, n-2\} \end{aligned} \quad (1)$$

holds for the forward direction of LR dependency. It expresses that the first request from a user device to the server (e.g., the web server in Figure 1) is followed by another request from that server to another server (e.g., the database server in Figure 1) to process the original request. Therefore, the first IP flow between the user device and the server will not end until the additional request to another server is processed.

The condition for return from LR dependency:

$$\begin{aligned} & t_1(v_{i+1}, v_{i+2}) \le t_1(v_i, v_{i+1}) \le t_2(v_i, v_{i+1}) \le \\ & \le t_2(v_{i+1}, v_{i+2}) \land \exists j : j \ne i \land v_{i+2} = v_j \land v_{i+1} = v_{j+1}, \\ & i \in \{1, \dots, n-2\}, j \in \{1, \dots, n-1\} \end{aligned} \quad (2)$$

describes the reverse direction of LR dependency (i.e., a chain of reverse IP flows), assuming that the random walk already contains its forward direction. Consider LR dependency from Figure 1. In this case, v_i denotes the database server, v_{i+1} the web server, and v_{i+2} the user device. The IP flow from the web server to the user device starts first, but the IP flow from the database server to the web server will end first. The second condition differs from the first one in the order of vertices.

The third condition:

$$\begin{aligned} & t_2(v_i, v_{i+1}) \le t_1(v_i, v_{i+2}) \land \\ & \land t_1(v_i, v_{i+2}) - t_2(v_i, v_{i+1}) \le \varepsilon, i \in \{1, \dots, n-2\} \end{aligned} \quad (3)$$

expresses the opening of RR dependency that should happen within the specified time ε. For example, let v_i be the user device from Figure 1, v_{i+1} the DNS server, and v_{i+2} the web server. The forward IP flow from the user device to the DNS server, represented as edge (v_i, v_{i+1}), and its reverse flow that ends approximately at the same time will end before the request to the web server. The third condition may also include sequences that accidentally fulfill the condition within ε, but they will be less frequent per time unit with longer observation and smaller input parameter ε.

The fourth condition for triplets of form v_1, v_2, and v_1 formalizes return over reverse edge representing reverse flow. In this case, two unidirectional flows are represented by edges (v_1, v_2) and (v_2, v_1), where the former is the initiator flow.

$$\begin{aligned} & s(v_1, v_2) = d(v_2, v_1) \land d(v_1, v_2) = s(v_2, v_1) \land \\ & \land t_1(v_1, v_2) \le t_1(v_2, v_1) \land \\ & \land |t_2(v_1, v_2) - t_2(v_2, v_1)| \le \varepsilon, i \in \{1, \dots, n-1\} \end{aligned} \quad (4)$$

expresses that the source (s) and destination (d) ports are equal but mutually exchanged, and we use time constraints for their starts and ends.

While the brute-force approach would check that the dependency materialized at least n_t times in data, the random walk exploration checks that there are at least n_t sampled IP flows between the last processed and the next vertex that should extend the random walk. Consequently, all vertices fulfilling one of the conditions are chosen with equal probability.

C. Splitting of Chains

Communication chains (i.e., random walks) obtained from the previous step should be divided into candidate dependencies (i.e., pairs of IP addresses) representing input for the neural network. For this purpose, we consider the sliding windows of the neighboring IP addresses in the communication chain, also called context in DeepWalk and Node2Vec [27], [14]. We use the one-sided context for the proposed approach based on the following argumentation, contrary to the general link prediction on undirected graphs.

Consider Condition 1 holding for triplet v_i, v_{i+1}, and v_{i+2} from forward random walk. Without loss of generality, assume that (v_i, v_{i+1}) and (v_{i+1}, v_{i+2}) are forward IP flows. If we consider the double-sided window, then we also consider reverse random walk, i.e., edges (v_{i+2}, v_{i+1}), (v_{i+1}, v_i), to the learning process as a valid sequence of IP addresses. In the optimal case, the forward IP flow has a lower start timestamp than the reverse IP flow, and they end approximately at the same time. Therefore, we obtain $t_1(v_i, v_{i+1}) \le t_1(v_{i+1}, v_i) \le t_2(v_{i+1}, v_i) \approx t_2(v_i, v_{i+1})$ and similarly $t_1(v_{i+1}, v_{i+2}) \le t_1(v_{i+2}, v_{i+1}) \le t_2(v_{i+2}, v_{i+1}) \approx t_2(v_{i+1}, v_{i+2})$.

We know that $t_1(v_i, v_{i+1}) \le t_1(v_{i+1}, v_{i+2}) \le t_2(v_{i+1}, v_{i+2}) \le t_2(v_i, v_{i+1})$ holds for forward edges from Condition 1. Denote $A = t_1(v_{i+1}, v_i)$, $B = t_2(v_{i+1}, v_i)$, $C = t_1(v_{i+2}, v_{i+1})$, and $D = t_2(v_{i+2}, v_{i+1})$. One possibility of ordering A, B, C, and D using $\le$ operator and assuming approximate equations with $\approx$ is $CADB$ that does not correspond to any of Conditions 1 – 4. It means that we would often incorrectly consider the opposite direction of dependency as a valid dependency for neural network training.

D. Embedding and Model Fitting

The context from the previous step is processed by the neural network in Python. All candidate dependencies (v_i, v_j) for the same first vertex v_i are processed simultaneously, not separately, because the neural network can adjust the embedding so that the first vertex is close to all specified vertices. The neural network also uses stochastic gradient descent, a skip-gram model, and negative sampling optimization to obtain IP address embedding [14].

We use positive dependency labels (ground truth) and an equal number of negative labels for random pairs of vertices to compute the embedding of their source and destination IP addresses. Consequently, the scalar product of these values creates an item for an IP address pair in dependency embedding. We use a random forest classifier in our implementation to fit dependency embedding to the dependency labels.

V. Evaluation

In this section, we describe the evaluation datasets and the ground truth. Correctness, time aspects, and other properties of the method were evaluated using a proof-of-concept implementation based on PyTorch Geometric [34], [32]. Moreover, we compare the proposed approach with local similarity indices and discuss lessons learned. The evaluation was accomplished using Python implementation on a personal computer with 64 GB RAM, 16 CPU cores, and a processor's clock speed of 2.5 GHz. All parameters of evaluation are available in supplementary materials [33].

A. Datasets

We used two kinds of IPFIX flow datasets captured in network topologies familiar to us in detail. The first kind was captured during a two-day cyber defense exercise [35], [36]. The cyber exercise involved six teams (denoted as T1 – T6) that had the same emulated network but behaved in a different way, which provides six partial datasets. Moreover, the exercise used one global network common for all teams. Bidirectional IP flows were captured at the edge of team networks. We converted them to unidirectional form prior to the evaluation. We did not consider IP addresses that represented attacker machines.

The second type was captured in the university campus network, which is assigned class B address space with /16 CIDR prefix. Two partial campus datasets contain ten-minute-long (denoted as U10m) and one-hour-long (denoted as U1h) time windows captured at the network edge during working hours on one Tuesday in March 2022 (U10m) and on one Wednesday in February 2023 (U1h). The campus network data contained unidirectional IP flows.

B. Ground Truth

We determined the ground truth by brute-forcing all possible sequences of unidirectional IP flows up to a limited length of four vertices, which is sufficient because of the context size we consider. Table I contains the number of direct dependencies (DD) and RR dependencies with two (RR) or three communication pairs (RR3) for considered datasets. The dependencies appeared at least n_t times. Two subsequent requests in one RR dependency were accomplished within ε. Many RR dependencies were DNS dependencies.

We do not distinguish LR dependencies because they are already present as DDs. However, we list transitive dependencies that can be created by chaining two (TD) or three direct dependencies (TD3) that fulfill LR conditions on timestamps. It is as if a specific user device solely caused the materialization of LR dependency in the data. Hence, we add its dependency on the supportive server, e.g., the database server in Figure 1. Duplicate TDs are allowed only when two distinct paths materialize the dependency.

TABLE I
Thresholds (n_t, ε) and number of dependencies (DD, RR, RR3, TD, TD3) for team networks from cyber exercise (T1 – T6) and datasets from campus network (U10m, U1h). U1h contains average values from twelve time windows and used different n_t thresholds for DD and TD dependencies (50) and RR dependencies (300).

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**U10m**	**U1h**
n_t	10	10	10	10	10	10	10	50/300
ε	1 s	1 s	1 s	1 s	1 s	1 s	0.5 s	0.5 s
DD	300	444	330	427	321	143	38,372	2,866
RR	248	131	177	310	98	35	1,731	164
RR3	1,875	113	697	2,067	161	26	23,905	2,347
TD	17	13	22	24	9	14	854	117
TD3	0	0	2	1	0	0	359	81

TABLE II
Count of LR dependencies for team networks (T1 – T6) by NSDMiner and dependencies found in the ground truth. C denotes confidence.

	T1	**T2**	**T3**	**T4**	**T5**	**T6**
IP Flows (thousands)	55.1	47.8	29.6	55.6	40.5	23.2
Dependencies for $C > 0$	7	5	0	2	1	0
Present in ground truth	6	5	0	2	1	0
All dependencies	46	50	29	57	36	15
Present in ground truth	27	18	17	28	20	11

We compared the ground truth with the results from the open-source implementation of NSDMiner [37]. NSDMiner used the number of bidirectional IP flows listed in Table II. We did not use transport ports from its output and limited the number of required appearances of dependencies to ten. Otherwise, we used the default options of NSDMiner.

Table II shows that almost all dependencies with non-zero confidence and approximately half of all dependencies by NSDMiner were revealed. It indicates the validity of the ground truth since NSDMiner uses a different approach. It compares only the overall timestamps of biflows, while we compared these timestamps for forward and reverse IP flows.

C. Properties of the Method

We evaluated the method's properties by exploring random walks with five vertices while the context size was four. The approach accomplished ten walks for each vertex, and each positive walk was balanced by finding one non-existing (i.e., negative) random walk. We used five learning epochs for training the model because only small changes in the computed loss between positive and negative random walks were observed after them. The number of values in the embedding vectors was 64, i.e., the embedding had 64 dimensions.

TABLE III
AMOUNT OF DATA AND MEASURED TIME FOR TEAM NETWORKS (T1 – T6) AND DATA FROM UNIVERSITY CAMPUS NETWORK (U10M, U1H) USING THE PROPOSED APPROACH. U1H CONTAINS AVERAGES FROM TWELVE TIME WINDOWS EXCEPT FOR IP FLOWS AND ADDRESSES.

		T1	**T2**	**T3**	**T4**	**T5**	**T6**	**U10m**	**U1h**
Data	IP flows	61,346	54,941	34,721	63,266	46,253	28,506	8,259,584	78,270,416
	IP addresses	689	1,421	654	1,190	1,047	247	451,365	1,235,300
	Vertices	111	96	99	102	95	103	129	93
	Edges (contains multiple edges)	21,026	21,720	20,035	22,905	20,986	16,080	15,076	18,411
Time	Preprocessing	12.80 s	14.88 s	8.41 s	13.86 s	11.54 s	6.19 s	27.15 s	27.32 s
	Creating embedding	15.93 min	14.02 min	14.06 min	14.83 min	12.98 min	13.95 min	6.23 min	6.93 min
	Computation	< 3 s	< 1 s	< 2 s	< 2 s	< 1 s	< 3 s	< 1 s	< 1 s

TABLE IV
EVALUATION METRICS FOR INDIVIDUAL TEAMS (T1 – T6) FROM CYBER DEFENSE EXERCISE AND TWO CAMPUS NETWORK DATASETS (U10M, U1H). ACCURACY, PRECISION, AND F1 SCORE CONTAIN AVERAGES FROM FIFTEEN TRAIN-TEST SPLITS EXCEPT FOR U1H THAT WAS ALSO SPLIT INTO TWELVE CONSEQUENT WINDOWS.

Test size		**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**U10m**	**U1h**
0.25	Accuracy	0.514	0.417	0.493	0.503	0.429	0.337	0.479	0.515
	Precision	0.612	0.521	0.597	0.605	0.530	0.444	0.604	0.615
	F1 score	0.666	0.566	0.644	0.656	0.572	0.462	0.625	0.664
0.50	Accuracy	0.535	0.453	0.529	0.532	0.485	0.403	0.515	0.545
	Precision	0.628	0.544	0.621	0.630	0.580	0.485	0.612	0.638
	F1 score	0.677	0.584	0.669	0.672	0.617	0.510	0.645	0.681
—	AUC	0.68	0.64	0.67	0.68	0.67	0.61	0.71	0.69
	AP	0.81	0.74	0.80	0.81	0.78	0.68	0.84	0.81

We used the test sizes of 25% and 50% of all labels for individual train-test splits. The number of positive and negative labels was the same, but we did not constrain which were chosen into the test set. All datasets represented one time window except for a one-hour-long dataset from the university campus network that was divided into twelve consequent (i.e., five-minute-long) time windows since such time windows can be obtained from IP flow collectors in practice. The method was executed on each window separately, i.e., nineteen times.

Datasets for correctness evaluation contained numbers of IP flows and IP addresses listed in Table III. A sampling of input IP flows outputted approximately one hundred of the most important IP addresses according to the number of IP flows in which they participated. Creating embedding (i.e., neural network training) took approximately a quarter of an hour, mainly due to the exploration of constrained random walks. A long learning time is usual for neural network approaches that infer complex relationships. Computation for considered test sizes was executed in one to three seconds and took a much shorter time than brute-forcing all possibilities.

Table IV contains accuracy, precision, F1 score, area under receiver operating characteristic (ROC) curve (denoted as AUC), and average precision (AP) for the datasets. Since the F1 score is a harmonic mean of precision and recall, it is a suitable measure for imbalanced datasets where most labels belong to one class. AP summarizes precision-recall (PR) curves. The curves for team networks are available in supplementary materials [33]. Examples for team five are depicted in Figure 4. The perfect ROC curve is very close to the upper left, and the PR curve to the upper right corner.

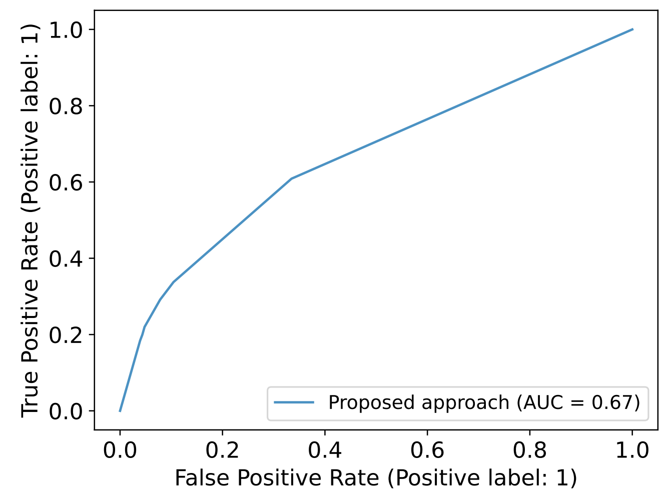
The proposed approach obtained approximately as good AUC, AP, and other metrics for small datasets from cyber defense exercise as for university campus network data. The average AUC for all windows from the one-hour-long campus network dataset was 0.69, while all values ranged between 0.63 and 0.74 AUC. The average AP for the twelve windows was 0.81, and the APs ranged from 0.74 to 0.88.

The accuracy, precision, and F1 score are not as comprehensive as AUC and AP because test sizes of 25% and 50% may not be fractions where they achieve their optimal values. Yang et al. [38] recommend using AP (i.e., the area under the PR curve) since it can better cope with a typical class imbalance of link prediction. The random classifier has an AUC of 0.5, but its AP equals the fraction of positive labels among all labels. The chance level is listed in all PR curves in supplementary materials for six teams [33]. It ranged from 61% to 73% for university data, i.e., the majority class was the positive one. Due to these facts, we conclude that the approach provides acceptable performance with respect to the class imbalance.

D. Comparison with Local Similarity Indices

The method should determine more complex dependencies, not only DDs or missing edges, even though the random walks were sampled over IP flows, which represent graph edges and possible DDs. Positive results from correctness evaluation could mean that it focused too much on DDs (i.e., existing edges with many occurrences).

Local similarity indices provide a substantially different approach because they use paths containing three vertices and determine only possible edges in the graph. We used four local



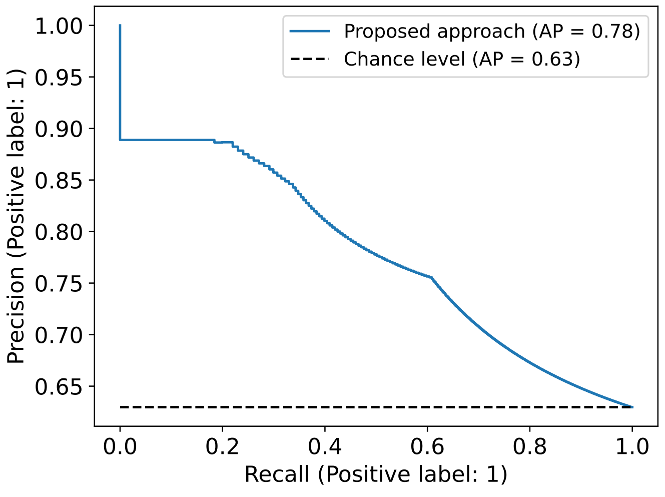


Fig. 4. The ROC and PR curves of the proposed approach for team five from the cyber defense exercise.

similarity indices – Adamic-Adar (AA), Common Neighbors (CN), Preferential Attachment (PA), and Resource Allocation (RA). We adjusted them to directed variants:

$$s_{AA}(x,y) = \sum_{v \in N_{out}(x) \cap N_{in}(y) : |N_{out}(v)| \neq 1} \frac{1}{log|N_{out}(v)|}$$
$$s_{CN}(x,y) = |N_{out}(x) \cap N_{in}(y)|$$
$$s_{PA}(x,y) = |N_{out}(x)| \cdot |N_{in}(y)|$$
$$s_{RA}(x,y) = \sum_{v \in N_{out}(x) \cap N_{in}(y)} \frac{1}{|N_{out}(v)|}$$

where $N_{out}(x)$ denotes a set of vertices v connected from vertex x by an edge (x, v) and $N_{in}(y)$ a set of vertices v that are connected to y by an edge (v, y). These indices are equal to zero for distant nodes and are higher when they are closer.

The comparison of local similarity indices on data from cyber defense exercise with the results of our method determined correlation using the Spearman correlation coefficient and Kendall tau [39]. Both were ranging from 0 to 11%. It means that each local similarity index outputted uncorrelated values to the method's predicted values and also to probabilities assigned to pairs of IP addresses. It indicates that even though the method uses graph edges as input, the conditions on timestamps of IP flows cause it to determine more complex dependencies that are not directly visible in the input.

E. Lessons Learned

The proposed approach had 0.61 to 0.74 AUC for directed graphs when predicting dependencies using an imbalanced set of labels due to counting with all possible pairs of IP addresses. A more general Node2Vec approach had 0.77 to 0.97 AUC for undirected graphs when only edges were predicted [14]. Since the task of link prediction for undirected graphs is much easier and its evaluation used an equal number of positive and negative labels, the measured AUCs for the proposed approach seem to be realistic.

The proposed approach can deal with multiple types of dependencies. However, the separate identification of LR dependencies may not be suitable for it due to the lack of labels (see Table II). Further tuning of the model can use a different classifier than random forest, and the parameters can be tuned for IP flow data using optimization methods, e.g., grid search and random search [40].

VI. Conclusion

This paper addressed device dependency identification based on passively collected IP flows. We used latent graph representation learning for a new use case of creating dependency embedding used by a dependency classifier. The novel core and the most complex part is based on creating communication chains fulfilling time conditions imposed on device dependencies.

The proposed device dependency identification achieves acceptable correctness and is suitable for use cases when the training time does not represent a disadvantage. However, the prediction of dependencies using an already trained model can be very fast compared to using brute force approaches. It can cope with all dependency types simultaneously and process large amounts of data split into batches.

Further research can couple the approach to device criticality detection, network topology discovery, and enterprise mission modeling. It can also be extended by results from active monitoring and adjusted to discover dependencies of network services deployed on stable ports and complex dependencies with redundant devices, such as alternative DNS servers.

The supplementary materials contain a proof-of-concept implementation, a detailed list of parameters used for evaluation, and plots drawn for data from the controlled environment [33]. All materials can be used to reproduce evaluation results.

Acknowledgment

This research was supported by project "MSCA-fellow5_MUNI" (No. CZ.02.01.01/00/22_010/0003229).

References

[1] SolarWinds, "Application Discovery and Dependency Mapping Software," 2023, accessed on Sep 6, 2023. [Online]. Available: https://www.solarwinds.com/server-application-monitor/use-cases/application-dependency-mapping

[2] "Service Dependency Mapping," The MITRE Corporation, 2022, accessed on Oct 2, 2023. [Online]. Available: https://d3fend.mitre.org/technique/d3f:ServiceDependencyMapping/

[3] A. Zand, A. Houmansadr, G. Vigna, R. Kemmerer, and C. Kruegel, "Know Your Achilles' Heel: Automatic Detection of Network Critical Services," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC '15. ACM, 2015, p. 41–50, doi: 10.1145/2818000.2818012.

[4] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions." in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, 2008, pp. 117–130. [Online]. Available: https://www.usenix.org/legacy/event/osdi08/tech/full_papers/chen_xu/chen_xu_html/

[5] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards Highly Reliable Enterprise Network Services via Inference of Multi-Level Dependencies," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 13–24, doi: 10.1145/1282380.1282383.

[6] A. Natarajan, P. Ning, Y. Liu, S. Jajodia, and S. E. Hutchinson, "NSDMiner: Automated discovery of Network Service Dependencies," in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 2507–2515, doi: 10.1109/INFCOM.2012.6195642.

[7] A. Zand, G. Vigna, R. Kemmerer, and C. Kruegel, "Rippler: Delay injection for service dependency detection," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, 2014, pp. 2157–2165, doi: 10.1109/INFOCOM.2014.6848158.

[8] Y. Lan, L. Fang, M. Zhang, J. Su, Z. Yang, and H. Li, "Service dependency mining method based on service call chain analysis," in *2021 International Conference on Service Science (ICSS)*, 2021, pp. 84–89, doi: 10.1109/ICSS53362.2021.00021.

[9] M. Lange and R. Möller, "Time series data mining for network service dependency analysis," in *International Joint Conference SOCO'16-CISIS'16-ICEUTE'16*, M. Graña, J. M. López-Guede, O. Etxaniz, Á. Herrero, H. Quintián, and E. Corchado, Eds. Cham: Springer International Publishing, 2017, pp. 584–594, doi: 10.1007/978-3-319-47364-2_57.

[10] S. Slimani, T. Hamrouni, and F. B. Charrada, "SCoRMiner: Automated Discovery of Network Service and Application Dependencies using a graph mining approach," *Procedia Computer Science*, vol. 176, pp. 985–994, 2020, Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 24th International Conference KES2020, doi: 10.1016/j.procs.2020.09.094.

[11] "The Complete Guide to Application Dependency Mapping," Faddom, accessed on Oct 9, 2023. [Online]. Available: https://faddom.com/wp-content/uploads/2023/08/FD-ADM_Guide.pdf

[12] A. Kumar, S. S. Singh, K. Singh, and B. Biswas, "Link prediction techniques, applications, and performance: A survey," *Physica A: Statistical Mechanics and its Applications*, vol. 553, p. 124289, 2020, doi: 10.1016/j.physa.2020.124289.

[13] D. Zhang, J. Yin, X. Zhu, and C. Zhang, "Network Representation Learning: A Survey," *IEEE Transactions on Big Data*, vol. 6, no. 1, pp. 3–28, 2020, doi: 10.1109/TBDATA.2018.2850013.

[14] A. Grover and J. Leskovec, "Node2vec: Scalable Feature Learning for Networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 855–864, doi: 10.1145/2939672.2939754.

[15] K. Kaynar, "A taxonomy for attack graph generation and usage in network security," *Journal of Information Security and Applications*, vol. 29, pp. 27–56, 2016, doi: 10.1016/j.jisa.2016.02.001.

[16] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: a survey," *Data Mining and Knowledge Discovery*, vol. 29, no. 3, p. 626–688, 2014, doi: 10.1007/s10618-014-0365-y.

[17] S. Noel, *A Review of Graph Approaches to Network Security Analytics*. Springer International Publishing, 2018, pp. 300–323, doi: 10.1007/978-3-030-04834-1_16.

[18] M. Husák, J. Khoury, Đ. Klisura, and E. Bou-Harb, *On The Provision of Network-Wide Cyber Situational Awareness via Graph-Based Analytics*. Springer, 2023.

[19] B. Bowman and H. H. Huang, "Towards Next-Generation Cybersecurity with Graph AI," *SIGOPS Operating Systems Review*, vol. 55, no. 1, p. 61–67, jun 2021, doi: 10.1145/3469379.3469386.

[20] M. Atzmueller and R. Kanawati, "Explainability in cyber security using complex network analysis: A brief methodological overview," in *Proceedings of the 2022 European Interdisciplinary Cybersecurity Conference*, ser. EICC '22. ACM, 2022, p. 49–52, doi: 10.1145/3528580.3532839.

[21] S. Lagraa, M. Husák, H. Seba, S. Vuppala, R. State, and M. Ouedraogo, "A review on graph-based approaches for network security monitoring and botnet detection," *International Journal of Information Security*, pp. 1–22, 2023, doi: 10.1007/s10207-023-00742-7.

[22] M. Laštovička and P. Čeleda, "Situational awareness: Detecting critical dependencies and devices in a network," in *Security of Networks and Services in an All-Connected World*. Cham: Springer International Publishing, 2017, pp. 173–178, doi: 10.1007/978-3-319-60774-0_17.

[23] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014, doi: 10.1109/COMST.2014.2321898.

[24] S. G. Aksoy, E. Purvine, and S. J. Young, "Directional Laplacian Centrality for Cyber Situational Awareness," *Digital Threats: Research and Practice*, vol. 2, no. 4, oct 2021, doi: 10.1145/3450286.

[25] A. S. Pope, D. R. Tauritz, and M. Turcotte, "Automated design of tailored link prediction heuristics for applications in enterprise network security," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1634–1642, doi: 10.1145/3319619.3326861.

[26] S. Noel and V. Swarup, "Dependency-based link prediction for learning microsegmentation policy," in *Information and Communications Security*. Cham: Springer International Publishing, 2022, pp. 569–588, doi: 10.1007/978-3-031-15777-6_31.

[27] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online Learning of Social Representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 701–710, doi: 10.1145/2623330.2623732.

[28] W. W. Lo, S. Layeghy, M. Sarhan, M. Gallagher, and M. Portmann, "E-GraphSAGE: A Graph Neural Network based Intrusion Detection System for IoT," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, doi: 10.1109/NOMS54207.2022.9789878.

[29] J. Busch, A. Kocheturov, V. Tresp, and T. Seidl, "NF-GNN: Network Flow Graph Neural Networks for Malware Detection and Classification," in *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management*, ser. SSDBM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 121–132, doi: 10.1145/3468791.3468814.

[30] H. Wang, Y. Wu, G. Min, and W. Miao, "A graph neural network-based digital twin for network slicing management," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 2, pp. 1367–1376, 2022, doi: 10.1109/TII.2020.3047843.

[31] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, p. 37–57, mar 1985, doi: 10.1145/3147.3165.

[32] "PyG Documentation," PyG Team, 2023, accessed: Jul 31, 2023. [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/

[33] L. Sadlek, M. Husák, and P. Čeleda, "Supplementary Materials: Identification of Device Dependencies Using Link Prediction," Zenodo, jan 2024, doi: 10.5281/zenodo.10548433.

[34] M. Fey and J. E. Lenssen, "Fast Graph Representation Learning with PyTorch Geometric," *arXiv preprint arXiv:1903.02428*, 2019, accessed on Sep 6, 2023. [Online]. Available: https://arxiv.org/abs/1903.02428

[35] D. Tovarňák, S. Špaček, and J. Vykopal, "Traffic and log data captured during a cyber defense exercise," *Data in Brief*, vol. 31, p. 105784, 2020, doi: 10.1016/j.dib.2020.105784.

[36] D. Tovarňák, S. Špaček, and J. Vykopal, "Traffic and Log Data Captured During a Cyber Defense Exercise," Zenodo, apr 2020, doi: 10.5281/zenodo.3746129.

[37] B. Peddycord, A. Natarajan, and P. Ning, "NSDMiner: Tool for identifying Network Service Dependencies," 2015, accessed on Sep 6, 2023. [Online]. Available: https://sourceforge.net/projects/nsdminer/

[38] Y. Yang, R. N. Lichtenwalter, and N. V. Chawla, "Evaluating link prediction methods," *Knowledge and Information Systems*, vol. 45, pp. 751–782, 2015, doi: 10.1007/s10115-014-0789-0.

[39] P. L. H. Yu, J. Gu, and H. Xu, "Analysis of ranking data," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 11, no. 6, p. e1483, 2019, doi: 10.1002/wics.1483.

[40] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020, doi: 10.1016/j.neucom.2020.07.061.