Fast and Configurable Detection of Device Dependencies in Network Traffic

Jakub Dusil*, Martin Husák*, Lukáš Sadlek†
*Faculty of Informatics, Masaryk University, Brno, Czech Republic
†Institute of Computer Science, Masaryk University, Brno, Czech Republic
536566@mail.muni.cz, husak@fi.muni.cz, sadlek@ics.muni.cz

***Abstract*—Device dependencies are recurring communication patterns between IP addresses that reveal how networked entities rely on one another. Understanding these relationships is essential for reliability, troubleshooting, and security, yet detecting them efficiently from operational traffic remains challenging. We propose a fast and accurate tool for dependency detection from passive flow-level data using a link prediction approach. In contrast to the prior implementation, the tool introduces a parallelized processing pipeline with early termination of stalled random walks, an expanded feature set that combines embedding-derived and graph-theoretic metrics, and a fully externalized configuration of sampling, embedding, and classification parameters. These design choices enable scalable execution and more reliable identification of dependencies across diverse network environments. Evaluation on synthetic traffic from cyber-defense exercises and real-world campus flows demonstrates up to 100× faster runtime and markedly higher classification accuracy compared to the prior implementation. Further analysis shows that structural graph features improve stability in sparse settings, while extended embedding training enhances accuracy in low-signal scenarios. Together, these results confirm that the proposed tool advances link prediction-based dependency detection toward practical, near-real-time use.**

***Index Terms*—network traffic analysis, device dependency, dependency detection, link prediction, graph-based machine learning**

I. Introduction

Modern computer networks are highly dynamic and complex, making it increasingly challenging for administrators and security analysts to determine how devices and services depend on one another. Accurate knowledge of these dependencies is essential for maintaining network reliability, diagnosing issues, and enhancing security.

Network monitoring approaches are generally classified as *active* or *passive* [1]. Active monitoring injects probe traffic, such as ICMP echo requests or traceroutes, to collect information about network paths, reachability, or latency. Passive monitoring instead observes and analyzes traffic generated during normal operations, avoiding additional network load.

Recent research has shown that device dependencies can be inferred from flow-level data using *graph-based machine learning*, specifically the *link prediction* technique [2]. In this approach, a communication graph is constructed from observed flows, and graph embeddings are learned to capture structural and temporal patterns indicative of dependencies. A prior Python-based prototype implemented this concept and demonstrated that combining passive flow analysis with embedding-based classification can uncover such relationships.

Although the Python prototype demonstrated the feasibility of the approach, its execution time on large or high-volume datasets makes it unsuitable for practical use. In addition, it offers limited configurability across processing stages and yields suboptimal classification accuracy in complex environments. These limitations are especially problematic in operational settings that require timely, adaptable, and reliable analysis.

To overcome the limitations of the prior Python prototype, we present a high-performance *C++* tool that offers several new contributions beyond the earlier implementation: a parallelized processing pipeline, an expanded configurable feature set that integrates graph-theoretic and embedding-derived metrics, and fully externalized parameter control for rapid adaptation to diverse network conditions. Evaluation of the tool using both synthetic traffic from cyber-defense exercises and real-world campus flows demonstrates substantial improvements in runtime efficiency, predictive performance, and flexibility compared to the original prototype.

The remainder of this paper is organized as follows. Section II reviews related work on dependency identification. Section III details the design and implementation of the enhanced C++ tool. Section IV presents the experimental setup, datasets, and results. Finally, Section V concludes and outlines future research directions.

II. Related Work

We review existing research on identifying dependencies between networked entities using passive flow-level traffic analysis. To provide the necessary background, we first outline the commonly recognized types of dependencies that form the basis of most detection techniques. We then examine two representative solutions selected for their relevance and methodological contrast: NSDMiner [3], a tool for identifying local-remote dependencies, and a graph-based machine learning prototype based on the link prediction principle [2]. These systems differ in scope, underlying algorithms, and performance characteristics, and their comparative analysis informs the design goals of our proposed solution.

A. Types of Network Dependencies

This work adopts a widely used classification scheme introduced in previous research [2]–[4], which categorizes

978-3-903176-75-1 ©2025 IFIP

dependencies into three principal types. Each type corresponds to a recurring communication pattern observed between network entities based on IP flow records:

- **Direct dependency (DD):** A direct dependency refers to a frequently recurring communication from one IP address to another, where all flow attributes, except for timestamps, are consistent, and the frequency exceeds a predefined threshold [2]. This typically suggests a stable and regular interaction between two entities.
- **Local-remote dependency (LR):** A local-remote dependency arises when a local entity initiates communication with a remote peer in response to an earlier inbound request. For example, a host processing an incoming client request may access an external database or DNS service as part of completing that request [2], [3]. These patterns highlight supportive interactions triggered by preceding communication.
- **Remote-remote dependency (RR):** A remote-remote dependency captures an indirect link between two external entities, where one is typically contacted as a prerequisite to interacting with the other. A typical example is querying a DNS resolver to obtain the address of a target server, followed by communication with that resolved address [2], [4].

B. Passive Dependency Identification Solutions

Various techniques have been proposed to identify dependencies between communicating hosts by analyzing network traffic. While some rely on active probing, our focus is on passive methods, which infer relationships without introducing additional traffic or altering system behavior. Operating solely on observed communication patterns, these approaches are well-suited for production environments where minimal intrusion is essential.

We focus on two representative passive solutions that have strongly influenced this work. The first, NSDMiner [3], is a flow-based tool that applies temporal correlation to infer service-level dependencies. It is valued for its accuracy and efficiency, but is limited to detecting LR dependencies. The second is a Python-based prototype based on the link prediction principle [2], which uses graph embeddings from constrained random walks to detect multiple dependency types. While offering broader coverage, its performance and scalability limit its applicability in real-world scenarios.

The following subsections describe these two systems in detail, outlining their algorithmic foundations, strengths, and limitations.

C. NSDMiner

NSDMiner (Mining for Network Service Dependencies), proposed by Natarajan et al. [3], is a passive flow analysis tool for identifying LR dependencies. It introduces a metric $dweight(A, B)$ to quantify the strength of a potential dependency between services A and B, based on how often flows from A to B occur within the active window of preceding inbound flows to A. A dependency is reported when $dweight(A, B)/accesses(A)$ exceeds a user-defined threshold α, filtering out weak or coincidental interactions.

Operating on chronologically ordered flows, NSDMiner increments $dweight(A, B)$ whenever an overlapping inbound-outbound flow pair is found. Its design is automated, non-intrusive, and memory-efficient, discarding inactive flows to maintain near-constant per-record processing time. It outperforms earlier tools such as Orion [4] and Sherlock [5] in accuracy and false-positive rate, and supports high-throughput environments with flexible scoring modes and service aggregation. Its effectiveness has been demonstrated on multi-week real-world campus network datasets [3].

However, NSDMiner is limited to LR dependencies and cannot detect RR relationships common in client-centric traffic [3]. Accuracy is sensitive to the threshold α, and rare dependencies or overlapping inbound flows can lead to false positives or false negatives. Sparse traffic may require extended observation, and persistent connections (e.g., SSH, Remote Desktop) can produce misleading patterns if not filtered, further reducing accuracy.

D. Link Prediction-Based Prototype

Sadlek et al. [2] developed a graph-based machine learning method for detecting dependencies between networked devices from passively collected flow-level traffic. Based on the link prediction principle, it uses latent representations of communication patterns to infer functional relationships between IP addresses. The approach was implemented as a Python prototype and evaluated on controlled and real-world datasets.

The system processes NetFlow/IPFIX-style records by constructing a directed communication graph, where nodes represent IP addresses and edges represent individual flow records annotated with timestamps, ports, and transport protocol (TCP/UDP). Dependency structures are modeled through constrained random walks over the graph, which simulate sequences of communication events under temporal rules reflecting realistic LR and RR interactions [2]. Candidate dependency pairs are extracted from these walks using a sliding window, pairing the first IP in the window with others to preserve temporal and structural context.

A neural embedding model then learns vector representations of IP addresses, with related devices positioned closely in the latent space. Two embeddings are combined via a scalar product to represent a dependency, and these feature vectors, labeled with known dependencies, train a classifier to predict dependency existence. Due to embedding symmetry, the method cannot infer the direction of dependency.

The prototype is passive, non-intrusive, and capable of detecting DD, LR, and RR dependencies, including indirect relationships. However, embedding training is computationally expensive, limiting real-time applicability. Performance is sensitive to flow quality and sampling configuration, with misconfigured sampling distorting the communication structure. Rare dependencies cause class imbalance, and missing or incomplete flow data can further degrade accuracy, particularly in large-scale or noisy environments.

III. Methodology

The proposed tool was designed to overcome the performance, scalability, and configurability limitations of the Python-based link prediction prototype [2]. Our design was guided by the needs of network administrators, security analysts, and other operational users who require accurate, efficient, and non-intrusive methods for identifying device dependencies in large-scale environments. The implementation is publicly available in the project's GitHub repository.[1]

Compared to the prior prototype, the proposed tool provides several new contributions: a high-performance C++ implementation optimized for high-throughput flow processing, extensive parallelization of computationally intensive stages with early termination of stalled random walks, an expanded configurable feature set that combines learned embeddings with graph-theoretic metrics to improve predictive accuracy, and a modular interface-driven architecture that allows independent replacement or extension of major components. It further introduces a fully externalized configuration of operational parameters, including sampling, embedding, features, and classifier behavior, enabling rapid adaptation to diverse network conditions.

A. Processing Pipeline

The proposed tool transforms raw flow-level records into dependency predictions through a sequence of interconnected stages (overview in Figure 1). The process begins with *flow preprocessing*, where JSON-formatted network flow records are parsed using the *Boost.JSON* library [6], blocked or irrelevant IP addresses are removed, and the most active internal and external hosts are selected according to communication frequency. A bounded eviction-based counter ensures that the highest-activity hosts are retained without excessive memory usage, and reservoir sampling [7] selects a representative flow subset for each host.

The filtered flows are then used to construct a directed communication graph, where vertices correspond to IP addresses and edges represent individual network flows annotated with timestamps, port numbers, and protocol types. This graph, implemented using the *Boost.Graph* library [8], serves as the structural foundation for all subsequent analysis.

To capture temporal and structural patterns indicative of dependencies, the tool generates constrained random walks over the communication graph. These walks follow domain-specific timing rules, introduced in the prior link prediction approach [2], that reflect realistic multi-step interactions, including the opening and returning transitions of LR and RR dependencies. Unlike the prior work, the walk generation is parallelized across multiple threads and applies early termination when no valid continuation exists, avoiding unnecessary computation and improving runtime efficiency.

The resulting sequences are passed to the *context extraction* stage, where a sliding window produces overlapping subsequences that preserve local temporal and structural order. From each subsequence, the first node is paired with subsequent nodes to form positive candidate dependencies, while negative samples are generated by selecting nodes outside the subsequence. This combination of positive and negative examples enables supervised embedding training.

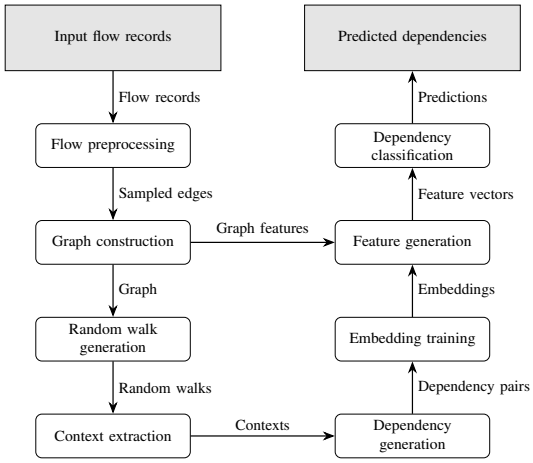

Fig. 1. High-level data flow in the dependency detection pipeline, illustrating the transformation of input flow records into predicted dependencies.

In the *embedding learning* stage, a Skip-Gram model is trained to produce low-dimensional vector representations of IP addresses based on the candidate pairs, similarly to the previous prototype. The embedding model is implemented using the *PyTorch C++ API (LibTorch)* [9], which provides efficient tensor operations. These embeddings capture both the structural position and communication behavior of nodes in the network.

The *feature generation* stage extends the original prototype by complementing the learned embeddings with a substantially expanded and fully configurable set of features. These include similarity-based metrics (e.g., cosine similarity, Euclidean distance), statistical descriptors (e.g., embedding norm ratios, component standard deviations), and element-wise (Hadamard) products. A major extension of our work is the integration of multiple graph-theoretic measures, such as the Jaccard coefficient, common neighbors, Adamic-Adar index, preferential attachment, and resource allocation scores. All feature extraction parameters are configurable, allowing the tool to tailor the feature space to different datasets and operational requirements.

The final stage, *dependency classification*, evaluates the feature vectors using a Random Forest classifier implemented with the *mlpack* machine learning library [10]. Numerical operations in this stage, including feature scaling and matrix preparation, are accelerated using the *Armadillo* C++ linear algebra framework [11], [12]. The classifier supports decision threshold calibration, class weighting to mitigate label imbalance, optional feature scaling, and hyperparameter optimization through grid search, enabling fine control over the precision–recall trade-off for different deployment requirements.

The tool operates in three distinct modes. In *training* mode, a model is learned from labeled flow data, which may include externally provided ground truth generated by other methods;

[1] https://github.com/xdusil/link-prediction-tool

TABLE I
FLOW AND IP ADDRESS COUNTS FOR EACH DATASET BEFORE AND AFTER PREPROCESSING, ALONG WITH THE NUMBER OF UNIQUE UNDIRECTED DEPENDENCIES IDENTIFIED POST-PREPROCESSING.

		T1	T2	T3	T4	T5	T6	MU
IP flows	Original	66,499	116,897	63,400	88,734	78,254	30,781	100,000
	Preprocessed	61,346	54,941	34,721	63,266	46,253	28,506	99,792
IP addresses	Original	705	1 457	672	1 209	1 059	256	342
	Preprocessed	689	1 421	654	1 190	1 047	247	305
Unique dependencies		2,735	1,224	1,674	3,772	605	206	27,206

in *prediction* mode, the model is applied to unseen flows to identify likely dependencies; and in *ground truth* mode, dependencies are derived directly from flows using rule-based heuristics [2]. This separation of modes enables both offline research workflows and near-real-time operational deployment.

IV. EVALUATION

The proposed tool was evaluated for predictive accuracy, execution efficiency, and robustness under varying configurations. First, we compare its performance to the prior work by Sadlek et al. [2], focusing on classification quality, runtime, and reliability. Second, we analyze how selected configuration parameters affect results across diverse datasets. Experiments were conducted on a workstation with six CPU cores at 2.8 GHz, 16 GB RAM, and OpenMP-enabled parallel execution. Unless stated otherwise, training and prediction used all available hardware threads.

A. Datasets

The evaluation used two sources of flow-level data. The first was synthetic traffic from the *Cyber Czech 2019* cyber defense exercise [13], capturing realistic attack and defense activity in six independent Blue Team networks. These datasets are denoted **T1**–**T6**. The second was real-world operational traffic from the Masaryk University campus network (Czech Republic), denoted **MU**, recorded on 17 February 2025 during a 28-minute interval and representing production activity.

In all datasets, preprocessing removed flows involving Red Team addresses (for T1–T6), non-TCP/UDP protocols, and incomplete records. Table I summarizes the number of flows and unique IP addresses before and after preprocessing, together with the number of unique undirected dependencies discovered afterwards. The datasets vary substantially in traffic volume and network complexity, which is essential for assessing the tool's adaptability.

B. Ground Truth Construction

Ground truth was generated using the brute-force enumeration method of Sadlek et al. [2], which scans all unidirectional flow sequences within a temporal window to identify valid dependency relationships. Only dependencies observed at least ten times were retained, with the maximum time gap between related flows limited to 1000 milliseconds in both forward and reverse directions. Our implementation follows this procedure but corrects a flaw in the original Python code that caused many valid dependencies to be omitted, as discussed in Section IV-D.

C. Evaluation Configuration

The baseline configuration was chosen to balance predictive performance, comparability, and computational efficiency. Several parameters were taken directly from the original Python prototype [2] to ensure consistency, while others were chosen through preliminary experiments to reflect the scale and characteristics of the evaluation datasets. In graph construction, the number of internal IP addresses was limited to 50 and external IP addresses to 100, with a cap of 500 flows per IP. Minimum occurrence thresholds for dependencies and next-step transitions in random walks were set to 10, ensuring that only frequent patterns were included and noise from rare or coincidental interactions was reduced.

Random walks used a length of 5 and a context size of 4. The embedding model employed 64-dimensional vectors, trained for 15 epochs with a learning rate of 0.01, and one negative sample per positive pair. The classifier used a Random Forest with 50 trees, a maximum depth of 30, a minimum leaf size of 1, and no minimum gain constraint for node splits. Class weighting was enabled to mitigate label imbalance, and min-max feature scaling was applied before classification. Automatic threshold calibration optimized the F1–score decision boundary, while grid search was disabled to reduce training time. No limit was imposed on processing threads, allowing embedding training and classification to fully exploit available hardware parallelism.

The active feature set combined embedding-derived and graph-theoretic descriptors: cosine similarity, Euclidean (L_2) distance, embedding norm ratio, Hadamard product sum, common neighbors count, Jaccard coefficient, Adamic–Adar index, preferential attachment score, and normalized node degree. These features capture both latent-space similarity and explicit structural connectivity between node pairs, maximizing the classifier's ability to detect diverse dependency types.

D. Evaluation against the Python Prototype

We compared our tool to the Python prototype by Sadlek et al. [2], executed with its default configuration from the original publication. To ensure comparability, our setup matched parameters common to both implementations, changing only the number of training epochs to five. Both tools used the same ground truth procedure described in Section IV-B, with our

TABLE II
CLASSIFICATION PERFORMANCE OF THE PROPOSED TOOL COMPARED TO THE PYTHON PROTOTYPE (TEST SPLIT = 25%) ACROSS ALL DATASETS.

Method	Metric	T1	T2	T3	T4	T5	T6	MU
Our tool	Accuracy	0.91	0.91	0.90	0.89	0.85	0.94	0.97
	Precision	0.74	0.32	0.47	0.44	0.24	0.17	0.98
	Recall	0.69	0.37	0.51	0.59	0.49	0.37	0.95
	F1–score	0.71	0.33	0.48	0.50	0.31	0.23	0.97
	ROC–AUC	0.92	0.87	0.87	0.89	0.82	0.87	0.99
Python prototype	Accuracy	0.61	0.57	0.60	0.60	0.46	0.40	0.98
	Precision	0.70	0.67	0.68	0.70	0.57	0.50	0.98
	Recall	0.82	0.78	0.82	0.80	0.66	0.61	1.00
	F1–score	0.75	0.72	0.75	0.74	0.61	0.55	0.99
	ROC–AUC	0.44	0.43	0.44	0.44	0.39	0.36	0.50

TABLE III
END-TO-END EVALUATION RUNTIMES (IN MINUTES) FOR THE PROPOSED TOOL AND THE PYTHON PROTOTYPE ACROSS ALL DATASETS. GROUND TRUTH GENERATION IS EXCLUDED.

Method	T1	T2	T3	T4	T5	T6	MU
Our tool	0.15	0.22	0.15	0.18	0.17	0.10	0.09
Python prototype	20.83	19.63	19.91	22.67	19.96	20.31	17.65

implementation correcting a flaw in the prototype that excluded many valid dependencies. Apart from this fix, the prototype's embedding, negative sampling, and classification logic were left unchanged.

To avoid structural bias, we temporarily adopted the prototype's practice of computing features before the train–test split, although our standard pipeline prevents such leakage. In our tool, duplicate dependency pairs (A, B) and (B, A) were filtered out to prevent memorization, whereas the prototype retains them. Evaluation was conducted on datasets **T1**–**T6** and **MU** with a 25% test split. A 50% split was also tested but yielded no additional insights and is omitted for brevity. Run time was measured from preprocessing start to evaluation end, excluding ground truth generation.

1) Identified Issues in the Python Prototype: Evaluation revealed four major flaws in the original implementation [2]. First, in `labels.py`, helper functions for counting LR and RR dependencies stopped after the first matching flow, omitting subsequent valid ones. Second, in `embedding.py`, the non-dependency check for negative-sample selection was inverted, mislabeling positives as negatives. Third, in `labels.py`, both (A, B) and (B, A) were stored for each dependency, and duplicates were added when multiple paths met the same condition, producing identical feature vectors across train–test splits, which enabled memorization. Finally, in `evaluation.py`, embeddings were computed on the full graph before splitting, causing structural leakage from test-set topology.

2) Discussion of Results: Tables II and III highlight three main benefits of the proposed implementation compared to the Python prototype: stronger ability to distinguish true from false dependencies, a more reliable precision–recall profile, and substantially faster execution.

First, as shown in Table II, our tool achieves consistently strong discriminative performance, with ROC–AUC scores ranging from 0.82 to 0.99 across datasets. ROC–AUC measures how well a classifier separates positive from negative instances across all thresholds: values near 1.0 indicate excellent separability, while 0.5 corresponds to random guessing. By contrast, the Python prototype remains close to chance level (0.36–0.50). Its seemingly high precision and recall stem mainly from structural flaws, duplicate feature vectors, and memorization of repeated (A, B) / (B, A) pairs, rather than genuine predictive ability.

Second, our implementation achieved F1–scores between 0.23 and 0.97 for the 25% test split, reflecting a more realistic balance between precision and recall. This difference becomes most evident on the **MU** dataset: both systems report high accuracy, yet our implementation attains near-perfect AUC (0.99) and F1–score (0.97), demonstrating reliable generalization. The prototype, in contrast, records an AUC of only 0.50 despite its nominal accuracy (0.98).

Finally, *runtime performance* (Table III) shows a major efficiency gain. End-to-end evaluation finishes in less than 15 seconds per dataset with the new implementation, compared to 17–23 minutes for the Python prototype. The resulting speedup of almost 100× stems from the optimized C++ design with parallel execution and reduced memory demands, making the system practical for both large-scale analyses and near-real-time deployment.

In summary, the evaluation highlights that the proposed system combines higher predictive accuracy with dependable generalization across datasets, while maintaining efficiency suitable for operational use. Compared to the prior work, it excels in both modeling capability and runtime scalability. Importantly, these gains are not merely the result of correcting flaws in the Python prototype. Even if those issues were resolved, the prototype would still lack parallelization, structural features, and externalized configuration, leaving it far less accurate and scalable than the proposed implementation.

E. Evaluation with Various Configurations

To examine the influence of configuration choices, we conducted experiments on datasets **T4** and **MU**, which retained the largest number of flows after preprocessing. For realism, each dataset was divided temporally: the first 75% of flows served for training, while the final 25% were reserved for testing. Ground truth was generated separately for each partition as described in Section IV-B, ensuring that dependencies were discovered strictly within their respective time spans. Although some IP addresses appeared in both partitions, temporal separation of flows eliminated leakage.

Although T4 and MU are comparable in overall flow volume (about 63k and 100k flows, respectively), they differ sharply in dependency density. T4's test split contains only a few hundred unique dependencies, making it a sparse, low-signal setting. MU, in contrast, yields several thousand dependencies in testing, dominated by remote–remote relations, and thus represents a dependency-rich scenario. Together, the two datasets provide complementary evaluation conditions:

TABLE IV
CLASSIFICATION PERFORMANCE ON DATASETS T4 AND MU UNDER DIFFERENT FEATURE CONFIGURATIONS AND EMBEDDING TRAINING DURATIONS. METRICS REPORTED ARE ACCURACY (ACC), PRECISION (PREC), RECALL (REC), F1–SCORE (F1), AND ROC–AUC (AUC).

Configuration	T4					MU				
	Acc	Prec	Rec	F1	AUC	Acc	Prec	Rec	F1	AUC
Base model	0.97	0.24	0.16	0.20	0.88	0.97	0.95	0.94	0.94	0.99
Structural only	0.97	0.26	0.17	0.21	0.91	0.97	0.98	0.92	0.95	0.99
Embedding only	0.98	0.00	0.00	0.00	0.49	0.54	0.31	0.45	0.36	0.52
All features	0.98	0.29	0.11	0.16	0.83	0.96	0.96	0.92	0.94	0.98
50 epochs	0.97	0.28	0.18	0.22	0.93	0.96	0.92	0.94	0.93	0.99
250 epochs	0.97	0.29	0.24	0.26	0.94	0.96	0.94	0.94	0.94	0.99
500 epochs	0.98	0.43	0.31	0.36	0.95	0.97	0.95	0.95	0.95	0.99
750 epochs	0.98	0.51	0.38	0.43	0.96	0.96	0.94	0.94	0.94	0.99

T4 stresses the system under scarcity, while MU reflects a dense operational environment.

Unless otherwise specified, experiments adopted the baseline settings of Section IV-C, with the classifier threshold fixed at 0.5 and automatic calibration disabled. Each run modified only a single factor, either the active feature set or the number of embedding training epochs, while all other parameters were held constant. Models were trained on the full training split and evaluated on the corresponding test split.

1) Results and Discussion: The outcomes in Table IV highlight distinct behaviors across feature sets and evaluation settings. On MU, accuracy and AUC remain near 0.97–0.99 under all conditions, reflecting the richness of available patterns. T4 proves more difficult: with only 232 positives in testing, class imbalance and sparse structure make performance highly sensitive to configuration.

Structural graph features deliver stable predictions in both datasets (F1–score = 0.21 on T4, 0.95 on MU), confirming their utility in sparse settings. Embeddings alone, however, collapse in performance (F1–score = 0.00 on T4, 0.36 on MU), demonstrating that they cannot support discrimination without complementary signals. Combining all features recovers some accuracy but falls short of the curated baseline, showing that unfiltered inclusion of descriptors is suboptimal.

Extended embedding training improves results in the sparse T4 case, raising the F1–score from 0.22 at 50 epochs to 0.43 at 750 epochs, with AUC increasing to 0.96. For MU, longer training provides little additional benefit, as most dependencies are already captured at lower epochs. Even the most demanding setting (750 epochs) remains computationally practical, requiring about 20 seconds for training and 10 seconds for prediction on T4, and roughly 10 seconds for training and 6 seconds for prediction on MU. For typical configurations ($\leq$250 epochs), both training and prediction complete in under 10 seconds, supporting use in real-time as well as batch analysis.

These findings emphasize that structural features are indispensable for stability, embeddings add complementary value in sparse regimes when sufficiently trained, and the system remains efficient across a range of operating conditions.

V. CONCLUSION

This work introduced a high-performance *C++* tool for identifying device dependencies from network flow data using a link prediction approach. The system addresses major shortcomings of prior solutions in scalability, configurability, and predictive reliability through a parallelized processing pipeline, an expanded feature set that integrates graph-theoretic and embedding-derived metrics, and fully externalized parameter control. Evaluation on both synthetic and real-world datasets demonstrated clear advantages over the baseline Python prototype, with up to two orders of magnitude faster execution, consistently higher ROC–AUC scores, and more stable precision–recall trade-offs. The tool proved robust across diverse network conditions and was easily adaptable to different configurations without architectural modifications. Future work may explore reducing reliance on explicit ground truth through semi-supervised learning and extending the model to capture dependency type and direction.

REFERENCES

[1] R. Hofstede, P. Čeleda, B. Trammell, I. Drago et al., "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.

[2] L. Sadlek, M. Husák, and P. Čeleda, "Identification of Device Dependencies Using Link Prediction," in *Proceedings of the 2024 IEEE Network Operations and Management Symposium*. IEEE, 2024, pp. 1–10.

[3] A. Natarajan, P. Ning, Y. Liu, S. Jajodia et al., "NSDMiner: Automated discovery of Network Service Dependencies," in *Proceedings of the IEEE INFOCOM 2012 Conference*. IEEE, 2012, pp. 2507–2515.

[4] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 117–130.

[5] P. Bahl, R. Chandra, A. Greenberg, S. Kandula et al., "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*. ACM, 2007, pp. 13–24.

[6] V. Falco, K. Stasiowski, and D. Arkhipov. (2024) Boost.JSON. Accessed: 2025-05-04. [Online]. Available: https://www.boost.org/doc/libs/1_86_0/libs/json/doc/html/index.html

[7] J. S. Vitter, "Random Sampling with a Reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, 1985.

[8] J. Siek, L.-Q. Lee, and A. Lumsdaine. (2001) The Boost Graph Library (BGL). Accessed: 2025-05-04. [Online]. Available: https://www.boost.org/doc/libs/1_86_0/libs/graph/doc/index.html

[9] J. Ansel, E. Yang, H. He, N. Gimelshein et al., "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, 2024, pp. 929–947.

[10] R. R. Curtin, M. Edel, O. Shrit, S. Agrawal et al., "mlpack 4: a fast, header-only C++ machine learning library," *Journal of Open Source Software*, vol. 8, no. 82, p. 5026, 2023.

[11] C. Sanderson and R. Curtin, "Armadillo: An Efficient Framework for Numerical Linear Algebra," in *Proceedings of the 2025 17th International Conference on Computer and Automation Engineering (ICCAE)*. IEEE, 2025, pp. 303–307.

[12] ——, "Practical Sparse Matrices in C++ with Hybrid Storage and Template-Based Expression Optimisation," *Mathematical and Computational Applications*, vol. 24, no. 3, p. 70, 2019.

[13] D. Tovarňák, S. Špaček, and J. Vykopal. (2020, Apr.) Traffic and Log Data Captured During a Cyber Defense Exercise . [Online]. Available: https://zenodo.org/records/3746129