International Journal of Advances in Applied Sciences (IJAAS)
Vol. 15, No. 2, June 2026, pp. 583~593
ISSN: 2252-8814, DOI: 10.11591/ijaas.v15.i2.pp583-593

Program defect prediction model based on topology aware node evaluation pool graph topology model

Dan Li[1,4], Poh Soon JosephNg[1], Peng Yin Choo[2], Koo Yuen Phan[2], Wong See Wan[3]
[1]Institute of Computer Science and Digital Innovation, UCSI University, Kuala Lumpur, Malaysia
[2]Faculty of Information and Communication Technology, Universiti Tunku Abdul Rahman, Kampar, Malaysia
[3]Vise Trading, Penang, Malaysia
[4]Shenzhen CEPREI Industry Technology Research Institute Co. Ltd., Shenzhen, China

Article Info

Article history:

Received Aug 19, 2025
Revised Apr 8, 2026
Accepted Apr 22, 2026

Keywords:

Artificial intelligence
Fuzzy testing
Graph neural network
Machine learning
Program defect prediction
Sustainable infrastructure
Vulnerability discovery

ABSTRACT

Traditional fuzzing struggles with efficiency, as maximizing code coverage does not guarantee the discovery of additional vulnerabilities. To solve this, the study introduces topology-aware node evaluation (TANE-Pool), a deep learning model that proactively predicts defects to guide the fuzzing process. The model analyzes the structural properties of a program's attributed control flow graph (ACFG) via a diffusion attention mechanism. This process identifies fragile code regions and generates a static vulnerability score (SVS) for each basic block. The fuzzer then uses this score to prioritize test cases, concentrating its efforts on the areas most likely to contain flaws. Evaluated on the Juliet test suite and a set of real-world programs, TANE-Pool demonstrates superior prediction accuracy. Its integration into a fuzzer significantly enhances the rate of vulnerability discovery, proving that a defect-prediction-guided approach is a more efficient and effective strategy for software security testing in sustainable infrastructure development.

Corresponding Author:

Poh Soon JosephNg
Institute of Computer Science and Digital Innovation, UCSI University
1st. UCSI, UCSI Heights (Taman Connaught), Cheras-56000, Kuala Lumpur, Malaysia
Email: joseph.ng@ucsiuniversity.edu.my

1. INTRODUCTION

Software systems have become indispensable to modern society, supporting critical domains such as aerospace, finance, and public infrastructure. As software scale and complexity continue to increase, ensuring system reliability has become a major challenge, particularly in security-sensitive environments. Among various reliability concerns, software vulnerabilities—defects introduced during design or implementation—pose severe risks because attackers may exploit them to compromise systems or obtain sensitive information [1], [2]. Therefore, early identification and mitigation of software vulnerabilities have become crucial for enhancing software security, and security testing has received growing attention from both academia and industry [3], towards a resilient infrastructure development.

Fuzzing has emerged as one of the most effective automated techniques for identifying vulnerabilities in real-world programs. By continuously mutating existing inputs and monitoring execution behaviors, fuzzing can detect crashes, abnormal logic, and anomalous states that indicate potential security flaws [3], [4]. Modern fuzzers, such as those based on evolutionary or feedback-driven strategies, prioritize test cases that trigger new execution paths, enabling deeper exploration of a program's behavior space [5]. However, the input spaces of complex software systems are often large and structurally constrained, making random mutation alone inefficient for vulnerability discovery. As a result, many unguided or weakly guided fuzzing strategies generate a large number of invalid inputs, reducing overall testing effectiveness.

To address this issue, coverage-guided grey-box fuzzing (CGF) was introduced, enabling fuzzers to retain only inputs that expand code coverage, thereby improving test efficiency and vulnerability detection [6].

Despite the success of CGF frameworks, they still exhibit important limitations. Unguided mutation wastes resources on unpromising seeds, scheduling strategies often rely solely on heuristics without predictive ability, and mutation operators rarely incorporate deeper semantic or structural characteristics of programs. As research on fuzzing has progressed, scholars have attempted to overcome these weaknesses through three primary directions: format-constraint-based test generation, information-feedback-based fuzzing, and symbolic-execution-based approaches [7]–[9]. Format-constraint-based methods (e.g., Peach, SPIKE, SNOOZE) generate structured inputs that adhere to strict syntactic specifications, improving coverage for protocols or highly structured file formats. However, these approaches require substantial manual effort and expert domain knowledge, limiting their scalability. Information-feedback-based fuzzing methods—represented by American fuzzy lop (AFL), AFLFast, and Syzkaller—use runtime data such as code coverage to guide test case mutation and selection, significantly enhancing search efficiency [10]–[12]. Yet, for programs with complex or deeply nested input structures, these approaches still suffer from the generation of invalid test cases, which lowers mutation efficiency and slows vulnerability discovery.

To further improve testing precision and coverage, symbolic execution–based approaches have been introduced to generate high-quality, path-directed test cases. Tools such as SAGE, KLEE, and TaintScope analyze execution paths symbolically and solve constraints to craft inputs that force exploration of specific program branches [13]–[15]. Although highly effective in theory, symbolic execution faces critical challenges such as path explosion and the high computational cost of constraint solving, which limit its applicability in large-scale real-world software systems.

Given these limitations across existing fuzzing paradigms, there remains a substantial need for more intelligent, structure-aware, and resource-efficient fuzzing guidance mechanisms. Motivated by this gap, this study introduces topology-aware node evaluation (TANE-Pool), a defect-prediction-enhanced fuzzing framework. Unlike previous works that rely solely on code metrics or flat neural networks, the core novelty of our approach lies in the fusion of TANE-based dependency extraction with a hierarchical graph pooling strategy. Specifically, program dependencies extracted via TANE capture latent structural relationships correlated with defect-proneness. By encoding these dependencies into graph representations and applying an adaptive pooling mechanism, the proposed model can identify seeds with a higher probability of triggering security-relevant behaviors. The predictions produced by the TANE-Pool model are further integrated into scheduling and mutation strategies, enabling more efficient allocation of fuzzing resources and deeper exploration of previously unreachable execution paths.

The main contributions of this study are threefold. First, we propose a dependency-driven defect prediction model based on TANE and graph pooling that captures structural cues associated with program vulnerabilities. Second, we design a prediction-guided scheduling strategy that prioritizes seeds using defect likelihood and fitness scoring, improving the efficiency of resource allocation within the fuzzing loop. Third, we introduce a mutation mechanism enhanced by learned representations, enabling deeper and more targeted path exploration. Experimental results demonstrate that the proposed approach significantly improves both code coverage and crash discovery compared with state-of-the-art fuzzers. This work provides a promising step toward the development of intelligent, scalable, and security-enhancing automated testing systems.

The remainder of this paper is organized as follows. Section 2 reviews the related work on deep learning in software testing and graph-based vulnerability detection. Section 3 details the proposed methodology, including the TANE-based dependency extraction, graph construction, and the TANE-Pool model architecture. Section 4 presents the experimental setup, dataset description, and a comprehensive analysis of the results compared to state-of-the-art baselines. Finally, section 5 concludes the paper and outlines directions for future research.

2. LITERATURE REVIEW

In recent years, the rapid advancement of deep learning in fields such as image recognition and natural language processing [16], [17] has spurred interest in its potential application to software testing [18]. As an artificial intelligence technology with powerful learning and generalization capabilities, deep learning offers new paradigms for the evolution of fuzzing techniques. Currently, a growing body of research is exploring the deep integration of these two fields to enhance the effectiveness and efficiency of fuzzing [19]. Given that the performance of fuzzing largely depends on the quality of test cases, optimizing test case generation and scheduling using deep learning has emerged as a key research direction [20].

In fuzzing, the test case scheduling strategy is a key determinant of testing efficacy, directly impacting resource allocation efficiency and the depth of vulnerability discovery. Traditional approaches typically evaluate each test case in the corpus based on its historical performance, assigning weights to determine the priority and frequency of subsequent mutations. However, these traditional evaluation strategies are often limited in their ability to capture complex program behaviors. The advancement of AI, particularly deep learning, has led to its application in software testing, where it has been recognized for its performance on complex tasks. In fuzzing, applying deep learning to key aspects such as test case screening and scheduling has yielded remarkable results, significantly improving both testing efficiency and vulnerability discovery [21]. This section reviews the research progress of deep learning in these two areas.

In the domain of test case scheduling, researchers build deep neural network models to parse a target program's source or binary code. The objective is to extract features closely related to program behavior and identify potentially vulnerable regions. By training models to learn the distributional characteristics of these critical regions, the test case scheduling strategy can be intelligently optimized, guiding the fuzzer to more effectively cover potentially high-risk execution paths. Typical research efforts in this area include NeuFuzz and V-Fuzz. NeuFuzz utilizes deep reinforcement learning methods to dynamically adjust its test case selection strategy, improving the validity of test samples as path exploration deepens [22]. V-Fuzz constructs a model based on an attention mechanism to measure the correlation between inputs and code coverage, thereby prioritizing the scheduling of high-value test cases [23]. This use of attention is part of a broader trend in applying sophisticated graph neural network (GNN) architectures to code. To handle the complexity of large code graphs, hierarchical pooling methods have been introduced to create coarse-grained representations of the graph. Models like DiffPool learn a differentiable cluster assignment to group nodes [24], while SAGPool uses a self-attention mechanism to score and select the most important nodes [25]. These techniques are vital for graph classification tasks, such as determining if an entire function is vulnerable, but their effectiveness depends heavily on how well they preserve the topological information relevant to security defects.

A significant subfield in this domain leverages graph-based representations of source code for vulnerability prediction. Researchers have utilized structures such as control flow graphs (CFGs), abstract syntax trees (ASTs), and code property graphs (CPGs) as inputs for deep learning models [26]. For instance, some models apply GNNs directly to these graphs to learn structural patterns indicative of vulnerabilities like buffer overflows or use-after-free errors. The outputs of these predictors are then used to create a "heat map" of potentially vulnerable functions or basic blocks, allowing fuzzers to prioritize inputs that exercise these high-risk code regions. Concurrently, mainstream fuzzing engines employ evolutionary search algorithms—such as coverage-guided genetic strategies or vulnerability-prioritized mutation—to direct test case generation toward uncovered and high-risk program paths [23], [27]. This approach marks a shift from coverage-guidance to vulnerability-guidance, though the accuracy of the underlying graph model is paramount to its success.

In terms of test case screening, existing research often employs neural network models as predictors to determine if a pending test case is likely to trigger a new program state, such as an exception or an unexplored code region. Some studies construct annotated datasets by collecting historical test cases and their resulting outcomes, which are then used to train a classification model [28], [29]. This model can subsequently predict whether new inputs are likely to trigger novel behaviors, providing a basis for test case screening in fuzzing. However, a systematic evaluation of such methods revealed that while they are effective in some scenarios, their overall contribution to enhancing fuzzing efficiency is limited. In contrast, the FuzzGuard framework introduced a prediction mechanism combined with a deep learning model. FuzzGuard utilizes previous test inputs and their runtime feedback as training data to build a model that screens out ineffective or redundant test cases, thereby enhancing resource utilization and improving overall testing efficiency [30]–[32]. Gan *et al*. [33] proposed GREYONE, a data flow sensitive fuzzer that uses fuzzing-driven taint inference and conformance-guided evolution to optimize mutation direction, enhancing execution efficiency and vulnerability discovery performance

3. RESEARCH METHOD

This section details the proposed TANE-Pool framework, designed to address the limitations of blind mutation in traditional fuzzing [34]–[36]. The methodology follows a pipeline approach: i) extracting latent program dependencies using the TANE algorithm; ii) constructing attributed control flow graphs (ACFGs) that encode both structural and semantic features; iii) applying a novel graph pooling mechanism to identify high-risk regions; iv) predicting defect probabilities; and v) integrating these predictions into a fuzzing scheduling strategy. This step-by-step procedure ensures the reproducibility of experiments [37]–[39].

3.1. TANE-based dependency extraction

To overcome the limitations of static analysis, which often misses complex data relationships, the TANE Pool, a dependency discovery algorithm, is employed to extract latent relationships within the target binary. While standard CFGs only capture execution paths, they fail to represent the functional dependencies between variables across different basic blocks. Justification: TANE was selected because it effectively mines functional dependencies and approximates functional dependencies from data, providing a richer semantic layer that complements the structural information of the CFGs. This step is crucial for identifying “deep” vulnerabilities that arise from complex data interactions rather than simple control flow errors.

3.2. Graph construction

The raw binary program is transformed into a graph-structured representation suitable for deep learning. Disassembly tools (e.g., IDA Pro) are utilized to parse the binary functions. The graph construction proceeds as follows. Node representation: each basic block in the function is treated as a node. To capture the local semantics, a multi-dimensional feature vector is extracted for each node, comprising the number of instructions, arithmetic operations, memory operations, and the distribution of opcodes. Edge establishment: edges are initially established based on the control flow jumps between basic blocks. Dependency enrichment: the dependencies extracted by TANE in sub-section 3.1 are superimposed onto the graph. If a functional dependency exists between basic blocks A and B, an additional edge (or weighted edge) is added. The resulting structure is an ACFG, which serves as a comprehensive representation of the function’s execution logic and semantic context3.

3.3. Pooling mechanism: diffusion attention strategy

Preserving topological information relevant to vulnerabilities. Traditional global pooling methods (e.g., summing all node features) often discard critical local structural cues. To address this, the TANE-Pool layer is introduced as shown in Figure 1, which employs a hierarchical pooling strategy based on diffusion attention.

i) Attention score calculation: first, the relevance between connected nodes is computed. The attention score $s_{i,j}^{(l)}$ for an edge between the nodes i and j at layer l is calculated as (1).

$$s_{i,j}^{(l)} = \delta(v_{\alpha}^{(l),T}(W^{(l)}h_i^{(l)}||W^{(l)}h_j^{(l)})) \tag{1}$$

Where || denotes vector concatenation, and $\delta(\cdot)$ is the LeakyReLU activation function. This mechanism allows the model to dynamically weigh the importance of immediate neighbors based on their feature compatibility.

ii) Diffusion attention for long-range dependencies: justification: vulnerabilities often involve interactions between basic blocks that are distant in the control flow (e.g., a memory allocation and a subsequent distant free). Standard graph convolution only aggregates local neighbors. To capture these long-range dependencies, we compute a diffusion attention score matrix (A_K) is computed as in (2).

$$A_K = \sum_{i=0}^{K} \theta_i A^i \tag{2}$$

Here A^i represents the transition matrix of i steps, and θ_i as a decay factor ($\theta_i > \theta_{i+1}$). This formulation expands the receptive field, allowing the model to "see" relationships up to KHops away, ensuring that distant but causally related code blocks influence the node importance score.

iii) Graph coarsening the pooling layer finally aggregates node features using A_K: a top-k selection method based on the aggregated importance scores is then employed to retain only the most critical nodes (basic blocks most likely to contain defects), forming a coarsened subgraph. This hierarchical approach preserves the most salient structural features while filtering out noise, as defined in (3).

$$Agg(G, H^{(l)}; \Theta) = A_K H^{(l)} \tag{3}$$

3.4. Prediction model architecture

The overall architecture, as shown in Figure 2, adopts a stacked graph convolutional network (GCN) backbone interleaved with TANE-Pool layers.

i) Feature extraction: the ACFG passes through three consecutive blocks. Each block consists of a GCN layer followed by a TANE-Pool layer. The GCN refines node features, while TANE-Pool reduces the graph size.

ii) Readout phase: after each pooling step, a readout function aggregates the features of the coarsened subgraph (using sum pooling) to generate a graph-level vector.

iii) Final prediction: the graph-level vectors from all three layers are concatenated to form a multi-scale representation. This vector is fed into a multi-layer perceptron (MLP) classifier, which outputs the probability of the function being vulnerable.

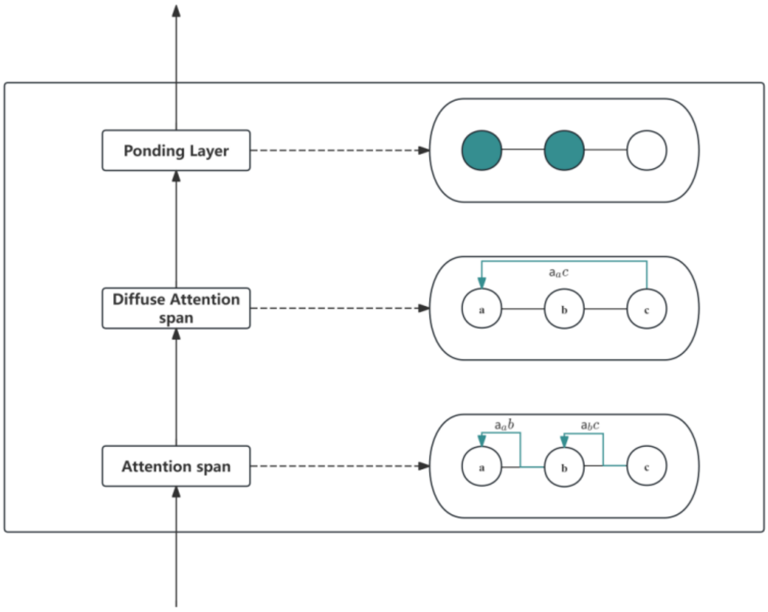

Figure 1. Detailed structure of the TANE-Pool layer incorporating diffusion attention mechanism

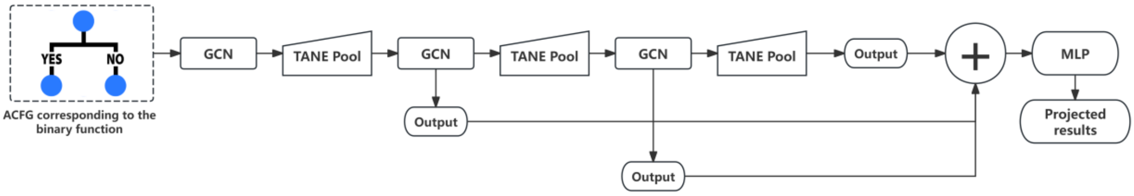

Figure 2. Architecture of the TANE-Pool program defect prediction model: from ACFG to probability output

3.5. Integration into fuzzing: prioritized scheduling

The ultimate goal of our model is to guide the fuzzer. The model's output is translated into a static vulnerability score (SVS) to prioritize test cases as illustrated in Figure 3. Justification: standard fuzzers (like AFL) treat all paths equally or focus solely on coverage. By integrating SVS, computational resources (energy) are directed toward paths that traverse high-risk basic blocks. For a binary function with vulnerability probability p_v, the SVS of its constituent basic blocks (b_i) is derived as (4).

$$SVS(b_i) = \alpha * p_v + \beta \tag{4}$$

Where $\alpha = 20$ and $\beta = 0.1$ are scaling constants determined empirically to differentiate risk levels.

During fuzzing, for a test case *t* executing a path $Path_t$, it is Fitness Score fit_t is the sum of the SVS of all visited blocks as in (5).

$$fit_t = \sum_{b_i \in Path_t} SVS(b_i) \tag{5}$$

Finally, mutation energy is allocated dynamically. Test cases with fit_t above the population average (fit_{avg}) are assigned double the standard energy $2p(t)$, effectively focusing the fuzzer's mutation efforts on the most suspicious code regions.

In order to further optimize the mutation energy scheduling, this paper adds the adaptation score indicator of the use cases to the original energy scheduling of AFL. A high adaptation score indicates that the use cases have a high priority and thus should be given more mutation energy. Mutation energy is the number of mutations in the HAVOC phase. HAVOC is a mutation phase of the fuzzifier represented by AFL, in which large-scale changes are made to the use cases.

The energy scheduling formula is shown in (6). For a given use case t, the energy $pf(t)$ assigned to it can be calculated from (6).

$$pf(t)\begin{cases}2p(t), if fit(t) \geq fit_{avg} \text{ and } p(t) \leq \frac{U}{2} \\ p(t), if\ fit(t) < fit_{avg} \\ U, otherwise\end{cases} \tag{6}$$

Where $p(t)$ is the mutation energy assigned to the use case t by traditional gray-box fuzzy testing tools (in particular, AFL), and U is the maximum energy that can be assigned by AFL. $fit(t)$ is the adaptation score corresponding to the use case t, and fit_{avg} is the average of the adaptation scores of all current use cases. When the fitness score $fit(t)$ of the use case t is greater than or equal to fit_{avg} and the raw energy $p(t)$ is less than or equal to half of U, $2p(t)$ of energy is assigned to the use case. When the fitness score $fit(t)$ of the use case t is less than fit_{avg}, the energy of $p(t)$ is assigned. If it is not either of the above, then allocate U the energy for the use case t.

Energy scheduling based on test case prioritization is designed to give more energy to the cases with a higher priority level, where the priority level is reflected by the fitness scores of the test cases. By using the average fitness score of the current test cases as a baseline, the cases with fitness scores higher than the baseline are given higher priority and are given more energy, and the cases lower than the baseline use the AFL's original allocation of energy. More mutation energy is assigned to use cases with higher fitness scores to improve the efficiency of mutation for fuzzy test cases.

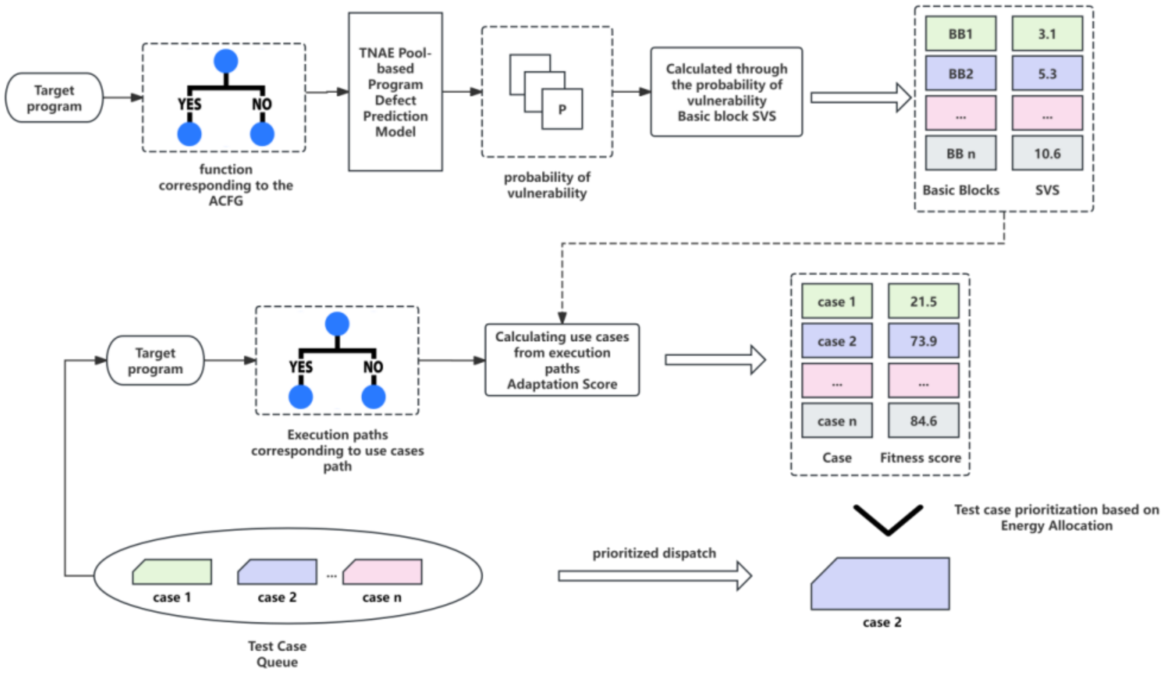

Figure 3. Schematic of the test case prioritization and energy scheduling framework guided by defect prediction

4. RESULTS AND DISCUSSION

4.1. Experimental setup and datasets

To comprehensively validate the TANE-Pool framework, two distinct datasets were utilized, as shown in Table 1: the Juliet test suite v1.3 and a curated real-world programs dataset. The Juliet suite provides a controlled environment with labeled memory-related weaknesses (e.g., buffer overflows), allowing for precise baseline comparisons. To evaluate generalization capability, the real-world dataset was constructed from open-source projects (via GitHub) and known exploits (via Exploit-DB), processed using IDA Pro for binary labeling. TANE-Pool was benchmarked against two categories of state-of-the-art models: i) standard GNNs: GCN, GraphSAGE, and graph attention network (GAT) (representing graph learning without hierarchical pooling) and ii) pooling-based methods*:* DiffPool and SAGPool (representing advanced topology-aware reduction).

Table 1. Training and testing dataset segmentation

Data set	Category	#Vulnerable samples	#Normal samples	Total
Juliet Test Suite v1.3	Training	20,000	20,000	40,000
	Testing	2,000	2,000	4,000
Real-world	Training	20,000	20,000	40,000
Programs	Testing	2,000	2,000	4,000

4.2. Performance analysis and comparison with literature

The comparative accuracy results are presented in Figure 4. TANE-Pool consistently outperformed all baselines on both datasets. Specifically, on the complex Real-world dataset, our model achieved a 3.2% accuracy gain over the strongest baseline, SAGPool.

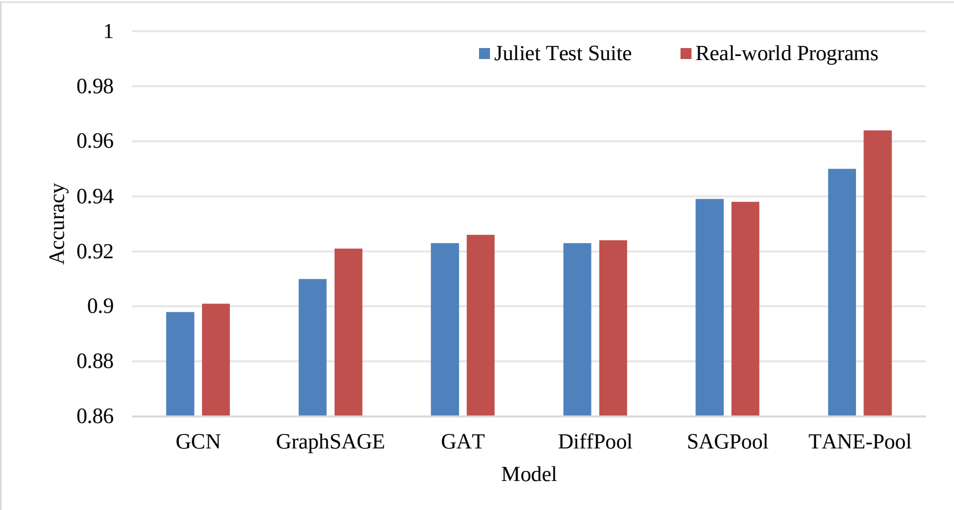

Figure 4. Comparison of the accuracy of different models on Juliet test suite and real-world programs datasets

Critical discussion on graph topology: the results highlight a significant performance gap between pooling-based methods (DiffPool, SAGPool, and TANE-Pool) and flat GNNs (GCN and GAT). Observation: GCN and GAT struggled to generalize on the real-world dataset. Literature validation: this finding aligns with the work of Zhou *et al*. [26], who demonstrated that GNN models capturing program semantics significantly outperform flat (non-graph) baselines for vulnerability detection, confirming that hierarchical structural information is essential. The contribution: By incorporating the TANE-Pool layer, our model effectively preserves the local substructures of defects. While Zhou *et al*. [26] demonstrated that graph-based program representations improve vulnerability detection, the results extend this by showing that combining dependency mining (TANE) with diffusion attention yields superior defect localization compared with topology learning alone. Why TANE-Pool beats SAGPool: although SAGPool is effective, it relies solely on self-attention for node dropping. In contrast, TANE-Pool leverages data dependencies extracted via TANE. Justification: a vulnerability often spans structurally distant but data-dependent basic blocks. Standard graph pooling might drop a "connector" node that seems topologically unimportant but carries critical data flow. TANE-Pool's diffusion mechanism ensures these latent dependencies are preserved.

4.3. Top-k accuracy and practical implications

For practical deployment in CI/CD pipelines, the "Top-k accuracy" (the proportion of actual vulnerabilities found within the top-k ranked candidates) is a more critical metric than overall classification accuracy. High top-k performance implies that fuzzer wastes less energy on false positives. Tables 2 and 3 present the top-k analysis. TANE-Pool shows a remarkable advantage in the "early discovery" phase. Result: on the real-world dataset, as shown in Table 3, TANE-Pool improved accuracy by 11.1% at top-800 compared to SAGPool.

Table 2. Comparison of top-k accuracy of different models on Juliet test suite dataset

Model	Top-200 accuracy	Top-400 accuracy	Top-600 accuracy	Top-800 accuracy	Top-1,000 accuracy	Top-1,200 accuracy
GCN	0.914	0.872	0.842	0.831	0.781	0.786
GraphSAGE	0.925	0.918	0.905	0.874	0.852	0.794
GAT	0.927	0.919	0.912	0.896	0.879	0.807
DiffPool	0.934	0.928	0.924	0.909	0.893	0.856
SAGPool	0.948	0.936	0.934	0.924	0.908	0.874
TANE-Pool	0.968	0.945	0.938	0.925	0.917	0.891

Table 3. Comparison of top-k accuracy of different models on Real-world programs dataset

Model	Top-200 accuracy	Top-400 accuracy	Top-600 accuracy	Top-800 accuracy	Top-1,000 accuracy	Top-1,200 accuracy
GCN	0.926	0.881	0.835	0.783	0.789	0.754
GraphSAGE	0.928	0.891	0.836	0.785	0.791	0.771
GAT	0.929	0.893	0.845	0.801	0.792	0.781
DiffPool	0.931	0.904	0.955	0.812	0.796	0.795
SAGPool	0.938	0.914	0.864	0.823	0.805	0.807
TANE-Pool	0.967	0.934	0.929	0.915	0.854	0.828

Practical relevance: as highlighted in the 2025 survey by Qiu *et al.* [4], the primary bottleneck in greybox fuzzing is the "scheduling of ineffective seeds," which dilutes the testing budget. Validation: the high top-k scores directly address this bottleneck. By accurately prioritizing the top 20% of suspicious blocks, TANE-Pool allows the integrated fuzzer (sub-section 3.5) to allocate energy where it is statistically most likely to yield crashes. This confirms that our defect-prediction-guided strategy is not just theoretically sound but operationally viable for large-scale software repositories.

4.4. Parameter sensitivity analysis

The impact of the diffusion distance (K) on model performance was further analyzed as shown in Figure 5. The accuracy peaks at $K = 4$ for both datasets. Interpretation: a distance of $K < 4$ captures insufficient context (missing long-range dependencies), while $K > 5$. It introduces noise from irrelevant code regions. This "sweet spot" at $K = 4$ suggests that memory-related vulnerabilities in C/C++ binaries typically manifest within a medium-range locality of 4 hops in the dependency graph. This empirical finding provides a reference for future researchers optimizing GNN depth for binary analysis.

Figure 5. Effect of different diffusion distances on the accuracy of prediction models

5. CONCLUSION

To address the inefficiency of traditional coverage-guided fuzzing, which often expends significant resources on low-risk paths without guaranteeing vulnerability discovery, this study successfully developed and validated TANE-Pool, an intelligent defect-prediction-guided fuzzing framework. By fusing TANE-based dependency extraction with a novel diffusion attention pooling mechanism, the proposed model effectively captures latent structural dependencies within binary programs that are typically missed by flat GNNs. Experimental results on both the Juliet test suite and real-world repositories demonstrated that TANE-Pool achieves superior accuracy compared to state-of-the-art baselines (including SAGPool and DiffPool), confirming that hierarchical topology preservation is essential for precise defect localization. Furthermore, the integration of SVS into the fuzzer's energy scheduling strategy proved to be a critical practical innovation; it enables the system to prioritize high-risk seeds, thereby directly addressing the bottleneck of ineffective mutation in continuous integration environments. While the current implementation focuses on C/C++ binaries, the framework's modular design establishes a solid foundation for future sustainable industrialization research, which will aim to enhance adaptability across diverse programming languages through transfer learning and advanced program analysis techniques.

FUNDING INFORMATION

Authors state no funding involved.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Dan Li	✓	✓	✓	✓	✓	✓		✓	✓				✓	
Poh Soon JosephNg	✓	✓		✓	✓	✓	✓			✓	✓	✓	✓	✓
Peng Yin Choo		✓			✓	✓	✓			✓				✓
Koo Yuen Phan		✓			✓	✓	✓			✓				✓
Wong See Wan		✓			✓	✓	✓			✓				✓

C : **C**onceptualization
M : **M**ethodology
So : **So**ftware
Va : **Va**lidation
Fo : **Fo**rmal analysis
I : **I**nvestigation
R : **R**esources
D : **D**ata Curation
O : Writing - **O**riginal Draft
E : Writing - Review & **E**diting
Vi : **Vi**sualization
Su : **Su**pervision
P : **P**roject administration
Fu : **Fu**nding acquisition

CONFLICT OF INTEREST STATEMENT

Authors state no conflict of interest.

INFORMED CONSENT

We have obtained informed consent from all individuals included in this study.

DATA AVAILABILITY

Data availability does not apply to this paper as no new data were created or analyzed in this study.

REFERENCES

[1] F. Y. Assiri and A. O. Aljahdali, "Software vulnerability fuzz testing: a mutation-selection optimization systematic review," *Engineering, Technology & Applied Science Research*, vol. 14, no. 4, pp. 14961–14969, Aug. 2024, doi: 10.48084/etasr.6971.

[2] B. A. Dapshima and S. K. Ahmad, "Evaluation and assessment of software security risks and vulnerabilities within the realm of secure DevOps," *International Journal For Multidisciplinary Research*, vol. 6, no. 4, Jul. 2024, doi: 10.36948/ijfmr.2024.v06i04.25026.

[3] V. Casola, A. De Benedictis, C. Mazzocca, and V. Orbinato, "Secure software development and testing: a model-based methodology," *Computers & Security*, vol. 137, Feb. 2024, doi: 10.1016/j.cose.2023.103639.

[4] J. Qiu, Y. Jiang, Y. Miao, W. Luo, L. Pan, and X. Zheng, "A survey of coverage-guided greybox fuzzing with deep neural models," *Information and Software Technology*, vol. 186, Oct. 2025, doi: 10.1016/j.infsof.2025.107797.

[5] R. Qian, Q. Zhang, C. Fang, and L. Guo, "Investigating coverage guided fuzzing with mutation testing," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, Jun. 2022, pp. 272–281, doi: 10.1145/3545258.3545285.

[6] R. Liang *et al.*, "Vulseye: detect smart contract vulnerabilities via stateful directed graybox fuzzing," *IEEE Transactions on Information Forensics and Security*, vol. 20, pp. 2157–2170, 2025, doi: 10.1109/TIFS.2025.3537827.

[7] G. Lacombe, D. Feliot, E. Boespflug, and M.-L. Potet, "Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities," *Journal of Cryptographic Engineering*, vol. 14, no. 1, pp. 147–164, Apr. 2024, doi: 10.1007/s13389-023-00310-8.

[8] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach, and J. Klein, "Dynamic security analysis on Android: a systematic literature review," *IEEE Access*, vol. 12, pp. 57261–57287, 2024, doi: 10.1109/ACCESS.2024.3390612.
[9] C. Zimmerman, J. DiVincenzo, and J. Aldrich, "Sound gradual verification with symbolic execution," *Proceedings of the ACM on Programming Languages*, vol. 8, pp. 2547–2576, Jan. 2024, doi: 10.1145/3632927.
[10] H. Yuan, X. Bo, Z. Lina, and L. Chengyang, "A control flow graph optimization method for enhancing fuzz testing," *IEEE Access*, vol. 12, pp. 169370–169378, 2024, doi: 10.1109/ACCESS.2024.3452938.
[11] Z. Zhang, "Systematic approaches for improving fuzzing evaluation and fuzzing program configurations." Ph.D. dissertation, The University of Texas at Dallas, Richardson, Texas, 2024.
[12] W. Chen *et al.*, "SyzGen++: dependency inference for augmenting kernel driver fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*, May 2024, pp. 4661–4677, doi: 10.1109/SP54263.2024.00269.
[13] A. Pokharel, "Detecting software vulnerability with symbolic execution and fuzzing." M.S. thesis, University of Idaho, Moscow, Russia, 2024.
[14] H. Tu, "Boosting symbolic execution for heap-based vulnerability detection and exploit generation," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, May 2023, pp. 218–220, doi: 10.1109/ICSE-Companion58688.2023.00059.
[15] N. Nascimento, "Safeguard analyzer: a fuzz testing tool." M.S. thesis, Universidade do Porto, Porto, Portugal, 2024.
[16] R. Archana and P. S. E. Jeevaraj, "Deep learning models for digital image processing: a review," *Artificial Intelligence Review*, vol. 57, no. 1, Jan. 2024, doi: 10.1007/s10462-023-10631-z.
[17] J. Liu *et al.*, "Application of deep learning-based natural language processing in multilingual sentiment analysis," *Mediterranean Journal of Basic and Applied Sciences*, vol. 8, no. 2, pp. 243–260, 2024, doi: 10.46382/MJBAS.2024.8219.
[18] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, Apr. 2024, doi: 10.1109/TSE.2024.3368208.
[19] N. S. Harzevili, M. M. Mohajer, M. Wei, H. V. Pham, and S. Wang, "History-driven fuzzing for deep learning libraries," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 1, pp. 1–29, Jan. 2025, doi: 10.1145/3688838.
[20] A. Mehmood, Q. M. Ilyas, M. Ahmad, and Z. Shi, "Test suite optimization using machine learning techniques: a comprehensive study," *IEEE Access*, vol. 12, pp. 168645–168671, 2024, doi: 10.1109/ACCESS.2024.3490453.
[21] J. Chen, L. Yan, S. Wang, and W. Zheng, "Deep reinforcement learning-based automatic test case generation for hardware verification," *Journal of Artificial Intelligence General Science*, vol. 6, no. 1, pp. 409–429, Nov. 2024, doi: 10.60087/jaigs.v6i1.267.
[22] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, "NeuFuzz: efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, pp. 36340–36352, 2019, doi: 10.1109/ACCESS.2019.2903291.
[23] Y. Li *et al.*, "V-Fuzz: vulnerability prediction-assisted evolutionary fuzzing for binary programs," *IEEE Transactions on Cybernetics*, vol. 52, no. 5, pp. 3745–3756, May 2022, doi: 10.1109/TCYB.2020.3013675.
[24] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," in *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, 2018, pp. 4805–4815.
[25] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *Proceedings of the 36th International Conference on Machine Learning*, Long Beach, California, 2019, pp. 1-10.
[26] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, 2019, pp. 10197–10207.
[27] M. Wu *et al.*, "One fuzzing strategy to rule them all," in *Proceedings of the 44th International Conference on Software Engineering*, May 2022, pp. 1634–1645, doi: 10.1145/3510003.3510174.
[28] Y. Feng, Z. Lu, and Q. Cao, "Secure sharing of private locations through homomorphic bloom filters," in *2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*, May 2018, pp. 107–113, doi: 10.1109/BDS/HPSC/IDS18.2018.00034.
[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2018, pp. 2123–2138, doi: 10.1145/3243734.3243804.
[30] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 2255–2269, doi: 10.5555/3489212.3489339.
[31] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 803–817, doi: 10.1109/SP.2019.00052.
[32] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "MTFuzz: fuzzing with a multi-task neural network," in *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Nov. 2020, pp. 737–749, doi: 10.1145/3368089.3409723.
[33] S. Gan *et al.*, "GREYONE: data flow sensitive fuzzing," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 2577–2594, doi: 10.5555/3489212.3489357.
[34] P. S. JosephNg, "EaaS optimization: available yet hidden information technology infrastructure inside medium size enterprise," *Technological Forecasting and Social Change*, vol. 132, pp. 165–173, Jul. 2018, doi: 10.1016/j.techfore.2018.01.030.
[35] P. S. JosephNg, Z. Fu, R. Zhang, and K. Y. Phan, "The impact and acceptance of large language models in healthcare: a perspective from China," *Journal of Advanced Research in Applied Sciences and Engineering Technology*, vol. 58, no. 4, pp. 110–158, Oct. 2024, doi: 10.37934/araset.59.2.110158.
[36] P. S. JosephNg, "Hotel room access control: an NFC approach ecotourism framework," *Journal of Science and Technology Policy Management*, vol. 15, no. 3, pp. 530–551, Apr. 2024, doi: 10.1108/JSTPM-10-2021-0153.
[37] M. Alias *et al.*, "Artificial intelligence-powered image recognition retail checkout systems," *International Journal of Advances in Applied Sciences*, vol. 15, no. 1, pp. 187–197, Mar. 2026, doi: 10.11591/ijaas.v15.i1.pp187-197.
[38] L. Junchang, J. P. Soon, P. K. Yuen, and W. S. Wan, "Every second counts for search and rescue: a systematic review of TinyML drone," *Edelweiss Applied Science and Technology*, vol. 10, no. 1, pp. 1177–1188, 2026, doi: 10.55214/2576-8484.v10i1.11904.
[39] S. Kaur, R. S. Sahota, S. Y. Ying, and Y. Haotian, "Emerging trends in Industry 4.0 and predictive maintenance," *Abhigyan*, vol. 43, no. 1, pp. 54–67, Mar. 2025, doi: 10.1177/09702385241280813.

BIOGRAPHIES OF AUTHORS

Dan Li graduated with an M.Sc. in Data Technology and Analytics at UCSI University. He worked as a software test engineer at China Ceprei Laboratory, obtained the title of intermediate engineer, and published several national papers in China as the first author and corresponding author. His research direction is the combination of software reliability and machine learning. He can be contacted at email: 13672738346m@sina.cn.

Poh Soon JosephNg graduated with a Doctor of Philosophy in Information Technology, master's in Information Technology (Aus), master's in Business Administration (Aus), and Associate Chartered Secretary (ICSA-UK) with various instructor qualifications, professional certifications, and industry memberships. Listed numerous times as the World's top 2% scientist in artificial intelligence and image processing by Stanford University, United States, Cisco's top 10% instructor excellence expert award, and with his blended technocrat mix of both business sense and technical skills, has held many multinational corporation senior management positions, global postings, and leads numerous 24×7 global mission-critical systems. He has appeared on LIVE national television prime time cybersecurity talk shows and overseas teaching exposure. His current research is on strategic artificial intelligence transformation. He can be contacted at email: joseph.ng@ucsiuniversity.edu.my.

Peng Yin Choo graduated with a Doctor of Philosophy (Ph.D.) in Electrical and Computer Engineering with a minor in Optical Science from the University of Arizona, United States, where he also earned his master's and dual bachelor's degrees in Electrical and Computer Engineering. Currently serving as deputy dean (Industrial Training and Student Development) at Universiti Tunku Abdul Rahman (UTAR), Malaysia. Through his leadership, he has spearheaded 40+ international and regional collaborations, including with leading organizations from China, Singapore, and Malaysia, to promote academic-industry synergy and innovation. He can be contacted at email: choopy@utar.edu.my.

Koo Yuen Phan graduated with a Ph.D. in Business Information Technology (Malaysia) and an M.Sc. in Information Studies (Singapore). He is currently working at Universiti Tunku Abdul Rahman as an assistant professor of computer science (Faculty of Information and Communication Technology). His research interests focus on the domains of information systems, information technology, business intelligence success, and firm performance. He can be contacted at email: phanky@utar.edu.my.

Wong See Wan holds a master's degree in Computer Science and a bachelor's degree in Information Technology, complemented by several industry-recognized instructor qualifications and professional certifications in enterprise networking, cybersecurity, and digital marketing. Leveraging this academic expertise, she transitioned into industry and is currently a co-founder of a textile company, where she leads strategic development and digital transformation initiatives. She can be contacted at email: karenwsw@gmail.com.