A Comparative Study of Program Graph Effectiveness for Binary Vulnerability Detection

Michael Kadoshnikov, Clemente Izurieta, Matthew Revelle
Montana State University
{mike.kadoshnikov, clemente.izurieta, matthew.revelle}@montana.edu

***Abstract*—Program graphs have become essential for vulnerability detection on program binaries, particularly for approaches based on machine learning. However, many researchers focus on comparing the performance of their technique with others, often neglecting the rationale behind the chosen graph structure used in their approach. This paper explores the comparative performance of various program graphs, such as abstract syntax trees (ASTs), control flow graphs (CFGs), data dependence graphs (DDGs), and their combinations. Each graph variation is evaluated by measuring the classification performance of representation-specific graph neural networks in detecting vulnerabilities at the program level in compiled programs from the NIST SARD Juliet dataset. By evaluating each combination's strengths and weaknesses, we identify the most effective graph structure for binary vulnerability detection. Performance is evaluated across all variations through a statistical analysis of the experimental results.**

I. Introduction

Many existing studies on vulnerability detection compare performance by using similar methods, such as leveraging program representations like control flow graphs (CFGs), data dependence graphs (DDGs), and abstract syntax trees (ASTs). However, prior research often adopts these graph structures, or their combinations, based on precedent, without critically assessing whether a particular combination is optimal in practice. While many studies assume that certain graph types or combinations improve classification accuracy, there is limited experimental evidence to support these claims. We aim to fill that gap by systematically exploring the performance differences across various combinations of CFGs, DDGs, and ASTs. Through an experimental evaluation, we assess how the inclusion of specific program information impacts the classification performance using graph neural networks (GNNs).

This work specifically evaluates the performance impacts of different program graphs by keeping the GNN architecture fixed and varying only the edges included as input. By comparing various combinations of ASTs, CFGs, and DDGs on compiled binaries under identical experimental conditions, we aim to quantify how the inclusion of additional edge information influences vulnerability detection and classification. We find evidence of the complementary contributions of program graph types.

Our approach operates directly on compiled program binaries, enabling analysis where source code is unavailable. This scenario is common in real-world applications, such as proprietary software, legacy systems, or third-party components. Moreover, some vulnerabilities may not be fully observable from source code alone. Compiler optimizations, platform-specific behavior, and low-level execution details can create discrepancies between source code and the actual executed instructions, leading to the "What You See Is Not What You eXecute" (WYSINWYX) phenomenon, where certain vulnerability relevant behaviors may only be visible after compilation [1].

We leverage intermediate representations (IRs), which provide a structured view of program behavior while abstracting away low-level details. Disassembled instructions contain memory address constants, stack offsets, and architecture-specific registers, which vary across systems and obscure generalizable patterns. IRs can represent programs at a higher level of abstraction, allowing the model to capture semantic patterns in the program without being tied to specific architectures. Disassembled instructions are translated into IRs through a process known as *lifting*, and program graphs can be constructed from this lifted representation instead of directly from disassembled instructions.

Operating on program binaries introduces additional challenges that directly influence how program representations are constructed. During compilation, information is often lost or transformed: storage locations can be reused to hold values from different source-level variables, and source-level control structures are lowered into control-flow primitives supported by the target instruction set architecture. Consequently, the choice of program graph and edge composition plays a critical role in what information is preserved for downstream vulnerability detection.

The study aims to address the following research questions regarding GNN-based classifiers for vulnerability detection on program binaries:

- **RQ1:** How does performance vary between each of the different combinations of graph types?
- **RQ2:** Which combination of graph types correspond to higher label accuracy?

In summary, this work makes the following contributions:

Workshop on Binary Analysis Research (BAR) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-08-4
https://dx.doi.org/10.14722/bar.2026.23091
www.ndss-symposium.org

- We train GNN-based classifiers on seven different program graph combinations containing different types of edges and evaluate their vulnerability detection performance.
- A statistical analysis is performed and a discussion of the experiment results and impact of including different types of graph information for vulnerability detection on program binaries is provided.

The source code used to conduct the experiment is available as open-source software[1] and includes code for dataset processing, the GNN-based classifier, and statistical analysis.

II. Background

A. Program Graphs

Representing programs as graphs allows for the preservation of program semantics, robustness to obfuscation techniques, and allows the graph to be language and performance agnostic [2]. In this work, we focus on three commonly used graph types: abstract syntax trees, control flow graphs, and data dependence graphs. Each graph type captures different aspects of program behavior, from syntactic structure to execution paths and data dependencies.

1) Abstract Syntax Trees (ASTs): Abstract syntax trees are produced by parsers and an intermediate representation used by compilers. As a result they form the basis for generating other code representations. Abstract syntax trees are a tree structure where nodes correspond to statements and expressions and the edges correspond to the parent-child relationship between nodes.

2) Control Flow Graphs (CFGs): Control flow graphs describe the order in which code statements are executed as well as describing the conditions that need to be met for a particular path of execution to be taken [3], [4]. Control flow graphs incorporate structure beyond that of ASTs by explicitly modeling execution paths and control dependencies, with the AST serving as the basis for construction. Depending on the scope, a control flow graph may be constructed at the statement level or at the basic block level. At the statement level, each statement is treated as a node with flow to the next instruction. At the basic block level statements get flattened into a singular node, statements only get flattened until a statement that changes flow through jumps or conditional logic is encountered. Control flow edges are used to indicate both conditional—true and false—and unconditional control flow. Control flow graphs have become a standard code representation in reverse engineering to aid in program understanding, however while they expose control flow, they fail to provide data flow information [3]. For vulnerability detection, this means that control flow graphs cannot be easily used to identify statements which process data influenced by an attacker.

[1]https://github.com/hacr-lab/Program-Graph-Study/

3) Data Dependence Graphs (DDGs): Data dependence graphs [5] consist of statements as nodes and variable-use edges, the edges between nodes capture the data dependence between statements [6]. Data dependence graphs require both the abstract syntax tree and control flow information in order to be generated. This is due to the identification of def-use relationships requiring knowledge of both the program's structure and possible execution paths, in order to determine where variables are defined and subsequently used. Data dependence graphs are particularly useful as they can help identify points in a program where user provided input is used, which can lead to identifying potentially vulnerable points of execution in a program.

B. Graph Neural Networks

Graph neural networks (GNNs) are a class of machine learning (ML) models designed to operate directly on graph-structured data. Unlike traditional neural networks, GNNs utilize relationships between nodes and edges to capture the structure and dependencies within a graph. In the context of program analysis, GNNs can learn representations that combine syntactic, control-flow, and data-flow information from program graphs, allowing the model to reason about complex program behavior. By iteratively aggregating information from a node's neighbors, GNNs produce embeddings that reflect both local and global structure, making them well suited for tasks such as vulnerability detection, function classification, and code similarity analysis.

III. Related Work

A. Vulnerability Detection

Vulnerability detection is used to identify weaknesses in software that could be exploited to compromise security. Common types of vulnerabilities include buffer overflow, command injection, and race conditions. Techniques for automated vulnerability detection can operate on source code or compiled binaries, and commonly represent programs as sequences of code statements or as graphs capturing structural relationships.

Approaches that operate on program source code may analyze raw source code or structured representations such as ASTs and CFGs to detect vulnerabilities. Sequence-based methods treat code as a token sequence and learn patterns over the sequences that are associated with vulnerable code [7], [8], [9], [10], [11]. Graph-based methods, on the other hand, leverage graph representations of programs which capture structural and semantic relationships in the code. For example, Zhou et al. [12] uses a merged control flow graph, data flow graph, and abstract syntax tree. Li et al. [13] uses a program dependence graph to generate program slices. VulChecker [14] uses an enhanced program dependence graph for vulnerability detection. FUNDED [15] combines the program and control dependence graphs with the abstract syntax tree. VGDetector [16] extracts the control flow graph from the abstract syntax tree. Wu et al. [17] converts program dependence graphs into images for use in convolution neural networks. While these methods demonstrate strong performance, they are typically

evaluated on a single representation or predefined combinations, leaving the relative combination of different graph types largely unexplored.

For compiled binaries, vulnerability detection can also be approached using sequence-based and graph-based methods. Both approaches require analysis to reconstruct meaningful representations from binaries. Sequence-based methods treat assembly instructions [18] or lifted instructions [19], [20] as a token sequence and learn patterns associated with vulnerable code. Unlike source sequences, these tokens may be influenced by the compiler or target architecture, and certain semantic information, such as variable names, is often unavailable.

Graph-based methods construct structural representations from disassembled [21], [22] or lifted code [23], [24], most often using control flow graphs as input to a GNN. TYGR [25] targets type inference rather than direct vulnerability detection; however, it constructs data flow graphs from CFGs that are applicable to GNN-based vulnerability detection. VulANalyzeR [26] combines sequence-based and graph-based approaches in their methodology.

In addition to ML-based approaches, vulnerability detection has been explored through rule-based static analysis and symbolic execution techniques. These methods typically depend on expert-defined rules or explicit path exploration to identify vulnerabilities.

Binary-level vulnerability detection introduces additional challenges that affect how program representations are constructed from compiled code. Even with lifted representations, some information is lost during compilation: variables names are often unavailable; storage locations may be reused and become associated with multiple original source variables; the same source code may result in different storage locations being used to hold variable values depending on the compiler used or architecture being targeted; and compound boolean expressions may be split across multiple conditional branches. Consequently, different program graph constructions preserve different subsets of semantic and structural information.

Our work addresses this gap by evaluating the vulnerability detection performance of a GNN for program binaries when different combinations of program graph edges are provided. This allows measurement of how each type of program graph contributes to classification performance for many common vulnerability classes.

B. Code Property Graphs (CPGs)

Code property graphs provide explicit structural information which can improve performance beyond sequence-based approaches. Code property graphs (CPGs), introduced by Yamaguchi et al. [3], are graphs which incorporate multiple program graph types. Originally, a CPG is a combination of an AST, CFG, and DDG; however, the term has become to more generally mean any graph representation that includes multiple layers of program graphs.

The authors describe the CPG as a means to enable expressive graph-based queries over code, allowing analysis to jointly reason about different program properties within one representation. In the original formulation, the CPG is presented as a general representation for program analysis that supports querying and exploring semantic relationships in code, rather than as a representation designed specifically for vulnerability detection. Subsequent work has adopted the CPG for security-related analyses by leveraging its ability to expose relationships that are relevant to vulnerability discovery.

Several early approaches leveraged CPGs for heuristic-based vulnerability finding. In these methods, expert-defined graph traversal queries were executed over the CPG to identify program patterns associated with potential vulnerabilities, such as tainted data flows or misuse of APIs [3]. The slice property graph introduced by Zheng et al. [27] extends this concept by focusing analysis on specific program slices, enabling more precise identification of code regions relevant to vulnerabilities.

More recent work has explored CPGs as input to machine learning models, particularly graph neural networks, to automatically detect vulnerabilities from data [28], [29], [30], [31]. These approaches treat the CPG as a rich structural representation, learning patterns across syntax, control flow, and data dependencies without relying on manually crafted rules. Liu et al. [28] shows that CPG-based learning models can achieve strong performance across multiple datasets and vulnerabilities types.

While CPGs provide a unified representation that captures multi-graph program relationships, prior work largely operates at the source level or treats the CPG as a single, monolithic graph. As a result, the individual contribution of each graph type of program relationship—syntax, control flow, or data dependence—remains largely unexplored. To build on these insights, we focus on program graphs reconstructed from compiled binaries, which allows us to examine structural information as it appears at the execution level.

IV. Approach

In order to answer our research questions, we need to extract program graphs from compiled programs. To prepare the program graphs, we first obtain an intermediate representation of functions from the program binaries in the dataset. We use Binary Ninja to lift disassembled functions to Binary Ninja's High-Level Intermediate Language (HLIL). This representation abstracts away details specific to any instruction set architecture while retaining general program information necessary for vulnerability detection. Once lifted, the program functions are used to construct program graphs which serve as inputs to the graph neural network.

We experiment with three distinct graph types: AST, CFG, and DDG; and seven distinct graph combinations: the three basic graphs as well as AST-CFG, AST-DDG, AST-CFG-DDG, and CFG-DDG. For a particular program, each of the graph types will contain the same nodes, with only the edges differing to mitigate any external factors from influencing any of the results. Using a standard GNN architecture, we train classification models for each graph combination in order to evaluate their relative performance.

A. Dataset

Despite substantial progress in vulnerability detection, commonly used datasets for training and evaluation may not capture the full diversity of real-world vulnerabilities, which can constrain model performance when deployed in practice. Chakraborty et al. [32] evaluated existing token and graph based techniques on real-world data and observed a 73% drop in F1 score for pretrained models and a drop of 54% in F1 score when models were trained directly on real-world data. These results highlight the difficulty of generalizing models trained on commonly used datasets to realistic settings.

Many prior approaches to vulnerability detection focus on source-level analysis, and a variety of datasets have been used in this context. Previous works introduced named datasets, such as Big-Vul [33], s-bAbI [34], D2A [35], CrossVul [36], CVEfixes [37], TrVD [9], FUNDED [15], VulDeePecker [7], and Draper VDISC [8]. Devign [12] proposes and evaluates a model on collections of code samples assembled for the study, and portions of these samples have been released as artifacts.

While these source-level datasets have enabled progress in vulnerability detection, they present significant challenges for use in binary analysis. In particular, many datasets do not guarantee that programs are self-contained or semantically complete, often omitting required header files, or build configurations. As a result, substantial manual effort is required to infer missing components and successfully compile the code, introducing ambiguity in how real-world projects should be reconstructed for analysis.

Some datasets attempt to mitigate these issues by restricting their scope. For example, the s-bAbI dataset is labeled at the line-of-code level with safe and unsafe buffer writes, but the scope of this dataset is limited to buffer writes. TrVD and VulDeePecker derive their samples from NIST's SARD [38], focusing primarily on buffer and resource management errors. μVulDeePecker [10] and SySeVR [11] similarly rely on data derived from NVD and SARD. FUNDED [15] aggregates samples from NVD, SARD, and GitHub; however, the build context of individual samples are not always available, limiting reproducibility.

The correctness of vulnerability datasets is also an important consideration. Studies such as Croft et al. [39] report label inaccuracies found in widely used datasets including Big-Vul, Devign, D2A, while noting that Juliet avoids such inaccuracies by construction, as its vulnerable cases are intentionally synthesized. However, Juliet suffers from duplicate functions, many of which arise from automatically generated test cases that introduce only minor variations in control flow logic. Additionally, non-vulnerable code samples often contain fixed statements with identical corrected implementations across samples. Additional limitations have been identified in both Draper [8] and Devign [12], including missing build information and labels inferred from commit messages or code diffs, which may be unreliable.

For compiled binary analysis, Juliet remains one of the most commonly used datasets due to its ease of compilation and consistent nature. A key limitation of the Juliet dataset, however, is its synthetic nature, which limits the generalizability of models trained exclusively on it. While a small number of labeled real-world binary datasets do exist, they are typically limited in size and scope, providing insufficient diversity and sample volume to effectively train data-hungry models such as graph neural networks.

In our approach, we utilize the Juliet Test Suite for C/C++ [38], a part of SARD, compiled for the x86_64 Linux platform. The dataset was compiled on a system running Debian 12. Since the original test suite relies on a Windows batch file for compilation, which is not compatible with Linux, we employed a reproducible Unix-based build process. This process ensured that all test cases could be correctly compiled, producing a compiled binary dataset suitable for evaluation on Linux systems [40].

B. Graph Generation

Our approach uses the reverse engineering tool Binary Ninja[2] to lift the disassembly from a program binary into Binary Ninja's High-Level Intermediate Language (HLIL). The choice of HLIL over Binary Ninja's Medium-Level or Low-Level Intermediate Languages (MLIL and LLIL) is due to HLIL performing transformations which simplify the code representation [41]. The HLIL statements contain specific variable identifiers which do not provide additional information for vulnerability detection and classification. For example, an HLIL variable declaration refers to the identifier of the declared variable, `int32_t var_10`. Statements and expressions are abstracted so that references to specific identifiers are removed. The use of generic nodes such as `HLIL_VAR_DECLARE` (variable declaration) and `HLIL_CALL` (call statement) to represent these program constructs prevent the GNN from using the identifiers for vulnerability detection. Associations between nodes in the graphs are maintained by constructing edges. For example, a variable declaration such as `int32_t var_10` is abstracted to a `HLIL_VAR_DECLARE` node. References to the `var_10` variable will be represented as distinct `HLIL_VAR` nodes and `HasDependent` edges will be formed between the `HLIL_VAR_DECLARE` node and every `HLIL_VAR` node. The `main` function of the example C program in Listing 1 is represented in HLIL—as shown in Figure 1—after being lifted from disassembly. The abstracted program graphs which represent the original HLIL statements are shown in the example graphs in Figure 2, Figure 3, and Figure 4.

1) Abstract Syntax Tree (AST): The Abstract Syntax Tree is generated for each function in a program, resulting in many smaller function ASTs rather than one large program AST. Each function is represented as a top-level `FUNCTION` node, which serves as a structured statement containing all top-level statements within the function's scope. Structured statements within the function, such as `if-else` or `while` statements, act as parent nodes for their constituent parts: for example,

[2]https://binary.ninja

```c
#include <stdio.h>

void main() {
    int x;
    scanf("%d", &x);
    int y = x * 2;
    if (y > 15) {
        puts("Condition Satisfied");
    } else {
        puts("Condition Not Satisfied");
    }
    puts("Program End");
}
```

Listing 1. An example C program that doubles input provided by the user and checks a condition.

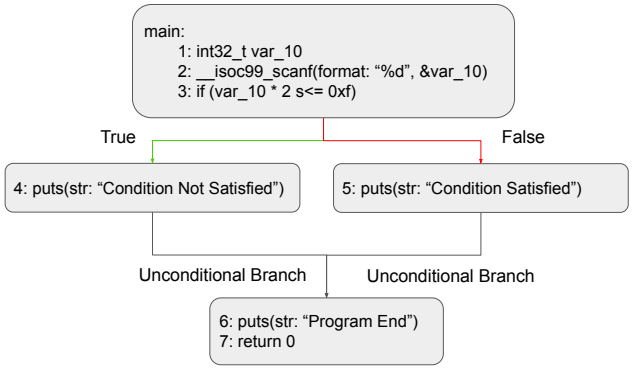

Fig. 1. The HLIL representation of the example program.

an `if-else` statement contains a comparison expression and two nested statements corresponding to the true and false branches. Similarly, loop statements also contain a comparison expression and a loop body statement. Figure 2 illustrates this hierarchical structure, showing how parent nodes connect to their nested child statements to represent program syntax. The AST structure is based on the parent-child relationship provided by Binary Ninja. ASTs contain only one graph edge type: `HasChild`.

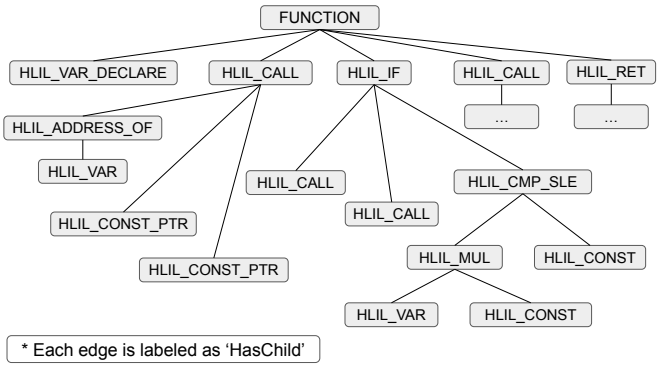

Fig. 2. The AST generated from the example program in Listing 1 (some edges hidden for readability).

2) Control Flow Graph (CFG): Using the previously generated AST, we construct the control flow graph (CFG). At the HLIL level, structured control-flow information is available through edges which link basic blocks. We leverage these edges by propagating them to the statement level: for each outgoing edge from a basic block, we introduce a corresponding edge from the last statement of the source block to the first statement of the destination block. These edges are labeled as `TrueBranch`, `FalseBranch`, or `UnconditionalBranch`, depending on the control flow structure.

To complete the CFG, we additionally insert edges between consecutive statements within each basic block. These edges are labeled as ε edges and represent implicit sequential execution within a block. Unlike `UnconditionalBranch` edges, which denote an explicit control-flow transfer between basic blocks, ε edges do not alter control flow and merely encode intra-block statement ordering. There is one more edge added to indicate function start resulting in five total edge types: `FunctionStart`, `TrueBranch`, `FalseBranch`, `UnconditionalBranch`, and ε. Figure 3 illustrates the generated CFG.

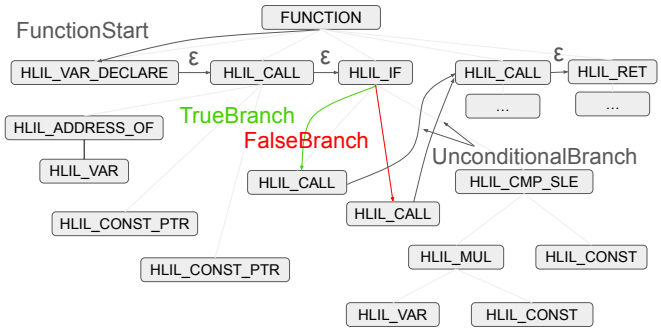

Fig. 3. The CFG generated from AST of the example program.

3) Data Dependence Graph (DDG): We use both the AST and CFG to obtain the data dependence graph for a function. For each statement in the graph, if a variable is used, we find all previous definitions of the variable by exploring all control-flow branches. Once the variable definition is reached, we add an edge between the AST `HLIL_VAR` node associated to the definition and the AST `HLIL_VAR` read node. Figure 4 shows the edges produced from this process. Data dependence graphs contain only one edge type: `HasDependent`.

4) Generating Aggregate Graphs: The graphs generated in the previous sections (AST, CFG, and DDG) each share the same nodes but have distinct edge types. To explore the relationships between these different graph structures, we combine edge types into new aggregate graphs. Specifically, we generate the following graph combinations:

- **AST-CFG**: This graph aggregates the edges from both the AST and the CFG, providing a combined view of the program's syntactic structure and control flow.

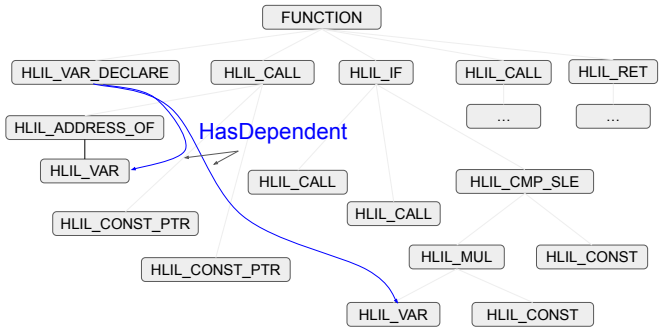

Fig. 4. The DDG generated from AST and CFG of the example program.

- **AST-DDG**: This graph combines the AST edges with those from the DDG, capturing both the syntactic structure of the program and the data dependencies between statements and expressions.
- **CFG-DDG**: This combination merges the CFGs' control flow edges with the dependencies from the DDG. However, since the data dependencies in the DDG are linked to AST nodes, the edges are disconnected, which may limit the graph's ability to fully capture the relationship between control flow and data dependencies.
- **AST-CFG-DDG**: The aggregate graph combines the edges from all three individual graphs, providing a comprehensive representation of the program's syntactic structure, control flow, and data dependencies.

Each of the graph types are a combination of their respective graph edges, the edges contained in the simple graphs can be seen in Table I. The AST and DDG graphs have a single edge type, while the CFG includes five types of edges. For each aggregate graph type, the edges included are the union of all edges between the smaller sub graphs. For example, $E_{AST\text{-}DDG} := E_{AST} \cup E_{DDG}$, where E_X denotes the set of edges in a graph X.

TABLE I
GRAPH TYPES AND THEIR CORRESPONDING EDGE TYPES.

Graph Type	Edge Types
AST	`HasChild`
CFG	`TrueBranch`, `FalseBranch`, `UnconditionalBranch`, `FunctionStart`, ε
DDG	`HasDependent`

The list of node types is extensive, a few examples of node names are: `HLIL_VAR_DECLARE` (variable declaration statement), `HLIL_CALL` (function call statement), `HLIL_RET` (return statement), and `HLIL_IF` (if statement), and `HLIL_CONST` (constant value expression).

5) Program Graph Construction and GNN Preprocessing: The constructed graphs represent entire programs, where each function is an individual disjoint subgraph in the program graph. For a given program containing multiple functions, all corresponding function-level graphs collectively represent the program during training and evaluation. This representation is required due to the vulnerability labels being assigned at the program level in the Juliet dataset, not at the function level. A program may contain multiple functions, some of which are labeled with names indicating potentially vulnerable behavior, while others implement supporting functionality. Since the label applies to the entire program, it is unclear which specific function contains the vulnerability, making it difficult to assign labels at the function level.

Preparing graphs for input to the GNN consists of several preprocessing steps. First, function-level information that is not relevant to vulnerability modeling is filtered out. Since the compiled test programs contain runtime support code which are not part of the original test program, the resulting graphs include extraneous functions which may introduce noise. To mitigate this, functions not associated with the program source code are removed using a simple name-based filtering strategy that exploits a common naming pattern found in Juliet.

Following preprocessing, node and edge label encoders are constructed by iterating over the graphs and mapping categorical labels to integer values. These encoders are then applied to transform each graph into a PyTorch `Data` object, which serves as the final input representation for the GNN.

C. Graph Neural Network

1) Labeling Data: To limit the total number of vulnerability labels and improve the number of samples for each label, we used ChatGPT [42] to help generate a set of generalized labels which may include multiple CWEs. One of the CWEs (CWE-500) had only one example and was removed from the experiments. By mapping the original Juliet CWE labels to these generalized labels, we group smaller groups of CWEs into a parent label, by reducing the amount of labels for the GNN, we have more instances for a particular label which can potentially share a similar graph structure, and ideally would reduce ambiguity. Table II shows the generalized labels and their CWE counterparts.

2) GNN Architecture: The architecture of the GNN remained very small. The GNN took advantage of RGCNConv [43], as it has been shown to be effective in link prediction and entity classification. The model has four layers of RGCNConv, starting with the input channel size, which goes to 192 channels, with each hidden layer reducing the dimensions by half until the out channels (in, 192, 96, 48, out). As for training, validating, and testing we make a 60/20/20 split. For each run of the GNN, we get a random state to partition the training and testing sets into an 80/20 split, then use another random state to split the remaining training set into a 60/20 split between training and validation. This was done to eliminate the possibility that a particular split in data would result in model outliers. For each of the seven graph types, each model was run with the same architecture, with the only difference being the random states for generating the dataset split, and the graph data. Additionally, since the labels are

TABLE II
GENERALIZED LABELS FOR CWE CLASSIFICATION.

Generalized Label	CWEs
Improper Input Validation	CWE-23, CWE-36, CWE-78, CWE-134, CWE-440
Buffer Overflow	CWE-121, CWE-122, CWE-124, CWE-126, CWE-127, CWE-588
Memory Corruption	CWE-123, CWE-188, CWE-415, CWE-416, CWE-464, CWE-467, CWE-468, CWE-469, CWE-475, CWE-476, CWE-562, CWE-590, CWE-761, CWE-762
Integer Overflow	CWE-190, CWE-680
Integer Underflow	CWE-191
Integer Error	CWE-194, CWE-195, CWE-196, CWE-197, CWE-681
Dangerous Function Usage	CWE-242, CWE-676
Error Handling Issue	CWE-252, CWE-253, CWE-390, CWE-391, CWE-396, CWE-397, CWE-690
Race Condition	CWE-364, CWE-366, CWE-367
Arithmetic Error	CWE-369
Insecure Resource Managment	CWE-377, CWE-426, CWE-427
Code Quality Issue	CWE-398, CWE-546, CWE-561, CWE-563, CWE-587
Resource Management Issue	CWE-400, CWE-401, CWE-404, CWE-459, CWE-605, CWE-665, CWE-672, CWE-675, CWE-773, CWE-775, CWE-789
Uninitialized Memory Usage	CWE-457
Logic Error	CWE-478, CWE-480, CWE-481, CWE-482, CWE-483, CWE-484, CWE-570, CWE-571, CWE-606, CWE-617, CWE-674, CWE-685, CWE-688, CWE-758, CWE-835, CWE-843
Concurrency Issue	CWE-479, CWE-666, CWE-667, CWE-832
Malicious Code	CWE-506, CWE-510, CWE-511
Information Exposure	CWE-526
Not-Vulnerable	Good Versions of All Programs

not normalized, we generate a class weight matrix for training to ensure instances are equally weighted. The experimental results are from running 350 total model instances, split among the seven graph types, taken from the 60th epoch, as we are not interested in the model's best-performing epoch but in the average performance across graph types.

V. RESULTS

Table III presents the weighted F1 scores for each graph representation across all vulnerability labels. Each graph type was evaluated using a GNN trained for 60 epochs and repeated over 50 independent runs. The reported values correspond to the weighed average F1 score aggregated across all runs for each graph type.

Across all vulnerability categories, the combined AST-CFG-DDG representation achieved the highest weighted average F1 score (0.95) followed by the AST-CFG (0.94), AST-DDG (0.79), AST (0.75), CFG-DDG (0.74), CFG (0.71), while the DDG-only representation achieved the lowest weighted average F1 score (0.20) [**RQ1**]. The complete ranking of graph representations based on weighted average F1 score is shown in Table IV.

In Table III, we see evidence of complementary contributions of different program graphs. For example in the vulnerability label **Improper Input Validation**, the CFG alone had the second highest average F1 score. We also see that the AST edges provide additional information useful for classification, as the AST-CFG graph had the highest average F1 score. For the results in **Arithmetic Error**, all merged graphs outperformed their individual subgraphs indicating that each edge type in conjunction with one another provided useful additional information for the label.

VI. ANALYSIS

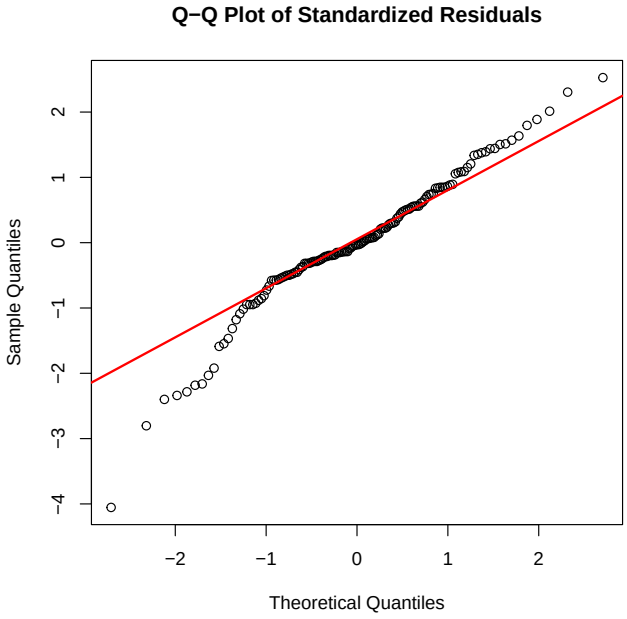

Fig. 5. Q-Q plot of residuals from the linear model. The red line shows the theoretical normal quantiles.

To evaluate the statistical significance of the observed performance differences, a two-factor analysis of variance (ANOVA) was performed using graph type and vulnerability label as factors shown in Table V. The ANOVA results indicate that both graph type and vulnerability label have statistically significant effects on the weighted F1 score ($p < 2.2 \cdot 10^{-16}$ for both factors). The model explains 87.41% of the total variance in the observed F1 scores, with the remaining 12.59% attributed to residual error.

To assess the validity of the ANOVA, the residuals of the fitted linear model were examined using a Q-Q plot shown in Figure 5. The residual distribution follows the theoretical normal quantiles, indicating no substantial deviation from normality.

The ANOVA results demonstrate that the observed differences in performance across graph representations are statistically significant. As a result, we reject the null hypothesis that all graph representations yield equivalent performance and

TABLE III
AVERAGE F1 SCORE WEIGHTED ACROSS ALL GNN INSTANCES. THE HIGHEST SCORE FOR EACH LABEL (ROW) IS SHOWN IN BOLD AND THE SECOND-HIGHEST SCORE IS UNDERLINED; TIES ARE FORMATTED IDENTICALLY.

	AST	**CFG**	**DDG**	**CFG-DDG**	**AST-DDG**	**AST-CFG**	**AST-CFG-DDG**
Arithmetic Error	0.85	0.28	0.54	0.82	0.87	**0.98**	0.97
Buffer Overflow	0.73	0.70	0.39	0.72	0.77	0.94	**0.95**
Code Quality Issue	0.47	0.26	0.15	0.43	0.52	0.68	**0.70**
Concurrency Issue	0.51	0.52	0.15	0.67	0.59	0.85	**0.94**
Dangerous Function Usage	0.55	0.31	0.03	0.55	0.69	**0.88**	**0.88**
Error Handling Issue	0.50	0.45	0.12	0.52	0.59	0.82	**0.85**
Improper Input Validation	0.82	0.88	0.55	0.86	0.88	**0.96**	**0.96**
Information Exposure	0.31	0.00	0.00	0.03	0.34	**0.73**	0.69
Insecure Resource Management	0.36	0.40	0.05	0.40	0.50	0.74	**0.76**
Integer Error	0.76	0.68	0.44	0.69	0.79	0.94	**0.95**
Integer Overflow	0.79	0.32	0.19	0.42	0.84	**0.93**	**0.93**
Integer Underflow	0.72	0.34	0.26	0.42	0.75	0.91	**0.92**
Logic Error	0.60	0.69	0.13	0.75	0.68	0.85	**0.89**
Malicious Code	0.47	0.64	0.11	0.77	0.57	**0.92**	0.91
Memory Corruption	0.71	0.74	0.27	0.78	0.78	0.92	**0.94**
Race Condition	0.61	0.86	0.06	0.89	0.72	0.93	**0.95**
Resource Management Issue	0.63	0.66	0.21	0.74	0.70	0.88	**0.91**
Uninitialized Memory Use	0.78	0.77	0.25	0.81	0.82	0.94	**0.97**
Not Vulnerable	0.78	0.79	0.08	0.80	0.81	**0.97**	**0.97**
Macro Average	0.62	0.54	0.21	0.63	0.69	0.88	**0.90**
Weighted Average	0.75	0.71	0.20	0.74	0.79	0.94	**0.95**

TABLE IV
RANKING OF GRAPH RESULTS BY WEIGHTED AVERAGE (BEST TO WORST).

Rank	**Item**	**Weighted Average**
1	AST-CFG-DDG	0.95
2	AST-CFG	0.94
3	AST-DDG	0.79
4	AST	0.75
5	CFG-DDG	0.74
6	CFG	0.71
7	DDG	0.20

support the alternative hypothesis that graph structure significantly influences a GNNs ability to detect vulnerabilities.

While differences across vulnerability labels are expected due to variation in program semantics, the strong statistical significance associated with graph type indicates that representation choice is a primary factor affecting model performance. The proportion of explained variance further reinforces the impact of graph representation on classification outcomes.

A pairwise comparison using Tukey's Honest Significant Difference (HSD) test provides further insight into the observed differences among graph representations shown in Table VI. The results show that only six graph pairs do not exhibit statistically significant differences at $\alpha = 0.05$. Notably, the AST-CFG-DDG representation differs significantly from all other graph types except AST-CFG, indicating that these two combined representations capture similar information content. A similar pattern is observed for AST-CFG, which also shows statistically significant differences relative to most simpler graph representations. These two graph types have the highest average label accuracy and the best classification label accuracy than the other representations, but minimal differences between each other [**RQ2**].

The superior performance of combined graph representations suggests that integrating complementary sources of program information improves the GNNs ability to capture features relevant to vulnerability detection. In contrast, the comparatively poor performance of single-graph representations, particularly DDG alone, indicates that data-dependence information in isolation is insufficient to fully characterize program behavior in this task.

Overall, the rejection of the null hypothesis and the statistically significant pairwise differences observed across graph representations demonstrate that information-rich graphs provide a measurable advantage for GNN-based vulnerability detection.

VII. DISCUSSION

The results of the study provide valuable insights into how different program graphs contribute to the performance of

TABLE V
ANALYSIS OF VARIANCE (ANOVA)

Source	Df	Sum Sq	Mean Sq	F value	Pr(>F)
graph_type	6	6.8204	1.13673	104.070	$< 2.2 \times 10^{-16}$ ***
vulnlabels	20	2.2816	0.11408	10.444	$< 2.2 \times 10^{-16}$ ***
Residuals	120	1.3107	0.01092		

Significance codes: 0 '***', 0.001 '**', 0.01 '*', 0.05 '.', 0.1 ' ', 1

TABLE VI
TUKEY-ADJUSTED PAIRWISE CONTRASTS BETWEEN GRAPH REPRESENTATIONS. (95% CONFIDENCE LEVEL; 120 DEGREES OF FREEDOM).

Contrast	Estimate	SE	t-ratio	p-value
AST vs. AST-CFG	-0.25095	0.0323	-7.781	< 0.0001
AST vs. AST-CFG-DDG	-0.26524	0.0323	-8.224	< 0.0001
AST vs. AST-DDG	-0.06524	0.0323	-2.023	0.4060
AST vs. CFG	0.08476	0.0323	2.628	0.1269
AST vs. CFG-DDG	-0.00571	0.0323	-0.177	1.0000
AST vs. DDG	0.42524	0.0323	13.184	< 0.0001
AST-CFG vs. AST-CFG-DDG	-0.01429	0.0323	-0.443	0.9994
AST-CFG vs. AST-DDG	0.18571	0.0323	5.758	< 0.0001
AST-CFG vs. CFG	0.33571	0.0323	10.409	< 0.0001
AST-CFG vs. CFG-DDG	0.24524	0.0323	7.604	< 0.0001
AST-CFG vs. DDG	0.67619	0.0323	20.965	< 0.0001
AST-CFG-DDG vs. AST-DDG	0.20000	0.0323	6.201	< 0.0001
AST-CFG-DDG vs. CFG	0.35000	0.0323	10.852	< 0.0001
AST-CFG-DDG vs. CFG-DDG	0.25952	0.0323	8.046	< 0.0001
AST-CFG-DDG vs. DDG	0.69048	0.0323	21.408	< 0.0001
AST-DDG vs. CFG	0.15000	0.0323	4.651	0.0002
AST-DDG vs. CFG-DDG	0.05952	0.0323	1.846	0.5204
AST-DDG vs. DDG	0.49048	0.0323	15.207	< 0.0001
CFG vs. CFG-DDG	-0.09048	0.0323	-2.805	0.0829
CFG vs. DDG	0.34048	0.0323	10.556	< 0.0001
CFG-DDG vs. DDG	0.43095	0.0323	13.362	< 0.0001

graph neural networks in classifying vulnerable programs. The AST-CFG-DDG graph generally outperformed the simpler graph types, showing that multifaceted representations of program data yield more accurate results for vulnerability detection across a diverse set of labels. One reason for the superior performance of the AST-CFG-DDG might be that vulnerabilities often require multiple dimensions of program behavior. For instance, integer overflow vulnerabilities often involve values that propagate through multiple data dependencies before being used in memory allocation or bounds checks; when these operations occur along specific control-flow paths, overflow can lead to unsafe memory accesses, making both DDG and CFG information essential.

The AST-CFG and AST-CFG-DDG perform similarly, indicating that between these two graph types the inclusion of DDG edges does not provide a statistically significant increase in performance, though a small increase in performance is observed. This minimal increase in performance with the use of DDG edges could be due to the simplicity of the Juliet dataset, and further experimentation would be necessary.

Examining the effects of adding different types of information, we see that including CFG information improves performance substantially: moving from AST to AST-CFG results in an increase of approximately 19%. In contrast, adding DDG information alone to the AST (AST $\rightarrow$ AST-DDG) yields only a 4% improvement. A similar trend is observed when combining both AST and CFG features: AST-CFG $\rightarrow$ AST-CFG-DDG shows only a 1% gain from adding DDG data. Likewise, for CFG alone, including DDG edges (CFG $\rightarrow$ CFG-DDG) increases performance by 3%. These results suggest that CFG information has a stronger impact on performance than DDG information in this setting. Future work may shed further insight into the importance of graph edge types in classification tasks.

However, there are a few limitations that could have lowered the performance for vulnerability detection. First, DDG edges provide limited information by themselves because they rely on the AST for context; without the AST, their contribution

to classification is minimal. Second, The model is relatively simple and is not fully optimized for any specific graph type, so further experimentation with specialized architectures could yield improved performance for certain graph types. Moreover, the Juliet dataset, while valuable for studying simple vulnerable code examples, does not fully capture the complexity of real-world vulnerabilities, and results may differ when using more diverse and complex datasets. Finally, because this study operates on compiled program binaries, the availability of large-scale, labeled binary-datasets that reflect the complexity and diversity of real-world vulnerabilities remains limited. As a result, model performance observed on synthetic benchmarks may not fully translate to practical deployment scenarios.

VIII. Threats to Validity

There are various threats to validity associated with ML-based vulnerability detection techniques, and we consider both external and internal threats. The most significant threat to validity is the generalization of the study to other datasets. This study utilized a popular dataset that is labeled as a synthetic/semi-synthetic dataset. Due to the nature of synthesized data, the results may not generalize to other datasets. However, the purpose of this study is to identify the value of different program graphs in vulnerability detection for program binaries. Experimentation with more datasets is necessary to conclude that different edge types affect vulnerability classification in real-world datasets similarly to the Juliet dataset.

Another potential threat to validity is a lack of random sampling. To mitigate internal threats to validity, the dataset was randomized into separate training, validation, and test groups. Additionally, for each GNN instance, the data splits were randomized to help mitigate biased results.

IX. Conclusion

This work evaluates the performance impact of various program graphs on binary vulnerability detection by using a standard GNN architecture and only varying which types of edges are included in the input graphs. By evaluating combinations of ASTs, CFGs, and DDGs recovered from compiled binaries under the same experimental conditions, we show that the structure of the program graph has a clear and statistically significant impact on classification performance.

Our results show that the combined program graphs outperformed the simpler graph representations, indicating that incorporating multiple forms of program information leads to more effective vulnerability detection. In particular, representations that include control flow information provide substantial performance gains, while data dependence information offers smaller but measurable improvements. This suggests that not all graph components contribute equally, and that representation choice plays a critical role in overall model performance. The results show that the design decision of graph representations can be evaluated directly instead of being inherited from prior work.

These findings generate directions for future research. Exploring more expressive GNN architectures or alternative graph constructions could reveal additional performance improvements and clarify the contributions of different program features. Additionally, validating these methods on larger and more diverse binary datasets would help determine how well these insights generalize to real-world vulnerability detection, particularly in settings where source code is unavailable.

References

[1] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.

[2] T. Bilot, N. El Madhoun, K. Al Agha, and A. Zouaoui, "A survey on malware detection with graph representation learning," *ACM Comput. Surv.*, vol. 56, no. 11, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3664649

[3] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[4] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: Association for Computing Machinery, 1970, p. 1–19. [Online]. Available: https://doi.org/10.1145/800028.808479

[5] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83. New York, NY, USA: Association for Computing Machinery, 1983, p. 177–189. [Online]. Available: https://doi.org/10.1145/567067.567085

[6] K. J. OTTENSTEIN, "Data-flow graphs as an intermediate program form," Ph.D. dissertation, 1978, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-07-26. [Online]. Available: https://www.proquest.com/dissertations-theses/data-flow-graphs-as-intermediate-program-form/docview/302912471/se-2

[7] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings 2018 Network and Distributed System Security Symposium*, ser. NDSS 2018. Internet Society, 2018. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23158

[8] R. L. Russell, L. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," 2018. [Online]. Available: https://arxiv.org/abs/1807.04320

[9] Z. Tian, B. Tian, J. Lv, Y. Chen, and L. Chen, "Enhancing vulnerability detection via ast decomposition and neural sub-tree encoding," *Expert Syst. Appl.*, vol. 238, no. PB, Mar. 2024. [Online]. Available: https://doi.org/10.1016/j.eswa.2023.121865

[10] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2021.

[11] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2022.

[12] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1909.03496

[13] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103 184–103 197, 2019.

[14] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, "VulChecker: Graph-based vulnerability localization in source code," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6557–6574. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky

[15] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2021.

[16] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2019, pp. 41–50.

[17] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "Vulcnn: an image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2365–2376. [Online]. Available: https://doi.org/10.1145/3510003.3510229

[18] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[19] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *NDSS*, 2023.

[20] A. Schaad and D. Binder, "Deep-learning-based vulnerability detection in binary executables," in *Foundations and Practice of Security*, G.-V. Jourdan, L. Mounier, C. Adams, F. Sèdes, and J. Garcia-Alfaro, Eds. Cham: Springer Nature Switzerland, 2023, pp. 453–460.

[21] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, R. Baldoni *et al.*, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019, pp. 1–11.

[22] Y. Ji, L. Cui, and H. H. Huang, "Vestige: Identifying binary code provenance for vulnerability detection," in *Applied Cryptography and Network Security*, K. Sako and N. O. Tippenhauer, Eds. Cham: Springer International Publishing, 2021, pp. 287–310.

[23] X. Shang, L. Hu, S. Cheng, G. Chen, B. Wu, W. Zhang, and N. Yu, "Binary code similarity detection via graph contrastive learning on intermediate representations," 2024. [Online]. Available: https://arxiv.org/abs/2410.18561

[24] S. Arakelyan, S. Arasteh, C. Hauser, E. Kline, and A. Galstyan, "Bin2vec: Learning representations of binary executable programs for security tasks," 2021. [Online]. Available: https://arxiv.org/abs/2002.03388

[25] C. Zhu, Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé, M. Naik, Y. Shoshitaishvili, R. Wang, and A. Machiry, "TYGR: Type inference on stripped binaries using graph neural networks," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4283–4300. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/zhu-chang

[26] L. Li, S. H. H. Ding, Y. Tian, B. C. M. Fung, P. Charland, W. Ou, L. Song, and C. Chen, "Vulanalyzer: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution," *ACM Trans. Priv. Secur.*, vol. 26, no. 3, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3585386

[27] W. Zheng, Y. Jiang, and X. Su, "Vu1spg: Vulnerability detection based on slice property graph representation learning," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 457–467.

[28] R. Liu, Y. Wang, H. Xu, J. Sun, F. Zhang, P. Li, and Z. Guo, "Vul-lmgnns: Fusing language models and online-distilled graph neural networks for code vulnerability detection," *Information Fusion*, vol. 115, p. 102748, Mar. 2025. [Online]. Available: http://dx.doi.org/10.1016/j.inffus.2024.102748

[29] Q. Feng, C. Feng, and W. Hong, "Graph neural network-based vulnerability predication," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 800–801.

[30] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," 2020. [Online]. Available: https://arxiv.org/abs/2006.08614

[31] M. F. Rozi, T. Ban, S. Ozawa, A. Yamada, T. Takahashi, and D. Inoue, "Securing code with context: Enhancing vulnerability detection through contextualized graph representations," *IEEE Access*, vol. 12, pp. 142 101–142 126, 2024.

[32] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2022.

[33] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*, 2020, pp. 508–512.

[34] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos, "Towards security defect prediction with ai," 2018. [Online]. Available: https://arxiv.org/abs/1808.09897

[35] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, and Z. Su, "D2a: A dataset built for ai-based vulnerability detection methods using differential analysis," 2021. [Online]. Available: https://arxiv.org/abs/2102.07995

[36] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos, "Crossvul: a cross-language vulnerability dataset with commit data," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1565–1569. [Online]. Available: https://doi.org/10.1145/3468264.3473122

[37] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE '21. ACM, Aug. 2021, p. 30–39. [Online]. Available: http://dx.doi.org/10.1145/3475960.3475985

[38] P. E. Black, "The software assurance reference dataset (sard)," National Institute of Standards and Technology, NIST Internal Report 8561, Jan. 2025.

[39] R. Croft, M. A. Babar, and M. Kholoosi, "Data quality for software vulnerability datasets," 2023. [Online]. Available: https://arxiv.org/abs/2301.05456

[40] A. Richardson, "Juliet test suite for c/c++," 2025, accessed: 2025-03-19. [Online]. Available: https://github.com/arichardson/juliet-test-suite-c

[41] V. 35, "Binary ninja intermediate languages (bnil/hlil)," https://docs.binary.ninja/dev/bnil-hlil.html, accessed: 2025-03-11.

[42] OpenAI, "ChatGPT (march 28 version)," 2025, [Large language model]. [Online]. Available: https://openai.com/chatgpt

[43] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," 2017. [Online]. Available: https://arxiv.org/abs/1703.06103