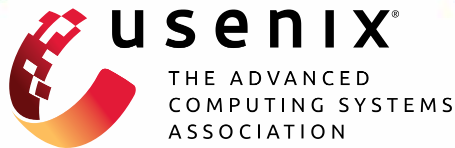


FVD-DPM: Fine-grained Vulnerability Detection via Conditional Diffusion Probabilistic Models

Miaomiao Shao and Yuxin Ding, *Harbin Institute of Technology, Shenzhen*

https://www.usenix.org/conference/usenixsecurity24/presentation/shao

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

FVD-DPM: Fine-grained Vulnerability Detection via Conditional Diffusion Probabilistic Models

Miaomiao Shao
Harbin Institute of Technology, Shenzhen

Yuxin Ding*
Harbin Institute of Technology, Shenzhen

Abstract

Software vulnerabilities pose a significant threat to software security. Nevertheless, existing vulnerability detection methods still struggle to effectively identify vulnerabilities and pinpoint vulnerable statements. In this paper, we introduce FVD-DPM: a novel Fine-grained Vulnerability Detection approach via a conditional Diffusion Probabilistic Model. FVD-DPM formalizes vulnerability detection as a diffusion-based graph-structured prediction problem. Firstly, it generates a new fine-grained code representation by extracting graph-level program slices from the Code Joint Graph. Then, a conditional diffusion probabilistic model is employed to model the node label distribution in the program slices, predicting which nodes are vulnerable. FVD-DPM achieves both precise vulnerability identification (slice-level detection) and vulnerability localization (statement-level detection). We evaluate FVD-DPM on five collected datasets and compare it against nine state-of-the-art vulnerability detection approaches. Experimental results demonstrate that FVD-DPM significantly outperforms the baseline approaches across various evaluation settings.

1 Introduction

Software vulnerabilities (or vulnerabilities for short) are security flaws resulting from errors in software design, development, or configuration, rendering them a primary source of software security issues. These vulnerabilities can be exploited by attackers to carry out malicious actions, posing risks to the privacy and property security of software users. Despite collaborative efforts from academia and industry to enhance software security, vulnerabilities remain a significant challenge [5]. One effective strategy for addressing vulnerabilities is the development of detection models. An ideal detection model should exhibit both robust identification and localization capabilities. This entails the detection model not only determining whether a code fragment is vulnerable but also predicting the specific code lines (i.e., statements) that are vulnerable. In this paper, *vulnerable* describes the status of a code fragment, such as a vulnerable function, a vulnerable statement. A vulnerable function represents a vulnerability, while a vulnerable statement is part of a vulnerability. A vulnerable function contains at least one vulnerable statement.

Many traditional static analysis-based methods have been proposed to detect vulnerabilities, showcasing their effectiveness. These methods encompass symbolic execution-based approaches [29], rule-based techniques [9, 19], and code similarity-based methods [16, 28, 36]. The intricate programming logic present in real-world software projects poses a challenge for manually identifying detection rules or patterns, significantly hampering the performance of traditional static analysis-based methods. In recent years, the wealth of data derived from open-source code has provided a favorable environment for constructing deep learning-based vulnerability detection models. Leveraging the feature extraction capabilities of deep learning facilitates the automatic identification of latent vulnerability patterns, offering a novel approach to vulnerability detection. Numerous studies [15, 17, 26, 32, 39] have demonstrated that deep learning-based methods outperform traditional static analysis-based methods. However, existing deep learning-based vulnerability detection approaches still exhibit some limitations, as outlined below.

The program semantics have not been fully leveraged. On one hand, some approaches [17, 26] convert source code into token sequences and employ language models to detect vulnerabilities. However, in contrast to natural language, source code contains rich structural information. Consequently, these approaches struggle to achieve good performance due to their inability to capture the structural information of the program. On the other hand, some works [21, 32, 33, 39] transform functions into graph-structured representations, e.g., Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), and Program Dependency Graph (PDG). While these graph-structured representations can capture structural information about certain aspects of the program, they often cannot obtain comprehensive and

*Corresponding author: yxding@hit.edu.cn

precise vulnerability semantics. Specifically, these methods typically extract program semantics from individual functions, disregarding call relationships between functions. In other words, these approaches lack the capability to perform inter-procedural analysis for source code.

The detection granularity is coarse-grained. The detection granularity of existing deep learning-based approaches is mostly at the file-level [6, 24], function-level [21, 26, 32, 39], or slice-level [2, 15, 17, 18]. Upon analyzing a considerable number of vulnerable samples, we observe that vulnerabilities typically involve only a few statements. This implies that software developers still need to invest a significant amount of time manually scanning numerous statements to identify vulnerable ones. Hence, it is crucial to devise a more fine-grained vulnerability detection approach capable of pinpointing vulnerable statements (i.e., vulnerability localization). However, constructing a fine-grained detection approach that covers the majority of vulnerabilities is challenging due to the substantial semantic differences between various vulnerabilities.

In this paper, we present a novel Fine-grained Vulnerability Detection approach based on a conditional Diffusion Probabilistic Model (FVD-DPM) to alleviate the aforementioned limitations.

Fully leveraging program semantics. To extract comprehensive and precise semantic features, we combine control flow, data flow, function calls, and code sequences to construct a joint code representation, i.e., Code Joint Graph (CJG). Firstly, we augment CFG with data flow based on the variable `define-use` relations to generate PDG, enabling the analysis of control and data dependencies within the function. Next, we extend the PDG using Call Graph (CG) to facilitate inter-procedural analysis. Furthermore, we incorporate code sequence information into the extended graph to capture the programming logic in the source code. CJG preserves both structural and unstructured program semantics. Overall, CJG enables both intra-procedural and inter-procedural control and data flow analysis, capturing richer vulnerability semantics.

Fine-grained vulnerability detection. To train a fine-grained vulnerability detection model, we initially generate graph-level program slices from CJG. Subsequently, we formulate vulnerability detection as a diffusion-based graph-structured prediction problem. Specifically, we identify vulnerable statements by predicting node labels in the graph-level slices using a conditional diffusion probabilistic model. Diffusion probabilistic models (DPMs) belong to a class of generative models known for their ability to produce high-quality images [12, 22]. The main idea of DPMs is to iteratively add noise to an image and train a neural network to learn the noise and restore the image. Motivated by that, we treat the node label in each graph as an image in our vulnerability detection task. Leveraging the graph information (nodes and edges), we train a conditional diffusion probabilistic model by utilizing the probabilistic diffusion of node labels to identify vulnerable nodes. Previous studies have demonstrated that DPMs excel at modeling complex image domains, describing the distribution of neighboring pixels in the image and generating diverse high-quality images. Hence, we are confident that our FVD-DPM can effectively model node labels, extracting implicit patterns of various vulnerabilities and achieving high-performance fine-grained vulnerability detection.

Additionally, training deep learning-based models for fine-grained detection requires large-scale datasets. Existing datasets primarily fall into two categories: synthesized and real-world. The granularity of real-world datasets [26, 38, 39] is consistently coarse-grained (e.g., file-level or function-level), making them unsuitable for training fine-grained detection models. In this paper, based on the utilization of existing datasets in [15, 17], we construct a real-world, fine-grained dataset. This dataset includes vulnerability labels at function, slice, and statement levels. The datasets and source code presented in this paper are available online. [1]

To the best of our knowledge, this paper makes the following contributions:

- We propose a novel fine-grained code representation that extracts graph-level program slices from CJG. The CJG is obtained by integrating CFG, DFG, CG, and code sequence, enabling it to capture comprehensive and precise vulnerability semantics within the source code.
- We formalize fine-grained vulnerability detection as a diffusion-based graph-structured prediction problem. We build a conditional diffusion probabilistic model to predict node labels. With this approach, we achieve vulnerability detection at both slice-level and statement-level.
- We employ Graph Attention Network (GAT) with hybrid time encoding to predict noisy label at each timestep during the diffusion process. Moreover, we enhance the model by concurrently learning the mean and variance of the noisy label distribution, and modifying loss function.
- We evaluate our FVD-DPM approach on collected datasets, demonstrating its effectiveness in detecting vulnerabilities compared to state-of-the-art methods.

2 Motivating Examples

To highlight the motivation behind FVD-DPM, we use a typical vulnerability in the Linux Kernel as an example. Figure 1 presents the simplified vulnerable function `ksmbd_conn_handler_loop` for clarity. We observe that `ksmbd` fails to validate the `SMB` request protocol ID at line 23, resulting in an out-of-bounds read vulnerability. Exploiting this vulnerability successfully could lead to sensitive information leakage or denial of service (DoS). The fix for this vulnerability involves adding validation for the `SMB` request

[1] https://github.com/VulDet/FVD-DPM.git

```
1    int ksmbd_conn_handler_loop(void *p)
2    {
3        struct ksmbd_conn *conn = (struct ksmbd_conn *)p;
4        unsigned int pdu_size, max_allowed_pdu_size;
5        ...
6        conn->request_buf = kvmalloc(size, GFP_KERNEL);
7        if (!conn->request_buf)
8             break;
9        memcpy(conn->request_buf, hdr_buf, sizeof(hdr_buf));
10 -     if (!ksmbd_smb_request(conn))
11 -          break;
12       ...
13       if (size != pdu_size) {
14           pr_err("PDU error. Read: %d, Expected: %d\n", size, pdu_size);
15           continue;
16       }
17 +     if (!ksmbd_smb_request(conn))
18 +         break;
19       ...
20   }
21   bool ksmbd_smb_request(struct ksmbd_conn *conn)
22   {
23 -     return conn->request_buf[0] == 0;
24 +     __le32 *proto = (__le32 *)smb2_get_msg(conn->request_buf);
25 +     if (*proto == SMB2_COMPRESSION_TRANSFORM_ID) {
26 +         pr_err_ratelimited("smb2 compression not support yet");
27 +         return false;
28 +     }
29 +     if (*proto != SMB1_PROTO_NUMBER &&*proto != SMB2_PROTO_NUMBER &&
30 +         *proto != SMB2_TRANSFORM_PROTO_NUM)
31 +         return false;
32 +     return true;
33   }
```

Figure 1: An out-of-bounds read vulnerability (CVE-2023-38430) in Linux Kernel.

protocol ID, as demonstrated in lines 24-32. This example provides the following observations:

Observation 1. Comprehensive and precise program semantic extraction is necessary. As illustrated in Figure 1, the semantics of vulnerable code and non-vulnerable code are different. Traditional static analysis-based detection methods [4, 9, 19] may not capture this semantic difference adequately because predefined rules or patterns are often simplistic and incomplete. Furthermore, we observe that this vulnerability involves multiple statements in two functions. It cannot be easily detected by some existing deep learning-based methods [21, 32, 39] because they may ignore the call relationships between functions. Additionally, the change from lines 10-11 in the vulnerable code to lines 17-18 in the non-vulnerable code underscores the significance of the natural order of code statements in identifying vulnerable code.

In this paper, we combine control flow, data flow, function calls and code sequence to construct a joint code representation (CJG). This graph not only denotes the control and data dependencies among statements within and between functions, but also captures the natural order of the source code.

Observation 2. Detection models must identify global contextual information of nodes in the CJG. As shown in Figure 1, only three statements are vulnerable. This emphasizes the necessity of building a fine-grained detection model. Existing GNN-based detection approaches [2,8,32,39] typically propagate a node's information to its first-order or second-order adjacent nodes. However, in a vulnerable sample, different vulnerable statements may be separated by dozens of non-vulnerable statements. This indicates that vulnerable nodes in graph-level code representations may not be in each other's first-order or second-order neighborhood. Consequently, these GNN-based models may struggle to infer whether a potential vulnerable node directly affects another, as they often ignore the global contextual information of nodes, thereby hindering their ability to detect vulnerabilities.

Based on the above observations, we formalize vulnerability detection as a diffusion-based graph-structured prediction problem to enhance the detection of fine-grained vulnerabilities. We extract graph-level program slices from CJG and then construct a conditional diffusion probabilistic model to predict node labels within these graph-level slices. This model considers all nodes in the entire graph as a whole, and then models the joint distribution of node labels, which can effectively capture the global contextual information of nodes.

3 Methodology

In this section, we introduce our vulnerability detection framework, FVD-DPM. As illustrated in Figure 2, FVD-DPM extracts Graph-based Vulnerability Candidate slices (GrVCs) and introduces a conditional diffusion probabilistic model for detecting vulnerabilities. The framework comprises two stages: 1) Generate CJG from source code and utilize program slicing technology to extract GrVCs from the CJG, thereby obtaining fine-grained vulnerability representations. 2) Construct a conditional diffusion probabilistic model to predict node label distribution for GrVCs, thereby achieving precise identification and localization of vulnerable code.

3.1 Feature Extraction

To capture comprehensive and precise program semantics, we introduce a novel graph-level code representation: Code Joint Graph (CJG). We construct the CJG by integrating CFG, DFG, CG, and the code sequence. The CJG not only incorporates control and data flows within and between functions but also preserves the natural sequential order of the code. However, a function typically comprises dozens or even hundreds of statements, with vulnerabilities concentrated in only a few of them. Training a detection model on the entire graph might diminish the ability to extract significant vulnerability features, making it challenging to precisely locate vulnerable statements. Consequently, our approach involves extracting graph-level program slices from the CJG to obtain a more fine-grained vulnerability representation.

3.1.1 Generating Code Joint Graph

We combine multiple code representations, including CFG, DFG, CG, and code sequence, to generate the CJG. Next, we will briefly introduce each type of code representation and explain how we integrate them into the CJG.

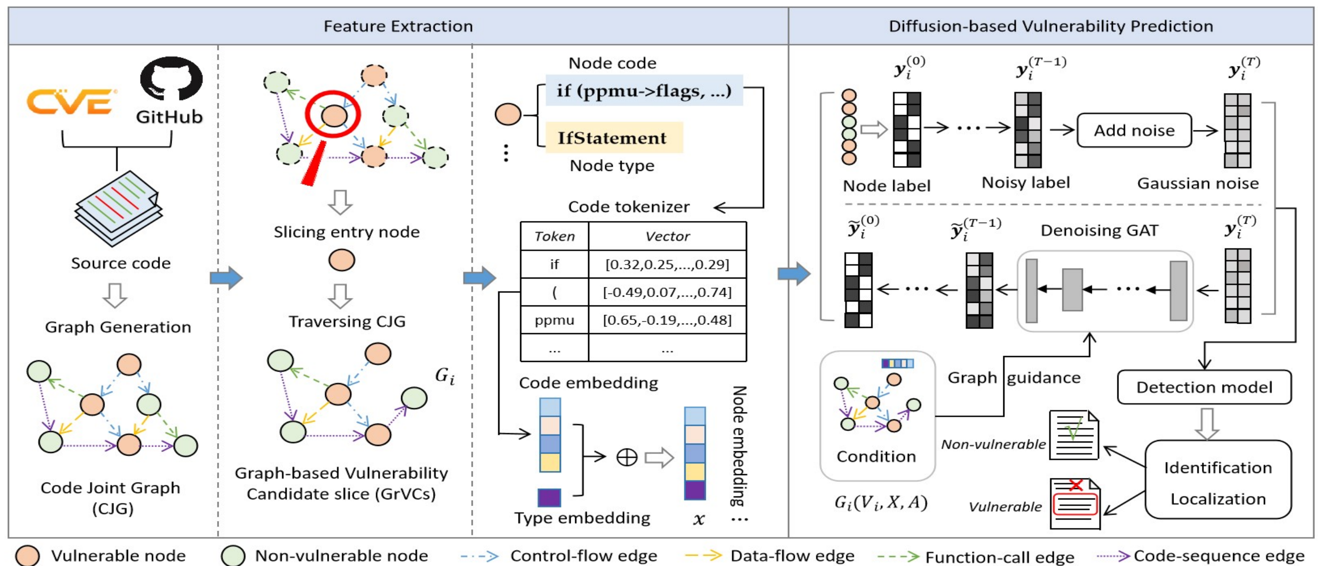


Figure 2: Overview of FVD-DPM.

Control Flow Graph (CFG). CFG explicitly delineates all paths a program will take during execution. Path selection is influenced by conditional statements such as `if`, `for`, and `switch`. In the CFG, nodes are linked by directed edges to signify the transfer of control.

Data Flow Graph (DFG). DFG monitors variable usage across the program. Any data flow entails accessing or modifying specific variables. An edge in the DFG signifies subsequent access or modification of related variables.

Call Graph (CG). Call Graph illustrates the relationships between function calls, tracing from the caller to the callee. Integrating the Call Graph allows for inter-procedural analysis, enabling the capture of more comprehensive control and data flow information.

Code Sequence (CS). To preserve the inherent order of the source code, we introduce CS edges to sequentially connect each statement. The inclusion of these edges serves to maintain the programming logic conveyed by the source code.

Specifically, we utilize the static analysis tool Joern[2] to automatically generate CFG and DFG for each function. By combining CFG and DFG, we can generate PDG. All generated graphs are stored in the database Neo4j[3]. Subsequently, we conduct data flow inspection via reaching definition analysis. This analysis tracks the definitions and uses of variables in the function to determine which definitions lead to a given point of use. To generate CG, we query all nodes related to functions from the database, which are named as function nodes and constitute the nodes of the CG. We can generate a callee list (the *type* of node is `Callee` in it) for each function node. If a function name in the callee list corresponds to a node in the CG, then an edge is added from the caller to the callee. Consequently, the CG can be constructed.

A function f can be denoted by a CJG $G = (V, E)$, where four distinct attribute edges (control-flow edges, data-flow edges, function-call edges, and code-sequence edges) share the same node set V. Each node $v \in V$ possesses two attributes: *code* signifies a statement in the source code, and *type* indicates specific node type. The edge set E comprises a set of directed edges, where each edge symbolizes a particular dependency relationship between a pair of nodes. Specifically, the node's *type* in CJG includes the following 23 types: `Function`, `ExpressionStatement`, `Label`, `Condition`, `CFGEntryNode`, `IdentifierDeclStatement`, `UnaryExpression`, `GotoStatement`, `Parameter`, `Callee`, `CallExpression`, `BreakStatement`, `InfiniteForNode`, `ContinueStatement`, `ClassDefStatement`, `Statement`, `IncDecOp`, `ReturnStatement`, `Expression`, `CFGExitNode`, `ForInit`, `AssignmentExpr`, and `IfStatement`. These node types are defined internally by the Joern. We list the most common eight node types and the average number of nodes of each type per function in the appendix (see Table 9).

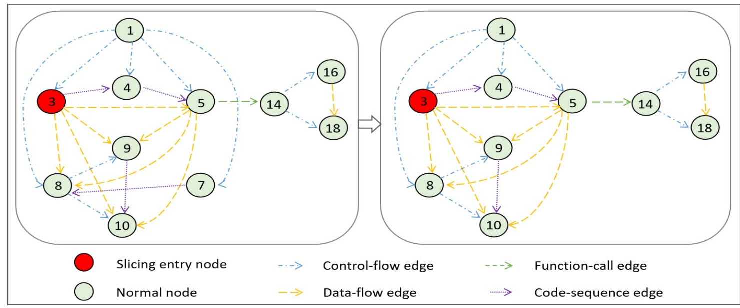
3.1.2 Extracting Slicing Entry Nodes

To achieve a more fine-grained representation of vulnerable code, we perform program slicing to extract graph-level program slices, i.e., GrVCs, from the CJG. During program slicing execution, a starting node is necessary to extract general program's characteristics from the CJG. In this context, we introduce the concept of slicing entry nodes. The slic-

[2] https://joern.io/
[3] https://neo4j.com/

```
1  struct vfsmount *collect_mounts(struct path *path)
2  {
3      struct mount *tree;
4      namespace_lock();
5      tree = copy_tree(real_mount(path->mnt), path->dentry, CL_COPY_ALL | CL_PRIVATE);
6
7      namespace_unlock();
8      if (IS_ERR(tree))
9          return ERR_CAST(tree);
10     return &tree->mnt;
11 }
12
13
14 struct mount *copy_tree(struct mount *mnt, struct dentry *dentry, int flag)
15 {
16     struct mount *res, *p, *q, *r, *parent;
17     ...
18     return res;
19 }
```

(a) Source code



(b) Program slicing

Figure 3: An example of generating Graph-based Vulnerability Candidate slices (GrVCs).

ing entry node is an entry point to extract features that may be associated with a vulnerability, but it cannot conclusively identify the presence of a vulnerability.

Through the analysis of numerous vulnerable codes, we observe that the majority of vulnerabilities are closely linked to API/library function calls, sensitive variables (array and pointer variables), and arithmetic expressions. Concurrently, many existing studies [2, 17, 18] highlight the misuse of API/library functions as a major vulnerability cause. Moreover, various vulnerability types (e.g., null pointer dereference, double free, buffer overflow) are frequently associated with array and pointer variables, extensively utilized in traditional static analysis methods [4, 9]. Furthermore, arithmetic expressions often contribute to vulnerabilities such as integer overflow. Hence, we select slicing entry nodes by matching these three types of characteristics that may be related to vulnerabilities.

In total, we selected 762 API/library function calls, all of which come from several static vulnerability detectors [3,4,9]. These functions primarily encompass input validation, encryption and decryption, access control and authorization, file and directory operations, and sensitive data processing. More details of these functions are available in our online dataset. We select slicing entry nodes based on the node's *type* and *code* attributes within CJG. For instance, if a node has the *type* `ExpressionStatement` and its *code* contains the character '=', it can be considered an arithmetic expression-related node, and thus, we consider it as a slicing entry node. According to our statistics, an average of 4% of the nodes in a CJG are slicing entry nodes.

3.1.3 Program Slicing

After selecting the slicing entry nodes, we extract GrVCs from the CJG. Specifically, starting from the slicing entry node, we iteratively perform forward and backward slicing until all nodes in the CJG are traversed, thus finding all nodes that are reachable from the slicing entry node. Finally, all reachable nodes are connected based on their dependency edges to generate the GrVCs. The GrVCs is essentially a subgraph of the CJG.

Figure 3 provides an example to illustrate the process of generating GrVCs. As shown in Figure 3a, the function `collect_mounts()` fails to adequately consider the possibility of execution after a path has been unmounted. This fault allows local users to exploit user-namespace root access for a `MNT_DETACH umount2` system call, causing a denial of service (DoS). There is only one slicing entry node in this example, represented by the red circle in Figure 3b. In our approach, we first extract the control flow, data flow, and code sequence information of the vulnerable code to construct the CJG, as depicted in Figure 3b. We then introduce function-call edges (i.e., Edge 5 $\rightarrow$ 14) through inter-procedural analysis. To reduce irrelevant nodes, we designate Node 3, associated with sensitive variables, as the slicing entry node for forward and backward slicing. After slicing, Node 7 is removed since there is no reachable path between Node 7 and Node 3.

3.1.4 Initial Node Embedding

After generating GrVCs, we convert all nodes in the graph into low-dimensional vectors to obtain acceptable inputs for deep learning models. We derive an initial node representation based on the *code* and *type* attributes. For *code*, we transform it into a token sequence and employ Word2Vec to train a token embedding model. Specifically, we use our training dataset to train a specific Word2Vec model for each dataset, ensuring that the vector representation is derived from domain-specific vocabulary. Subsequently, we obtain the code representation of each node by averaging the vectors of all tokens. Regarding the node *type*, we employ label encoding. Finally, we concatenate the vector representations of the *code* and *type* to obtain the initial node embedding.

3.2 Diffusion-based Vulnerability Prediction

In the following, we describe the detailed implementation process of using a conditional diffusion probabilistic model

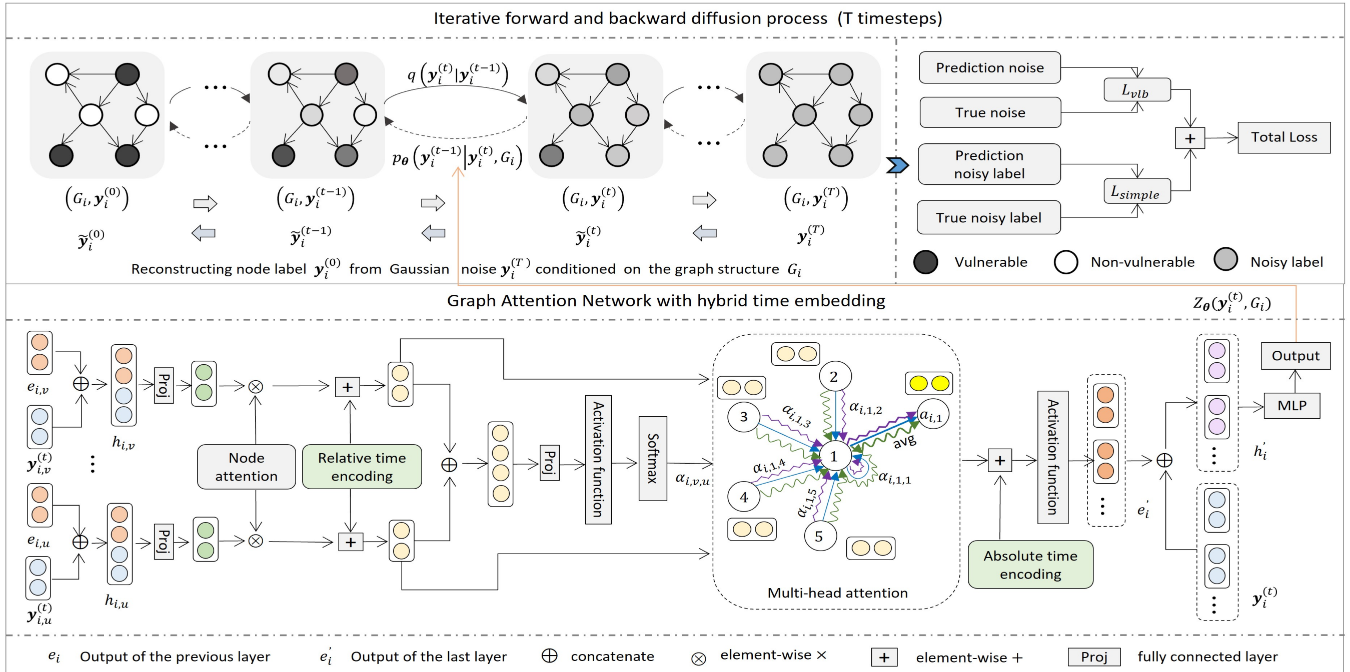


Figure 4: Illustration of FVD-DPM training on GrVCs.

for fine-grained vulnerability detection. Our goal is to develop a diffusion-based vulnerability detection method that captures the node label dependencies characterized by a graph.

Specifically, we formalize this diffusion process using a GrVCs, denoted as $G_i(V_i, E_i)$. The graph $G_i(V_i, E_i)$ consists of a node set V_i and an edge set E_i. The node label of the graph $G_i(V_i, E_i)$ is represented by y_i, with values of 0 (vulnerable) and 1 (non-vulnerable). We perform supervised learning on $G_i(V_i, E_i)$ to predict the node label y_i. Additionally, given that the node label y_i is discrete, we relax it into an one-hot vector to yield continuous values. The initial node feature matrix X and adjacency matrix A of $G_i(V_i, E_i)$ have been obtained. Thus the formal representation of the graph can be denoted as $G_i(V_i, X, A)$, abbreviated as G_i.

As depicted in Figure 4, the training process of FVD-DPM comprises two sub-processes: forward diffusion and reverse diffusion. Forward diffusion involves gradually injecting Gaussian noise into the node labels until converging to a standard Gaussian distribution. In contrast, reverse diffusion entails progressively denoising from the standard Gaussian distribution, conditioned on the graph structure of GrVCs, thereby reconstructing node labels. Within each GrVCs, we anticipate the presence of similar features between adjacent nodes. Graph Attention Network (GAT) is suitable for capturing the structural information of these graphs. Therefore, we employ GAT to learn the abstract features of GrVCs. Additionally, we aim to predict the noisy label distribution at each timestep based on the GAT.

3.2.1 Forward Diffusion Process

Following the models of [12, 13], the specific process of forward diffusion is as follows. Given the node label $y_i^{(0)} = y_i$ and the number of diffusion timesteps T, we assume that $y_i^{(0)}$ conforms to the initial data distribution $q(y)$. Gaussian noise is continuously injected into the data distribution during the forward diffusion process. The variance of the Gaussian noise injected at each timestep is fixed, and the mean depends only on the noisy label at the previous timestep. The forward diffusion process is defined as a Markov process, where $y_i^{(t)}$ depends only on $y_i^{(t-1)}$. Therefore, the forward diffusion process can be defined as follows:

$$q\left(y_i^{(1)}, \cdots, y_i^{(T)} \middle| y_i^{(0)}\right) = \prod_{t=1}^{T} q\left(y_i^{(t)} \middle| y_i^{(t-1)}\right). \tag{1}$$

$$q\left(y_i^{(t)} \middle| y_i^{(t-1)}\right) = N\left(y_i^{(t)}; \sqrt{1-\beta_t} y_i^{(t-1)}, \beta_t I\right). \tag{2}$$

where I is an identity matrix, and $\beta_t \in [0, 1]$ is the fixed variance schedule for the noise injected at the timestep t. The variance schedule β_t increases linearly with the increase of t, and $\beta_T = 1$. If we sample from the Gaussian distribution $N(\mu, \sigma^2)$ and the parameters μ and σ are learned from the neural network, there is a risk of gradient vanishing during the back-propagation process because the sampling process is non differentiable. To address this issue, we exploit the reparameterization technique [14]. Specifically, we sample z from

a standard Gaussian distribution and calculate $\mu + \sigma * z$. This operation is equivalent to performing an affine transformation, transferring randomness to the constant z. Given the notion $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$, we have:

$$q\left(y_i^{(t)} \middle| y_i^{(0)}\right) = N\left(y_i^{(t)}; \sqrt{\bar{\alpha}_t} y_i^{(0)}, (1 - \bar{\alpha}_t) I\right). \tag{3}$$

When $t = T$, the final noisy label obtained from the forward diffusion process confirms to a standard Gaussian distribution, i.e., $q\big(y_i^{(T)} | y_i^{(0)}\big) \approx N\big(y_i^{(T)}; 0, I\big)$.

3.2.2 Conditional Reverse Process

The reverse diffusion process is essentially a conditional probabilistic diffusion process, involving the reconstruction of the node label $y_i^{(0)}$ from Gaussian noise conditioned on the graph structure G_i and $y_i^{(T)}$. The $y_i^{(T)}$ is sampled from the standard Gaussian distribution $N(0, I)$. The reverse diffusion process can be defined as the conditional probability distribution $p_\theta\big(y_i^{(0)}, \cdots, y_i^{(T-1)} \big| y_i^{(T)}, G_i\big)$, learning to reconstruct the forward diffusion process given by $q(\cdot)$. It is also a Markov process and can be defined as:

$$p_\theta\big(y_i^{(0)}, \cdots, y_i^{(T-1)} \big| y_i^{(T)}, G_i\big) = \prod_{t=1}^{T} p_\theta\big(y_i^{(t-1)} \big| y_i^{(t)}, G_i\big). \tag{4}$$

$$p_\theta\big(y_i^{(t-1)} \big| y_i^{(t)}, G_i\big) = N\big(y_i^{(t-1)}; \mu_\theta(y_i^{(t)}, G_i), \Sigma_\theta\big). \tag{5}$$

3.2.3 Learning the Mean μ_θ and the Variance Σ_θ

To estimate the mean μ_θ and the variance Σ_θ, we first calculate the inverse distribution $q\big(y_i^{(t-1)} \big| y_i^{(t)}, y_i^{(0)}\big)$. Based on equations (2), (3) and Bayes theorem, we can calculate $q\big(y_i^{(t-1)} \big| y_i^{(t)}\big)$, which is also a Gaussian distribution denoted as $N\big(\hat{\mu}_t, \hat{\Sigma}_t\big)$. Assuming $\bar{Z}_t \sim N(0, I)$, the calculation for the mean $\hat{\mu}_t$ and variance $\hat{\Sigma}_t$ are as follows:

$$\hat{\mu}_t = \frac{1}{\sqrt{\alpha_t}} \left(y_i^{(t)} - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \bar{Z}_t \right). \tag{6}$$

$$\hat{\Sigma}_t = \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t} \beta_t. \tag{7}$$

Then we employ GAT to parameterize the distribution $p_\theta\big(y_i^{(t-1)} \big| y_i^{(t)}, G_i\big)$, and apply it to match the data distribution $q\big(y_i^{(t-1)} \big| y_i^{(t)}\big)$, gradually reconstructing the target distribution from the standard Gaussian distribution. The mean μ_θ can be calculated based on the following equation:

$$\mu_\theta(y_i^{(t)}, G_i) = \frac{1}{\sqrt{\alpha_t}} \left(y_i^{(t)} - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} Z_\theta\big(y_i^{(t)}, G_i\big) \right). \tag{8}$$

In the limit of infinite diffusion timesteps, the mean μ_θ is more effective in determining the distribution than the variance Σ_θ. However, Nichol and Dhariwal [22] pointed out that the first few timesteps of the diffusion process contribute the most to the variational lower bound of the log-likelihood. Therefore, it appears that we could improve the log-likelihood by selecting a better Σ_θ. To achieve this, we set Σ_θ to the weighted sum of β_t and $\hat{\Sigma}_t$ as follows.

$$\Sigma_\theta = \exp\big(\kappa \log \beta_t + (1 - \kappa) \log \hat{\Sigma}_t\big) \tag{9}$$

Here, κ is a learnable parameter vector.

As the number of diffusion timesteps T is usually large, FVD-DPM needs to sequentially calculate dozens or even hundreds of timesteps to obtain an accurate node label distribution in the reverse diffusion process. This results in training one graph taking several tens or even hundreds of times longer than general deep learning models, significantly reducing the training speed of the model. To address this issue, we sample timestep T during the model training process. Detailed description is in Appendix A.

3.2.4 GAT with Hybrid Time Encoding

Due to the randomness of sampling timesteps, we may not effectively predict the noise at each sampled timestep t using GAT. To enhance noise prediction, we aim to make the GAT model remember the noise level at the timestep t. Therefore, we follow the position encoding method in Transformer [30] and inject time encoding information into the learning process of GAT. However, absolute time encoding alone cannot estimate the relative positions between sampled timesteps. To overcome this limitation, we combine relative time encoding and absolute time encoding to construct hybrid time encoding. As shown in Figure 4, for absolute time encoding, we directly concatenate the time encoding with the input embedding of GAT. For relative time encoding, we incorporate it into each layer of GAT by modifying the self attention calculation process. Both the absolute time encoding $abs(t)$ and relative time encoding $rel(t)$ consist of sine and cosine functions of different frequencies and fully connected layers. Thus, we parameterize $Z_\theta\big(y_i^{(t)}, G_i\big)$ using an improved GAT as follows:

$$\alpha_{i,v,u}^{m} = \frac{\exp\big(\phi(\omega^{\mathsf{T}}[(W^m h_{i,v} \oplus W^m h_{i,u}) + rel(t)])\big)}{\sum_{l \in N_v} \exp\big(\phi(\omega^{\mathsf{T}}[(W^m h_{i,v} \oplus W^m h_{i,l}) + rel(t)])\big)}. \tag{10}$$

$$a_{i,v} = \sigma\Big(\frac{1}{M} \sum_{m=1}^{M} \sum_{u \in N_v} \alpha_{i,v,u}^{m} \big(W^m h_{i,u} + rel(t)\big)\Big). \tag{11}$$

$$h_{i,v}^{'} = \varphi\big(a_{i,v} + abs(t)\big) \oplus y_{i,v}^{(t)}. \tag{12}$$

$$Z_\theta(y_i^{(t)}, G_i) = \mathrm{MLP}(h_i^{'}). \tag{13}$$

where M is the number of attention heads, $\alpha_{i,v,u}^{m}$ is the weight coefficient calculated by the m-th attention head, W^m is a learnable weight matrix, and ω is a learnable weight vector. The $\phi(\cdot)$ and $\varphi(\cdot)$ represent the LeakyReLU and ELU activation functions, respectively. The N_v denotes the first-order

neighboring node set of node v. Here, $h_{i,v}$ is initialized by concatenating the node feature $x_{i,v} \in X$ and the noisy label $y_{i,v}^{(t)}$. Additionally, the symbol $.^{\top}$ represents transposition. After learning the graph representation using the GAT model, we apply an MLP to map the graph representation onto a final two-dimensional output vector to estimate the noise $Z_\theta\left(y_i^{(t)}, G_i\right)$.

3.2.5 Training Objective

The training goal of FVD-DPM model is to maximize the log-likelihood of the target distribution. DPMs generally optimize log-likelihood based on their variational lower bound, and employ KL divergence to derive L_{vlb}. However, directly predicting Z_θ can be better than predicting μ_θ, especially when combined with a reweighted loss function [12]:

$$L_{simple} = E_{t, y_i^{(0)}, Z}\left[\left\|\bar{Z}_t - Z_\theta\left(y_i^{(t)}, G_i\right)\right\|^2\right]. \tag{14}$$

This goal can be viewed as a reweighted form of L_{vlb}, which does not include the term of affecting Σ_θ. Since L_{simple} does not rely on Σ_θ, we define a new hybrid training objective:

$$L_{hybrid} = L_{simple} + \lambda L_{vlb}. \tag{15}$$

To achieve the identification and localization of vulnerabilities, we employ two distinct schemes. Firstly, we select the node with the highest probability, and its label is used as the prediction for the graph, achieving vulnerability identification. Additionally, by mapping the predicted node labels back to the statements in the source code, we can pinpoint the vulnerable statements, achieving vulnerability localization.

4 Evaluation

4.1 Experimental Setup

4.1.1 Research Questions

We aim to answer the following research questions:

- RQ1: How effective is FVD-DPM when compared to state-of-the-art vulnerability detection approaches?
- RQ2: How effective is CJG in vulnerability detection compared to existing code representations?
- RQ3: Can FVD-DPM perform better in vulnerability detection by incorporating hybrid time encoding into GAT, and simultaneously learning mean and variance of the noisy label distribution?
- RQ4: How effective and precise is FVD-DPM in locating different types of vulnerabilities?

To answer these questions, we implement FVD-DPM in Python using Pytorch.[4] We employ the static analysis tool Joern-0.3.1 to parse the source code and generate CFG, DFG, CG, and PDG. We utilize the database Neo4j-2.1.5 to store and access these graphs. The experiments are conducted on a Linux server equipped with NVIDIA GeForce RTX 4090 GPU and Intel(R) Core(TM) i9-13900K CPU running at 4.60 GHz. We utilize a word embedding vector size of 128 to train the Word2Vec model, from which the embedding representation of node codes can be obtained. The embedding dimension of the nodes in GrVCs is 129. The main hyper-parameters for training FVD-DPM are as follows: the Adam optimizer with a learning rate of 5e-3 is used for training the model up to 15000 epochs; the number of diffusion timesteps T is 40; the batch size is set to 64; the number of GAT layers is 2; the hidden dimensions and the number of heads in GAT are 32 and 2, respectively. We use an optional linear skip connection between each GAT layer and apply a dropout probability of 0.5. The number of nodes in each GrVCs is set to 400.

[4] https://pytorch.org/

4.1.2 Datasets

To evaluate the performance of FVD-DPM in detecting vulnerabilities in various scenarios and to compare it with existing state-of-the-art detection methods, we established our dataset using two sources: (1) Two well-known datasets, i.e., NVD and SARD, which have been widely used in previous detection approaches [2,15,17,35]. (2) Three real-world open-source projects, namely OpenSSL, Libav, and Linux Kernel, were manually collected from the GitHub repository.

We followed the methodology in [38] to collect the three open-source projects. Since only a small proportion of commits on GitHub are related to vulnerabilities, we selected vulnerability-related fixing commits by matching a set of predefined vulnerability keywords (e.g., vulnerability, out of bound, use after free, memory leak, null pointer, dereference, buffer overflow) in the commit messages. For each vulnerability-related fixing commit, we first extracted the vulnerable code file, patched code file, and `diff` file. Subsequently, if a function originates from a vulnerable code file and has at least one statement deleted or changed (indicated by "-" in the `diff` file), it is labeled as vulnerable; otherwise, it is labeled as non-vulnerable. Deleted or changed statements are considered vulnerable statements. Each node in GrVCs represents a statement in the source code; nodes corresponding to vulnerable statements are labeled as vulnerable, while the rest are labeled as non-vulnerable. If at least one node in a GrVCs is vulnerable, the entire GrVCs is labeled as vulnerable. More details are in Appendix B.

Real-world data often includes more non-vulnerable examples than vulnerable ones. A model trained on such imbalanced datasets may tend to classify samples into the majority class. Additionally, DPM model treats each GrVCs as an image. If all pixels in an image are identical, having an abundance of such images would not benefit model training. To address these challenges, we employ resampling on datasets

Table 1: Statistics on datasets.

Dataset	#Version	#Vul. Fs	#Fs	#Vul. GrVCs	#Non-Vul. GrVCs	#GrVCs	#Nodes	#Edges
NVD	-	937	2,011	4,355	8,526	12,881	870,855	4,633,355
SARD	-	2,851	5,879	4,742	22,720	27,462	240,202	580,908
OpenSSL	0.9.6-3.0.7	2,009	2,302	6,677	3,362	10,039	221,262	684,357
Libav	0.6-11.5	1,666	1,956	7,710	4,334	12,044	334,964	1,372,749
Linux Kernel	2.6-5.17	1,178	1,528	4,036	2,287	6,323	272,267	1,099,651
Total	-	8,641	13,676	27,520	41,229	68,749	1,939,550	8,371,020

with severely imbalanced samples. Since the positive and negative sample sizes in the NVD are already balanced, we do not perform any resampling on it. For other datasets, we randomly selected approximately 10% of the original non-vulnerable functions to construct our negative samples.

Table 1 displays the statistics of our datasets, indicating a total of 13,676 C/C++ functions collected, with 8,641 being vulnerable. Column 2 provides information on the scope of project versions affected by vulnerabilities in our datasets. Column 5 shows the count of vulnerable GrVCs, and column 6 represents the count of non-vulnerable GrVCs. Columns 8 and 9 present the total numbers of nodes and edges in GrVCs, respectively. For each dataset, we randomly select 80% of the GrVCs for training, with the remaining 20% allocated for validation and testing.

4.1.3 Evaluation Metrics

We employ widely used metrics, including Recall (R), F1 score (F1), Area Under Curve (AUC), and Matthews Correlation Coefficient (MCC), to evaluate vulnerability identification and localization systems. Additionally, Intersection over Union (IoU) is also used to evaluate vulnerability localization results. The metric $IoU = \frac{|U \cap V|}{|U \cup V|}$ is proposed in [15], where U is the set of truly vulnerable statements and V is the set of detected vulnerable statements.

4.1.4 Baseline Methods

To evaluate the effectiveness of FVD-DPM in vulnerability identification, we compare it with seven state-of-the-art detection methods. Cppcheck [4] and Flawfinder [9] are both rule-based detection tools. Devign [39] builds an extended Code Property Graph (CPG) and employs a GNN model to identify vulnerabilities at the functional-level. VulDeePecker [18] and SySeVR [17] are slice-level detection methods that extract sequence-level program slices from PDG and use BGRU to identify vulnerabilities. VulDeeLocator [15] converts source code into intermediate code and employs BGRU to detect vulnerabilities. MVD [2] extracts graph-level program slices from PDG and utilizes a flow-sensitive GNN model to learn code representations and identify vulnerabilities.

Moreover, to assess the effectiveness of FVD-DPM in vulnerability localization, we compare it with four state-of-the-art localization methods. Cppcheck [4] performs semantic checks on nodes in the AST by calling various rule classes. VulDeeLocator [15] applies an attention layer to build a granularity refinement module. It selects the top k statements with the highest attention weight as the vulnerability localization result. DeepLineDP [25] employs a Hierarchical Attention Network (HAN) to calculate the vulnerability risk of the code token and then sums the risks of all tokens in each statement to form statement-level risks, thereby ranking all statements to achieve vulnerability localization. VulChecker [20] is a tool that performs statement-level vulnerability detection, utilizing message-passing graph neural networks to learn code embedding representations. It constructs a new code representation (i.e., ePDG) based on LLVM intermediate representation (IR).

4.2 RQ1: Performance of Proposed Approach

To answer this RQ, we compare FVD-DPM with seven state-of-the-art vulnerability identification approaches and four state-of-the-art vulnerability localization approaches. Specifically, we train and test all approaches except for Devign, VulDeeLocator, and MVD on five datasets. For each approach, we adjust the hyper-parameters through grid search to obtain the optimal model. Due to the unavailability of complete code disclosure by Design, VulDeeLocator, and MVD, we directly cite research results from their articles.

Vulnerability identification (slice-level detection). Table 2 shows the average comparative results of our FVD-DPM with state-of-the-art vulnerability identification methods. Additionally, Figure 5 displays the F1 score of Flawfinder, Cppcheck, VulDeePecker, SySeVR, and FVD-DPM on each dataset. Overall, FVD-DPM achieves better results and outperforms state-of-the-art vulnerability identification methods. On average, FVD-DPM attains an F1 score of 85.73%, a Recall of 82.93%, an AUC of 86.40%, and an MCC of 72.14%. Flawfinder and Cppcheck, as representatives of rule-based detectors, perform the worst among all approaches. The reason is that predefined detection rules or patterns cannot cover the majority of vulnerabilities, limiting the effectiveness of these rule-based detectors. Furthermore, FVD-DPM improves the F1 score over Devign by 12.47%. This indicates that, although GNN's powerful capability in extracting program semantics can make Devign perform outstandingly, the existence of redundant nodes unrelated to vulnerabilities in CPG limits

Devign's ability to detect vulnerabilities.

Table 2: Comparison with state-of-the-art vulnerability identification approaches (metrics unit: %).

Method	F1	R	AUC	MCC
Flawfinder	49.73	52.86	-	10.07
Cppcheck	61.09	71.43	-	-
MVD	65.20	61.50	-	-
VulDeePecker	71.48	77.62	77.65	51.20
Devign	73.26	-	-	-
SySeVR	79.72	81.26	-	60.49
VulDeeLocator	**85.90**	82.07	-	-
FVD-DPM (ours)	85.73	**82.93**	**86.40**	**72.14**

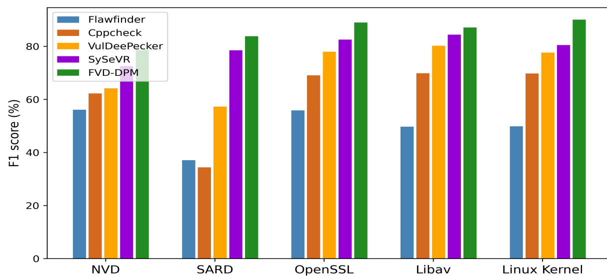


Figure 5: Comparing FVD-DPM with state-of-the-art vulnerability identification approaches on each dataset (F1 score).

VulDeePecker, SySeVR, MVD, and VulDeeLocator are all slice-based detection approaches. As shown in Figure 5, VulDeePecker performs worse than SySeVR on all datasets. The reason for this is that VulDeePecker only considers vulnerability characteristics related to API function calls and ignores control dependencies in the source code. Furthermore, we observe that FVD-DPM improves F1 score and MCC over SySeVR by 6.01% and 11.65%, respectively. One of the main reasons is that our graph-level program slices (i.e., GrVCs) can extract more comprehensive and precise program semantics than the sequence-level slices generated by SySeVR. Notably, compared to the best vulnerability detection approach VulDeeLocator, FVD-DPM achieves competitive result. In addition, MVD is most similar to our approach, both of which are vulnerability detection methods based on graph-level program slices. MVD performs worse than FVD-DPM in terms of each metric. There are two main reasons for this result. On one hand, GrVCs extracted from CJG is more effective than the graph-level slices extracted from PDG by MVD in extracting vulnerability features. On the other hand, our proposed conditional diffusion probabilistic model, which can capture the global contextual information of nodes, is more effective than the GNN-based model in node classification.

Vulnerability localization (statement-level detection). After evaluating the vulnerability identification capability of FVD-DPM, we assess its effectiveness in locating vulnerable statements. Table 3 presents comparative results on the metric IoU between FVD-DPM and three state-of-the-art localization methods. Here, IoU is the average value measured on IoUs between the detected vulnerable code and the ground-truth vulnerable code in the test dataset. We observe that FVD-DPM significantly outperforms the three existing state-of-the-art vulnerability localization methods on all datasets.

Table 3: The performance comparison of different vulnerability localization approaches (metrics unit: %).

Method	NVD	SARD	OpenSSL	Libav	Linux Kernel
Cppcheck	15.27	9.89	48.79	42.82	27.33
DeepLineDP	31.05	14.67	18.53	24.31	30.02
VulDeeLocator	32.60	36.30	-	-	-
FVD-DPM	59.04	72.35	63.13	62.95	72.70

FVD-DPM outperforms Cppcheck in terms of IoU by an average of 37.21%. The fundamental reason for this performance gap is that the predefined detection rules in Cppcheck cannot cover the majority of vulnerabilities, resulting in low localization performance. FVD-DPM, on average, improves IoU by 42.32% over DeepLineDP. One of the main reasons is that DeepLineDP adopts HAN to learn the sequence features of the source code, ignoring the control and data dependencies of the program. FVD-DPM is roughly 31.59% higher than VulDeeLocator. This difference can be attributed to the limitations of the localization module in VulDeeLocator. Specifically, VulDeeLocator applies an attention layer to construct a code granularity refinement module, which selects the most suspicious k statements based on attention weights as the localization result. Due to both the varying lengths of functions and the varying numbers of vulnerable statements they contain, fixing the value of k significantly reduces the localization performance of VulDeeLocator. Instead, FVD-DPM applies a DPM model to predict node labels in graph-level program slices and corresponds each node label to a statement in the source code, thereby achieving more effective localization.

Table 4: Comparative results of VulChecker and FVD-DPM on the Juliet dataset (metrics unit: %).

Method	CWE190	CWE121	CWE122	CWE415	CWE416
VulChecker	97.00	85.40	79.00	100.00	90.90
FVD-DPM	97.87	88.30	90.93	94.83	88.23

Additionally, to achieve a fair comparison with VulChecker, we use the Juliet dataset introduced in [20] to evaluate FVD-DPM. The dataset includes the following five vulnerability types: integer overflow (CWE-190), stack overflow (CWE-121), heap overflow (CWE-122), double free (CWE-415), and use-after-free (CWE-416). As shown in Table 4, FVD-DPM achieves competitive results with VulChecker overall on these

Table 5: Effectiveness of CJG in vulnerability identification and localization (metrics unit: %).

Dataset	Model Structure	Vulnerability Identification				Vulnerability Localization				
		F1	R	AUC	MCC	F1	R	AUC	MCC	IoU
NVD	DPM+CPG	70.87	67.23	79.09	60.25	73.60	63.61	81.73	74.20	47.26
	DPM+PDG	76.56	76.15	82.81	65.80	**78.99**	**78.50**	**89.21**	**78.92**	55.55
	DPM+CJG	**78.63**	**78.54**	**83.79**	**67.61**	77.21	75.17	87.54	77.14	**59.04**
SARD	DPM+CPG	77.32	74.52	83.93	69.55	72.34	64.57	82.23	72.72	62.81
	DPM+PDG	76.20	69.47	82.27	68.92	75.93	69.51	84.69	76.14	63.03
	DPM+CJG	**83.82**	**81.01**	**89.09**	**80.34**	**83.66**	**80.87**	**90.40**	**83.64**	**72.35**
OpenSSL	DPM+CPG	77.12	67.67	78.60	57.20	58.98	45.48	72.60	60.93	43.92
	DPM+PDG	83.17	78.66	81.54	62.02	72.36	70.07	84.98	72.05	53.63
	DPM+CJG	**88.99**	**86.80**	**85.15**	**68.80**	**79.09**	**81.27**	**90.56**	**78.97**	**63.13**
Libav	DPM+CPG	80.59	72.80	81.89	64.25	67.19	53.67	76.72	68.64	53.29
	DPM+PDG	84.68	81.42	84.37	**68.55**	77.85	**76.98**	**88.43**	77.74	60.27
	DPM+CJG	**87.11**	**82.70**	**85.10**	68.22	**79.62**	75.40	87.65	**79.64**	**62.95**
Linux Kernel	DPM+CPG	84.63	79.74	86.41	74.10	76.91	68.21	84.01	77.15	62.97
	DPM+PDG	86.82	80.08	87.01	73.14	81.03	78.23	89.08	81.00	67.61
	DPM+CJG	**90.09**	**85.59**	**88.85**	**75.71**	**81.97**	**80.88**	**90.39**	**81.88**	**72.70**

five vulnerability types. This demonstrates that both our CJG built from source code and ePDG built from LLVM IR are effective in capturing fine-grained program semantics. Compared to VulChecker, FVD-DPM achieves an 11.93% higher AUC on CWE-122. This is likely because heap memory allocation and release typically involve many API/library function calls. FVD-DPM generates program slices by analyzing relevant API/library functions, enabling it to extract fine-grained vulnerability characteristics more effectively, thereby better detecting this vulnerability type.

4.3 RQ2: Effectiveness of Code Joint Graph

The two most widely used and effective code representations are PDG and CPG. Many state-of-the-art vulnerability detection approaches [2, 8, 17, 18] extract program slices from PDG or CPG to detect vulnerabilities. Therefore, to evaluate the effective of our CJG in vulnerability detection, we replace CJG with PDG and CPG. We then extract graph-level program slices from these two graph structures using the same program slicing technique. Finally, we apply our designed conditional diffusion probabilistic model to detect vulnerabilities.

We evaluate the effectiveness of CJG in vulnerability detection using the same settings as those for answering RQ1. As shown in Table 5, DPM+PDG, DPM+CPG and DPM+CJG denote that we apply our designed DPM model to train vulnerability detection models based on PDG, CPG, and CJG, respectively. From the table, we observe that, compared with the state-of-the-art baseline approaches in RQ1, the DPM+CPG, DPM+PDG, and DPM+CJG models obtain competitive or better detection performance in most cases. This phenomenon further demonstrates the effectiveness of our DPM model for detecting vulnerabilities. Overall, for each dataset, CJG outperforms CPG and PDG in most metrics. A main reason is that CJG captures more comprehensive and precise code semantics, including control and data flows within and between functions, as well as the natural order of the code. Moreover, the relationships between statements preserved in CJG enable the DPM trained on CJG to better capture the joint dependencies of node labels.

It is worth noting that CPG achieves the worst detection performance in most cases. There are two main reasons for this performance gap. Firstly, CPG may not perform well in the graph model learning process, as information from AST cannot flow across the graph. Another important reason is the issue of node labeling errors. Because there is a one-to-one correspondence between nodes in CJG/PDG and statements in the source code, statement-level vulnerability labels can be directly assigned to nodes in these graphs, with almost no annotation errors. However, AST nodes in CPG are usually composed of partial tokens in a statement, which may not necessarily indicate vulnerabilities, making it difficult to determine the labels of such nodes. Therefore, compared to CPG, we believe that PDG and CJG are more suitable for combining our designed diffusion probabilistic model to train a graph-based fine-grained vulnerability detection model.

Additionally, we provide an evaluation of the impact of different types of edges in CJG on model performance. Overall, the model's performance gradually improved as we added different types of dependency edges to the CFG. Detailed analysis can be found in Appendix C.

4.4 RQ3: Ablation Study

One of the main contributions of our approach FVD-DPM, is the utilization of GAT with hybrid time encoding to predict

Table 6: Comparative experiments on models with and without hybrid time encoding (metrics unit: %).

Time Encoding	Vulnerability Identification				Vulnerability Localization				
	F1	R	AUC	MCC	F1	R	AUC	MCC	IoU
Without	77.20	69.72	80.28	60.64	74.96	72.21	86.04	74.97	58.22
With	86.05	83.34	86.15	71.90	79.72	78.05	88.97	79.65	66.00

Table 7: Experimental results achieved by different objectives (metrics unit: %).

Objective	Vulnerability Identification				Vulnerability Localization				
	F1	R	AUC	MCC	F1	R	AUC	MCC	IoU
L_{simple}	84.98	81.64	85.32	71.05	77.82	74.76	87.32	77.88	63.67
L_{hybrid}	86.41	83.62	86.30	72.61	79.62	77.08	88.48	79.63	66.05

noisy labels at each timestep. By incorporating hybrid time encoding into GAT, FVD-DPM can effectively retain information about the noise level for each timestep. In addition, we estimate the noisy label distribution by simultaneously learning its mean and variance. Correspondingly, we define a new hybrid training objective, which is the weighted sum of the simplified loss L_{simple} and the variational lower bound (VLB). To evaluate the effectiveness of these two improvements, we conduct two groups of comparative experiments: 1) With time encoding vs. without time encoding; 2) Hybrid objective L_{hybrid} vs. simplified objective L_{simple}.

Impact of time encoding. Table 6 illustrates the impact of hybrid time encoding on FVD-DPM's performance. We observe that after removing the time encoding in GAT, FVD-DPM shows a significant decrease in all metrics. In terms of vulnerability localization, F1 score, AUC, MCC, and IoU decrease by an average of 4.76%, 2.93%, 4.68%, and 7.78%, respectively. This can be explained that the time encoding injected to GAT helps the model predict noisy labels, thereby achieving more effective fine-grained vulnerability detection.

Impact of different objectives. As shown in Table 7, overall, the hybrid objective L_{hybrid} significantly achieves better detection results than the simplified objective L_{simple}. Therefore, we usually prefer to employ L_{hybrid} over L_{simple} as it can improve detection performance without reducing likelihood.

Moreover, an evaluation of the performance of FVD-DPM under diverse training epochs is provided in Appendix D.

4.5 RQ4: Results on Different CWE Types

In this experiment, we evaluate the performance of FVD-DPM in locating different types of vulnerabilities. By counting the number of samples of different CWE vulnerability types on the SARD dataset, we select the most common eight types (i.e., CWE-78, CWE-121, CWE-122, CWE-124, CWE-126, CWE-127, CWE-195, CWE-134) for evaluation.

Figure 6 presents the evaluation metrics for each vulnerability type. We observe that FVD-DPM demonstrates good performance in locating vulnerable statements across different vulnerability types. Overall, FVD-DPM excels in vulnerability type CWE-195 and performs less effectively in CWE-121. The primary reason for this performance gap is that CWE-195 (i.e., signed to unsigned conversion error) exhibits a straightforward vulnerability pattern, usually associated with only one code statement. However, the vulnerability pattern of CWE-121 is complex and may involve multiple statements in various functions, making it more challenging to identify. Additionally, CWE-121 vulnerabilities often lead to program crashes, emphasizing the need for increased attention to such vulnerabilities. In future research, exploring methods to further enhance the model's ability for locating CWE-121 vulnerabilities would be valuable.

4.6 Applications to New Versions and Projects

To test the detection capability of FVD-DPM on new versions of the same project and on new projects, we manually collect 18 known vulnerabilities from new versions of OpenSSL, Libav, and Linux Kernel, as well as from new projects like OpenBSD, Asterisk, FreeRDP, and Wireshark.

As shown in Table 8, we identify 7 of the 8 known vulnerabilities in the new versions and 7 of the 10 known vulnerabilities in the new projects. Moreover, we detect 5 unknown vulnerabilities in the new versions, which are similar to some already disclosed vulnerabilities (as confirmed by our manual inspection). Additionally, two vulnerabilities in the FreeRDP project are missed by our model because the CWE types of these examples, CWE-20 and CWE-787, are not included in our training dataset. This implies that if new vulnerability types are encountered in new projects or versions not covered by the original dataset, the model may need to be retrained with these additional datasets.

5 Insights and Findings

We have successfully introduced the DPM model to vulnerability detection, yielding promising performance. This extends the applicability of DPMs to new domains. With

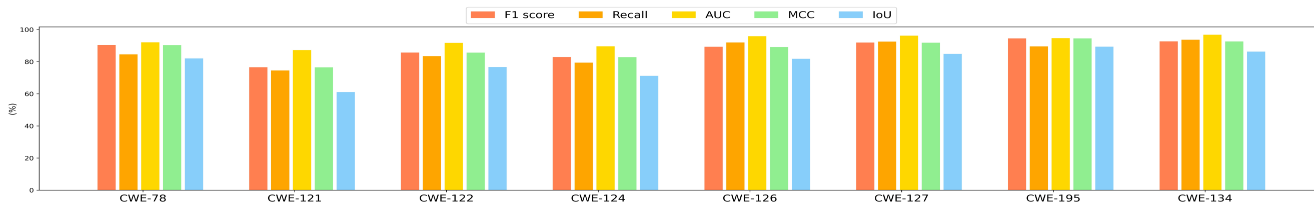


Figure 6: Locating results for the most common eight CWE vulnerability types on the SARD dataset.

Table 8: The 19 vulnerabilities detected and the 4 vulnerabilities missed by FVD-DPM in new versions and new projects.

Project	CVE ID	Vulnerable file	Status
OpenSSL 3.0.8-3.3	CVE-2023-6237	rsa_sp800_56b_check.c	Detected
	CVE-2023-0465	x509_vfy.c	Detected
	-	v3ext.c	Detected
	-	smime.c	Detected
	-	drbg_ctr.c	Detected
Libav 12	CVE-2017-9051	nsvdec.c	Missed
	-	mpegvideo_parser.c	Detected
	-	avconv_vaapi.c	Detected
Linux Kernel 6.2-6.5	CVE-2024-27388	gss_rpc_xdr.c	Detected
	CVE-2024-26593	i2c-i801.c	Detected
	CVE-2024-26592	transport_tcp.c	Detected
	CVE-2023-40791	scatterlist.c	Detected
	CVE-2023-38430	connection.c	Detected
OpenBSD 5.8	CVE-2022-48437	x509_verify.c	Detected
	CVE-2021-46880	x509_verify.c	Detected
	CVE-2020-16088	ca.c	Detected
Asterisk 18.2.0	CVE-2023-49294	manager.c	Detected
FreeRDP 2.0-3.5	CVE-2023-39354	nsc.c	Detected
	CVE-2022-39318	libusb_udevice.c	Missed
	CVE-2020-13398	crypto.c	Missed
Wireshark 4.2.5	CVE-2024-24476	addr_resolv.c	Detected
	CVE-2016-5358	packet-ppi.c	Detected
	CVE-2016-5353	packet-umts_fp.c	Missed

our training dataset containing numerous real-world vulnerabilities, FVD-DPM can effectively tackle complex projects. Moreover, our collected real-world vulnerability dataset, containing ground-truth labels at function, slice, and statement levels, offers data support for future research in vulnerability detection, localization, and repair.

Table 5 illustrates that the performance enhancement of FVD-DPM is not substantial when transitioning from PDG to CJG. This might be due to the lack of inter-procedural dependencies among many functions in our training datasets. We calculated the proportion of functions with inter-procedural dependencies in each dataset: NVD 2.5%, SARD 34.5%, OpenSSL 35.7%, Libav 27.8% and Linux Kernel 19.6%. We can see that the dataset with the smallest proportion of functions with inter-procedural dependencies corresponds to the worst model performance. This phenomenon indicates that when there are too few functions with inter-procedural dependencies in the training dataset, FVD-DPM struggles to learn useful program information from inter-procedural analysis effectively. Further analysis can be found in Appendix E.

In the training phase of FVD-DPM, we employ a relatively balanced training dataset, ensuring that the DPM model can effectively learn diverse vulnerability patterns. This approach equips FVD-DPM with the ability to identify vulnerabilities comprehensively. Consequently, even in real-world vulnerability scenarios, FVD-DPM effectively leverages learned knowledge of vulnerability patterns to identify potential vulnerabilities. Table 8 shows that FVD-DPM accurately detects the majority of vulnerabilities in new versions and new projects, proving its applicability in real-world situations.

6 Threats to Validity

External validity. External validity of our approach is influenced by two factors. First, we focus on detecting vulnerabilities in C/C++ source code within specific software projects, implying that the detection model might require adjustments when applied to other programming languages or software projects. Nonetheless, our approach is generic and has the potential to be extended to other programming languages and software projects. Second, attackers may employ adversarial machine learning techniques to poison our training datasets, potentially leading to the model missing certain vulnerabilities during deployment. Although this attack is possible, it is challenging because we use code samples collected from large-scale open-source repositories to enhance the coverage and diversity of our datasets. It is unlikely to inject adversarial samples into some large, actively-maintained open-source projects without being noticed by developers and users.

Internal validity. We may encounter label errors during the process of automatically generating ground-truth labels for nodes. These errors can impact the model's ability to identify real vulnerabilities. To avoid the harmful effects of incorrect node labeling on the model's performance, future research should develop more effective and precise node label-

ing methods. While we can partially explain the effectiveness of FVD-DPM, further research is needed in this interpretable direction. Additionally, in the future, we will explore the potential of leveraging popular large language models (LLMs), such as ChatGPT, in fine-grained vulnerability detection.

7 Related Work

7.1 Classical Vulnerability Detection Methods

Existing detection methods can be categorized into dynamic detection methods and static detection methods. Dynamic detection methods, such as symbolic execution [1] and fuzzing [10], are commonly used to identify vulnerabilities in binary code. These methods detect vulnerabilities by monitoring the program's running state, execution paths, and registration status. However, when applied to large-scale software systems, these methods encounter issues related to significant time and space overhead. In contrast, static analysis does not require the program to be running. Consequently, an increasing number of static detection methods have been proposed. Among these, source code-based static vulnerability detection stands out as a prominent technique, encompassing symbolic execution-based methods, rule-based methods, and code similarity-based methods.

Symbolic execution-based methods [29] represent source code using an intermediate language, combining symbolic execution and constraint solving to analyze vulnerabilities. Rule-based detection methods (e.g., ITS4 [31], Flawfinder [9], RATS and Yasca [19]) maintain an internal library of features for various vulnerability types and use lexical analysis algorithms to match these entries for identifying vulnerable code. Due to their reliance on simple parsers and predefined rules, these tools often yield a high false positive rate. The fundamental process of code similarity-based methods [16, 28, 36] involves transforming source code into an easily analyzable intermediate representation, and using a matching algorithm for similarity detection. However, in these methods, the manual extraction process of features is notably time-intensive, and the features often exhibit task-specific tendencies.

7.2 Deep Learning-based Detection

Deep learning-based approaches can automatically capture vulnerability patterns. Considering the scope of detection granularity, existing deep learning-based methods can be roughly divided into function-based methods and slice-based methods. Function-based methods typically focus on a single function to identify vulnerabilities. These methods always extract graph-level representations, such as AST, CFG and PDG, from the function to analyze vulnerable code [21, 32, 33, 39]. Li *et al.* introduced SySeVR [17], and VulDeePecker [18] as slice-based solutions for detection. Thereafter, an increasing number of slice-based methods were proposed [2, 8, 11]. Additionally, VulCNN [35] converts source code into an image and uses CNN to identify vulnerabilities.

Most of the aforementioned detection approaches can determine whether a file, function or code slice is vulnerable. However, these methods generally lack the capability to locate vulnerable statements. To address this issues, Wattanakriengkrai *et al.* [34] employed local interpretable model-agnostic Explanations (LIME) to identify risky code tokens, thus statements containing these tokens are predicted as vulnerable. Both VulDeeLocator [15] and DeepLineDP [25] identify vulnerable statements by designing a granularity refinement module behind the recurrent neural network based on attention mechanisms. Recently, Mirsky *et al.* [20] proposed a tool that achieves instruction-level vulnerability detection.

7.3 Diffusion Probabilistic Models

Taking inspiration from non-equilibrium statistical physics, Sohl-Dickstein *et al.* [27] introduced the concept of DPMs. This idea gained prominence with the development of denoising diffusion probabilistic models (DDPMs) [12], bringing DPMs to the forefront of image generation. DPMs have disrupted the longstanding dominance of generative adversarial networks (GANs) in demanding tasks of image synthesis, showing immense potential across various domains. To achieve more precise predictions and enhance the efficiency of the sampling process, researchers have introduced several improved versions of DDPMs [7, 22]. Recent studies have demonstrated the remarkable achievements of DDPMs across diverse domains, including computer vision [7], natural language processing [37], and multi-modal modeling [23].

The fundamental idea behind all these approaches is to progressively introduce random noise to the data and then remove the noise step by step to generate new data samples. Recently, researchers have uncovered the capability of DPMs to make predictions on complex graph-structured data, such as molecular structure prediction [13]. Inspired by this, we aim to employ DPMs in the field of vulnerability detection.

8 Conclusion

Fine-grained vulnerability detection is a challenging and largely unexplored task. In this paper, we propose an effective fine-grained detection approach, FVD-DPM, by formalizing vulnerability detection as a diffusion-based graph-structured prediction problem. FVD-DPM generates a new fine-grained code representation (GrVCs) by extracting program slices from the Code Joint Graph to capture more precise vulnerability semantics. Based on this, we design a conditional diffusion probabilistic model to predict node labels. Overall, FVD-DPM can simultaneously achieve vulnerability identification and localization.

Acknowledgments

This research was partially supported by the National Natural Science Foundation of China (Grant No.61872107) and Scientific Research Foundation in Shenzhen (Grant No. JCYJ20230807094318038).

References

[1] Eman Alatawi, Tim Miller, et al. Leveraging abstract interpretation for efficient dynamic symbolic execution. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 619–624. IEEE, 2017. https://doi.org/10.1109/ASE.2017.8115672.

[2] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1456–1468, 2022. https://doi.org/10.1145/3510003.3510219.

[3] Checkmarx. https://www.checkmarx.com/.

[4] Cppcheck. https://cppcheck.sourceforge.io/.

[5] CVEdetails.com. *# Of Vulnerabilities*. https://www.cvedetails.com/.

[6] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 47(1):67–85, 2018. https://doi.org/10.1109/TSE.2018.2881961.

[7] Prafulla Dhariwal and Alexander Nichol. Diffusion models beat gans on image synthesis. *Advances in neural information processing systems*, 34:8780–8794, 2021. https://doi.org/10.48550/arXiv.2105.05233.

[8] Yukun Dong, Yeer Tang, Xiaotong Cheng, Yufei Yang, and Shuqi Wang. SedSVD: Statement-level software vulnerability detection based on relational graph convolutional network with subgraph embedding. *Information and Software Technology*, 158:107168, 2023. https://doi.org/10.1016/j.infsof.2023.107168.

[9] Flawfinder. https://dwheeler.com/flawfinder/.

[10] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018. https://doi.org/10.1109/SP.2018.00040.

[11] Wenbo Guo, Yong Fang, Cheng Huang, Haoran Ou, Chun Lin, and Yongyan Guo. HyVulDect: A hybrid semantic vulnerability mining system based on graph neural network. *Computers & Security*, page 102823, 2022. https://doi.org/10.1016/j.cose.2022.102823.

[12] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020. https://dl.acm.org/doi/abs/10.5555/3495724.3496298.

[13] Hyosoon Jang, Seonghyun Park, Sangwoo Mo, and Sungsoo Ahn. Diffusion probabilistic models for structured node classification. In *ICML 2023 Workshop on Structured Probabilistic Inference {\&} Generative Modeling*, 2023. https://openreview.net/forum?id=CxUuCydMDU.

[14] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *stat*, 1050:1, 2014. https://hdl.handle.net/11245/1.434281.

[15] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. VulDeeLocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2821–2837, 2021. https://doi.org/10.1109/TDSC.2021.3076142.

[16] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*, pages 201–213, 2016. https://doi.org/10.1145/2991079.2991102.

[17] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021. https://doi.org/10.1109/TDSC.2021.3051525.

[18] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018. https://doi.org/10.14722/ndss.2018.23158.

[19] Rahma Mahmood and Qusay H Mahmoud. Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code. *arXiv preprint arXiv:1805.09040*, 2018. http://arxiv.org/abs/1805.09040.

[20] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. VulChecker: Graph-based vulnerability localization in source code. In *31st USENIX Security Symposium, Security 2022*, 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky.

[21] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. ReGVD: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 178–182, 2022. https://doi.org/10.1145/3510454.3516865.

[22] Alexander Quinn Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models. In *International Conference on Machine Learning*, pages 8162–8171. PMLR, 2021. https://proceedings.mlr.press/v139/nichol21a.html.

[23] Alexander Quinn Nichol, Prafulla Dhariwal, Aditya Ramesh, Pranav Shyam, Pamela Mishkin, Bob Mcgrew, Ilya Sutskever, and Mark Chen. GLIDE: Towards photorealistic image generation and editing with text-guided diffusion models. In *International Conference on Machine Learning*, pages 16784–16804. PMLR, 2022. https://proceedings.mlr.press/v162/nichol22a.html.

[24] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, 150:22–36, 2019. https://doi.org/10.1016/j.jss.2018.12.001.

[25] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. DeepLineDP: Towards a deep learning approach for line-level defect prediction. *IEEE Transactions on Software Engineering*, 49(1):84–98, 2022. https://doi.org/10.1109/TSE.2022.3144348.

[26] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018. https://doi.org/10.1109/ICMLA.2018.00120.

[27] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International conference on machine learning*, pages 2256–2265. PMLR, 2015. https://dl.acm.org/doi/abs/10.5555/3045118.3045358.

[28] Hao Sun, Lei Cui, Lun Li, Zhenquan Ding, Zhiyu Hao, Jiancong Cui, and Peng Liu. VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. *Computers & Security*, 110:102417, 2021. https://doi.org/10.1016/j.cose.2021.102417.

[29] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. Search-driven string constraint solving for vulnerability detection. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 198–208. IEEE, 2017. https://doi.org/10.1109/ICSE.2017.26.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. https://dl.acm.org/doi/abs/10.5555/3295222.3295349.

[31] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267. IEEE, 2000. https://doi.org/10.1109/ACSAC.2000.898880.

[32] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2020. https://doi.org/10.1109/TIFS.2020.3044773.

[33] Wenhan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. Modular tree network for source code representation learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–23, 2020. https://doi.org/10.1145/3409331.

[34] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering*, 48(5):1480–1496, 2020. https://doi.org/10.1109/TSE.2020.3023177.

[35] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. VulCNN: An image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2365–2376, 2022. https://doi.org/10.1145/3510003.3510229.

[36] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo,

Table 9: The most common eight node types and the average number of nodes of each type in each function.

ExpressionStatement	Condition	CFGEntryNode	IdentifierDeclStatement	Statement	Parameter	ReturnStatement	IncDecOp
82	51	35	20	19	11	4	4

Wei Zou, et al. MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020. https://dl.acm.org/doi/abs/10.5555/3489212.3489278.

[37] Peiyu Yu, Sirui Xie, Xiaojian Ma, Baoxiong Jia, Bo Pang, Ruiqi Gao, Yixin Zhu, Song-Chun Zhu, and Ying Nian Wu. Latent diffusion energy-based model for interpretable text modelling. In *International Conference on Machine Learning*, pages 25702–25720. PMLR, 2022. https://par.nsf.gov/biblio/10351401.

[38] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 111–120. IEEE, 2021. https://doi.org/10.1109/ICSE-SEIP52600.2021.00020.

[39] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019. https://doi.org/10.48550/arXiv.1909.03496.

Appendix

A Improving Model Training Speed

We note that $y_i^{(t)}$ can be sampled from the original data $y_i^{(0)}$. Thus, in the actual training process, we do not need to train the model on all timesteps. By directly sampling to timestep t, we can obtain the $y_i^{(t)}$ at that timestep and use the GAT to predict the injected noise. Specifically, we sample T' timesteps from T diffusion steps. In our experiments, the value of T' is 16. This method can significantly improve the training speed of the model. Its training process is described as: 1) Randomly sample a node label $y_i^{(0)}$ from the initial data distribution $q(y)$. 2) Randomly sample a timestep t from 1 to T, to represent the level of injected Gaussian noise. 3) Randomly sample a t-level Gaussian noise Z_t, and then inject Z_t into $y_i^{(0)}$ using Equation (3). 4) Train the GAT model to predict the noise acting on $y_i^{(0)}$ based on $y_i^{(t)}$.

B Details of Data Collection and Labeling

Our dataset comprises two parts: (1) NVD and SARD. NVD focuses on 19 popular C/C++ open-source software projects (as in [18]), providing corresponding vulnerable code files and patched code files. SARD is essentially a synthetic dataset with a smaller complexity of vulnerable samples compared to those present in real-world scenarios.[5] As vulnerable functions and statements are already labeled in these two datasets, we only need to construct labels for GrVCs and its nodes. (2) Three open-source projects (i.e., OpenSSL, Libav, and Linux Kernel). These projects are popular among developers and offer diverse functionalities. Specifically, we followed the data collection methodology in [38]. The dataset construction and labeling process entails the following steps:

- Commit message filtering. We identify vulnerability-related fixing commits by matching a set of predefined vulnerability keywords.
- Code preprocessing. For each vulnerability-related fixing commit, we extract vulnerable code file, patched code file, and `diff` file. Then we extract functions and statements involving changes from the `diff` file.
- Automatic data labeling. If a function from a vulnerable code file has at least one statement that was deleted or changed (indicated by "-" in the `diff` file), it is labeled as vulnerable; otherwise, it is labeled as non-vulnerable. Deleted or changed statements are considered vulnerable statements.
- Manual verification. We conduct manual verification on a small proportion of samples to ensure the accuracy and reliability of the annotations.

C Analysis of the Various Components of CJG

In this experiment, we progressively added data-flow edges, function-call edges, and code-sequence edges to CFG. Then, we extracted GrVCs based on graph structures at different stages and applied our DPM-based detection model for training and testing. As shown in Table 10, overall, the model's performance gradually improved as we added different types of dependency edges to the CFG. Notably, the model's performance with `CFG+DF` significantly surpassed that of the `CFG`, highlighting the substantial contribution of data flow to extracting vulnerability features. However, the inclusion

[5] https://samate.nist.gov/SARD

Table 10: Contributions of different edge types in Code Joint Graph (metrics unit: %).

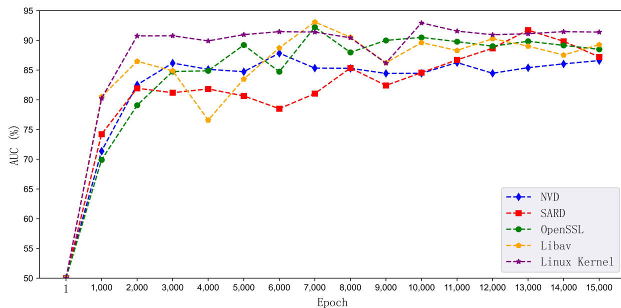
Code representation	Vulnerability Identification				Vulnerability Localization				
	F1	R	AUC	MCC	F1	R	AUC	MCC	IoU
CFG	82.45	76.33	82.72	60.03	71.81	55.97	82.68	72.17	60.76
CFG+DF	82.69	79.22	84.73	69.16	79.29	77.90	88.91	79.22	61.14
CFG+DF+CG	82.74	80.02	85.02	69.10	78.95	78.94	89.41	78.88	61.77
CFG+DF+CG+CS (CJG)	85.28	82.28	85.91	70.51	79.60	77.15	88.53	79.55	64.90

Table 11: Performance comparison of FVD-DPM on original and augmented datasets (metrics unit: %).

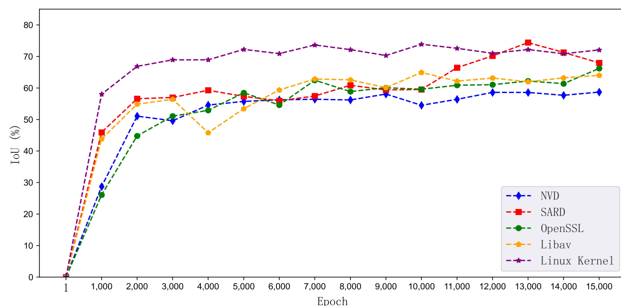
Dataset	Vulnerability Identification				Vulnerability Localization				
	F1	R	AUC	MCC	F1	R	AUC	MCC	IoU
Original	78.63	78.54	83.79	67.61	77.21	75.17	87.54	77.14	59.04
Augmented	81.52	79.63	86.54	74.15	84.07	78.82	89.37	84.18	60.14

of inter-procedural analysis in `CFG+DF+CG` did not yield a significant improvement in detection performance. One possible reason for this is the small proportion of functions with inter-procedural dependencies in our dataset (less than 1/4 of the total functions), which may restrict the model's ability to benefit from inter-procedural analysis. We provide further analysis of this phenomenon in Appendix E.

D Results under Diverse Training Epochs



(a) AUC



(b) IoU

Figure 7: localization results of FVD-DPM on the test set under diverse training epochs.

Figure 7 illuminates the dynamic evolution of model metrics across various epochs for each dataset. Notably, the scores for all metrics exhibit a pronounced and swift upsurge during the initial 2000 epochs. Subsequently, within the epoch range of 2000 to 7000, the scores of all metrics show a fluctuating and slow growth trend. Beyond the 7000-epoch mark, a noteworthy observation is the model's gradual convergence. This detailed analysis enhances our understanding of the temporal dynamics governing the model's localization capabilities, offering valuable insights into its evolving behavior throughout the training process.

E Effectiveness of Inter-procedural Analysis

To study the impact of inter-procedural analysis on FVD-DPM in vulnerability detection, we augmented the NVD dataset with new functions featuring inter-procedural dependencies and retrained FVD-DPM on the augmented dataset. Due to the challenge of re-collecting real-world functions with inter-procedural dependencies, we utilized data augmentation techniques to create new functions. Specifically, we employed multiple code refactoring methods to generate transformed code that retained the same labels as the original function. We used a total of 14 types of code refactoring methods, including adding arguments, renaming methods, renaming local variables, and adding print statements. Through this data augmentation, we added 120 new functions with inter-procedural dependencies to the NVD dataset.

We retrained and tested FVD-DPM on the augmented dataset and compared the results with those from the original dataset. As shown in Table 11, FVD-DPM achieves better performance on the augmented dataset. This demonstrates that by adding more functions with inter-procedural dependencies, FVD-DPM can capture more comprehensive vulnerability semantics. Therefore, function-call edges included in CJG are effective for FVD-DPM in detecting vulnerabilities.