Enhancing Automated Vulnerability Repair through Dependency Embedding and Pattern Store

Qingao Dong, Yuanzhang Lin, Hailong Sun, Xiang Gao†
Beihang University
{dongqa930, yuanzhanglin, sunhl, xiang_gao}@buaa.edu.cn

***Abstract*—In recent years, the proliferation of software vulnerabilities has significantly increased the complexities and costs associated with manual remediation efforts. Although AI-based methods for automated vulnerability repair are gaining traction, many existing approaches have two limitations: 1) treat code as a sequence of tokens, neglecting critical structural information like control flow and data flow, and 2) do not fully utilize the repair patterns of vulnerabilities. To address these limitations, we introduce FAVOR, an innovative tool that utilizes both the vulnerable function's code and its control flow graph (CFG) as inputs. FAVOR incorporates a dependency embedding module to capture structural and dependency information and leverages CodeT5, a state-of-the-art model pre-trained for code generation tasks. To further enhance the repair process, we introduce a pattern store that uses KNN search to retrieve similar past repair patterns, which helps guide the model toward generating more contextually accurate patches. In our experiments, FAVOR, trained on a dataset of 6548 faulty C/C++ functions, repaired 45 more vulnerabilities compared to VULREPAIR, demonstrating improved accuracy and efficiency in automated vulnerability repair.**

***Index Terms*—Software Vulnerability, Automatic Program Repair, Dependency Embedding, Automatic Vulnerability Repair**

I. INTRODUCTION

Software vulnerabilities, such as the prominent log4j vulnerability CVE-2021-44228, have emerged as a critical concern, posing substantial security risks and financial implications. According to the National Vulnerability Database (NVD) Dashboard, the Common Vulnerabilities and Exposures (CVEs) database includes 264,460 reported vulnerabilities, with 29,066 new CVEs recorded throughout 2023 [1], marking a significant increase over the 25,081 CVEs published in 2022 [2], [3]. This trend underscores the escalating scope of software security threats. The increasing volume and complexity of these vulnerabilities have rendered manual detection and remediation efforts both challenging and costly.

In response to these challenges, neural network-based methods have shown promising outcomes in automatic vulnerability detection [4], [5], [6], [7], [8], [9]. However, these advancements are insufficient for completing reassurance. Detecting new vulnerabilities is crucial, but addressing them effectively to mitigate potential software system risks is even more important and resource-intensive. According to the statistics of Edgescan reports [10], "Mean time to remediation for critical severity vulnerabilities is 65 days", leaving programs in the risk of being attacked. Manual vulnerability resolution is time-consuming and requires specialized expertise. Consequently, there is an increasing demand for automatic vulnerability repair tools [11], [12], [13].

AI-based methodologies, exemplified by neural network-powered solutions like VREPAIR [14] and VULREPAIR [15], have been developed to automatic vulnerability repair. Specifically, VREPAIR delivers the automated vulnerability repair (AVR) task into two phases (1) pre-training a vanilla-transformer language model in a large buggy program corpus, and (2) fine-tuning the pretrained model in the target vulnerability datasets, including CVEfixes [16] and BigVul [17]. Specifically, VULREPAIR relies on the whole faulty source code as CodeT5's [18] input to generate the fixed pair function. Although both approaches have achieved promising results in automatically generating vulnerability patches, they have two main shortcomings:

- **Overlooking structural information of the code:** Existing tools, such as VREPAIR and VULREPAIR, treat source code as a flat token sequence akin to natural language, overlooking the rich structural and semantic information inherent in programming languages — such as control flow branches and loops. While VULMASTER ultilizes Abstract Syntax Tree (AST) information, which offers some syntactical insights into the vulnerable code, it fails to capture critical elements of the program's control flow and data flow. Although some works [19] have utilized Call Flow Graph (CFG) for vulnerability detection, they have excessively abstracted node information. When their approaches are applied to the vulnerability repair domain, the lack of detailed node information hinders the effective generation of patches.
- **Limited Use of Learned Repair Patterns.** Current works primarily rely on models from other domains, such as CodeBert [20] or CodeT5 [18], which are trained on large datasets (e.g., text datasets). However, vulnerability datasets are often relatively small in size, with the classic dataset, CVEfixes-BigVul [17], [16] used in previous work [15], [14] containing only 5,800 distinct samples. Fine-tuning on such limited datasets makes it difficult for models to effectively catch the characteristics of vulnerabilities and generate accurate repair patches. Since many similar vulnerabilities share similar repair patches, these patterns could be leveraged to inform new patch generation. However, existing approaches

†Corresponding author.

have not considered how to utilize repair patterns from previous patches to aid in generating new ones.

To address these two shortcomings, we propose FAVOR, based on the following two key ideas: 1) Embedding dependency information helps the pre-trained model better understand the semantic structure and root causes of vulnerable functions, and 2) inspired by data stores [21], [22], we utilize a pattern store to retain patterns from the training set, which can effectively help adjust the model's output. For the first idea, we fine-tune CodeT5 model in two phases. Specifically, the primary fine-tuning enable CodeT5 to learn vulnerability and its repair features. while the secondary fine-tuning to enable the model to learn program dependency features. Specifically, FAVOR first embeds the flow information from the CFG, the node details from the CFG, and the vulnerable function itself. These three components are then used as inputs to the dependency-aware model, which ultimately generates the patch. This embedding method enables a deeper comprehension of node characteristics within vulnerable code. For the second idea, drawing inspiration from the concept of a data store, FAVOR utilizes the dependency-aware model and the vulnerability dataset to construct the pattern store, and then leverages the pattern store to adjust and refine the output of the dependency-aware model. Specifically, FAVOR obtains the penultimate decoder layer's hidden state as the keys. FAVOR then passes the patches through our dependency model, using the target patch token indices as the values. These key-value pairs will assist in adjusting the patch generation.

To measure the effectiveness of FAVOR in generating vulnerability patches, we evaluate it using a substantial real-world C/C++ vulnerability dataset, BigVul [17], which is previously employed in both VREPAIR and VULREPAIR studies. This dataset encompasses 9,333 distinct vulnerability resolutions, with a temporal coverage extending from 2002 to 2019. We compared FAVOR with the state-of-the-art tool VULREPAIR and GPT-4o. Through experiments, we found that our proposed FAVOR correctly fixes 45 more vulnerabilities compared to VULREPAIR, and 239 more than GPT-4o. Further analysis revealed that our dependency embedding method and pattern store contributed significantly to the improvement, allowing us to repair an additional *36* and *9* vulnerabilities, respectively.

Our contributions are summarized in the following points:

- We proposed a dependency embedding strategy to capture code dependencies and trained a dependency-aware model.
- We designed a pattern store strategy, which enhances patch generation accuracy by saving patterns from the existing dataset and adjusting the output during inference.
- We implemented these strategies in the tool FAVOR, which is publicly available[1].
- We compared FAVOR with existing approaches, and the experiments demonstrated that FAVOR improved vulnerability repair accuracy from 18.81% to 21.20%.

[1]This work is available at https://github.com/qadong/SANER25FAVOR.

II. PRELIMINARIES AND MOTIVATION

In this section, we will introduce the background knowledge of NMT-based automated vulnerability repair, CodeT5, and graph embedding with GNN, which will help in understanding the content of the paper.

A. NMT-based Automated Vulnerability Repair

The task of automated vulnerability repair aims to find a feasible patch that fixes existing vulnerabilities in programs. With the development of deep learning in recent years, an increasing number of works [15], [14], [23] are adopting Neural Machine Translation (NMT) for vulnerability repair. According to previous work [15], [14], AVR tools usually treat repair tasks as Neural Machine Translation task. They [15], [14] employ a neural network as the generation engine, using vulnerable code snippets $Snip_i^{vul}$ and CWE types as inputs, and ultimately outputting the corresponding vulnerability repair patches for the programs. The mapping that the model needs to learn can be expressed as follow:

$$f : (Snip_i^{vul}, \text{CWE}) \rightarrow Patch_i$$

However, there exist some major technical limitations. On the one hand, existing work [15], [14] simply takes code snippets and CWE types as inputs, without incorporating dependency information, which limits the model's ability to effectively understand the code's structure and the true root-cause of the vulnerabilities. On the other hand, without referencing similar historical repair patterns, the model may struggle to generate accurate patches, as it lacks the ability to leverage past repair knowledge, making it harder to handle complex vulnerabilities effectively.

Compared to existing work, we have introduced a dependency embedding module that incorporates function-level dependency information related to vulnerabilities as part of the input. This enables the model to capture intra-function relationships, allowing a deeper understanding of the structural and data flow context within the vulnerable function. Additionally, we introduce a pattern store to reference similar historical repair patterns, further enhancing the model's ability to generate more accurate patches based on past repair knowledge. By doing so, the model can more effectively identify and address the root causes of vulnerabilities, leading to more precise repairs.

B. CodeT5

CodeT5 [18] is built on a Text-To-Text Transfer Transformer (T5) architecture, consisting of 6 encoder blocks and 6 decoder blocks that are sequentially connected. Let the input sequence be represented as $X = \{x_1, x_2, \ldots, x_n\}$, where each x_i represents a token in the input (e.g., a code snippet or textual description).

Encoder: The encoder consists of 6 sequentially stacked layers (or blocks). Each encoder block applies a multi-head self-attention mechanism and a feed-forward network (FFN). Let h_E^l represent the hidden state at the l-th encoder layer, where $l = 1, \ldots, 6$.

The hidden state for the first encoder layer is initialized as:

$$h_E^0 = X \tag{1}$$

For each subsequent layer, the hidden state is updated as follows:

$$h_E^l = \mathit{FFN}\left(\mathit{SelfAttention}(h_E^{l-1})\right), \quad l = 1, \ldots, 6 \tag{2}$$

This process progressively refines the input representation, capturing increasingly detailed syntactic and semantic features.

Decoder: The decoder also consists of 6 sequentially stacked layers. The decoder generates the output sequence $Y = \{y_1, y_2, \ldots, y_m\}$ token-by-token. Let h_D^l represent the hidden state at the l-th decoder layer.

The input of the first decoder layer at step t is the combine of the encoder's outputs and the generated tokens at $t-1$ step from the decoder:

$$h_D^0 = y_{\text{prev}} + h_E^6 \tag{3}$$

For each subsequent decoder layer, the hidden state is updated as:

$$h_D^l = \mathit{FFN}\left(\mathit{CrossAttention}(h_D^{l-1}, h_E^6)\right), \quad l = 1, \ldots, 6 \tag{4}$$

CrossAttention refers to the attention mechanism that allows the decoder to attend to both the previously generated tokens and the encoded input from the encoder. The final output y_t at each step t is generated based on the decoder's hidden state h_D^6. Specifically, the decoder predicts the next token y_t by:

$$y_t = \arg\max\left(\mathit{softmax}(W h_D^6 + b)\right) \tag{5}$$

where W and b are learnable parameters, and h_D^6 is the hidden state from the final decoder block at time step t. The process continues until the entire sequence is generated.

C. Graph embedding with GNN

Graph embedding using Graph Neural Networks (GNNs) involves learning low-dimensional vector representations for nodes, edges, or entire graphs while preserving the structural and relational information of the original graph. GNNs operate on graph-structured data, where the core process relies on message passing and aggregation [24], allowing each node to update its representation by incorporating information from its neighbors.

Formally, for a graph $G = (V, E)$, where V represents the set of nodes and E the set of edges, the hidden state of a node v at layer l, denoted as $h_v^{(l)}$, is updated based on its neighbors $\mathcal{N}(v)$. The node embedding update can be expressed as:

$$h_v^{(l)} = \mathit{UPDATE}\left(h_v^{(l-1)}, \mathit{AGGREGATE}\left(\{h_u^{(l-1)} : u \in \mathcal{N}(v)\}\right)\right) \tag{6}$$

Where:

- *AGGREGATE* is a function that gathers information from the neighboring nodes $u \in \mathcal{N}(v)$.
- *UPDATE* is a function that updates the hidden state of the current node using the aggregated information and its previous hidden state $h_v^{(l-1)}$.

```
1  bool initiate_stratum(struct pool *pool){
2      ...
3      n2size = json_integer_value(json_array_get(
           res_val, 2));
4  -   if (!n2size){
5  +   if(n2size < 1)  // patch generated by FAVOR
6  +   if(n2size > 0)  // patch generated by VulRepair
7          applog(LOG_INFO, "Failed to get n2size in
               initiate_stratum");
8          free(sessionid);
9          free(nonce1);
10         goto out;
11     }
12     ...
13     pool->n2size = n2size;
14     ...
15 out:
16     if (val)
17         json_decref(val);
18         Dereference JSON object
19     ...
20 }
```

Fig. 1: The repair results of a sample of CVE-2014-4502 by FAVOR and VULREPAIR.

Following DeepDFA [19], in this work, we implement our approach using a Gated Graph Sequence Neural Network (GGNN) [25], where *AGGREGATE* is a Multi-Layer Perceptron (MLP) and *UPDATE* is a Gated Recurrent Unit (GRU).

D. Motivating Example

Figure 1 shows the patch generated for fixing CVE-2014-4502 [2]. This CVE causes a heap-based buffer overflow in the function *initiate_stratum* of *bfgminer* project (a blockchain resource miner) before version 4.2.2, allowing remote pool servers to have unspecified impact via a negative value in the *Extranonc2_size* parameter. *Extranonc2_size* corresponds to variable *n2size* in line 3 of the code snippet.

The vulnerability happens because developers only terminated the program execution (line 8-10) when *n2size* is 0, but failed to check whether *n2size* is negative.

To solve this problem, developers changed condition *if(!n2size)* to *if(n2size < 1)* in commit [3], and it is marked with the rationale that *"if the size of extranonce2 is negative, it could lead to the allocation of too little memory, potentially causing subsequent security vulnerabilities or program crashes"*.

To manually repair this segment of code for developers, there are two important steps: 1) **Understanding the dependency information.** From the data flow perspective, the value of n2size is obtained through *json_integer_value(json_array_get(res_val, 2))* from an external JSON file. This means that n2size depends on external input, which may lead to unexpected values. Moreover, n2size is subsequently assigned to *pool → n2size*, which is used for further allocation and creation of the pool. If n2size has an improper value, it could result in serious issues such as buffer overflow. From the control flow perspective, when executing the *if (!n2size)* condition, it means that if n2size is 0 or

[2]https://nvd.nist.gov/vuln/detail/CVE-2014-4502
[3]https://github.com/luke-jr/bfgminer/commit/ff7f3012

not properly initialized, the function will enter the error-handling process, release resources, and exit. This control flow is designed to ensure that n2size is valid before proceeding with the subsequent logic, preventing invalid data from causing instability or security risks in the system. Therefore, developers should understand that this if condition serves as a safeguard to prevent improper data from continuing through the workflow.

2) **Referencing historical repair patterns for similar vulnerabilities.** By leveraging repair patterns from similar vulnerabilities, developers can effectively address issues such as the n2size variable. These types of fixes typically involve implementing safety checks for critical variables, such as null value or boundary condition validation. In this case, adding a validity check for n2size ensures that the variable is safely used in subsequent operations. These common repair patterns help developers reduce trial-and-error, quickly apply proven repair strategies, and improve the accuracy and efficiency of patches.

Existing AVR approaches fail to fix this vulnerability, for instance, VULREPAIR generates a wrong patch that modifies *!n2size* to *n2size > 0*). Since VULREPAIR cannot identify the control flow and characteristics of the vulnerability, it can only perform repairs based on the labeled vulnerable line (line 4) and generates a generalized type of fix.

III. APPROACH

The overall architecture of FAVOR is shown in Figure 2. More specifically, FAVOR can be divided into the following four phases:

Primary Fine-Tuning: For pre-trained model CodeT5, given a set of vulnerabilities S and its corresponding patch set P, we fine-tune CodeT5 to better understand the vulnerable code and extract repair-relevant features from the vulnerability code.

Dependency-Aware Model Adaption: At this stage, we train a model capable of understanding both the source code information and its dependency information. Specifically, we divide the embedding component into two parts: source code embedding and dependency embedding (including graph and node embedding), and combine these embedding vectors to provide inputs to the decoder. For the source code, we continue to employ the fine-tuned encoder from CodeT5 for embedding. To embed the program dependencies, we use Gated Graph Sequence Neural Network (GGNN), as it has been proven to be effective [19] in learning program semantics from both data flow graph (DFG) and control flow graphs (CFG). To capture CFG node information such as, node types, operations, we utilize the fine-tuned encoder from CodeT5 along with a pooling linear for embedding.

To improve the model's overall performance in capturing and utilizing dependency information, we employ second-stage fine-tune: Different from the first-stage fine-tune, the second-stage fine-tune is to adapt the model to the dependency information that CodeT5 has not previously encountered, we freeze the encoder and focus on training the decoder.

Pattern Store Construction: To enable the model to reference historical repair patterns during the patch generation phase, thereby adjusting the output token probability distribution and improving the accuracy of the repair, we construct a pattern store that maps vulnerability function patterns to corresponding patch token indices.

Patch Generation: In the inference phase, FAVOR extracts CFG and node information, which, along with the vulnerability function, will serve as input to the dependency-aware adapted model. The output from the penultimate decoder layer h_D^5 is then used to query the pattern store for stored values, and these values are then transformed into the token distribution of the corresponding pattern to adjust the model's output distribution. Based on the final token distribution, the generated repair function is obtained.

A. Primary Fine-Tuning

In alignment with prior research, we select CodeT5 as our encoder-decoder framework due to its demonstrated proficiency and advancements in code comprehension and generation tasks [26]. CodeT5's architecture follows a unified text-to-text approach, enabling the model to handle a wide range of programming languages and software engineering tasks with remarkable flexibility. Its pre-training on a diverse code corpus allows it to generate both syntactically and semantically accurate code, making it particularly well-suited for our task, where capturing both textual features and dependency information is critical for vulnerability repair.

To enable CodeT5 to better capture the repair-relevant features of vulnerable functions, at this stage, we begin by inputting the vulnerable function into the model and fine-tuning it to generate the correct patch. Specifically, for a given vulnerable function s_i and patch p_i in the training set, the model learns the mapping by minimizing the cross-entropy loss between the model's prediction and the target patch. This is formulated as an optimization problem, where the model's parameters θ are updated to minimize the average loss over the training examples:

$$\theta' = \underset{\theta}{\operatorname{argmin}} \frac{1}{M} \sum_{j=1}^{M} \mathit{CrossEntropy}(f(s_i;\theta), p_i)$$

B. Model Adaption for Understanding Dependency

We first present our strategy to embed source code and its dependency information, and then our secondary model fine-tuning to enable the model to understand code dependencies.

*1) **Source Code Embedding**:* In this phase, we focus on embedding the vulnerable source code s_i of the vulnerable function. Using the fine-tuned CodeT5 encoder, we transform the source code s_i into a high-dimensional embedding vector representation $\mathbf{E}^{\text{code}}$. This vector captures the syntactic and semantic details of the code. The embedded representation is crucial as it forms the foundation for capturing the repair-relevant context.

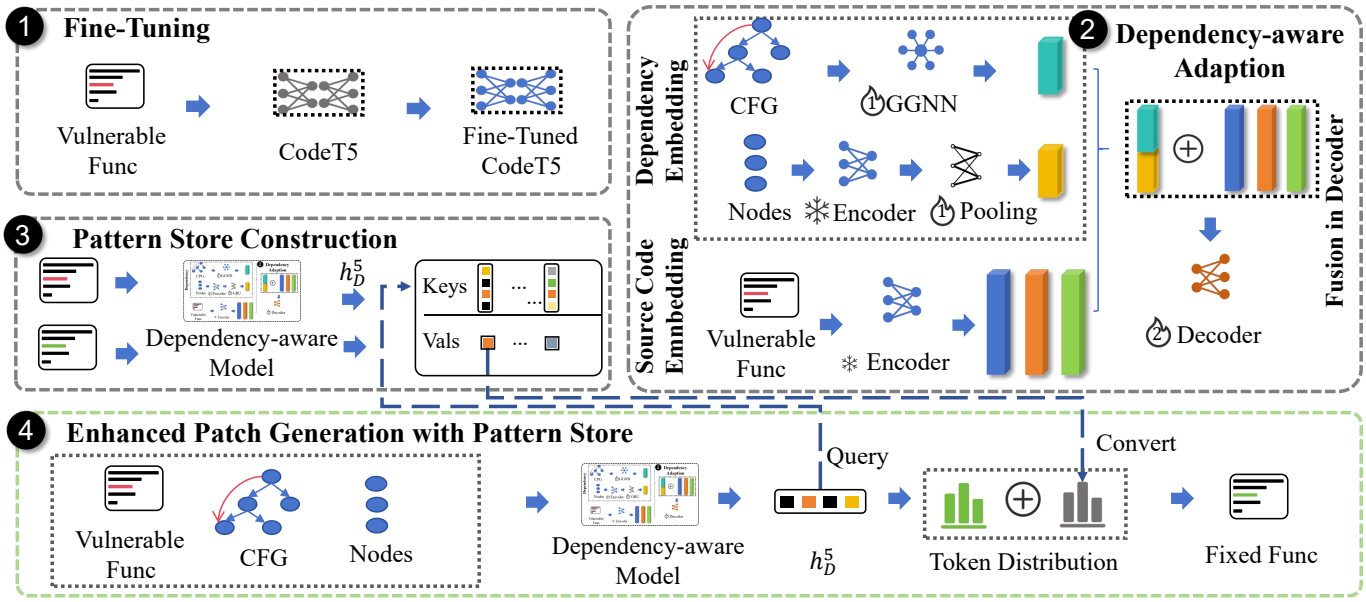

Fig. 2: The overall architecture of FAVOR. 1) Fine-tune CodeT5 using a vulnerability dataset. 2) Secondary fine-tuning produces a dependency-aware model. 3) Use the dependency-aware model and the vulnerability dataset to obtain the pattern store. 4) Use the pattern store to adjust the output of the dependency-aware model.

*2) **Dependency Embedding**:* The Dependency embedding consists of two parts: graph embedding and nodes embedding. The graph embedding denoted as $\mathbf{E}^{\text{graph}}$ is used to integrate information from both the control flow and data flow, while the nodes embedding denoted as $\mathbf{E}^{\text{node}}$ is responsible for incorporating the information from the CFG nodes (operations). This approach ensures that each node not only captures abstract data flow and control flow information but also retains specific textual details, such as API calls and variable names.

Graph Embedding. To obtain control flow and data flow, FAVOR first uses Joern [27] to extract the corresponding Control Flow Graph (CFG) for each function. This comprehensive structure enables a detailed analysis of both syntactic and semantic dependencies within the code, facilitating a deeper understanding of vulnerabilities.

To specifically capture both control flow and data flow, we utilize an abstract data flow encoding method inspired by DeepDFA [19]. FAVOR abstracts useless program details and focuses only on four key properties derived from DeepDFA: 1) API calls (e.g., *malloc*), 2) Data types (e.g., *int*, *char**), 3) Constants (e.g., *NULL*, -1), and 4) Operators (e.g., +, -). These key properties are then mapped to a fixed-size hot-vector representation, allowing for scalable and efficient analysis of data flow and control flow patterns. We then apply graph learning using a Gated Graph Neural Network (GGNN) [25] to learn the graph representation. The GGNN propagates both data flow and control flow information through the graph using a message-passing mechanism, where an MLP [28] aggregates information from neighboring nodes and a GRU [29] updates the node states.

The GGNN embedding, denoted as $\mathbf{E}^{\text{graph}}$, primarily focuses on capturing abstract patterns in the graph, such as the overall data flow and control flow, which aids in understanding the structural dependencies and root causes of vulnerabilities. However, this abstract approach may overlook the detailed textual context required for generation tasks, such as CFG nodes types, specific variable names, API calls, and function-level details.

Nodes Embedding. To address the limitations of abstract data flow embeddings and further enrich the graph representation, we additionally use the encoder from the fine-tuned CodeT5 model, as mentioned in Phase 1, to capture the textual features of each node. The fine-tuned CodeT5 model is specifically chosen to better extract repair-relevant features from the CFG node's textual information and ensure alignment in the vector representation space for subsequent stages. The node embeding, denoted as $\mathbf{E}^{\text{node}}$, is then concatenated with the GGNN-generated vectors $\mathbf{E}^{\text{graph}}$ at the hidden dimension (hid_dim) level to form the complete dependency embedding $\mathbf{E}^{\text{d}}$, i.e., $\mathbf{E}^{\text{d}} = \left[\mathbf{E}^{\text{graph}} \parallel \mathbf{E}^{\text{node}}\right]$.

*3) **Fusion in Decoder**:* To enable CodeT5 to simultaneously capture both the source code information of the vulnerability and its dependency information, following the idea of Fusion-in-Decoder, we concatenate $\mathbf{E}^{\text{code}}$ with $\mathbf{E}^{\text{d}}$ before passing them to the decoder, obtaining the final input vector $\mathbf{E}$ for the decoder, i.e., $\mathbf{E} = \left[\mathbf{E}^{\text{code}}; \mathbf{E}^{\text{d}}\right]$.

*4) **Secondary fine-tuning**:* Through the Dependency Embedding, we obtained the vector representation of the dependency $\mathbf{E}^{\text{d}}$. However, CodeT5 alone is not well-suited to learning relevant repair features from this CFG representation. Therefore, in this phase, we apply a secondary fine-tuning process specifically aimed at enabling CodeT5 to effectively

use the dependency information.

To achieve this, we employ a stagewise training strategy. This strategy ensures that each module learns its specific task independently, allowing them to focus on different objectives at each stage, reducing interference between modules, and enhancing the overall performance. Such a stagewise approach has been successfully used in various works [30], [31].

Specifically, the secondary fine-tuning process consists of two stages. The first stage aims to gradually adjust the GGNN model's parameters to improve the embedded representation of the function dependency. The second stage focuses on fine-tuning the decoder parameters, enabling it to generate more accurate patches.

In the first stage, since the vector space obtained from the dependency embedding may differ from the one obtained from the source code embedding, it is necessary to align the vector representations of GGNN and CodeT5. This alignment ensures that the information from the graph and the textual data are compatible for subsequent processing. To achieve this, we introduced additional $\mathbf{X}$ epochs adaptation phase. In this stage, we freeze the encoder and decoder of CodeT5 and train the parameters of the GGNN and Pooling Linear.

In second stage, we freeze the encoder, and train the parameters of the decoder. The graph vector, representing data flow and control flow information, enriches the model's understanding of the structural dependencies within the function. By combining this with the textual features learned by CodeT5, the model gains a more comprehensive view of both the code's semantics and its underlying graph-based structure. During this phase, decoder parameters are updated to jointly optimize the use of textual and structural information. After $\mathbf{N}$ epochs of training, we obtain the final dependency-aware adapted model.

C. Pattern Store Construction

Traditional transformer-based automated vulnerability repair tools treat the repair process as an isolated generation task, without referencing any past repair experiences. This lack of historical repair memory often results in suboptimal repairs, as the model fails to draw from previously successful fixes for similar vulnerabilities.

To address this issue, our approach, inspired by the advantages of KNN-machine translation [21], [22], introduces a repair pattern store, denoted as $R = (\mathcal{K} : \mathcal{V})$, which references past repairs. This store acts as a repository of similar fixes, enabling the model to retrieve patterns from previous repairs. As a result, it significantly enhances the model's ability to generate accurate and contextually relevant patches.

Specifically, at this stage, we use the data from the training set as the primary source for building the repair pattern store. First, we freeze the parameters of dependency-aware adapted models and only perform forward propagation. FAVOR takes vulnerable code and its corresponding CFG as input, with the patch serving as the label, for the forward propagation process. At each time step t_i during sequence generation, we capture the output from the second-to-last decoder layer, h_D^5, to serve as the keys. Since in the second phase, we fuse the vulnerable code and CFG graph vector before feeding them into the decoder, and they are processed through the cross-attention mechanism of the initial decoder layers (as shown in Equation 4), h_D^5 encapsulates the highest-level vulnerability features. We define h_D^5 as the repair pattern. The label token indices at each time step are collected as the values, forming key-value pairs $\{k_t : v_t\}$. Once all the time steps are processed, the construction of the pattern store is complete.

D. Enhanced Patch Generation with Pattern Store

Given a vulnerable function s_i, FAVOR extracts its CFG G_i, and uses both as inputs, $x = (s_i, G_i)$,to the adapted model (as discussed in Section III-B).

During the patch generation phase, at each time step t_i, FAVOR captures the output from the penultimate decoder layer, h_D^5, as the query, and performs a K-Nearest Neighbors(KNN) search in the pattern store R, retrieving the K-nearest values.

The retrieved values are converted into a probability distribution over the vocabulary by applying a softmax function with temperature T to the negative distances, while aggregating multiple occurrences of the same vocabulary item. Using a temperature greater than one flattens the distribution, helping to prevent overfitting to the most similar retrievals.

This process can be formalized as follows:

$$p_{\text{KNN}}(y_i|x, \hat{y}_{1:i-1}) \propto \sum_{(k_j,v_j)\in\mathcal{R}} \mathbb{I}_{y_i=v_j} \exp\left(-\frac{d(k_j, f(x,\hat{y}_{1:i-1}))}{T}\right)$$

Where:

- i represents the i^{th} time step.
- y_i is the target token at time step i.
- x represents the input, which includes the vulnerable code and the corresponding CFG.
- $\hat{y}_{1:i-1}$ denotes the sequence of tokens generated up to time step $i-1$.
- $f(x, \hat{y}_{1:i-1})$ refers to the hidden state h_D^5, which represents the combination of the input x (vulnerable code and CFG) and the previously generated tokens $\hat{y}_{1:i-1}$. This function f is derived from the second-to-last decoder layer's internal representation at the current time step, capturing both the input context and the history of previously generated tokens.

Then, FAVOR obtains the probability $p_{\text{Model}}(y_i|x, \hat{y}_{1:i-1})$ from the adapted model. The model and KNN distributions are interpolated using a tuned parameter λ, resulting in the final vocabulary distribution:

$$p(y_i|x, \hat{y}_{1:i-1}) = \lambda\, p_{\text{kNN}}(y_i|x, \hat{y}_{1:i-1}) + (1-\lambda)\, p_{\text{Model}}(y_i|x, \hat{y}_{1:i-1})$$

Executing the above process until the adapted model generates the End-of-Sequence Token (EOS), we obtain the corresponding patch p_i.

IV. EXPERIMENT

We utilize these experiments to answer the following research questions:

- **RQ1. How is the effectiveness of FAVOR compared to state-of-the-art (SOTA) methodologies?**
  We compare our model FAVOR with the SOTA approaches, Vulrepair [15] and GPT-4o [32].
- **RQ2. How does each component of FAVOR affect the model performance?**
  We conduct a series ablation experiments to find out the influence of each part of FAVOR.
- **RQ3. What is the extent of the impact that hyperparameter settings have on the performance of our model?**
  We conduct a series experiments to find out the influence of the hyperparameter of FAVOR.

A. Dataset

We choose BigVul [17] as our evaluation dataset, since BigVul has been widely used in vulnerability detection and repair approaches, including VREPAIR and VULREPAIR. This dataset encompasses 10,900 C/C++ vulnerable code and patch pairs, with a temporal coverage extending from 2002 to 2019.

During our evaluation, we observe that BigVul has several issues. Specifically, just as previous work [15], [23], we also observe data leakage issues. Moreover, we also observe that the labeling in these datasets is inconsistent, with some samples lacking the paired <*S2SV_StartBug*> and <*S2SV_EndBug*> labels, we have made efforts to refine the dataset. To solve this problem, we refine the vulnerable function data from BigVul [17], and annotated the data based on the location of the vulnerability repairs. To simplify the annotation process, we use <*Vul_Start*> and <*Vul_End*> to mark the start and end positions of the vulnerability, as well as the positions of the patch. Although we focused only on C/C++ vulnerable functions, our approach is applicable to data in any programming language, as FAVOR's input is not dependent on any language-specific features.

From the BigVul dataset, we initially collected a total of 10,900 samples. After removing all duplicate vulnerable function samples, we retained 10,455 unique data points. We then used Joern to extract the Control Flow Graph (CFG) for each function. However, for some functions, Joern was unable to generate a CFG, so we discarded those cases, leaving us with 9,333 valid samples, denoted as BigVul*. We denote th Following VulRepair's methodology, we divided the data into training, validation, and test sets with a ratio of 7:1:2. The detailed statistics of the dataset are illustrated in Table I.

TABLE I: Statistics of Dataset

Dataset	Train	Valid	Test	Total
BigVul* [17]	6548	903	1882	9333

B. Baselines

We compare FAVOR with two baseline models, VULREPAIR [15] and GPT-4o [32]. VULREPAIR is the State-Of-The-Art (SOTA) approach utilizing CodeT5 in AVR, Whichs take a vulnerable function code concatenated with a CWE ID as input and generates the fixed code as output. We directly reuse the original setting of VULREPAIR. GPT-4o is the most well-known model proposed by OpenAI [33]. While, it stands at the forefront of AI-driven code generation and comprehension. It excels in creating functional code snippets and fully-fledged applications across various programming languages. With a deep understanding of code semantics, GPT-4o can debug, optimize, and explain code, acting as an invaluable assistant for developers. Its ability to translate complex coding tasks into executable code and provide real-time programming support marks a notable advancement in AI-assisted software development. We use a prompt to guide GPT-4o to generate fixed code.

The prompt we used is *"The code {code} is suffering from a vulnerability {name}, could you generate the fixed code to address this vulnerability?"* In the prompt, '*{code}*' denotes the vulnerable function, while '*{name}*' signifies the CWE identifier coupled with the CWE name, for instance, 'cwe-119: Improper Restriction of Operations within the Bounds of a Memory Buffer'.

Additionally, we did not compare our approach with VULMASTER, primarily because VULMASTER used a model adapted on an additional dataset of 560,000 bug-fixing instances. However, we were unable to obtain the corresponding training data and code snippet necessary to reproduce this phase. As a result, we opted not to include VULMASTER in our comparison.

C. Experimental Setting

Implementation details. Following previous work [15], [14], we only focus on fixing C/C++ vulnerabilities. However, the versatility of our approach lies in its language-agnostic design, enabling straightforward adaptation to additional programming languages such as Java and Python.

We conducted experiments on a server with an x86_64 architecture, equipped with an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz and an A100-SXM4 GPU. We used Joern version 1.1.1072 for our CFG extraction. We utillize CodeT5 as our encoder-decoder framework. We configure the training and validation batch sizes to 16. The input token limits for the path was uniformly set to 512. During the dependency embedding phase, we selected 200 as the maximum number of nodes, as our analysis showed that the average number of CFG nodes across all functions is approximately 200. Any vector exceeding this limit is truncated after embedding. The hyperparameter settings for each module are shown in the table below. Throughout the training phase, the AdamW optimizer was employed, with a learning rate set to 1e-4.

With these settings in place, our model is estimated to utilize approximately 25GB of VRAM during the training stage. To maintain consistency with VULREPAIR [15], we train our

TABLE II: Hyperparameter settings for each module in FAVOR

Module	*Hyperparameter*	*Setting Value*
Dependency-aware Encoder	Steps	5
	Maximum Nodes	200
Pattern Store	K	64
	λ	0.4
	Temperature	20

model for 30 epochs (20 epochs for primary fine-tuning, 5 epochs each for secondary fine-tuning **X** and **N**), evaluating it on the validation set after each epoch. The model that performs the best on the validation set is then designated as the final model.

Experimental Metrics. In accordance with established research protocols, our model evaluation employs the *%Perfect Prediction* metric. This metric precisely measures the accuracy with which an algorithm generates repairs for vulnerable functions that exactly match the human-generated ground truth. In addition, we also utilize the CodeBLEU [34] score, a specialized variant of the BLEU [35] score designed specifically for source code, which takes code structure into account in addition to lexical matching. We compare the pass@1 values directly to evaluate the accuracy of the repair generation for each vulnerable function in our testing dataset, rather than adopting a beam width strategy.

V. EXPERIMENTAL RESULTS

A. RQ1. How does the efficacy of our model surpass that of the current state-of-the-art (SOTA) methodologies?

Setup. We evaluate both baselines (VULREPAIR, GPT-4o) and our proposed tool, FAVOR, using the BigVul* dataset. Notably, FAVOR does not use any additional training data. Both the fine-tuning phase and the pattern store construction phase are solely based on the BigVul* dataset. The evaluation metrics include *%Perfect Prediction* and CodeBLEU, where higher scores indicate better performance.

To align with VULREPAIR, we adopted the same training schedule, consisting of 30 epochs. The first phase included 20 epochs for initial tuning, followed by a secondary tuning phase with 10 epochs, which were evenly split between the embedding module (5 epochs) and the CodeT5 decoder (5 epochs).

TABLE III: Evaluation of baselines and FAVOR

Approach	#Correct Patches	*%Perfect Prediction*	CodeBLEU
GPT-4o [32]	160	8.50	17.62
VULREPAIR (SOTA) [15]	354	18.81	33.48
FAVOR	**399**	**21.20**	**36.67**

Result. As shown in Table III, FAVOR achieves the best performance with 21.20 *%Perfect Prediction*, and 36.67 CodeBLEU, surpassing all baselines. Specifically, FAVOR repaired 45 more vulnerabilities compared to VULREPAIR. Additionally, FAVOR uniquely fixed 66 vulnerabilities that were not addressed by VULREPAIR. Furthermore, Table IV highlights FAVOR's Perfect Prediction numbers for the top 10 most dangerous CWEs. In total, FAVOR successfully repaired 71 vulnerabilities in the top 10 CWE ranks, representing 15.30% of the 464 samples.

TABLE IV: Numbers of Perfect Prediction for the Top-10 most dangerous CWEs

Rank	CWE-Type	Name	VULREPAIR	FAVOR	#Sample
1	CWE-787	Out-of-bounds Write	5	7	36
2	CWE-79	Cross-site Scripting	2	3	14
3	CWE-89	SQL Injection	0	0	0
4	CWE-416	Use After Free	5	10	60
5	CWE-78	OS Command Injection	0	0	1
6	CWE-20	Improper Input Validation	34	36	214
7	CWE-125	Out-of-bounds Read	12	15	136
8	CWE-22	Path Traversal	0	0	3
9	CWE-352	Cross-Site Request Forgery	0	0	0
10	CWE-434	Dangerous File Type	0	0	0
		TOTAL	58 (12.50%)	**71 (15.30%)**	464

The reason for GPT-4o's exceptionally low accuracy in vulnerability repair: Table III shows an exceptionally low accuracy in vulnerability repair, which does not meet the expectations for GPT-4o. Moreover, some previous works have reported similar issues in code summarization [36] and vulnerability repair [23] tasks. We believe the main reasons are as follows:

- **Limited Context Understanding**: GPT-4o requires extensive prompt engineering, which limits its ability to fully comprehend the context of vulnerable code and the complexity of code structures within a restricted input length.
- **Lack of Fine-Tuning for Code Repair**: GPT-4o is a closed-source model, and we cannot access its source code to further fine-tune it using the training set for vulnerability repair tasks, making it less effective in generating precise patch.

Answer to RQ1: FAVOR achieves the best performance (21.20 *%Perfect Prediction* and 36.67 CodeBLEU) among all baselines. Specifically, FAVOR repaired 45 more vulnerabilities than VULREPAIR. Additionally, among all the vulnerabilities repaired by FAVOR, 66 were not fixed by VULREPAIR.

B. RQ2. How does each component of FAVOR *affect the model performance?*

TABLE V: Ablation study on FAVOR

Type	Variants	*%Perfect Prediction*
Tool Designs	FAVOR	21.20 *(399)*
	-w/o Repair pattern store	20.72 *(390)*
Embedding Designs	Full embedding module	20.72 *(390)*
	-w/o Abstract Dataflow Embedding	19.76 *(372)*
	-w/o CodeT5 Nodes Embedding	19.23 *(362)*
	-w/o Dependency Embedding	18.81 *(354)*

Setup. We conduct a series of ablation experiments to determine the influence of each part of FAVOR as shown in Tabel V.Specifically, we divided the experiments into two groups.

1) The first group, labeled as the Tool Design group, focuses on exploring the impact of the pattern store on enhancing the patch generation capabilities of FAVOR. 2) The second group, labeled as the Embedding Design group, investigates the effects of the introduced dependency-embedding module on FAVOR's performance. Note that, to minimize the interaction between modules, the experiments in this group did not use the pattern store to enhance patch generation. **Result.** Accroding to Tabel V, in the Tool Design setup, the probability correction method using similar historical repair patterns fixed 9 more vulnerabilities than the approach without the pattern store, resulting in a 0.48% improvement. This suggests that the pattern store is effective to some extent.

In the Embedding Design group, the performance with only abstract data flow embedding was less than ideal, as expected. Although it captures CFG edge flows, it lacks node-level textual details crucial for vulnerability repair, leading to reduced effectiveness. Moreover, adding the nodes embedding improved performance, likely due to the retention of operation-related information, which provides helpful context for generating accurate patches.The best results were achieved when both embeddings were combined, showing a significant improvement of 1.91%.

> **Answer to RQ2:** The Dependency Embedding Module showed the most significant improvement, fixing 36 more vulnerabilities. The Pattern Store Module further enhanced the best-performing model by fixing an additional 9 vulnerabilities.

C. RQ3. What is the extent of the impact that hyperparameter settings have on the performance of our model?

Setup. In this RQ, we explore the impact of hyperparameters on the performance of FAVOR. Specifically, we divide the experiments into two groups: 1) The first group focuses on testing the hyperparameters of the dependency embedding module. To minimize interference from other modules, patch generation in this group is performed without enhancement from the pattern store. 2) The second group evaluates the effect of hyperparameters related to the pattern store. Similarly, to reduce the impact of other components, we use the dependency-aware adapted model that performed best on the validation set to apply the enhancement during patch generation.

TABLE VI: Hyperparameters study on dependency embedding module of FAVOR

Hyperparameter	Value	*%Perfect Prediction*	#Correct Patches
Steps	3	19.63	*369*
	4	19.93	*375*
	5	20.72	*390*
Maximum Nodes	50	19.77	*372*
	200	20.72	*390*

Result. For the denpendency embedding module settings, Table VI shows that increasing both the number of steps and the maximum node count leads to improved performance. The best accuracy, 20.72%, is achieved with 5 steps and 200 maximum nodes, indicating that a greater number of steps allows for deeper dependency embedding, while a higher node limit helps capture more complex graph structures, both of which contribute to better repair generation by FAVOR.

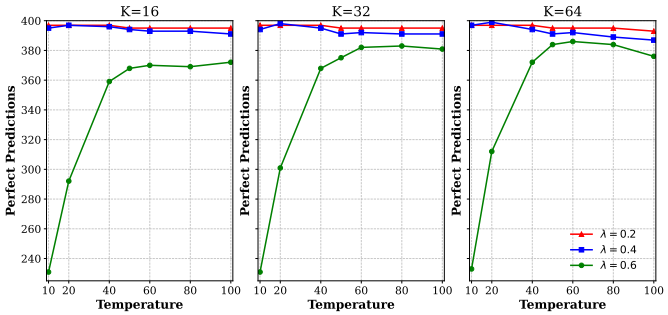

Fig. 3: Hyperparameters study on pattern store of FAVOR

For the pattern store, based on Figure 3, the best performance is achieved when $K = 64$, $\lambda = 0.2$, and the temperature is 20, resulting in 399 vulnerabilities being repaired. As K increases, the model retrieves more relevant patterns, leading to higher perfect prediction accuracy. In terms of λ, values of $\lambda = 0.2$ perform better across all K values, while $\lambda = 0.6$ consistently shows the lowest performance. Moreover, selecting an appropriate temperature (around 20-40) enhances the robustness of the model, as performance improves significantly up to this range, after which the improvement plateaus.

> **Answer to RQ3:** For the dependency module, increasing both the number of steps and maximum nodes in the dependency embedding module improves performance, repairing 390 vulnerabilities. For the pattern store, the optimal settings of $K = 64$, $\lambda = 0.2$, and a temperature of 20 provide the best results, repairing 399 vulnerabilities. with larger K values retrieving more relevant patterns and moderate temperature ranges enhancing model robustness.

VI. DISCUSSION

A. Explanations for low repair accuracy

The main reasons of FAVOR for the generating incorrect patches including: 1) Poor dataset quality: Many patches applied to functions do not directly address the root cause of the vulnerabilities [37]. In some cases, patches involve rewriting large sections of code or replacing entire functions with new APIs, rather than resolving the underlying security issue in these vulnerable functions. 2) Multiple block repairs: Vulnerabilities, unlike bugs, often require repairs across multiple code blocks, significantly increasing the complexity of the repair task. 3) Limitations of the pattern store: The limited size of the dataset and unclear annotations hinder the pattern store's ability to capture relevant repair patterns effectively, reducing its overall impact in guiding accurate fixes.

B. Future Work

Large models have garnered significant attention due to their remarkable generative capabilities. However, when applied directly to vulnerability repair tasks, they may not always deliver satisfactory results. This limitation often stems from the models' inability to fully grasp the nuanced details of vulnerability root causes. In future work, we aim to focus not only on optimizing input context to help large models better understand vulnerabilities, but also on developing methods that allow these models to analyze and debug vulnerability patches more like expert security engineers. By incorporating capabilities such as dynamic analysis and real-time debugging, we believe large models can achieve a deeper comprehension of complex vulnerabilities, ultimately improving their accuracy and efficiency in generating effective patches.

VII. THREATS TO VALIDITY

Threats to internal validity. We note that our FAVOR consists of many hyperparameters, and different hyperparameters could potentially impact the evaluation results [38], [39]. However, finding an optimal set of hyperparameters is a task with significant costs. To alleviate this threat, for existing work [15], [14], we adopt their hyperparameters and choose the same ones. Meanwhile, we also analyze the impact of some parameters on FAVOR, such as the max number of nodes in CFG provided to the model, all of which will be beneficial for future research efforts.

Threats to external validity. The threat comes from evaluating the correctness of patches. Although we utilized CodeBLEU [34], existing studies [26] have demonstrated that this metrics can be misleading when evaluating APR technologies. A higher CodeBLEU score does not necessarily correlate with a higher number of successfully repaired patches. Given the similarities between AVR and APR technologies, we decided not to rely on CodeBLEU as our primary evaluation metric.

VIII. RELATED WORK

NMT based automatic program repair. The APR work based on NMT has gained significant attention from the academic and industrial communities in recent years [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52]. Neural Machine Translation (NMT)-based Automated Program Repair tools can be categorized into three distinct groups [53], [54], Sequence-based approach [42], [43], [44], [45], [46], Tree-base approach [47], [48], [49], [50], Graph-based approach [51], [52].

Sequence-based approach: These approaches [42], [43], [44], [45], [46] conceptualizing Automated Program Repair as a task of token-to-token translation, which directly feed the buggy code segments, which contains buggy line and context into models to generate fixed patch. Tufano et al. [42] chooses method-level code snippet input to RNN-based model to generate fixed functions. Chen et al. [45] introduce SequenceR based on long short-term memory (LSTM) model with method-level input, and choose word copy mechanism to handle Out-Of-Vocabulary (OOV) issue. Lutellier et al. [43], [44] employs a pair of fully convolutional encoders, each dedicated to representing the buggy lines and their surrounding context independently.Bug-Transformer [46] adopt transformer-based model architecture in the APR tasks and achieved promising result. Xia et al. [55] proposed AlphaRepair, which is the first cloze-style APR method that directly utilizes a large pre-trained code model for APR without any fine-tuning or retraining on historical bug fixes.

Tree-based approach: The predominant approach [47], [48], [49], [50] of tree-based category involves parsing source code into Abstract Syntax Trees (ASTs), which offer more precise syntactic information and richer structural details. However, Seq2Seq models are incapable of directly processing tree-based inputs without additional modifications. To address this, there are primarily two methods. The first [47], [48] is to transform the tree-structured ASTs into a sequential format, and the second [49], [50] is to employ models, such as Tree-based LSTM, that are designed to analyze and process tree-based structure.Devlinet al. [47] use depth-first traversal to change the ASTs into nodes sequences, then using a LSTM-based model encoding them. Tang et al. [48] formulate a set of rules to transform tree-structured ASTs into sequence AST nodes, which can be easily captured by vanilla transformer. Li et al. [49], [50] employ a tree-based RNN model to learn the context and changes within ASTs.

Graph-based approach: Similar to the tree-based approach, the majority of graph-based tools utilize ASTs or dependence graph as a code representation, aiming to compensate for the lack of structural information inherent in sequence-based inputs. Unlike the tree-based approach, however, most of these tools employ a specialized Graph Neural Network (GNN) designed to learn from the information encapsulated within the edges and nodes of the graph. Recorder [51] conceptualizes the ASTs as direction graph, wherein nodes represent AST elements, and edges signify the relationships between each node. Additionally, Xu et al. [52] extend this context by incorporating both data and control dependencies, which are captured through data dependence graph (DDG) and control dependence graph (CDG), and adopt a GNN to capture these enriched information.

The task of automatic program repair focuses on fixing bugs in programs, which may not fall into any specific category of software vulnerabilities. Therefore, due to the different nature of the tasks, we have not directly compared with these automatic program repair tools.

NMT based automatic vulnerability repair. Most of the previous NMT-based AVR tools [56], [14], [15] are based on seq2seq model with the whole function input. SeqTrans [56] implements a standard transformer model, augmented with a copy mechanism, to rectify Java vulnerabilities. Similarly, Vrepair [14] employs a conventional transformer, but uniquely pre-trained on an extensive corpus of bug fixes in C/C++ to address vulnerabilities in these languages. In a comparable vein, Vulrepair [15] utilizes the CodeT5 [18] framework, incorporating a Byte Pair Encoding (BPE) [57] tokenizer. However, existing work [14], [15], [56] simply treats the AVR task as a

Seq2Seq task, ignoring the intrinsic structural information of the program. By incorporating dependency information and repair pattern store, FAVOR significantly improves the success rate of repairs.

IX. CONCLUSION

In this paper, we introduce FAVOR, a tool designed to enhance patch generation by embedding dependency information and leveraging a pattern store. Specifically, FAVOR takes a vulnerable function and its corresponding CFG as input, using a dependency embedding module to capture structural and dependency information. Additionally, we build a pattern store that employs KNN search during the generation phase to find the most similar patterns, improving repair accuracy. In the evaluation, FAVOR repaired 45 more vulnerabilities compared to VULREPAIR.

X. ACKNOWLEDGEMENTS

This work was supported by National Natural Science Foundation of China under Grant No (62202026).

REFERENCES

[1] "2023 cve data review," 2023, accessed: 2024. [Online]. Available: https://www.cvedetails.com/browse-by-date.php

[2] "National vulnerability database dashboard," 2023, accessed: 2023. [Online]. Available: https://nvd.nist.gov/general/nvd-dashboard

[3] "National vulnerability database dashboard," 2022, accessed: 2022. [Online]. Available: https://nvd.nist.gov/general/nvd-dashboard

[4] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Aug 2021. [Online]. Available: http://dx.doi.org/10.1145/3468264.3468597

[5] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: Statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 596–607.

[6] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[7] 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, 2021.

[8] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," *Cornell University - arXiv,Cornell University - arXiv*, Jul 2018.

[9] C. Pornprasit and C. K. Tantithamthavorn, "Deeplinedp: Towards a deep learning approach for line-level defect prediction," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 84–98, 2022.

[10] "2023 vulnerability statistics report," https://www.edgescan.com/intel-hub/stats-report/, accessed: 2023-11-04.

[11] Y. Song, X. Gao, W. Li, W.-N. Chin, and A. Roychoudhury, "Provenfix: Temporal property-guided program repair," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 226–248, 2024.

[12] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–27, 2021.

[13] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 8–18.

[14] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, 2022.

[15] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 935–947.

[16] 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*, 2021, pp. 30–39.

[17] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, Jun 2020. [Online]. Available: http://dx.doi.org/10.1145/3379597.3387501

[18] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[19] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[20] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[21] U. Alon, F. Xu, J. He, S. Sengupta, D. Roth, and G. Neubig, "Neuro-symbolic language modeling with automaton-augmented retrieval," in *International Conference on Machine Learning*. PMLR, 2022, pp. 468–485.

[22] U. Khandelwal, O. Levy, D. Jurafsky, L. Zettlemoyer, and M. Lewis, "Generalization through Memorization: Nearest Neighbor Language Models," in *International Conference on Learning Representations (ICLR)*, 2020.

[23] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[24] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[25] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks."

[26] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *arXiv preprint arXiv:2302.05020*, 2023.

[27] "Joern," 2024, accessed: 2024. [Online]. Available: https://github.com/joernio/joern

[28] H. Taud and J.-F. Mas, "Multilayer perceptron (mlp)," *Geomatic approaches for modeling land change scenarios*, pp. 451–455, 2018.

[29] K. Cho, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[30] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57–81, 2020.

[31] Z. Yuan, Y. Yan, R. Jin, and T. Yang, "Stagewise training accelerates convergence of testing error over sgd," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[32] OpenAI, "Gpt-4 technical report," *ArXiv*, vol. abs/2303.08774, 2023. [Online]. Available: https://arxiv.org/abs/2303.08774

[33] "Openai," 2023, accessed: 2023. [Online]. Available: https://openai.com/

[34] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[35] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[36] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang *et al.*, "Automatic code summarization via chatgpt: How far are we?" *arXiv preprint arXiv:2305.12865*, 2023.

[37] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai, "Vulgen: Realistic vulnerability generation via pattern mining and deep learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2527–2539.

[38] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction models," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 321–332.

[39] C. Tantithamthavorn, S. McIntosh, A. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 683–711, 2018.

[40] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.

[41] X. Li, S. Liu, R. Feng, G. Meng, X. Xie, K. Chen, and Y. Liu, "Transrepair: Context-aware program repair for compilation errors," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[42] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[43] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[44] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[45] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[46] J. Yao, B. Rao, W. Xing, and L. Wang, "Bug-transformer: Automated program repair using attention-based deep neural network," *Journal of Circuits, Systems and Computers*, vol. 31, no. 12, p. 2250210, 2022.

[47] J. Devlin, J. Uesato, R. Singh, and P. Kohli, "Semantic code repair using neuro-symbolic transformation networks," *arXiv preprint arXiv:1710.11054*, 2017.

[48] Y. Tang, L. Zhou, A. Blanco, S. Liu, F. Wei, M. Zhou, and M. Yang, "Grammar-based patches generation for automated program repair," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2021, pp. 1300–1305.

[49] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[50] Y. Li, S. Wang, and T. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 511–523.

[51] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 341–353.

[52] X. Xu, X. Wang, and J. Xue, "M3v: Multi-modal multi-view context embedding for repair operator prediction," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 266–277.

[53] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *arXiv preprint arXiv:2301.03270*, 2023.

[54] X. Gao, Y. Noller, and A. Roychoudhury, "Program repair," *arXiv preprint arXiv:2211.12787*, 2022.

[55] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[56] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin, "Seqtrans: automatic vulnerability fix via sequence to sequence learning," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 564–585, 2022.

[57] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.