DiffusionAVR: Automated Software Vulnerability Repair Via Diffusion Models

Anonymous ACL submission

Abstract

The automated vulnerability repair technology aims to reduce security risks for end users by repairing vulnerable code. Currently, machine translation-based models are becoming the mainstream approach in the field of vulnerability repair, focusing on modeling the conditional probability distribution functions of the patch subspace. However, the distribution of repair patches is complex and unknown, and excessive complexity makes it difficult for traditional models to effectively fit unknown data. To address this, this paper introduces a novel automated vulnerability repair method based on diffusion models, called DiffusionAVR. Compared to traditional models, DiffusionAVR progressively adds noise, allowing the complex distribution of repair patch data to gradually approximate a standard Gaussian distribution, and then achieves the process of going from simple to complex distribution through denoising and reverse modeling. In this context, the model can more easily learn the characteristics of vulnerability patch distributions, significantly improving the success rate of repairing unknown code. Furthermore, this paper optimizes the loss function in the traditional diffusion model for repair tasks, significantly enhancing repair accuracy. In tests involving 8,482 real vulnerability cases, DiffusionAVR achieved a perfect repair rate in single predictions that was 14% higher than existing Transformer-based models, with a generation speed increase of 20 times.

1 Introduction

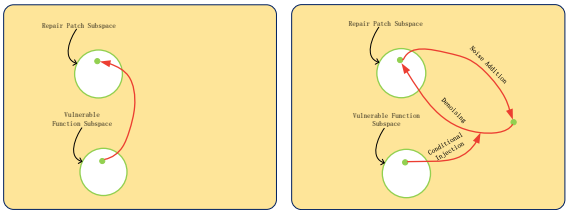
As the scale and complexity of software systems continue to increase, the number of software vulnerabilities is rising at an unprecedented rate, presenting a severe challenge to global cybersecurity. Software vulnerabilities refer to code defects or security weaknesses that allow attackers to bypass normal security mechanisms, perform unauthorized operations or access, and even cause system crashes. According to statistics from NDV[1], the number of vulnerabilities has been climbing year by year, forcing engineers to spend a significant amount of time and resources on repairs. However, traditional manual repair methods are inefficient and struggle to meet the growing demand.

Vulnerability detection is a prerequisite for vulnerability repair, and deep learning has been widely applied in this area Recently (Chakraborty et al., 2021), automatic vulnerability repair (AVR) methods have been extensively used to address the growing cybersecurity threat (Fu et al., 2024). By learning the mapping between vulnerable code and corresponding repair patches, these models generate patches for new vulnerabilities, accelerating the manual repair process(Britton et al., 2012; Mu et al., 2018).

Currently, AVR methods based on the Transformer (Vaswani et al., 2017) architecture have shown certain advantages in predicting vulnerabilities. For example, VulRepair (Fu et al., 2022), by fine-tuning the pre-trained CodeT5 model (Wang et al., 2021), can correctly predict 30% of the repair patches in the test set; in contrast, decoder-only architecture models like ChatGPT-4 have failed to successfully predict any repair patches (Fu et al., 2023). Previous methods can be summarized as learning a conditional generation model for repair patches. As shown in Figure 1.a, VulRepair is dedicated to modeling the conditional probability distribution of repair patches for specific vulnerable functions. However, the distribution of repair patches is complex and unknown, and directly modeling such a complex distribution severely impacts the model's performance, leading to poor results on unknown data (i.e., data outside the training set). Additionally, the fine-tuning scheme heavily relies on pre-trained models, which, while providing some performance improvement, also in-

[1] https://nvd.nist.gov/vuln

troduces limitations in adaptability. For example, the CodeT5 model based on the Transformer architecture restricts text length during pre-training, and the fine-tuned CodeT5 model cannot handle longer vulnerability codes or generate longer repair patches. Currently, the best-performing model can only accept vulnerability code lengths of 512 tokens and generates repair code that does not exceed 256 tokens.



(a) (b)

Figure 1: (a) Traditional vulnerability remediation methods. (b) Diffusion-based vulnerability remediation methods.

Additionally, repair patches are generally highly related to the source code and only require modifications to a portion of the code. For example, as shown in Figure 2, the repair patch retains a large amount of the original source code, removing only a small part. However, existing models use the decoder module to generate repair patches sequentially and calculate the attention between the pre-generated tokens and the already generated tokens, which severely affects the model's generation speed.

Vulnerable Function

```
void modifyBytes(unsigned char *byte, int length) {
    if (byte == NULL || length <= 0) {
        printf("Invalid input.\n");
        return;
    }

    int i, j;
    for (i = 0; i < length; i++) {
        if (byte[i] & 0x7f) {
            break;
        }
    }

    if (byte[i] & 0x40) {
        i--;
    }

    byte[i] &= 0x7f;

    for (j = 4; j >= i; j--) {
        byte[j] &= 0x7f;
    }
}
```

Repaired Function

```
void modifyBytes(unsigned char *byte, int length) {
    if (byte == NULL || length <= 0) {
        printf("Invalid input.\n");
        return;
    }

    int i, j;
    for (i = 0; i < length; i++) {
        if (byte[i] & 0x7f) {
            break;
        }
    }

    - if (byte[i] & 0x40) {
    + if (byte[i] & 0x40 &&i>0)
        i--;
    }

    byte[i] &= 0x7f;

    for (j = 4; j >= i; j--) {
        byte[j] &= 0x7f;
    }
}
```

Figure 2: Example of Vulnerability Repair

Traditional fine-tuning methods focus on modeling the conditional probability model of complex distributions (Repaired patches) under another complex distribution (vulnerability function). To learn the characteristics of vulnerability patches in complex distributions, this paper proposes an automated vulnerability repair method based on diffusion models—DiffusionAVR. DiffusionAVR models the conditional probability of complex distributions (fix patches) under a simple Gaussian distribution during the denoising process and injects the fix patches as conditional information into the denoising process. This allows the model to learn the more essential distribution characteristics of vulnerability patches. Additionally, we improve the loss function of traditional diffusion models to ensure the generation of high-precision fix patches.

As shown in Figure 1.b, DiffusionAVR gradually approaches a standard Gaussian distribution by progressively adding noise to the complex distribution of repair patches, and subsequently achieves the process of moving from a simple distribution to a complex distribution through denoising and reverse modeling. Compared to directly modeling complex distributions, the model can more easily learn the deep characteristics of the repair patch distribution, significantly improving the success rate of repairing unknown code. To enable controllable generation of repair patches, we map the vulnerability patches to the same latent space as the repair patches during the denoising process and inject this as conditional information into the denoising process. The trained model can start from any point in the latent space that conforms to the Gaussian distribution, using source code representations as conditions. The model then iteratively denoises the data, bringing it back to specific points in the repair patch subspace to generate repair patches. Experimental results indicate that in single-round generation, DiffusionAVR achieved a perfect generation rate of 44.4%, which is 14.4% higher than VulRepair and slightly above the 44% result from its provision of 50 candidate patches.

Compared to Transformer-based models, DiffusionVR's denoising module uses a parallel generation Encoder-only structure. When generating repair patches, this module does not need to incrementally calculate the attention weights between the generated tokens and the pre-generated tokens, thereby improving the generation speed by approximately 20 times.

In summary,we highlight the following contributions of our proposed DiffusionAVR

- **Introducing Diffusion Models for Automated Vulnerability Repair:** Compared to existing Transformer-based methods, DiffusionAVR can learn the more essential distri-

bution characteristics of vulnerability patches by modeling the conditional probability distribution functions of complex vulnerability patches under a simple Gaussian distribution. This significantly improves the accuracy of repair patch generation.

- **Optimizing the Diffusion Model's Loss Function for Vulnerability Repair Tasks:** We propose a novel loss function specifically tailored for automated vulnerability repair tasks, improving model accuracy in generating repair patches and increasing adaptability to complex code scenarios.

- **Eliminating Dependency on Pre-Trained Models to Optimize Generation Efficiency and Accuracy:** DiffusionAVR overcomes the token length limitations of Transformer architectures, enabling it to handle more complex repair scenarios. Through its parallel generation mechanism, DiffusionAVR achieves a repair patch generation speed approximately 20 times faster than the best existing models, with a 14% improvement in single-round perfect repair rates.

2 DiffusionAVR

In this section, we will provide a detailed introduction to the architecture of DiffusionAVR, which is a diffusion-based automatic vulnerability repair method that consists of three main parts: code representation, forward noise injection, and backward denoising. Finally, we will explain how to use the trained DiffusionAVR to generate repair patches.

2.1 Code Representation

The purpose of code representation is to transform code into a vectorized format, shifting the state space of the code from discrete to continuous for the subsequent diffusion process. The code representation consists of two components: a subword tokenization algorithm and an embedding layer.

- **BPE Subword Tokenization.** BPE (Sennrich, 2015) can split uncommon tokens into meaningful subwords while preserving the format of common words. For example, the variable name [unix_dgram_peer_wake_disconnect] can be represented as [un, ix, d, gram, peer, wake, dis, connect]. BPE not only accommodates long variable names but also constrains the vocabulary size to prevent excessive vocabulary leading to sparse semantic information. In DiffusionAVR, we use the BPE pre-trained by Feng et al. (Feng et al., 2020) on CodeSearchNet (CSN) (Husain et al., 2019), hereinafter referred to as CodeBERT Tokenization. The tokenization algorithm of CodeBERT is based on BERT's WordPiece (Wu, 2016) and optimizes for the characteristics of both code and natural language. This paper retains the four special subwords added in VulRepair: ["<StartLoc>" ,"<EndLoc>" ,"<ModStart>" ,"<ModEnd>"]. The use of BPE ensures the model's ability to handle unseen words and constrains the vocabulary size.

- **Embedding Layer.** After the source code is processed by the BPE algorithm, it is transformed from the original long text into multiple independent subword tokens. The meaning of each token depends on its context (i.e., surrounding tokens) and its position within the function. Therefore, it is necessary to capture the context of the tokens and their positional information. The purpose of this section is to generate embedding vectors to capture the semantic meaning of code tokens and their positions within the function.

We generate a 1×768 embedding vector for each token and combine them into an embedding matrix to represent the relative dependencies between the selected token and its surrounding tokens. To capture the positional information of the tokens in the code, we adopt the absolute position embeddings from BERT, which explicitly encode the positional information of the tokens. To address the token length limitation imposed by the pre-trained model, both the embedding layer and position embeddings in DiffusionAVR are randomly initialized and adjusted during training without using any pre-trained parameters. As shown in Figure 3, the repair patches and vulnerability functions share the same tokenization algorithm and embedding layer, so their representation vectors X and Y exist in the same vector space. This facilitates the direct injection of Y as conditional information into the denoising process during subsequent diffusion, eliminating the need for additional alignment operations.

2.2 Forward Noise Injection

Unlike traditional methods, after obtaining the representation vectors X and Y for repair patches and

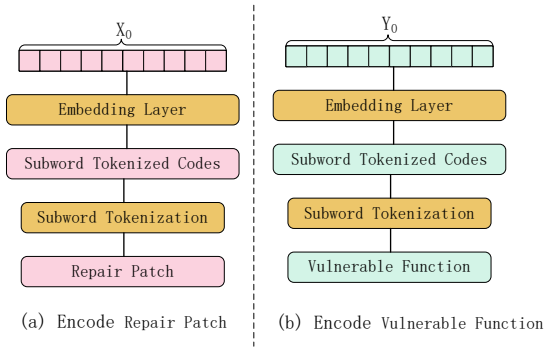


Figure 3: Embedding Layer. (a) is the embedding vector X for the repair patch, and (b) is the repair patch Y for the vulnerability function.

vulnerability functions, DiffusionAVR does not directly model $P(X \mid Y)$. Instead, it first disrupts X: gradually and iteratively adding Gaussian noise to the repair patches in the training set, causing them to “move out of or away" from the existing subspace. This process can be represented by a closed-form formula, which allows for the direct calculation of noisy data at a specific time step without the need for iterative noise addition.

$$X_t = \sqrt{\alpha_t} X_0 + \sqrt{1-\alpha_t} I \tag{1}$$

Equation 1 represents the noise addition formula for the repair patches, where X_0 denotes the original representation vector of the repair patches, α_t indicates the noise scale, and I is standard Gaussian noise. The value of α_t decreases as the time step t increases, ranging from [0,1]. The longer the time step, the greater the noise scale, causing the disrupted representation vector of the repair patches to become closer to the standard Gaussian distribution.

At the end of the noise addition process, the repair patches become completely unrecognizable. The complex distribution is transformed into a simple Gaussian distribution, with each repair patch being mapped to a space outside the data subspace. In the forward process, there exists a simple conditional probability distribution function $P(X_t|X_0)$ that expresses the mapping from the original repair patch to the external space, which can be described by Equation 1.

2.3 Backward Denoising

The core idea of the denoising process is to iteratively reverse the damage inflicted on the repair patches by the forward process. The reversal process begins with the simple distribution X_t generated by the forward process, which has the advantage that we know how to sample a point from this simple distribution. The goal of the reverse process is to find the path back to the repair patch subspace, that is, to model $P(X_0|X_t)$. However, the problem lies in the fact that we can start from a point in this “simple" space and advance along countless paths, but only a small fraction of those paths can lead us back to the repair patch subspace.

Therefore, at this stage, we need to use neural networks to reverse fit the parameters of the conditional probability distribution function $P(X_0|X_t)$ of the forward process. To achieve parallel generation of repair patches, DiffusionAVR adopts the Bert (Kenton and Toutanova, 2019) architecture for denoising. At the same time, to ensure the controllability of the generated repair patches, we need to inject the corresponding vulnerability code as conditioning information during the denoising process. The specific approach is to concatenate the vulnerability code representation vector Y generated during the code representation stage with the vector X_t generated during the noise-adding stage to serve as the model input Z_0.

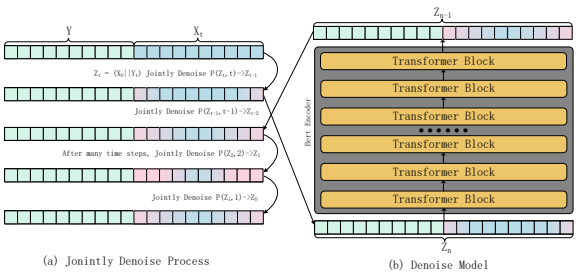


Figure 4: Denoising Process

Figure 4 provides a detailed description of the model architecture for the denoising process. The model consists of 12 Transformer encoder blocks, with each block starting with layer normalization (Lei Ba et al., 2016), followed by a multi-head attention layer (Li et al., 2018) and a feedforward neural network. During the stepwise denoising process, the self-attention mechanism calculates the weights between various tokens in Z_n. Meanwhile, the undamaged vulnerability code information is multiplied by the computed weights and integrated into the damaged repair patch vector X_t, thereby injecting conditional information into the mapping from X_t to X_0. DiffusionAVR employs timestep encoding, with the denoising model input being Z_n augmented with positional and timestep encodings. Unlike traditional diffusion models that

predict noise, DiffusionAVR directly predicts Z_0 . Furthermore, to ensure the generation of high-precision repair patches, we have improved the common loss function used in text diffusion; the original loss function is as follows:

$$\begin{aligned} &\min_\theta \mathcal{L}_{\mathrm{VLB}} = \min_\theta \Big[\textstyle\sum_{t=2}^{T} \|z_0 - f_\theta(z_t, t)\|^2 \\ &+ \|EMB(w^{x \oplus y}) - f_\theta(z_1, 1)\|^2 \\ &- \log p_\theta(w^{x \oplus y} \mid z_0)\Big] \end{aligned} \tag{2}$$

Where Z_0 is formed by concatenating the vulnerability code Y and the patch X. The L_{VLB} loss minimizes the distance between Z_0 and the denoising model's output by calculating the mean squared loss between the model output and the original Z_0. Gong et al. suggest that during the reconstruction of Z_0, the model learns the distributional relationship between X_0 and Y_0. However, in vulnerability repair tasks that require precise patch generation, an L_{VLB} loss based on Euclidean distance may lead the model to generate tokens that are similar but incorrect. Therefore, we compute the word probability of each vector in the predicted code patch sequence X_0^θ output by the denoising model using a linear layer, resulting in a probability vector p aligned with the vocabulary length. This vector is then concatenated to form a matrix P where $P \in R^{n \times c}$, with n as the number of tokens and c as the vocabulary size. One-hot vectorize each token in X as labels; we calculate the cross-entropy loss:

$$L_{nll} = -\sum_{i=1}^{C} X_i \log\left(\widehat{X_i}\right) \tag{3}$$

where C is the vocabulary length, and $\widehat{X_i} \in P$. Finally, the loss function used in DiffusionAVR is:

$$\mathcal{L}_{\mathrm{VR}} = \mathcal{L}_{\mathrm{VLB}} + L_{nll} \tag{4}$$

The necessity of this additional loss for vulnerability repair is discussed and verified in Appendix A.

2.4 Generating Repair Patches

After being trained through the denoising process, DiffusionAVR is able to learn how to map a simple Gaussian distribution to the repair patch subspace. During this process, the corresponding vulnerability source code serves as conditioning information in the mapping. Thus, the generation process begins with a matrix of randomly sampled vectors that conform to a Gaussian distribution, which is then concatenated with the embedding matrix of the vulnerability function. This undergoes iterative denoising, ultimately resulting in a repair patch corresponding to the input vulnerability function.

DiffusionAVR employs the diffusion ODE method proposed by DPM-solver++ (Lu et al., 2022) during the generation process. This method, applied in DiffuSeq-v2 (Gong et al., 2023), significantly accelerates inference speed, using DDIM (Zhang et al., 2022) to further accelerate the generation process and achieving results similar to 2000 steps with only 10 sampling steps.

We also experimented with an unaccelerated generation process, which required over 4 hours to generate 20 repair patches. This high time cost makes the original diffusion method infeasible for vulnerability repair tasks. In contrast, with the ODE method, generating 1,706 repair patches takes only 107 seconds, making the generation speed approximately 20 times faster than other non-diffusion methods.

3 Evalution

In this section, we present the baseline models used for comparison, the dataset, and the experimental setup.

3.1 Baseline Models

In recent years, the Transformer architecture has been widely adopted for vulnerability repair. VulRepair, proposed by Fu et al. (Fu et al., 2022), achieved the highest perfect repair rate and provides reproducible code for VRepair and CodeBERT, making comparison straightforward. Therefore, we selected baseline models consistent with those used by Fu et al. , as follows:

1. **VulRepair** (Fu et al., 2022): This model uses CodeT5 (Wang et al., 2021) for vulnerability repair. Based on the T5 (Raffel et al., 2020) text-to-text Transformer model, VulRepair fine-tunes CodeT5 on a vulnerability dataset in a fully parameterized manner.

2. **VRepair** (Chen et al., 2022): This standard Transformer model is first trained on a patch dataset and then fine-tuned on vulnerability-related repair code.

3. **CodeBERT** (Feng et al., 2020): Developed by Microsoft Research, this Transformer-based model is a decoder-only variant similar to

BERT, pre-trained on CodeSearchNet (Husain et al., 2019). The version of CodeBERT used in this study comprises 12 Transformer encoder blocks (CodeBERT) and a six-layer Transformer decoder for generation tasks. Hemmati et al. (Mashhadi and Hemmati, 2021) employed this CodeBERT model with a stacked decoder for automated program repair of Java vulnerabilities, achieving significantly better performance compared to RNN models.

3.2 Dataset

We used the CVEFixes_BigVul vulnerability dataset[2], consistent with previous studies. This dataset is available on Hugging Face, and we retained 8,428 pairs of vulnerable functions and repair patches without additional filtering.

To ensure accurate comparison with the baseline models, we strictly adhered to the dataset preprocessing methods provided by Chen et al. (Chen et al., 2022). Each vulnerable function includes a special label indicating its CWE type, with "<StartLoc>" and "<EndLoc>" tags marking the start and end positions of the vulnerable code segments. Similarly, the repair patches use "<ModStart>" and "<ModEnd>" tags to denote the modified sections. To prevent the tokenizer from splitting these special tags, we explicitly added them to the tokenizer vocabulary. These tags help the model focus on the vulnerable code segments and the corresponding repair content.

3.3 Experimental Setup

Following Fu et al. (Fu et al., 2022), we split the dataset into 70% for training, 10% for validation, and 20% for testing, ensuring that the test set exactly matches the one used by Fu et al.

We implemented DiffusionAVR using the Transformers (Wolf, 2019) and PyTorch (Collobert et al., 2011) libraries. The Transformers library was used to initialize BertEncoder for the denoising model, and PyTorch handled computations during training, including backpropagation and parameter optimization. The model was trained on an NVIDIA A100 with Ubuntu 20.04 and CUDA 11.4.

We initialized the denoising model using the configuration file from CodeBERT, increasing the maximum position encoding length from 512 to 768 and excluding any pre-trained weights.

[2] https://huggingface.co/datasets/MickyMike/cvefixes_bigvul

3.4 Evaluation Metrics

We used the percentage of perfect predictions (%pp) to assess the model's accuracy in generating software vulnerability repairs. This metric measures how closely the generated repair patch matches the manually created patch.

4 Experimental Result

We conducted a detailed evaluation of DiffusionAVR and baseline models through experiments to address the following four research questions:

1. **RQ1:** What is the accuracy of DiffusionAVR in generating software vulnerability repairs?
2. **RQ2:** What is the time cost of DiffusionAVR?
3. **RQ3:** How do sampling steps affect DiffusionAVR's generation capability?

4.1 RQ1: What is the accuracy of DiffusionAVR in generating software vulnerability repairs?

We compared DiffusionAVR with baseline models using configurations provided by Fu et al. DiffusionAVR's inference model is based on an encoder-only architecture that supports parallel outputs but does not support beam search; therefore, we generated only a single candidate vulnerability patch in this study.

Table 1 shows the superiority of DiffusionAVR; compared to VulRepair, it has achieved a 14.4% improvement in %pp on the same dataset, and it also significantly outperforms all models based on the Transformer architecture. Unlike VulRepair, DiffusionAVR only utilizes the Encoder part, resulting in a reduction of model parameters by half. Traditional Transformer architectures extract latent space vectors from vulnerability code using attention mechanisms and use a Decoder to generate the corresponding sequence, establishing a mapping relationship between vulnerability code and vulnerability repair code. However, the existing vulnerability repair dataset is too small (only 8482 samples), and the cost of collecting high-quality data is excessively high, making it difficult for all Transformer-based automatic vulnerability repair methods to model the conditional probability of complex distributions (repair code) under another complex distribution (vulnerability patches) in practical applications.

Table 1: Performance Metrics and Generation Times of DiffusionAVR and Baseline Models.

	DiffusionAVR	VulRepair	CodeBERT	VRepair
%pp	44.4%	30%	18%	21%
IT(ms)	167	2356	2785	1987

DiffusionAVR, using a limited dataset, gradually transfers the repair patch from the original distribution space to a simple Gaussian distribution through forward noise addition, and then models the process of restoring the vulnerability patch from the Gaussian distribution. We can restore the vulnerability patch starting from any sample that conforms to a Gaussian distribution; therefore, this modeling approach of first disrupting and then restoring can learn a more intrinsic distribution on a limited set of vulnerability patches, achieving stronger structural generalization ability.

4.2 RQ2: What is the Time Cost of DiffusionAVR?

VulRepair, which employs a beam search algorithm (Freitag and Al-Onaizan, 2017), achieves an accuracy rate of 44% with a 50-round search. However, beam search primarily enhances output diversity, and generating 50 candidate repair codes requires manual screening, increasing labor and time costs. This additional effort makes it less suitable for practical deployment in automated vulnerability repair. In this study, we consider time cost as one of the model evaluation criteria. By setting the beam search size to 1, we compare the time required for each baseline model and DiffusionAVR to generate an equal number of candidate repair patches. Results are shown in Table 1.

DiffusionAVR not only achieves the highest perfect generation rate but also requires only about 1/20 of the generation time compared to other models. This advantage in time cost is due to DiffusionAVR's model architecture: it uses only the encoder component and generates tokens in parallel during inference. In Transformer-based models, token generation is sequential, with each token depending on previously generated ones, which increases time costs. Since repair code is highly related to the original vulnerable code and does not depend on previously generated tokens, using only the encoder to generate repair code from noise maintains model performance while significantly reducing generation time. VulRepair relies on beam search to produce 50 candidate repair codes to approach DiffusionAVR's perfect generation rate; however, its high time cost and the manual selection required make it impractical.

4.3 RQ3: How Does the Number of Sampling Steps Affect DiffusionAVR's Generation Capability?

To investigate how DiffusionAVR acquires repair capabilities through diffusion, we examined the relationship between sampling steps and %pp. As shown in Figure 5, there is a clear positive correlation between sampling steps and perfect prediction rate. Notably, shorter repair patches require fewer sampling steps to reach peak performance. For example, repair patches of 0–10 tokens reach peak %pp at 25,000 sampling steps, while patches with 50 or more tokens achieve only 1% %pp at that point.

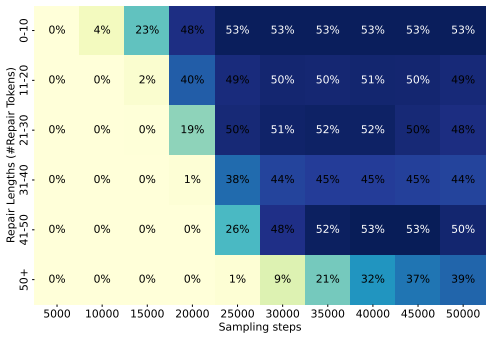


Figure 5: Relationship Between Sampling Steps and Repair Patch Length

DiffusionAVR learns to generate relatively simple short repair patches through a diffusion process. As the number of sampling steps increases, the amount of data involved in denoising sharply rises, and the model gradually learns how to reconstruct long repair patches from a standard Gaussian distribution. In the final convergence scenario, it demonstrates similar repair capabilities when dealing with repair patches of various lengths.

5 Related Work

5.1 Automated Vulnerability Repair (AVR) Methods

Auto vulnerability repair(AVR) can be viewed as a sequence-to-sequence task, aiming to learn the mapping between vulnerable code X and repair code Y by establishing the following relationship:

$$F_\theta(X) = Y \quad (5)$$

where F_θ is the mapping function. The core of AVR lies in using neural networks to approximate this function. Typically, the neural network comprises an encoder and a decoder: the encoder receives the vulnerable code X and maps it to a vector H in latent space; the decoder then generates the repair code Y based on H.

Early approaches widely used RNNs (Sutskever, 2014) for implementing encoders and decoders (Chen et al., 2019; Tufano et al., 2019). However, as the length of vulnerable code increased, RNN-based methods often suffered from catastrophic forgetting, where early information is lost in long sequences.

The introduction of the Transformer enabled self-attention mechanisms (Shaw et al., 2018), allowing dependencies to be established between any positions in the sequence, unlike the sequential processing of RNNs. This solves the long-range dependency problem, and Transformer-based sequence-to-sequence methods outperform RNNs in generation capability. Consequently, Transformer-based AVR methods (e.g., VRepair (Chen et al., 2022) and VulRepair (Fu et al., 2022)) have been proposed, achieving state-of-the-art perfect repair rates.

Unlike NMT-based methods, DiffusionAVR introduces diffusion models to the AVR field, addresses differences between natural language and code text, and provides solutions. Through denoising, the model learns deeper syntactic information, highlighting diffusion models' advancements over NMT-based methods.

5.2 Diffusion Models

Diffusion models have achieved notable success in image generation, as exemplified by Stable Diffusion, which generates high-quality images from brief prompts. Inspired by this, Gong et al. (Li et al., 2022) proposed adding continuous Gaussian noise in discrete text space, introducing the sequence-to-sequence diffusion model DiffuSeq (Gong et al., 2022). DiffuSeq consists of three parts:

1. **Forward Noise injection Process:** In the forward process, DiffuSeq adds noise only to the target sequence, leaving the source sequence unaltered, and concatenates the noisy target with the source sequence as a noisy sequence for denoising.

2. **Backward Denoising Process:** During denoising, the BERT (Kenton and Toutanova, 2019) model is used as the denoising model, with input as the selectively noisy sequence and target as the original sequence. The model learns to generate the target sequence from the unnoised source sequence and the noisy target sequence.

3. **Generation Process:** The input for generation is the source sequence and a standard Gaussian noise sequence. The trained denoising model iteratively removes noise at each timestep, ultimately generating the target sequence.

While DiffuSeq shows good generation capabilities, its high training and generation times limit its applicability to short-sequence tasks. Recently, Gong et al. proposed DiffuSeq-V2 (Gong et al., 2023), significantly improving training speed and generation efficiency, making it suitable for more complex, long-sequence generation. DiffusionAVR builds on DiffuSeq-V2, further optimizing loss calculation to enhance generation accuracy, making DiffusionAVR suitable for AVR, where perfect repair is required.

6 Conclusion

In this paper, we proposed DiffusionAVR, a diffusion model-based automated vulnerability repair method, and adapted the loss function to account for the unique characteristics of code text. In experimental evaluations, DiffusionAVR achieved a 14.4% higher accuracy in single-round generation compared to the previous SOTA model, VulRepair, while improving generation speed by approximately 20 times, showcasing the advantages of diffusion models in vulnerability repair. DiffusionAVR successfully generated 758 repair patches out of 1,706 real-world vulnerabilities, compared to only 513 generated by VulRepair. Notably, for the top 10 most dangerous CWE types, DiffusionAVR generated 46.54% of repair patches, 8.54% higher than VulRepair, highlighting its practicality.

In the discussion, we conducted an in-depth analysis of the impact of the loss function, CWE types, and other factors on model performance. We concluded that the distribution of CWE types in the training set does not introduce bias in model capability, providing future researchers with a pathway for further exploration.

References

Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. 2012. Quantify the time and cost saved using reversible debuggers. *Cambridge Judge Business School, Tech. Rep*.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*.

Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Markus Freitag and Yaser Al-Onaizan. 2017. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806*.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. 2024. Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering*, 29(1):4.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. 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*, pages 935–947.

Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chatgpt for vulnerability detection, classification, and repair: How far are we? In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 632–636. IEEE.

Shansan Gong, Mukai Li, Jiangtao Feng, Zhiyong Wu, and LingPeng Kong. 2022. Diffuseq: Sequence to sequence text generation with diffusion models. *arXiv preprint arXiv:2210.08933*.

Shansan Gong, Mukai Li, Jiangtao Feng, Zhiyong Wu, and Lingpeng Kong. 2023. Diffuseq-v2: Bridging discrete and continuous text spaces for accelerated seq2seq diffusion models. *arXiv preprint arXiv:2310.05793*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1. Minneapolis, Minnesota.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *ArXiv e-prints*, pages arXiv–1607.

Jian Li, Zhaopeng Tu, Baosong Yang, Michael R Lyu, and Tong Zhang. 2018. Multi-head attention with disagreement regularization. *arXiv preprint arXiv:1810.10183*.

Xiang Li, John Thickstun, Ishaan Gulrajani, Percy S Liang, and Tatsunori B Hashimoto. 2022. Diffusion-lm improves controllable text generation. *Advances in Neural Information Processing Systems*, 35:4328–4343.

Cheng Lu, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. 2022. Dpm-solver++: Fast solver for guided sampling of diffusion probabilistic models. *arXiv preprint arXiv:2211.01095*.

Hans Marmolin. 1986. Subjective mse measures. *IEEE transactions on systems, man, and cybernetics*, 16(3):486–489.

Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509. IEEE.

Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Rico Sennrich. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*.

I Sutskever. 2014. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 25–36. IEEE.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.(nips), 2017. *arXiv preprint arXiv:1706.03762*, 10:S0140525X16001837.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

T Wolf. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Yonghui Wu. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Qinsheng Zhang, Molei Tao, and Yongxin Chen. 2022. gddim: Generalized denoising diffusion implicit models. *arXiv preprint arXiv:2206.05564*.

A Impact of the Loss Function

Before discussing the impact of the loss function on the vulnerability repair task, it is essential to clarify the fundamental differences between code text and natural language text. In natural language, part-of-speech tags help group semantically similar words in neighboring positions upon model convergence, forming clusters based on part of speech. Code text, however, lacks this property. Although MSE loss (Marmolin, 1986) calculates vector distances, these distances are insufficient to reflect similarity in code's semantic space. Additionally, code generation tasks demand higher logical accuracy, as generated outputs must precisely match the dataset answers; even minor character deviations can render code non-functional. Therefore, we added a cross-entropy loss term to the original diffusion model loss to constrain the model's generation scope. To validate the necessity of this added loss, we compared the performance of models trained with each loss type (see Table 3).

Table 2: Model Performance with Different Loss Functions.

Method	%pp
DiffusionAVR+$\mathcal{L}_{\text{VLB}}$	0%
DiffusionAVR+$\mathcal{L}_{\text{VLB}}$ + L_{nll}	44.4%

We found that when using only MSE loss, DiffusionAVR failed to generate any correct repair patches. The output contained a large number of "_", which, based on training set statistics, is the most frequent token. Supervising model updates with only $\mathcal{L}_{\text{VLB}}$ loss causes the model to "take shortcuts"; due to the data's uneven distribution, the distances between all tokens in the converged vocabulary space and "_" become similar. By outputting a large number of "_" tokens, the model minimizes the loss. Adding cross-entropy loss to MSE loss as a constraint restores the model's generation capability.

In DiffuSeq, which handles natural language text-to-text tasks, MSE loss alone can capture syntactic information. During training with MSE loss, the cross-entropy loss also decreases simultaneously. However, this phenomenon does not occur with code text; only by explicitly backpropagating cross-entropy loss can the model generate correct and valid content.

B Impact of CWE Types on Vulnerability Repair

CWE is a classification standard used to assess software security issues and their severity. To evaluate the practical effectiveness of DiffusionAVR, we analyzed its repair capability on the Top 10[3] most dangerous vulnerabilities and across various CWE categories. The experimental results are presented in Table 3.

Table 3: Perfect Predictions for Top 10 CWE Types.

Rank	CWE Type	Name	%PP	Proportion
1	CWE-787	Out-of-bounds Write	39.62%	21/53
2	CWE-79	Cross-site Scripting	100.00%	1/1
3	CWE-89	SQL Injection	60.00%	3/5
4	CWE-20	Improper Input Validation	40.79%	62/152
5	CWE-125	Out-of-bounds Read	48.82%	83/170
6	CWE-78	OS Command Injection	66.67%	2/3
7	CWE-416	Use After Free	56.36%	31/55
8	CWE-22	Path Traversal	50.00%	4/8
9	CWE-352	Cross-Site Request Forgery	0.00%	0/2
10	CWE-434	Dangerous Type	-	-
TOTAL			46.54%	209/449

As shown in Table 3, DiffusionAVR achieved a perfect generation rate of 46.54% on the top 10 high-risk vulnerabilities. The top three CWE types by perfect generation rate are CWE-79 (Cross-Site Scripting), CWE-78 (OS Command Injection), and CWE-89 (SQL Injection). Notably, although these three CWE types represent a small portion of the test set, another similarly small category, CWE-352, has a perfect generation rate of 0%. This suggests that the proportion of instances is not a decisive factor in influencing the perfect generation rate.

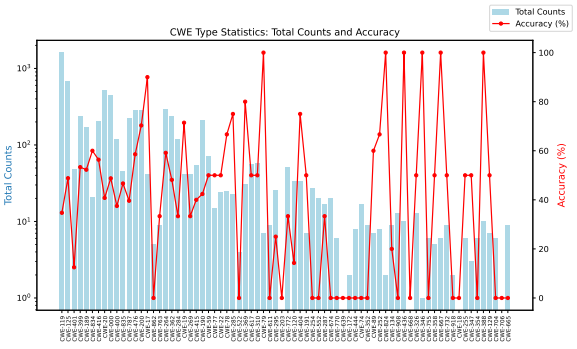


Figure 6: Relationship Between Generation Accuracy and Quantity for Each CWE Type.

To further investigate the impact of different CWE types on model performance, we analyzed the relationship between the quantity of each CWE type in the training set and the prediction accuracy metric %pp. The results are shown in Figure 6.

From the fig 6, we observe no clear correlation between %pp and the number of data samples in the training set. For example, CWE-119, which has the largest number of samples in the training set, achieves a perfect generation rate of less than 40%. In contrast, CWE-79, with fewer than 10 samples, has a perfect generation rate of 100%. The diffusion model's noise-augmented approach flattens distribution differences across various CWE types. Notably, DiffusionAVR fails to generate any correct patch code for the five CWE types absent in the training set. This analysis suggests that DiffusionAVR exhibits varying repair capabilities across different CWE types and cannot handle previously unseen vulnerability types.

C Limitations

DiffusionAVR represents the first attempt to apply diffusion models to automated vulnerability repair, achieving significant performance improvements over non-diffusion models. However, more than half of the vulnerable code samples still fail to generate correct repair patches. Currently, DiffusionAVR uses continuous Gaussian noise, which prevents the use of pre-trained parameters and limits its ability to generate tokens outside the training set vocabulary, leaving the out-of-vocabulary issue unresolved.

Although pre-training can restrict model flexibility, incorporating a pre-trained model could partially address the OOV issue when the training set cannot be effectively expanded. Our next step is to explore the use of discrete Gaussian noise and introduce pre-trained weights to mitigate the OOV problem.

D Impact of Token Length on Model Performance

VulRepair is constrained by its pre-trained model, with an input token length of 512 and an output of 256. To maintain consistency with VulRepair, DiffusionAVR sets the input token length to 768 (vulnerable function concatenated with the repair patch). To study the effect of token length on model performance, we tested three variants of DiffusionAVR:

1. **DiffusionAVR+768+No Pre-training:** Input length is 768, without pre-trained weights.

[3] https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

2. **DiffusionAVR+512+No Pre-training:** Input length is 512, without pre-trained weights.

3. **DiffusionAVR+512+Pre-training:** Input length is 512, using the pre-trained CodeBERT.

The experimental results are shown in Table 4. We observed that reducing the input token length from 768 to 512 nearly halved performance, with the number of perfectly generated repair patches decreasing from 758 to 451. Using pre-trained CodeBERT model parameters further reduced correct patch generation to almost zero (only 25 correct patches).

Table 4: Model Performance of DiffusionAVR Variants with Different Input Lengths.

Method	%pp	Proportion	IT(ms)
DiffusionAVR+768+No Pre-training	44.4%	758/1706	167
DiffusionAVR+512+No Pre-training	26.4%	451/1706	134
DiffusionAVR+512+Pre-training	1.4%	25/1706	148

This performance reduction occurs because DiffusionAVR uses noise addition in a continuous space, while the pre-trained BERT model operates in a discrete space, with its final converged state also being discrete. Adding continuous Gaussian noise to a discrete state space disrupts the data's original distribution, preventing the denoising model from effectively removing continuous noise in this discrete setting. As a result, the perfect generation rate drops to only 1.4%. In contrast, the randomly initialized state space in DiffusionAVR converges to a noise-consistent space during training, allowing the denoising model to learn the essential data distribution within this unified space.