VQR: AUTOMATED SOFTWARE VULNERABILITY REPAIR THROUGH VULNERABILITY QUERIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Recently, automated vulnerability repair (AVR) approaches have been widely adopted to combat the increasing number of software security issues. In particular, transformer-based models achieve competitive results. While existing models are learned to generate vulnerability repairs, existing AVR models lack a mechanism to provide their models with the precise location of vulnerable code (i.e., models may generate repairs for the non-vulnerable areas). To address this problem, we base our framework on the VIT-based approaches for object detection that learn to locate bounding boxes via the cross-matching between object queries and image patches. We cross-match vulnerability queries and their corresponding vulnerable code areas through the cross-attention mechanism to generate more accurate repairs. To strengthen our cross-matching, we propose to learn a novel vulnerability query mask that greatly focuses on vulnerable code areas and integrate it into the cross-attention. Moreover, we also incorporate the vulnerability query mask into the self-attention to learn embeddings that emphasize the vulnerable areas of a program. Through an extensive evaluation using the real-world 5,417 vulnerabilities, our approach outperforms all of the baseline methods by 3.39%-33.21%. The training code and pre-trained models are available at `https://github.com/AVR-VQR/VQR`.

1 INTRODUCTION

Software vulnerabilities are security flaws, glitches, or weaknesses found in software code that could lead to a severe system crash or be leveraged as a threat source by attackers (CSRC, 2020). According to National Vulnerability Database (NVD), the number of vulnerabilities discovered yearly has increased from 6,447 in 2016 to 20,156 in 2021 and 18,017 vulnerabilities have been found in 2022. This trend indicates more vulnerabilities are being discovered and released every year, meaning that there will be more workloads for security analysts to track down and patch those vulnerabilities. In particular, it may take 58 days on average to fix a vulnerability based on vulnerability statistics reported in 2022 (Edgescan, 2022). Recently, Deep Learning (DL)-based approaches have been proposed to automate the vulnerability repair process by learning the representation of vulnerable programs and generating repair patches accordingly, which may potentially accelerate manual security analysis processes. Specifically, the transformer architecture has been widely adopted to generate accurate vulnerability patches that repair the vulnerable code automatically (Chen et al., 2022; Chi et al., 2022; Berabi et al., 2021; Fu et al., 2022). The attention-based transformer is shown to be more effective than RNNs because its self-attention mechanism learns global dependencies when scanning through each word embedding rather than processing input sequentially.

For the software vulnerability repair (SVR) problem, awareness and attention to the vulnerable code areas including vulnerable statements are crucially important. This further helps to guide an SVR model to emphasize and focus more on the vulnerable statements for producing better repairs. However, it is challenging because the vulnerable areas locate spatially in a source code. Toward this challenge, we observe that object detection in computer vision intuitively shares a similar concept to vulnerability repair because both approaches need to localize specific items in the input. Particularly, by linking the vulnerable code areas in a source code to the objects in an image, we hope to borrow the principles from the VIT-based objection detection approaches (Carion et al., 2020; Zhu et al., 2020; Wang et al., 2021a) to propose a novel solution for the SVR problem.

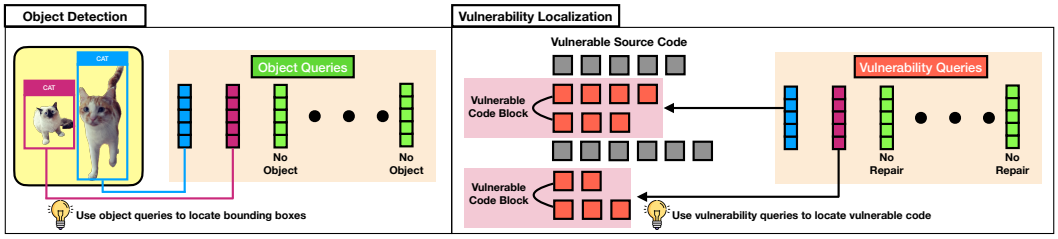

Figure 1: A high-level concept of using the principle of VIT-based object detection approaches (Carion et al., 2020; Zhu et al., 2020; Wang et al., 2021a) for vulnerability repairs. The VIT-based object detection approaches use object queries to locate bounding boxes, while we leverage vulnerability queries to locate vulnerable code tokens. Specifically, we aim to train the vulnerability queries to attend more to the vulnerable code tokens in the vulnerable code areas.

Specifically, our approach is inspired by the VIT-based approaches for object detection (Carion et al., 2020; Zhu et al., 2020; Wang et al., 2021a) where we connect detecting spatial objects in an image for predicting bounding boxes to localizing vulnerable code tokens in a source code for generating the repair tokens. Our model consists of a vulnerability repair encoder to produce code token embeddings for code tokens and a vulnerability repair decoder to generate repair tokens. Similar to the object queries in the VIT-based approaches for object detection aiming to attend to objects in an image for predicting the corresponding bounding boxes, we devise vulnerability queries (VQ) aiming to attend to the vulnerable areas in a source code for predicting repair tokens. Additionally, the cross-attention mechanism employed in the vulnerability repair decoder assists the VQs in cross-matching and paying more attention to the vulnerable code areas.

Furthermore, for real-world software vulnerabilities, not all code tokens in a source code are considered vulnerable, meaning that only some of the code tokens are likely to be more vulnerable than the others (Nguyen et al., 2021; Fu & Tantithamthavorn, 2022). To strengthen the attention of the VQs to the vulnerable code areas or code tokens, we train an additional model to learn a vulnerability mask. Specifically, given a source code, the vulnerability mask has significantly higher vulnerability scores for the vulnerable code tokens. We then apply the vulnerability mask to both the vulnerability repair encoder/decoder to enrich our approach, named *Vulnerability Query based Software Vulnerability Repair* (VQR). Finally, Figure 1 presents a conceptual overview of our VQR.

In summary, our contributions are **(i)** a novel vulnerability repair framework based on object detection that uses vulnerability queries to generate repair patches; **(ii)** a novel vulnerability query mask that facilitates the repair model to locate vulnerable code tokens more accurately during vulnerability query; **(iii)** a comprehensive evaluation of our proposed approach against other automated vulnerability repair approaches using a benchmark dataset including real-world vulnerabilities.

2 Related Work

Automated Vulnerability Repair (AVR) is a task that uses machine learning models to generate repair patches for vulnerable programs. RNN-based models such as SequenceR (Chen et al., 2019) have been proposed to encode the vulnerable programs and decode corresponding repairs sequentially. SequenceR used Bi-LSTMs as encoders with unidirectional LSTMs to generate repairs. Recently, attention-based Transformer models have been leveraged in the AVR domain, which was shown to be more accurate than RNNs. For instance, VRepair (Chen et al., 2022) relied on an encoder-decoder Transformer with transfer learning using the bug-fix data to boost the performance of the vulnerability repair on C/C++ programs. SeqTrans (Chi et al., 2022) constructed code sequences by considering data flow dependencies of programs and leveraged an identical architecture as VRepair. On the other hand, Berabi et al. (2021) proposed to use a T5 model pre-trained on natural language corpus (i.e., T5-large (Raffel et al., 2020)) to fix JavaScript programs and Fu et al. (2022) utilized a T5 model pre-trained on source code (i.e., CodeT5 (Wang et al., 2021b)) to repair C/C++ programs. Additionally, Mashhadi & Hemmati (2021) applied the CodeBERT (Feng et al., 2020) model to repair Java bugs. Those large pre-trained language models have demonstrated

strong improvement over RNNs and non-pretrained transformers because the pre-training steps help the models gain better initial weights for the vulnerability repair downstream task than training from scratch. Different from the sequence-based methods mentioned above, Dinella et al. (2020) proposed to learn the graph transformation based on the Abstract Syntax Tree (AST) of source code, which used GNNs to represent the program and LSTMs to generate repairs for JavaScript programs.

Vulnerable Function — CWE-787 (Out-of-bounds Write)

```
41  41  GPMF_ERR IsValidSize(GPMF_stream *ms, uint32_t size)
42  42  {
43  43      if (ms)
44  44      {
-   45          int32_t nestsize = (int32_t)ms->nest_size[ms->nest_level];
46  46          if (nestsize == 0 && ms->nest_level == 0)
47  47              nestsize = ms->buffer_size_longs;
50  50      }
51  51      return GPMF_ERROR_BAD_STRUCTURE;
52  52  }
```

Repaired Function

```
41  41  GPMF_ERR IsValidSize(GPMF_stream *ms, uint32_t size)
42  42  {
43  43      if (ms)
44  44      {
+   45          uint32_t nestsize = (uint32_t)ms->nest_size[ms->nest_level];
46  46          if (nestsize == 0 && ms->nest_level == 0)
47  47              nestsize = ms->buffer_size_longs;
50  50      }
51  51      return GPMF_ERROR_BAD_STRUCTURE;
52  52  }
```

Subword-tokens of the vulnerable function $x_i = [t_1, \ldots, t_n]$

```
['GP', 'MF', '_', 'ERR', 'IsValid', 'Size', '(', 'GP', 'MF', '_', 'stream', '*', 'ms', ',', 'uint', '32', '_', 't', 'size', ')', '{', 'if', '(', 'ms', ')', '{', 'int', '32', '_', 't', 'nest', 'size', '=', '(', 'int', '32', '_', 't', ')', 'ms', '->', 'nest', '_', 'size', '[', 'ms', '->', 'nest', '_', 'level', ];', 'if', '(', 'nest', 'size', '==', '0', '&&', 'ms', '->', 'nest', '_', 'level', '==', '0', ')', 'nest', 'size', '=', 'ms', '->', 'buffer', '_', 'size', '_', 'l', 'ongs', ';', 'if', '(', 'size', '+', '2', '<=', 'nest', 'size', ')', 'return', 'GP', 'MF', '_', 'OK', ';', '}', 'return', 'GP', 'MF', '_', 'ERROR', '_', 'BAD', '_', 'STRUCT', 'URE', ';', '}']
```

Subword-tokens of the vulnerability repair $y_i = [r_1, \ldots, r_k]$

```
['ms', ')', '{', 'uint', '32', '_', 't', 'nestsize', '=', '(', 'uint', '32', '_', 't', ')', 'ms', '->']
```

Figure 2: (CWE-787 Out-of-bounds Write) A real-world example (GoPro, 2019) of vulnerability in a C function is caused by an inappropriate variable type definition, which could lead to serious security breaches or system crashes. The left column presents the vulnerable function where below are sub-word tokens x_i used as input for our repair model. It can be seen that only some of the tokens highlighted in red (i.e., tokens corresponding to Line 45) are vulnerable. The right column presents the corresponding repaired function where below are sub-word tokens y_i as the repair patch output by our repair model.

3 Our Proposed Approach

3.1 Problem Statement

Assuming we have a source code data set consisting of vulnerable source code functions along with corresponding repair patches that repair the vulnerable parts of those functions. We denote the data set as $D = \{(x_1, y_1), ..., (x_N, y_N)\}$, where x_i is a vulnerable function and y_i is its repair patch. Note that each y_i is not a complete function but a patch used to repair the vulnerable part in the corresponding x_i as shown in Figure 2. The mapping between x_i and y_i has been completed by Chen et al. (2022) through parsing the code difference between the vulnerable and the fixed version of the source functions. In this paper, we leverage BPE algorithm (Sennrich et al., 2016) to tokenize x_i and consider x_i as a sequence of code tokens denoted as $x_i = [t_1, t_2, ..., t_n]$ where the code token $t_j, j = 1, ..., n$ could be a clean token or vulnerable token (i.e., the tokens highlighted in red in Figure 2). Similarly, a repair patch $y_i = [r_1, ..., r_k]$ where y_i consists of k number of repair tokens $r_j, j = 1, ..., k$. Each code token t_j and repair token r_j will be embedded into a vector for the model to learn its representation as detailed in Section 3.2. We define this problem as a sequence-to-sequence code generation task with an objective to capture vulnerable code tokens in x_i to generate corresponding repair patch y_i.

3.2 Vulnerability Query based Software Vulnerability Repair

Our approach is inspired by the VIT-based approaches (Carion et al., 2020; Zhu et al., 2020; Wang et al., 2021a) for object detection where we link detecting spatial objects in an image for predicting bounding boxes to localizing vulnerable code tokens in a source code for generating the repair tokens. Our model consists of an encoder to produce code token embeddings for code tokens and a decoder to generate repair tokens.

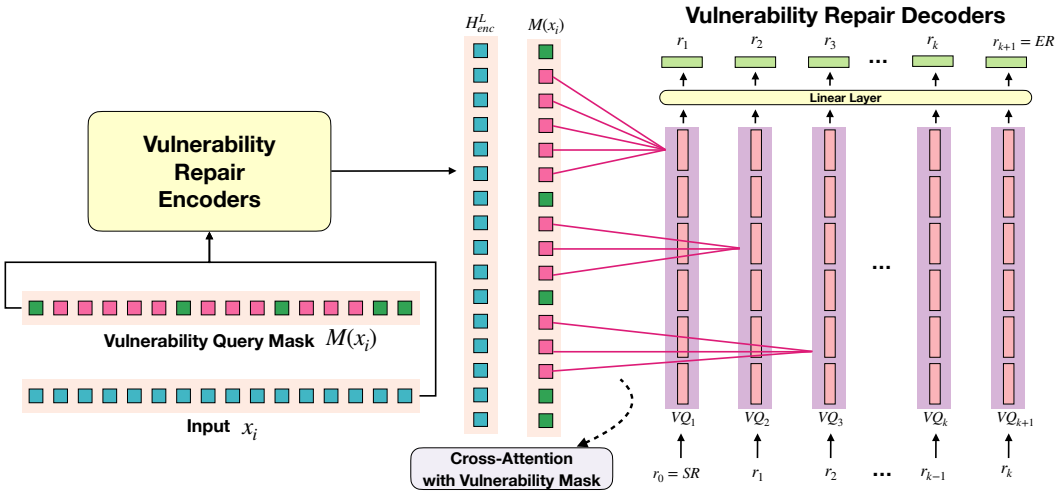

Figure 3: An overview architecture of our VQR approach. Input tokens $x_i = [t_1, ..., t_n]$ and a vulnerability query mask $M(x_i)$ are input to repair encoders that output the embeddings of input tokens H^L_{enc}, wherein $M(x_i)$ helps to emphasize the vulnerable embeddings. In repair decoders, each vulnerability query VQ_i is initialized from the previous repair token r_{i-1}, which is forwarded through multiple decoder layers followed by a linear layer to generate a repair token r_i. In each repair decoder, a cross-attention with $M(x_i)$ to emphasize vulnerable embeddings is leveraged to cross-match VQ_i and H^L_{enc} and generate repairs corresponding to the vulnerable tokens.

Both encoder and decoder are developed based on the transformer architecture (Vaswani et al., 2017). The main component of the encoder is multi-head self-attentions with the aim to learn code token embeddings. Similar to DeTR (Carion et al., 2020), the decoder utilizes both multi-head self-attentions and cross-attentions. The purpose of the cross-attentions is to cross-match vulnerability queries and their corresponding vulnerable code tokens in a vulnerable area. Ideally, when vulnerability queries achieve good matches with their vulnerable code tokens, they possess sufficient information to generate repair tokens.

Additionally, to orient the matching process for attending more to vulnerable code tokens inside a source code, we propose to learn a vulnerability query mask and apply it to both the encoder self-attention and decoder cross-attention mechanism. Particularly, we rely on the information of vulnerable tokens to train an additional model that outputs the possibility of a code token being a vulnerable token. We then base on these vulnerable scores to conduct a vulnerability query mask.

In what follows, we present the technicality of the vulnerability repair encoder, the vulnerability repair decoder, and how to conduct and incorporate vulnerability query masks into our framework.

3.2.1 Vulnerability Repair Encoder

The purpose of the encoder is to produce code token embeddings for a given source code. Each token is embedded into a vector in $\mathbb{R}^{d=768}$ by an embedding layer and input to the first encoder block. A stack of encoder blocks is leveraged to encode the representation for an embedded sequence through their self-attention layers followed by feed-forward neural networks, and each encoder block can be described as follows:

$$A^t = LN(MultiAttn(H^{t-1}_{enc})) + H^{t-1}_{enc}$$
$$H^t_{enc} = LN(FFN(A^t) + A^t)$$

where the hidden states from the previous encoder block H^{t-1}_{enc} forwards through a multi-head self-attention $MultiAttn$ followed by a 2-layer feed-forward neural network FFN, and a layer normalization LN. The process will iterate until we obtain the last encoder hidden states H^L_{enc} to represent the vulnerable function. Here we note that L is the number of encoder blocks applied and H^L_{enc} contains the code token embeddings.

3.2.2 VULNERABILITY REPAIR DECODER

Input to the vulnerability repair decoder is the vulnerability queries (VQ), each of which aims to match and capture information of vulnerable code tokens in a given source code.

The first VQ embeddings $Q^0 = [q_1^0, ..., q_k^0]$ are conducted and fed through several following decoder blocks. In each block, we apply both multi-head self-attention and cross-attention as follows:

$$\hat{Q}^t = LN(MultiAttn(Q^{t-1})) + Q^{t-1}$$
$$A_{cross}^t = LN(CrossAttn(\hat{Q}^t, H_{enc}^L)) + Q^{t-1}$$
$$Q^t = LN(FFN(A_{cross}^t) + A_{cross}^t)$$

where H_{enc}^L is the encoder output.

It is worth noting that the cross-attention $CrossAttn$ assists us in cross-matching the vulnerability query embeddings $Q^t = [q_0^t, ..., q_k^t]$ and the code token embeddings. If trained appropriately, the vulnerability query embeddings $q_0^t, ..., q_k^t$ attend and emphasize more the vulnerable code token embeddings in the vulnerable function, which finally contain sufficient information to generate the repair tokens.

Eventually, we obtain the output VQ embeddings $Q^U = [q_0^U, ..., q_k^U]$ where U is the number of the decoder blocks applied. On top of these VQ embeddings, we predict the repair tokens $r_1, ..., r_k$. Specifically, we dedicate a linear layer on each VQ embedding $q_0^U, ..., q_k^U$ and aim to predict $r_1, ..., r_k$ and $r_{k+1} = ER$ (i.e., the end repair token) by maximizing the likelihood with respect to a mini-batch of x_i:

$$p(y_i \mid x_i) = p(r_1, ..., r_k \mid t_1, ..., t_n) = \prod_{j=0}^{k} p(r_{j+1} \mid q_j^U) \tag{1}$$

where $x_i = [t_1, ..., t_n]$ is the source code and $y_i = [r_1, ..., r_k]$ is the corresponding repair patch.

The next arising question is how to initialize the first VQ embeddings $Q^0 = [q_0^0, ..., q_k^0]$. Different from VIT-based object detection approaches (Carion et al., 2020; Zhu et al., 2020; Wang et al., 2021a), we do not initialize the first VQ embeddings $Q^0 = [q_0^0, ..., q_k^0]$ randomly. Indeed, we initialize $Q^0 = [q_0^0, ..., q_k^0]$ more informatively by setting $q_0^0 = SR$ (i.e., the specific embedding for the start repairing token), $q_j^0 = r_j, j = 1, ..., k$. By this informative initialization, we cast the vulnerability repair problem to the source code to repair-patch generation task.

The inference process is hence very natural. Given a source code $x_i = [t_1, ..., t_n]$, we pass it through the vulnerability repair encoder to work out the encoder output H_{enc}^L. We start with the first VQ embedding $q_0^0 = SR$ and feed to the vulnerability repair decoder to generate the first repair token r_1. We then set VQ embedding $q_1 = r_1$ and feed it to the vulnerability repair decoder to generate the second repair token r_2. We repeat this process until reaching the ER token.

As mentioned before, the key factor to the success of our approach is how to accurately cross-match between the vulnerability queries and the vulnerable code tokens of a given source code. Currently, we expect that the cross-attention mechanism guided by maximizing the likelihood in Eq. (1) supports us in realizing this. To further strengthen the cross-matching, we learn a vulnerability query mask that highly focuses on the vulnerable code tokens and then apply it to the encoder self-attention and the decoder cross-attention mechanism.

3.2.3 LEARNING AND INCORPORATING VULNERABILITY QUERY MASK

In what follows, we present how to learn a vulnerability query mask and then apply it to our model.

Learning vulnerability query mask. We note that for our dataset $D = \{(x_1, y_1), ..., (x_N, y_N)\}$, each source code representation $x_i = [t_1, ..., t_n]$ is a sequence of code token embeddings in which we know exactly the vulnerable scope or information if a code token t_j belongs to a vulnerable statement. In other words, we also possess the token-level vulnerable label $v_i = [u_1, ..., u_n]$ wherein $u_j = 1$ means that the code token t_j belongs to a vulnerable statement and otherwise. For example,

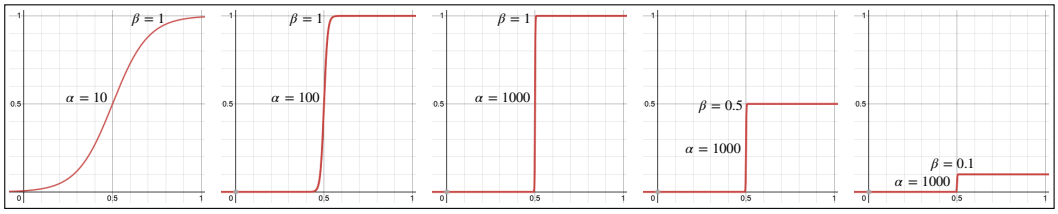

Figure 4: The plots of the vulnerability mask transformation to see how α, β control the sharpness. It can be seen that α controls the sharpness (i.e., how fast the curve gets saturated), while β controls the gap between vulnerable and non-vulnerable scores.

in the source code presented in Figure 2, the code tokens highlighted in red are the vulnerable code tokens labelled 1.

We now take advantage of this crucial information to learn vulnerability query masks. Basically, we train an additional model to predict the vulnerability query masks. Specifically, we leverage a pre-trained CodeBERT (Feng et al., 2020) model in learning the vulnerability query masks. Each $x_i = [t_1, ..., t_n]$ is forwarded through 12 layers of the BERT architecture. We then use a global max pooling layer and a sigmoid activation to obtain the probability mask $m(x_i)$ and minimize the following loss with respect to a mini-batch of x_i

$$H(x_i, v_i) = \sum_{j=1}^{n} \left[u_j \log m_j(x_i) + (1 - u_j) \log \left(1 - m_j(x_i)\right) \right] \tag{2}$$

Finally, to sharpen the vulnerability query mask, we apply the following transformation

$$M(x_i) = \frac{\beta}{1 + \exp\{-\alpha(m(x_i) - 0.5)\}}$$

where $\alpha > 0$ and $\beta > 0$ are two parameters to control the sharpness of the vulnerability query mask.

In Figure 4, we visualize how α and β affect the vulnerability query masks. It can be seen that α controls the sharpness (i.e., how fast the curve gets saturated), while β controls the gap between vulnerable and non-vulnerable scores.

Applying vulnerability query mask to our model. We incorporate our vulnerability query mask (VQM) into both encoder output and the cross-attention. For the encoder, we apply as follows:

$$A^t = LN\big(MultiAttn(H_{enc}^{t-1}) + M(x_i) \otimes MultiAttn(H_{enc}^{t-1})\big) + H_{enc}^{t-1}$$

$$H_{enc}^t = LN(FFN(A^t) + A^t)$$

where $\otimes$ is the element-wise product which returns $[M_j(x_i)B_j^t]_{j=1}^n$ with $B^t = MultiAttn(H_{enc}^{t-1})$.

For the cross-attention in the decoder, we apply as follows:

$$\hat{Q}^t = LN(MultiAttn(Q^{t-1})) + Q^{t-1}$$

$$A_{cross}^t = LN\big(CrossAttn(\hat{Q}^t, H_{enc}^L + M(x_i) \otimes H_{enc}^L)\big) + Q^{t-1}$$

$$Q^t = LN(FFN(A_{cross}^t) + A_{cross}^t)$$

Finally, the entire framework of our approach encapsulated the vulnerability repair encoder, vulnerability repair decoder, and how to incorporate vulnerability masks are summarized in Figure 3.

4 Experiments

4.1 Baseline approaches

We compare our proposed method VQR with the following baselines:

- **VRepair**: The vanilla transformer architecture is first pre-trained on a bug-fixing corpus in a supervised manner, VRepair (Chen et al., 2022) is then fine-tuned to fulfil the vulnerability repair task which shares a similar nature to the bug-fixing task.
- **VulRepair**: The transformer encoder-decoder approach learns the representation of vulnerable functions and generates the corresponding repair patches. VulRepair (Fu et al., 2022) relies on the CodeT5 (Wang et al., 2021b) model pre-trained through multi-task learning on the Codesearchnet (Husain et al., 2019) dataset to learn the characteristics of programming languages.
- **TFix**: Similar to VulRepair (Fu et al., 2022), TFix (Berabi et al., 2021) relies on a transformer encoder-decoder model, however, pre-trained on natural language using denoising objectives (Raffel et al., 2020) (i.e., T5-large).
- **SequenceR**: An RNN-based approach that learns the representation of vulnerable functions token by token, SequenceR (Chen et al., 2019) leverages bi-directional LSTM encoders with unidirectional LSTM decoders.
- **CodeBERT**: A bi-directional self-attention pre-trained on the Codesearchnet (Husain et al., 2019) dataset using masked language modelling and replaced token detection objectives (Feng et al., 2020). Mashhadi & Hemmati (2021) leveraged CodeBERT (Feng et al., 2020) for automated program repair of Java bugs and presented substantial improvement over RNN-based models.
- **GraphCodeBERT**: An extensive version of CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021) considers the Data Flow Graph (DFG) of code using the graph-guided masked attention during both pre-training and fine-tuning stages.

4.2 EXPERIMENTAL DATASET

We use the same experimental dataset provided by Chen et al. (2022) to evaluate our approach. The dataset consists of Big-Vul (Fan et al., 2020) and CVEfixes (Bhandari et al., 2021) vulnerability fix corpus written in C/C++. The Big-Vul dataset was collected from 348 open-source GitHub projects by crawling the Common Vulnerabilities and Exposures (CVE) database. In total, Big-Vul contains 3,754 code vulnerabilities from 2002 to 2019. On the other hand, the CVEfixes dataset was constructed similarly to the Big-Vul, which consists of 5,365 vulnerabilities collected from 1,754 projects from 1999 to 2021. Specifically, we leverage both datasets pre-processed by Chen et al. (2022) and obtain 5,417 samples spanning 2,095 different vulnerabilities (i.e., CVE-ID) after dropping null and duplicate samples.

4.3 PARAMETER SETTING

We split the data into 70% for training, 10% for validation, and 20% for testing. We follow the best hyperparameter setting as reported by the original authors to obtain the best results of each baseline method. We use a pre-trained T5 model provided by Wang et al. (2021b), which was pre-trained using multiple denoising objectives related to programming languages to strengthen its ability to understand the semantics of programming languages and generate meaningful output for source code-related tasks. More details of the model architecture we used can be found in this configuration file [1]. We use an AdamW optimizer and a linear training schedule with warm-up steps set to 10% of total training steps. More details of the hyperparameter settings and the model training scheme for our approach during both pre-training and fine-tuning can be found in Table 1.

Table 1: Training scheme of our VQR approach.

Training	Data	Seq_{enc}	Seq_{dec}	Optim	Sch.	LR	Grad Clip	Bhz	Epo
Pre-train	Bug Fix	512	256	AdamW	Linear	1e-4	1.0	8	75
Fine-tune	Vul Fix	512	256	AdamW	Linear	1e-4	1.0	8	75

[1] https://huggingface.co/Salesforce/codet5-base/blob/main/config.json

4.4 MODEL TRAINING

Given that the existing vulnerability repair dataset only contains limited samples, pre-training on a larger bug fix dataset can further enhance the performance of a vulnerability repair model as demonstrated by Chen et al. (2022). The intuition is that the software vulnerability is a sub-domain of the software defect (i.e., bugs) domain which increases the transferability between the two tasks. Thus, for each model including our VQR, we first pre-train on the bug fix dataset provided by Chen et al. (2022), which consists of 23,607 samples to obtain more meaningful pre-trained weights for the vulnerability repair downstream task. Note that the bug fix dataset is not overlapping with our experimental dataset introduced in Section 4.2. We train our model through specific epochs as reported in Table 1 and select the best model based on the lowest CE loss on the validation set. We run our experiments on a Linux machine with an AMD Ryzen 9 5950X with 16C/32T, 64 GB of RAM, and an NVIDIA RTX3090 GPU with 24GB of RAM.

4.5 EXPERIMENTAL RESULTS

4.5.1 MAIN RESULTS

We conduct our experiment using the dataset described in Section 4.2 and compare our proposed method with the baselines introduced in Section 4.1. We leverage the same percentage of perfect prediction (%PP) evaluation measure as used by Chen et al. (2022) and Fu et al. (2022). %PP is computed as the total correct predictions divided by the total testing samples, specifically, a prediction is considered correct if any of the beam search outputs is exactly the same as the ground-truth repair. During beam search, We use $beam \in [1, 3, 5]$ to evaluate all of the methods. Such beam settings lead to fewer repair candidates generated by the models, which would be more practical in real-world scenarios so developers will not need to inspect many repair candidates. The experimental results are shown in Table 2. Our method outperforms all baselines regardless of the number of beams. When comparing only the top-1 repair candidates (i.e., $beam = 1$), our VQR is 3.39%-17.52% better than large pre-trained transformer approaches (i.e., VulRepair and TFix) while our method is 23.58%-24.68% better when comparing with BERT-based approaches (i.e., CodeBERT and GraphCodeBERT). These results confirm that our proposed method of cross-matching vulnerability queries with vulnerable code tokens can help the model encode a more meaningful representation for a vulnerable function and decode the corresponding repair more accurately.

Table 2: (Main results) The comparison between our VQR approach and other baselines. Accuracy presented in percentage.

Methods	Beam=1	Beam=3	Beam=5
VQR(**Ours**)	**33.21**	**44.04**	**46.06**
VulRepair	29.82	39.72	43.67
TFix	15.69	27.71	31.28
GraphCodeBERT	9.63	17.52	21.83
CodeBERT	8.53	14.22	16.88
VRepair	5.96	10.73	13.76
SequenceR	0.0	0.64	1.01

4.5.2 ABLATION STUDY

(1) Study the effectiveness of our proposed vulnerability query mask. We aim to study the effectiveness of the vulnerability query mask applied in our VQR, thus, we compare our proposed method with other variants as follows:

- **Perfect Vulnerability Masking in Encoders and Decoders**: This method uses an identical architecture as our VQR, however, the perfect vulnerability masks (i.e., the exact location of each vulnerable token) are provided instead of predicted by a localization model.
- **Vulnerability Query Masking in Encoders**: This method only applies vulnerability query masks on the self-attention output of each encoder to help the model focus more on vulnerable tokens when encoding the representations for a vulnerable function.

- **Vulnerability Query Masking in Decoders**: This method only applies vulnerability query masks on the decoder cross-attention when cross-matching vulnerability queries and vulnerable code tokens to support the model to focus more on vulnerable tokens when generating repair tokens.
- **Without Vulnerability Query Masking**: This method is a plain transformer encoder-decoder architecture that applies no vulnerability query mask.

The experimental results are shown in Table 3. It can be seen that applying the vulnerability query mask is beneficial for both encoder self-attention and decoder cross-attention Specifically, it enhances the %PP by 2.93% when applied to encoders while gaining a %PP of 3.02% when applied to decoders. While the vulnerability query mask benefits both encoders and decoders, our proposed method to leverage the mask on both sides achieves better results for $beam \in [1, 3]$. These results confirm that using our mask on both encoders and decoders is more beneficial than using it on either side. Last but not least, the perfect vulnerability query mask achieves the highest %PP, highlighting that our vulnerability query mask is significant information that helps the model locate vulnerable tokens and generate more accurate corresponding repairs.

Table 3: (Ablation results) The comparison between our proposed method and four other variants. Accuracy presented in percentage.

Methods	Beam=1	Beam=3	Beam=5
Perfect Mask Encoder + Perfect Mask Decoder	*33.76*	*44.31*	*46.88*
Vul Mask Encoder + Vul Mask Decoder	**33.21**	**44.04**	46.06
Vul Mask Encoder	32.75	43.49	**46.15**
Vul Mask Decoder	32.84	43.85	45.69
w/o Vul Mask	29.82	39.72	43.67

(2) Study the effectiveness of pre-training on bug fix corpus. We aim to study whether pre-training on a larger bug fix corpus support AVR models to perform better vulnerability repairs. To this end, we directly train each baseline approach and our VQR on the vulnerability repair data without pre-training on the bug fix data. The experimental results are shown in Table 4. These results correspond to the finding by Chen et al. (2022) that the knowledge from the general bug fix corpus can be transferred to benefit the performance of AVR models. Last but not least, our VQR method still outperforms all baseline approaches.

Table 4: The comparison between our VQR approach and other baselines, wherein all of the methods are not pre-trained on the bug-fix data set provided by Chen et al. (2022). Accuracy presented in percentage.

Methods	Beam=1	Beam=3	Beam=5
VQR	**5.32**	**8.81**	**9.72**
VulRepair	4.13	6.06	7.43
TFix	2.75	4.4	4.68
GraphCodeBERT	2.57	4.13	5.23
CodeBERT	1.56	2.29	2.75
VRepair	0.09	0.55	0.92
SequenceR	0	0	0

5 CONCLUSION

In this paper, we have introduced a new AVR method inspired by VIT-based approaches for object detection to enhance awareness and attention to vulnerable code areas in a vulnerable function for producing better repairs. In our repair model, we cross-match vulnerability queries and their corresponding vulnerable code areas and their corresponding repairs via the cross-attention mechanism. To strengthen such cross-matchings, we propose to learn a vulnerability query mask that highly focuses on vulnerable code areas and incorporate it into the cross-attention. Additionally, we also apply the vulnerability query mask in the self-attention of encoders to help our model focus more on vulnerable code tokens when learning the embeddings of each token. Through an extensive evaluation of 5,417 real-world vulnerabilities, our approach outperforms all of the baseline approaches.

REFERENCES

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *the Proceedings of the International Conference on Machine Learning (ICML)*, pp. 780–791. PMLR, 2021.

Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *the Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 30–39, 2021.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *the Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 213–229. Springer, 2020.

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

Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering (TSE)*, 2022. doi: 10.1109/TSE.2019.2940179. URL `https://arxiv.org/pdf/2104.08308`.

Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. Seqtrans: Automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering (TSE)*, 2022.

CSRC. Definition of software vulnerability. `https://csrc.nist.gov/glossary/term/software_vulnerability`, 2020.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *the Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

Edgescan. 2022 vulnerability statistic report. `https://www.edgescan.com/2022-vulnerability-statistics-report-lp/`, 2022.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *the Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 508–512, 2020.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *the Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1536–1547, 2020.

Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *the Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, 2022.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dhung Dinh. Vulrepair: A t5-based automated software vulnerability repair. In *the Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

GoPro. An example software vulnerability from gopro systems. `https://github.com/gopro/gpmf-parser/commit/341f12cd5b97ab419e53853ca00176457c9f1681`, 2019.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. In *the Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

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

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

Van Nguyen, Trung Le, Olivier de Vel, Paul Montague, John Grundy, and Dinh Phung. Information-theoretic source code vulnerability highlighting. In *the Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, 2021.

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

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *the Proceedings of the Association for Computational Linguistics (ACL)*, pp. 1715–1725, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *the Proceedings of the Advances in neural information processing systems (NIPS)*, volume 30, 2017.

Yingming Wang, Xiangyu Zhang, Tong Yang, and Jian Sun. Anchor detr: Query design for transformer-based object detection. *arXiv preprint arXiv:2109.07107*, 2021a.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *the Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 8696–8708, 2021b.

Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable detr: Deformable transformers for end-to-end object detection. In *the Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.