# Vision Transformer-Inspired Automated Vulnerability Repair

MICHAEL FU, Monash University, Australia
VAN NGUYEN, Monash University, Australia
CHAKKRIT TANTITHAMTHAVORN*, Monash University, Australia
DINH PHUNG, Monash University, Australia
TRUNG LE, Monash University, Australia

Recently, automated vulnerability repair (AVR) approaches have been widely adopted to combat increasing software security issues. In particular, transformer-based encoder-decoder models achieve competitive results. While vulnerable programs may only consist of a few vulnerable code areas that need repair, existing AVR approaches lack a mechanism guiding their model to pay more attention to vulnerable code areas during repair generation. In this paper, we propose a novel vulnerability repair framework inspired by the Vision Transformer (VIT)-based approaches for object detection in the computer vision domain. Similar to the object queries used to locate objects in object detection in computer vision, we introduce and leverage vulnerability queries (VQs) to locate vulnerable code areas and then suggest their repairs. In particular, we leverage the cross-attention mechanism to achieve the cross-match between VQs and their corresponding vulnerable code areas. To strengthen our cross-match and generate more accurate vulnerability repairs, we propose to learn a novel vulnerability mask and integrate it into decoders' cross-attention, which makes our VQs pay more attention to vulnerable code areas during repair generation. In addition, we incorporate our vulnerability mask into encoders' 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 AVR baseline methods by 2.68%-32.33%. Addtionally, our analysis of the cross-attention map of our approach confirms the design rationale of our vulnerability mask and its effectiveness. Finally, our survey study with 71 software practitioners highlights the significance and usefulness of AI-generated vulnerability repairs in the realm of software security. The training code and pre-trained models are available at https://github.com/awsm-research/VQM.

CCS Concepts: • **Software and its engineering**; • **Security and privacy → Software and application security**;

Additional Key Words and Phrases: Software Security, Automated Vulnerability Repair



## 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 the 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

*The Corresponding Author

Authors' addresses: Michael Fu, Monash University, Clayton, Australia, yeh.fu@monash.edu; Van Nguyen, Monash University, Clayton, Australia, van.nguyen1@monash.edu; Chakkrit Tantithamthavorn, Monash University, Clayton, Australia, chakkrit@monash.edu; Dinh Phung, Monash University, Clayton, Australia, dinh.phung@monash.edu; Trung Le, Monash University, Clayton, Australia, trunglm@monash.edu.



1049-331X/2023/11-ART $15.00
https://doi.org/10.1145/3632746

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 [Berabi et al. 2021; Chen et al. 2022; Chi et al. 2022; 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.

In the automated vulnerability repair (AVR) problem, a deep learning model consists of encoders to encode the code representations of the vulnerable function and the decoders generate repair code for vulnerable code areas in the function. Commonly, vulnerabilities in a function are caused by a few vulnerable code areas, hence previous studies have proposed various techniques to localize vulnerable code areas in a vulnerable function [Ding et al. 2022; Fu and Tantithamthavorn 2022b; Li et al. 2021; Nguyen et al. 2021]. For instance in Figure 1 and 2, the decoders only need to generate the repair code for specific vulnerable code areas. Thus, awareness and attention to the vulnerable code areas including vulnerable statements are crucially important. This further helps to guide an AVR model to emphasize and focus more on the vulnerable statements for producing better repairs. However, existing AVR approaches lack a mechanism to enhance awareness of vulnerable code areas during the vulnerability repair process. It is also challenging because vulnerable code areas can appear in different spatial locations. 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; Wang et al. 2021b; Zhu et al. 2020] to propose a novel solution for the AVR problem.

In this paper, we propose an AVR approach that can guide the encoders and decoders to focus more on vulnerable code areas during the repair process. In particular, our approach is inspired by the VIT-based approaches for object detection [Carion et al. 2020; Wang et al. 2021b; Zhu et al. 2020]. Figure 3 presents our analogy between object detection from the computer vision domain and vulnerability repair from the NLP domain. We connect detecting spatial objects in an image for predicting bounding boxes to localizing spatial vulnerable code areas in a vulnerable function for generating the corresponding repair code. Our model consists of a vulnerability repair encoder to produce code token embeddings for vulnerable functions and a vulnerability repair decoder to generate repair patches. As presented in Figure 3, the object queries are used in the VIT-based approaches for object detection aiming to attend to objects in an image for predicting the corresponding bounding boxes, on the other hand, we devise vulnerability queries (VQs) aiming to attend to the vulnerable code blocks 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 blocks.

To further strengthen the attention of the VQs to the vulnerable code areas and facilitate the repair generation for vulnerable code areas, we train an additional model to learn a vulnerability mask (VMs). The VMs is a probability distribution used to emphasize vulnerable code areas in vulnerable functions. We then incorporate the VMs with the cross-attention in our repair decoder that guides our VQs to attend more to vulnerable code areas and generate corresponding repairs. In addition, we apply the VMs to the self-attention of our repair encoder to pay attention to vulnerable code areas when encoding the representations of vulnerable functions. We name our approach VQM - *Vulnerability Repair Through Vulnerability Query and Mask*.

We conduct an experiment and compare our VQM approach with six competitive AVR baseline approaches (i.e., VRepair [Chen et al. 2022], VulRepair [Fu et al. 2022], TFix [Berabi et al. 2021], CodeBERT [Feng et al. 2020], GraphCodeBERT [Guo et al. 2021], and SequenceR [Chen et al. 2019]). Through an extensive evaluation of our

**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 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         int32_t nestsize = (int32_t)ms->nest_size[ms->nest_level];
+  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', '_', 't', 'ongs', ';', 'if', '(', 'size', '+', '2', '<=', 'nest', 'size', ')', 'return', 'GP', 'MF', '_', 'OK', ';', '}', 'return', 'GP', 'MF', '_', 'ERROR', '_', 'BAD', '_', 'STRUCT', 'URE', ';', '}']

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

['<S2SV_ModStart>', 'ms', ')', '{', 'uint', '32', '_', 't', 'nestsize', '=', '(', 'uint', '32', '_', 't', '<S2SV_ModEnd>', ')', 'ms', '->']

**Note.**
**'ms', ')', '{'** are **context tokens before** the fix
**')', 'ms', '->'** are **context tokens after** the fix

*Those context tokens highlighted in blue are used to match the repair patches to the vulnerable parts in a vulnerable function.*

Fig. 1. (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 red tokens are vulnerable tokens; the green tokens are tokens used to repair; and the blue tokens are context tokens used to locate where the repair tokens should be implemented. It is worth noting that the model may not always match repairs to their correct locations when repeated context tokens are present. Nonetheless, in our experiments, we adopt the approach of Chen *et al.* Chen et al. [2022] by utilizing three context tokens, which effectively align all repairs in our studied dataset. 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.



approach on 5,417 C/C++ vulnerable functions involving 2,095 different vulnerabilities spanning from 1999 to 2021, we empirically evaluate our approach by answering the following two research questions:

**(RQ1) What is the accuracy of our VQM approach for generating software vulnerability repairs?**
**Results**. Among all approaches included in our experiment, our VQM approach achieves the best percentage of perfect predictions of 32%, 43%, and 45% respectively when using beam=1,3,5 during repair generation.

**(RQ2) What are the contributions of each component of our VQM approach?**
**Results**. Our method of applying vulnerability queries with vulnerability masks performs the best when compared with other variants in the ablation study. In addition, we find that using perfect vulnerability masks can achieve optimal performance, highlighting the effectiveness of our proposed vulnerability masks.

While our RQ1 and RQ2 delve into performance evaluations of our approach, the practical utility of AI-generated repairs for software developers remains a question unexplored. Consequently, in pursuit of this understanding, we address RQ3 by conducting a user study specifically aimed at gauging the perceptions of software developers possessing a security background towards AI-generated vulnerability repairs.

**Vulnerable Function — CWE-125 (Out-of-bounds Read)**

```
809 809 static inline unsigned short ReadPropertyUnsignedShort(const
        EndianType endian,
810 810 const unsigned char *buffer)
811 811 {
812 812   unsigned short
813 813     value;
814 814   if (endian == LSBEndian)
815 815     {
816 816       value=(unsigned short) ((buffer[1] << 8) | buffer[0]);
817 817       return((unsigned short) (value & 0xffff));
818 818     }
819 819   value=(unsigned short) ((((unsigned char *) buffer)[0] << 8) |
820 820     ((unsigned char *) buffer)[1]);
821 821   return((unsigned short) (value & 0xffff));
822 822 }
```

**Repaired Function**

```
809 809 static inline unsigned short ReadPropertyUnsignedShort(const
        EndianType endian,
810 810 const unsigned char *buffer)
811 811 {
812 812   unsigned short
813 813     value;
814 814   if (endian == LSBEndian)
815 815     {
  -   816       value=(unsigned short) ((buffer[1] << 8) | buffer[0]);
  -   817       return((unsigned short) (value & 0xffff));
  +   816       value=(unsigned short) buffer[1] << 8;
  +   817       value|=(unsigned short) buffer[0];
  +   818       return(value & 0xffff);
819 819     }
  -   819   value=(unsigned short) ((((unsigned char *) buffer)[0] << 8) |
  -   820     ((unsigned char *) buffer)[1]);
  -   821   return((unsigned short) (value & 0xffff));
  +   820   value=(unsigned short) buffer[0] << 8;
  +   821   value|=(unsigned short) buffer[1];
  +   822   return(value & 0xffff);
823 823 }
```

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

```
['static', 'inline', 'unsigned', 'short', 'Read', 'Property', 'Unsigned', 'Short', '(',
'const', 'End', 'ian', 'Type', 'endian', ',', 'const', 'unsigned', 'char', '*', 'buffer',
')', '{', 'unsigned', 'short', 'value', ';', 'if', '(', 'endian', '==', 'L', 'SB', 'Endian', ')',
'{', 'value', '=', '(', 'unsigned', 'short', ')', '(', '(', 'buffer', '[', '1', ']', '<<', '8', ')', '|',
'buffer', '[', '0', ']', ')', ';', 'return', '(', '(', 'unsigned', 'short', ')', '(', 'value', '&', '0',
'xffff', ')', ')', ';', '}', 'value', '=', '(', 'unsigned', 'short', ')', '(', '(', '(', '(', 'unsigned',
'char', '*', ')', 'buffer', ')', '[', '0', ']', '<<', '8', ')', '|', '(', '(', 'unsigned', 'char', '*', ')',
'buffer', ')', '[', '1', ']', ')', ';', 'return', '(', '(', 'unsigned', 'short', ')', '(', 'value', '&',
'0', 'xffff', ')', ')', ';', '}']
```

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

```
['<S2SV_ModStart>', 'unsigned', 'short', ')', '<S2SV_ModEnd>', 'buffer', '[', '1',
'<S2SV_ModStart>', ']', '<<', '8', ';', 'value', '|=', '(', 'unsigned', 'short', ')',
'<S2SV_ModEnd>', 'buffer', '[', '0', '<S2SV_ModStart>', '[', '0', ']',
'<S2SV_ModEnd>', ';', 'return', '(', '<S2SV_ModStart>', ';', 'return', '(',
'<S2SV_ModEnd>', 'value', '&', '0', 'xffff', '<S2SV_ModStart>', 'value', '&', '0',
'xffff', '<S2SV_ModEnd>', ')', ';', '}', '<S2SV_ModStart>', 'unsigned', 'short', ')',
'buffer', '<S2SV_ModEnd>', '[', '0', ']', '<S2SV_ModStart>', ']', '<<', '8', ';',
'value', '|=', '<S2SV_ModEnd>', '(', 'unsigned', 'short', '<S2SV_ModStart>',
'unsigned', 'short', ')', 'buffer', '[', '1', ']', ';', 'return', '<S2SV_ModStart>', '&', '0',
'xffff', ')', '<S2SV_ModEnd>', ';', '}']
```

Fig. 2. (CWE-125 Out-of-bounds Read) A real-world example [ImageMagick 2016] of vulnerability in a C function. In the vulnerable function on the left, the *value* is calculated using *(buffer[1] « 8) | buffer[0]* (i.e., line 816), which shifts the second byte of the buffer by 8 bits to the left and then tries to combine it with the first byte. This operation could lead to accessing memory beyond the buffer's bounds and result in undefined behavior. A similar vulnerability also occurs in the second vulnerable block (i.e., lines 819-821). In the repaired function, the problematic byte-order conversion operations in both vulnerable blocks have been restructured to ensure proper handling of byte manipulation and boundary checks. The key change is in the handling of the byte-order conversion, where instead of performing the bit shift and combination in a single step, the code is split into separate steps. This ensures that the operations involving byte manipulation are performed sequentially and within the buffer's boundaries.

**(RQ3) Are AI-generated vulnerability repairs perceived as useful by software developers?**

**Results**. Our survey study with 71 participants shows that 86% of participants perceive AI-generated vulnerability repairs as useful. In addition, 80% of them consider adopting AI-generated repairs if they are readily available and free of charge.

**The Novelty & Contributions** of this paper are as follows:

- A novel vulnerability repair framework based on object detection that uses vulnerability queries to generate repair patches;

- A novel vulnerability mask that facilitates the repair model to locate vulnerable code tokens more accurately during vulnerability query;
- A comprehensive evaluation of our proposed approach against other AVR approaches using a benchmark dataset including real-world vulnerabilities; and
- An ablation study to assess the effectiveness of each component in our proposed approach.
- A user study to assess the usefulness of AI-generated vulnerability repairs from software developers' perspective.

**Paper Organization.** Section 2 describes the problem definition along with the technical details of our proposed approach. Section 3 presents the experimental setup and Section 4 presents the experimental results. Section 5 presents an additional discussion of our approach. Section 7 presents the related works. Section 8 discloses the threats to validity. Section 9 draws the conclusions.

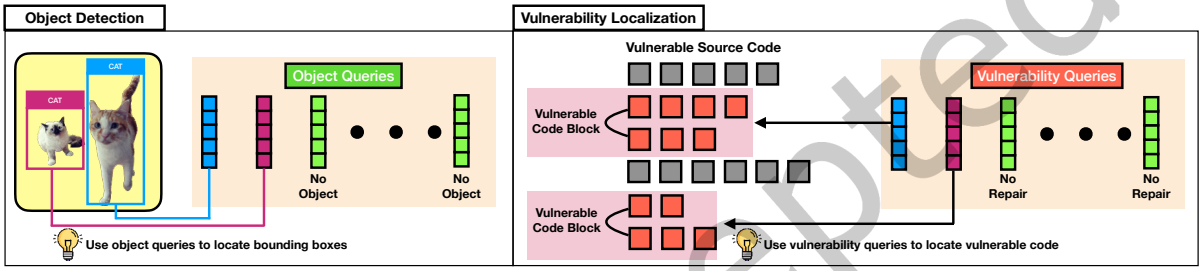


Fig. 3. Intuitively, not all code tokens in a program need to be repaired and the repair can be in multiple areas. Similarly, not all pixels in an image has objects and the objects can appear in multiple locations in an image. Thus, in object detection, object queries are used in VIT-based approaches [Carion et al. 2020; Wang et al. 2021b; Zhu et al. 2020] to predict bounding boxes and locate objects. With a similar principle of object detection, we leverage vulnerability queries to attend more to the vulnerable code tokens in the vulnerable code areas and generate repairs for them.

## 2 OUR PROPOSED APPROACH

### 2.1 Usage Scenario

Imagine a software development team that is particularly concerned about identifying and addressing vulnerabilities within their functions and has decided to leverage our proposed VQM approach. In practice, they analyze source code using static analysis tools like Cppcheck [Cppcheck [n. d.]] or deep learning-based vulnerability prediction tools [Fu et al. 2023c] to identify potential vulnerabilities. However, such tools cannot suggest vulnerability repairs. Thus, they leverage our VQM framework to obtain repair patches suggested by AI models. Finally, security experts on the team will validate the AI-generated patches before implementing them into their software system.

### 2.2 Problem statement

Similar to previous studies [Chen et al. 2022; Fu et al. 2022], we focus on function-level vulnerability repair, assuming all vulnerabilities can be resolved within the function-level scope. Our vulnerability repair approach can handle fixing multiple vulnerable parts in a vulnerable function and satisfy three key behaviors (i.e., add, delete, and replace) to fix vulnerabilities. The special tokens, "<S2SV_ModStart>" and "<S2SV_ModEnd>" are added into repair patches to determine the behavior of add, delete, or replace, as detailed in Section 3.2 in Chen *et al.*'s work [Chen et al. 2022]. In particular, the model will learn to generate three context tokens to match the repair patches back to the vulnerable function and implement the repairs. For instance, in the right part of Figure 1, the

first three repair tokens (“ms”, “)”, and “{”) and the last three repair tokens (“)”, “ms”, and “->”) are context tokens used to match the repair tokens to the vulnerable function. For simplicity, we use a vulnerable function with one vulnerable part as an example in Figure 1, however, the same repair manner can be repeated to fix vulnerable functions with multiple vulnerable parts.

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 1. The mapping between vulnerable functions, $x_i$, and repair patches, $y_i$, has been completed by Chen *et al.* Chen et al. [2022] through parsing the code difference between the vulnerable and the fixed version of the source functions from real-world vulnerability datasets [Bhandari et al. 2021; Fan et al. 2020]. 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 1). 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 2.3. 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 $y_i$.

As presented in Figure 1, the vulnerable function *IsValidSize* only consists of one vulnerable code area (i.e., statement 45). In particular, those repair tokens (i.e., $[r_1, ..., r_k]$) are only related to a few vulnerable tokens in the vulnerable function. Thus, it is a challenging task to generate repair tokens specifically for the vulnerable code area. To address this challenge, we propose vulnerability queries and masks to guide our repair model to pay more attention to the vulnerable code areas when generating their corresponding repair tokens. In what follows, we illustrate the technical details of our approach.

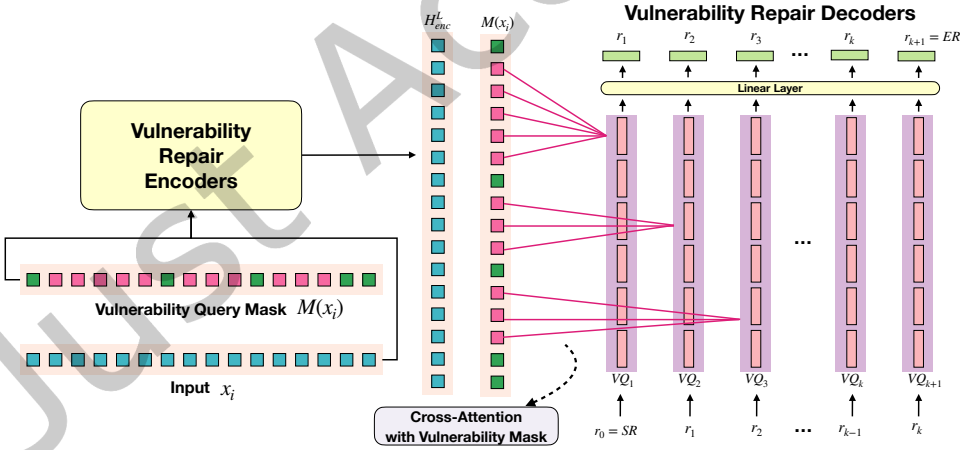


Fig. 4. An overview architecture of our VQM approach. Input tokens $x_i = [t_1, ..., t_n]$ and vulnerability masks $M(x_i)$ are input to encoders that output the embeddings of input tokens $H^L_{enc}$, where $M(x_i)$ helps to emphasize the vulnerable embeddings. In decoders, each vulnerability query $VQ_i$ 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 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.

## 2.3 Vulnerability Repair Via Vulnerability Query and Mask

Our approach is inspired by the VIT-based approaches [Carion et al. 2020; Wang et al. 2021b; Zhu et al. 2020] 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.

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 of learning 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 mask and apply it to both the encoder self-attention and decoder cross-attention mechanism. Particularly, we rely on the information on 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 mask.

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

*2.3.1 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:

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

where the hidden states from the previous encoder block $H_{enc}^{t-1}$ 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 $H_{enc}^L$ to represent the vulnerable function. Here we note that $L$ is the number of encoder blocks applied and $H_{enc}^L$ contains the code token embeddings.

*2.3.2 Vulnerability Repair Decoder.* Input to the vulnerability repair decoder is the vulnerability queries (VQ), each of which aims to match and capture information on 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; Wang et al. 2021b; Zhu et al. 2020], 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 reframe the vulnerability repair problem as the task of generating repair patches in the source code.

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 we reach 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 mask that highly focuses on the vulnerable code tokens and then apply it to the encoder self-attention and the decoder cross-attention mechanism.

*2.3.3 Learning and applying vulnerability mask.* In what follows, we present how to learn a vulnerability mask and then apply it to our model.

***Learning vulnerability mask.*** We note that for our dataset $D = \{(x_1, y_1), ..., (x_N, y_N)\}$, each vulnerable function $x_i = [t_1, ..., t_n]$ is a sequence of code token 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, in the source code presented in Figure 1, the code tokens highlighted in red are the vulnerable code tokens labelled 1.

We now take advantage of this crucial information to learn vulnerability masks. Basically, we train an additional model to predict the vulnerability masks. Specifically, we leverage a pre-trained CodeBERT [Feng et al. 2020] model in learning the vulnerability masks. Each $t_i$ in $x_i$ is embedded into a vector in $\mathbb{R}^{d=768}$ and 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 cross-entropy 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 mask, we apply the following transformation with a threshold value of 0.5:

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

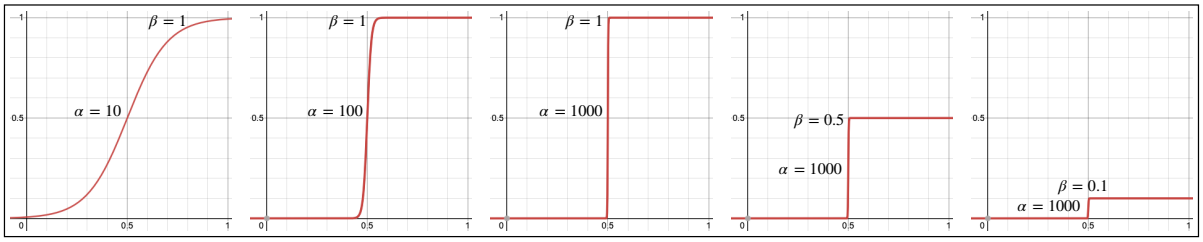


Fig. 5. The plots of the vulnerability mask transformation to see how $\alpha, \beta$ control the sharpness of vulnerability mask.

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

In Figure 5, we visualize how $\alpha$ and $\beta$ affect the vulnerability masks. It can be seen that $\alpha$ controls how fast the curve gets saturated and a large $\alpha$ value forms a line approaching vertical at $x = 0.5$. On the other hand, $\beta$ controls the gap between vulnerable and benign scores. We use a high value of $\alpha = 1000$ so the model prediction threshold can approach a common threshold value of 0.5. Tokens are classified as vulnerable by the model if their prediction probability surpasses 0.5; otherwise, they are categorized as benign. We chose a relatively low value of $\beta = 0.1$ to transform our vulnerability mask. Notably, a high $\beta$ value would result in higher masking values. However, such high masking values may interfere with the hidden representation of our main repair encoders and decoders. Thus, we select a low $\beta$ value to slightly adjust the self-attention weights and guide the repair model.

***Applying vulnerability mask to our model.*** Our vulnerability mask finds application in rectifying vulnerable code segments requiring either "replace" or "delete" actions for resolution. In scenarios involving replacement, our vulnerability mask emphasizes the vulnerable section to be replaced by repair tokens. In the context of deletion, the mask underscores the vulnerable code slated for removal. However, it is important to note that in scenarios requiring "addition," our vulnerability mask will not highlight anything, as no vulnerable code needs to be highlighted.

We incorporate our vulnerability mask into both the encoders' self-attention output and the decoders' 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 4.

## 3 EXPERIMENTAL DESIGN

### 3.1 Research Questions

In this paper, we aim to evaluate the effectiveness of our proposed VQM approach by answering our two research questions. In RQ1, we compare our approach with existing baseline methods for vulnerability repairs as described

in Section 3.2. In RQ2, we focus on studying the proposed components in our VQM approach and present an ablation study. Below, we present the motivation for our two research questions.

**(RQ1) What is the accuracy of our VQM approach for generating software vulnerability repairs?** Recently, transformer-based approaches have been leveraged for the automated vulnerability repair (AVR) problem [Chen et al. 2022; Chi et al. 2022; Fu et al. 2022]. While a vulnerable function may only consist of a few vulnerable codes to repair as shown in Figure 1, previous approaches can only implicitly learn the matching between vulnerable code areas and their repairs. On the other hand, we propose our VQM approach to explicitly guide the repair model to attend to those vulnerable areas and generate corresponding repairs. We formulate this RQ to assess the accuracy of VQM when comparing it to six other baseline approaches for the AVR introduced in Section 3.2.

**(RQ2) What are the contributions of each component of our VQM approach?** Our VQM involves two key components, i.e., vulnerability queries (VQs) and vulnerability masks (VMs), to help the transformer model focus more on vulnerable code areas during the generation of repairs. However, little is known about the effectiveness of applying our proposed components on top of the transformer encoder-decoder model. Thus, we formulate this RQ and conduct an ablation study regarding the proposed VQs and VMs to assess their effects on the transformer model.

## 3.2 Baseline approaches

Our approach is evaluated against the leading automated vulnerability repair (AVR) methods, namely VRepair [Chen et al. 2022] and VulRepair [Fu et al. 2022], to assess its effectiveness. Moreover, we consider seq2seq-based AVR methods like TFix [Berabi et al. 2021] and SequenceR [Chen et al. 2019] as baseline approaches for comparison. Additionally, we compare our approach with state-of-the-art pre-trained transformer models designed for source code, such as CodeBERT [Feng et al. 2020] and GraphCodeBERT [Guo et al. 2021], which are widely employed for addressing tasks related to source code. This comprehensive evaluation allows us to establish the advantages and novel contributions of our approach in the context of software vulnerability repair. The GPT2-CSRC approach is also included to compare our method with a decoder-only pre-trained model. In addition, we include automated program repair (APR) approaches such as CURE [Jiang et al. 2021] and DLFix [Li et al. 2020]. The details of baseline methods included in our evaluation are as follows:

- **VRepair**: A vanilla transformer architecture for the AVR task [Chen et al. 2022]. We replicate VRepair by following the instructions provided by Chen *et al.*to build and train the model.
- **VulRepair**: A T5-based approach [Fu et al. 2022] that relies on a large language model pre-trained on source code corpus [Wang et al. 2021a]. We reproduce VulRepair using the repository provided by Fu *et al.*.
- **TFix**: A T5-based approach [Berabi et al. 2021] that relies on a large language model pre-trained on natural language corpus [Raffel et al. 2020]. We reproduce TFix using the repository provided by Berabi *et al.*.
- **SequenceR**: An RNN-based approach with bi-directional LSTM encoders and unidirectional LSTM decoders [Chen et al. 2019]. We replicate SequenceR by following the instructions provided by Chen *et al.*to build and train the model.
- **CodeBERT**: A BERT-based large language model for source code [Feng et al. 2020], which has been leveraged to repair Java programs [Mashhadi and Hemmati 2021]. We reproduce CodeBERT using the repository provided by Feng *et al.*.
- **GraphCodeBERT**: An extensive version of CodeBERT by considering the Data Flow Graph (DFG) during training [Guo et al. 2021]. We reproduce GraphCodeBERT using the repository provided by Guo *et al.*.
- **GPT2-CSRC**: Utilizing a decoder-only model based on GPT-2 and a BPE tokenizer, GPT2-CSRC has undergone pre-training on an extensive dataset of approximately 17GB of C/C++ code. This dataset was curated from the top 10,000 most widely adopted Debian packages [Pearce et al. 2023].

- **CURE**: An automated program repair approach driven by code awareness [Jiang et al. 2021], this method harnesses the CoNut architecture [Lutellier et al. 2020] and integrates a decoder-only CodeGPT model alongside a BPE tokenizer. Moreover, during beam search, a code-aware search strategy is applied to enhance the output generation process.
- **DLFix**: DLFix is a program repair technique built upon Tree-LSTM [Li et al. 2020], a framework that utilizes abstract syntax trees (ASTs) as its input, standardized across variable names. To transform textual input into a vector space, Word2Vec embeddings [Mikolov et al. 2013] are used. Additionally, DLFix incorporates strategies such as patch re-ranking and program analysis filters to enhance the output generated by the model.

### 3.3 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.

Table 1. Training scheme of our VQM approach. Note. #: Scheme for training the mask prediction model; *: Scheme for training the repair model.

| Training | Data | $Seq_{enc}$ | $Seq_{dec}$ | Optim | Sch. | LR | Grad Clip | Bhz | Epo |
|---|---|---|---|---|---|---|---|---|---|
| #Pre-train | Bug Fix | 512 | N/A | AdamW | Linear | 1e-4 | 1.0 | 16 | 75 |
| #Fine-tune | Vul Fix | 512 | N/A | AdamW | Linear | 1e-4 | 1.0 | 16 | 75 |
| *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 |

### 3.4 Parameter Setting

We split the data into 70% for training, 10% for validation, and 20% for testing. We use a pre-trained T5 model provided by Wang et al. [2021a], which was pre-trained using multiple denoising objectives related to programming languages. The hyperparameter settings used to reproduce our mask model and repair model are presented in Table 1.

### 3.5 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 ours, 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 3.3. We report the details of our training settings in our replication package at https://github.com/awsm-research/VQM.

## 4 EXPERIMENTAL RESULTS

Table 2. (Main results) The comparison between our VQM approach and other baselines. Accuracy is presented in percentage. Beam=k shows the measure of %PP. We conducted the experiments five times with different random seeds and reported the mean performance plus minus standard deviation.

| Methods | Beam=1 | Beam=3 | Beam=5 |
|---|---|---|---|
| VQM(**Ours**) | **32.33±1.12** | **42.72±0.86** | **45.14±0.86** |
| VulRepair | 29.65±1.27 | 39.85±1.31 | 42.79±1.15 |
| TFix | 15.41±1.96 | 26.7±1.69 | 30±1.77 |
| GPT2-CSRC | 11.93±0.71 | 19.27±0.65 | 24.77±0.81 |
| CURE | 11.19±0.53 | 20.55±1.03 | 26.06±0.65 |
| GraphCodeBERT | 9.15±0.43 | 16.83±0.85 | 21.38±0.54 |
| CodeBERT | 7.47±0.61 | 13.69±0.37 | 16.85±0.17 |
| VRepair | 5.36±0.55 | 10.31±0.29 | 13.12±0.53 |
| DLFix | 0.51±0.08 | 1.05±0.15 | 1.53±0.23 |
| SequenceR | 0.0±0 | 0.44±0.13 | 0.53±0.27 |

***(RQ1) What is the accuracy of our VQM approach for generating software vulnerability repairs?***

**Approach**. To answer this RQ, we evaluate the accuracy of vulnerability repair approaches using the percentage of perfect predictions (%PP) similar to previous AVR studies [Chen et al. 2022; Fu et al. 2022]. If any of the repairs generated by the beam search is exactly the same as the ground-truth label (i.e., human-written vulnerable repair), it is considered as a correct prediction. Thus, the overall %PP across all testing data is computed as the number of correct predictions divided by the number of testing samples. The %PP measures how much of the predictions generated by each approach can be applied to the vulnerable functions, where the quality and the applicability of those correct repairs are guaranteed by the human-written ground truths.

We use the dataset described in Section 3.3 and compare our proposed method with the baselines introduced in Section 3.2. To ensure the robustness of our experimental results, we run our experiment five times for each approach by setting different random seeds. 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.

**Result**. The experimental results are presented in Table 2. **Regardless of the number of beams, our VQM method outperforms all baselines and achieves the best %PP.** When comparing only the top-1 repair candidates (i.e., $beam = 1$), our VQM is 2.68% (VulRepair) and 16.92% (TFix) better than pre-trained transformer encoder-decoder approaches, 21% (CURE) and 31% (DLFix) better than APR approaches, 23.18% (CodeBERT) and 24.86% (GraphCodeBERT) better than BERT-based approaches, and 20% better than the decoder-only approach, GPT2-CSRC.

The improvement of our VQM approach over previous state-of-the-art transformer encoder-decoder methods (e.g., VulRepair, TFix) has to do with our proposed mechanism to help the repair model focus on vulnerable areas while generating the repairs. The decoders in VulRepair and TFix attend to each token embedding (including both vulnerable and benign tokens) encoded by encoders without explicitly learning the cross-match between vulnerable token embeddings and their repair token embeddings. On the other hand, we introduce vulnerability queries (VQs) and vulnerability masks (VMs) in our VQM to explicitly learn the cross-match between vulnerable token embeddings and their repair token embeddings. Our VQs and VMs help the decoder bias toward vulnerable token embeddings encoded by encoders during repair generation, hence leading to more accurate vulnerability repairs.

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

| Methods | Beam=1 | Beam=3 | Beam=5 |
|---|---|---|---|
| Perfect Mask Enc + Perfect Mask Dec | *33.76* | *44.31* | *46.88* |
| Vul Mask Enc + Vul Mask Dec (ours) | **33.21** | **44.04** | 46.06 |
| Vul Mask Enc | 32.75 | 43.49 | **46.15** |
| Vul Mask Dec | 32.84 | 43.85 | 45.69 |
| w/o Vul Mask | 29.82 | 39.72 | 43.67 |
| with Vul Query Randomly Initialized | 12.57 | 24.95 | 28.81 |
| with No Bug Pre-trained Vul Mask | 26.42 | 39.63 | 41.74 |

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 of a vulnerable function and decode the corresponding repair more accurately. In what follows, we provide a comprehensive ablation study of our proposed vulnerability queries and masks.

Table 4. Compare our method with baselines, where all the methods are not pre-trained on the bug-fix data [Chen et al. 2022]. Accuracy is presented in percentage.

| Methods | Beam=1 | Beam=3 | Beam=5 |
|---|---|---|---|
| VQM | **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 |

***(RQ2) What are the contributions of each component of our VQM approach?***

**Approach**. We introduce five variants of our approach to assess the effectiveness of our proposed vulnerability queries (VQs) and vulnerability masks (VMs) as follows:

- **Perfect Vulnerability Masking in Encoders and Decoders**: This method uses an identical architecture as our VQM, however, the perfect vulnerability masks (i.e., the exact location of each vulnerable token) are provided instead of predicted by a localization model.
- **Vulnerability Masks in Encoders**: This method only applies vulnerability 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 Masks in Decoders**: This method only applies vulnerability 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 Masks**: This method is a plain transformer encoder-decoder architecture that applies no vulnerability masks.
- **With Vulnerability Query Randomly Initialized**: This method applies vulnerability masks in both encoders and decoders while vulnerability queries are randomly initialized at the start of training.

- **With No Bug Pre-trained Vul Mask**: This method applies vulnerability masks in both encoders and decoders while the vulnerability masks are only trained on the vulnerability repair dataset without pre-training on the bug-fix dataset.

In VQM, we train a separate model to predict VMs that apply to both encoders and decoders. In theory, the repair model should achieve better repair accuracy when applying more accurate VMs to its encoders and decoders. Thus, in the first variant, we aim to study whether leveraging perfect VMs (i.e., using the ground truths of vulnerability localization as VMs) can achieve better performance. Our proposed VMs can be applied to both the self-attention of encoders and the cross-attention of decoders. In particular, our VQM approach leverages VMs for both encoders and decoders. We further introduce three variants to study the effectiveness of VMs, i.e., (1) applying VMs to the self-attention of encoders; (2) applying VMs to the cross-attention of decoders; (3) without applying VMs; and (4) applying VMs trained only on the vulnerability repair dataset. Moreover, we proposed to initialize VQs based on the repair tokens in Section 2.3.2. We introduce a variant that randomly initializes VQs during training to compare with our proposed initialization method. In addition, the last variant is used to study the effectiveness of pre-training on the bug-fix corpus as described in Section 3.5.

We conduct the experiment using the dataset introduced in Section 3.3 and the same %PP measure as in RQ1 with $beam \in [1, 3, 5]$.

**Result**. The experimental results are shown in Table 3. **Our approach to using vulnerability queries (VQs) along with vulnerability masks (VMs) inside both encoders and decoders achieves the best performance for $beam \in [1, 3]$ and comparable performance for $beam = 5$.**

It can be seen that our proposed approach can achieve the best performance no matter the beam size when using the vulnerability localization ground truths as VMs (i.e., perfect masks). This result highlights the effectiveness of our proposed vulnerability masks. Moreover, applying the VMs is beneficial for both encoder self-attention and decoder cross-attention. It enhances the %PP by 2.93% when applied to encoders while gaining a %PP of 3.02% when applied to decoders, and the variants with VMs consistently outperform the variant without using any VMs. While applying the VMs on either encoders or decoders is beneficial, our proposed method to leverage the mask on both sides achieves better results for $beam \in [1, 3]$ and comparable results for $beam = 5$.

Furthermore, "With No Bug Pre-trained Vul Mask" attains a beam 5 accuracy of 41.74%, registering a 4% reduction compared to our proposed method. These results underscore a crucial insight: the training process that encompasses the bug-fix dataset, featuring a broader spectrum of general bugs with a larger sample size, carries paramount significance.

In terms of the vulnerability queries (VQs), it can be seen that our approach to initialize the VQ embeddings based on repair tokens during training consistently outperforms the randomly initialized VQs. However, the random VQ embeddings method still outperforms baselines such as CodeBERT and GraphCodeBERT, highlighting the effectiveness of using vulnerability queries with cross-attention (as proposed in Section 2.3.2) for our vulnerability repair task.

In addition, the results shown in Table 4 indicate the effectiveness of pre-training on bug-fix corpus and 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. Our method still outperforms all baseline approaches.

## 5 DISCUSSION

Our experiments have confirmed the performance advancement of our VQM approach over other AVR baseline approaches. In this section, we conduct further analysis to discuss whether our approach applies to repairing common vulnerabilities. Moreover, the data imbalance problem is common in vulnerability datasets [Das et al. 2021; Wang et al. 2020] where some vulnerability types (i.e., CWE-IDs) are common and easy to collect into the dataset while others are rare. Thus, we analyze the impact of imbalanced data frequencies across different

Table 5. The %PP of our VQM approach across the top 25 most dangerous CWE-IDs in 2022. The %PP is shown based on the beam search results where Beam=5.

| Rank | ID | %PP |
|---|---|---|
| 1 | CWE-787 (Out-of-bounds Write) | 50% (13/26) |
| 3 | CWE-89 (SQL Injection) | 100% (2/2) |
| 4 | CWE-20 (Improper Input Validation) | 33% (24/72) |
| 5 | CWE-125 (Out-of-bounds Read) | 42% (48/113) |
| 6 | CWE-78 (OS Command Injection) | 33% (1/3) |
| 7 | CWE-416 (Use After Free) | 31% (9/29) |
| 8 | CWE-22 (Path Traversal) | 50% (1/2) |
| 11 | CWE-476 (NULL Pointer Dereference) | 31% (11/36) |
| 13 | CWE-190 (Integer Overflow or Wraparound) | 54% (19/35) |
| 16 | CWE-862 (Missing Authorization) | 100% (1/1) |
| 17 | CWE-77 (Command Injection) | 67% (2/3) |
| 19 | CWE-119 (Memory Corruption) | 75% (223/296) |
| 22 | CWE-362 (Race Condition) | 9% (3/34) |
| 23 | CWE-400 (Uncontrolled Resource Consumption) | 55% (11/20) |
| | Average | 55% (368/672) |

CWE-IDs on our VQM approach. Last but not least, we analyze the impact of our vulnerability mask on repair decoders' cross-attention to answer whether our vulnerability masks can truly help enhance the awareness of decoders' vulnerability queries in vulnerable code areas as proposed in Section 2.3.

### 5.1 Can our VQM repair the common dangerous vulnerability types (i.e., CWE-IDs)?

To investigate whether our VQM approach can repair common dangerous real-world vulnerabilities, we evaluate our approach using testing data based on the 2022 CWE Top-25 Most Dangerous Software Weaknesses released by the CWE community [CWE 2022]. Among the Top-25 dangerous CWE-IDs, 14 of them are involved in our testing data. The results are presented in Table 5 with $beam = 5$ (the repair model generates 5 repair candidates for each vulnerable function). We find that our approach can generate correct human-written repairs for 41% ($\frac{87}{213}$) of the Top-5 dangerous CWE-IDs. On average, our approach can correctly repair 55% ($\frac{368}{672}$) of the vulnerable functions affected by the Top-25 most dangerous CWE-IDs, which is better than the average performance of our approach across all CWE-IDs (i.e., 45.14%). These results imply the potential applicability of our VQM approach which could be used to repair common vulnerabilities automatically.

### 5.2 How does our VQM perform across different vulnerability types that have different data frequencies in our experimental dataset?

We visualize the %PP across all CWE-IDs in our testing data as a bar graph to explore our VQM's performance for different CWE-IDs. In addition, we show the frequency of each CWE-ID for both training and testing data as two line graphs to explore the relationship between the frequency of samples and the performance of our method. Note that the ticks of the Y axis on the left are for the %PP metric while those on the right are for the data frequency of each CWE-ID.

We found that the frequency of training and testing samples are not highly correlated with the performance of our method. This indicates that automated vulnerability repair (AVR) is a challenging problem in that high-frequency samples may not guarantee the repair model's performance.

As shown in Figure 6, the performance of our approach varies for each CWE-ID. Our approach performs well on some of the CWE-IDs that all testing samples can be correctly repaired. In particular, we found that our approach

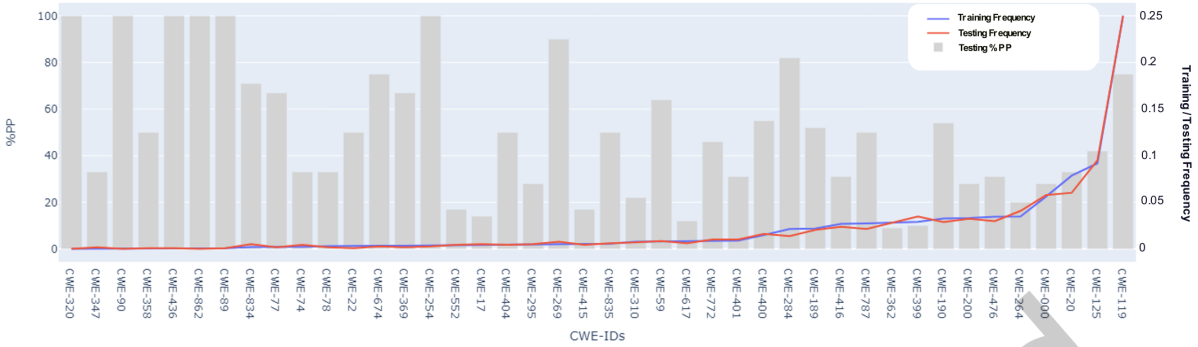


Fig. 6. The performance analysis of our VQM accross different CWE-IDs. The bar chart represents the %PP while the blue line is the training frequency and the red line is the testing frequency across all vulnerability types. Note that the ticks of the Y axis on the left are for the %PP metric while those on the right are for the data frequency of each CWE-ID.

can achieve better accuracy for buffer-related errors such as CWE-119, CWE-190, CWE-787, and CWE-125, where our approach achieves 52% better than its average performance. To comprehensively assess the landscape, we also focus on two representative baseline methods, namely VulRepair and TFix, and evaluate their performances across various CWE-IDs. It is noteworthy that these baseline methods exhibit analogous characteristics to the performance trends of our approach. Specifically, both VulRepair and TFix showcase superior performance in the realm of buffer-related errors, surpassing their average performance by 53% and 47%, respectively.

These buffer-related CWE-IDs, which our approach and baseline methods can address with higher accuracy, are primarily centered around memory management concerns. These vulnerabilities encompass improper management of memory buffers, including arrays, strings, and other data structures, ultimately leading to potential memory corruption, crashes, or unauthorized memory access. Despite the criticality of these vulnerabilities and the complexity involved in identifying and mitigating them, they tend to arise from specific code segments and memory management practices.

These analyses collectively underscore the strides achieved by deep learning-based methodologies in advancing the proficiency of generating code repairs, particularly in the domain of buffer-related errors in C/C++ code.

In addition, we found that there are two vulnerability types (i.e., CWE-310 – Cryptographic Issues and CWE-552 – Files or Directories Accessible to External Parties) that can only be fixed by our VQM approach but not other baselines.

Compared to the aforementioned buffer-related CWE-IDs, CWE-310 and CWE-552 address broader concerns related to cryptography and access controls. CWE-310 involves vulnerabilities related to cryptographic operations, which can be complex due to the intricacies of cryptography algorithms, key management, and proper implementation of cryptographic functions. CWE-552 pertains to vulnerabilities related to the exposure of files or directories to external parties, which can involve complex access control mechanisms, permissions management, and overall system design. These findings underscore the potential of our proposed technique to comprehend code structures better, enhancing transformer encoders and decoders in generating accurate repairs for intricate cryptography and access control vulnerabilities.

Finally, it's worth noting that despite the introduction of our vulnerability query and masking techniques, certain vulnerabilities that are infrequent within our dataset pose a persistent challenge for accurate repair. For instance, our approach faced difficulties in effectively addressing vulnerabilities like CWE-444 (Inconsistent Interpretation of HTTP Requests) and CWE-285 (Improper Authorization). This highlights the challenge of

automated vulnerability repair, demanding the model to address a wide spectrum of vulnerabilities, including those that manifest infrequently, thus obliging the model to glean insights from a limited dataset.

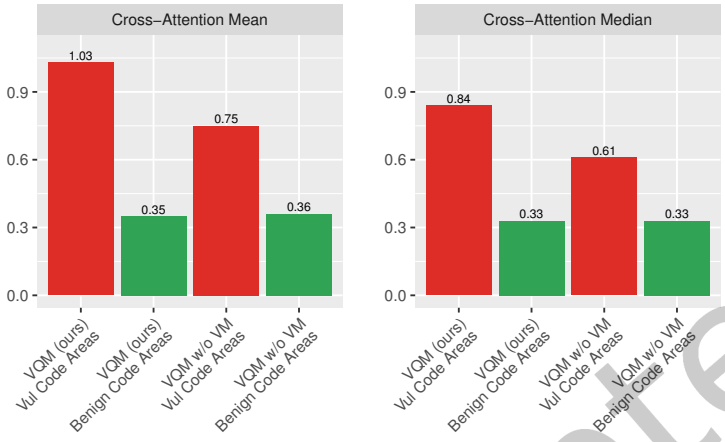


Fig. 7. The comparison of the cross-attention weights of our VQM approach and the variant, VQM w/o VM. The red bars indicate the cross-attention weights for vulnerable code areas while the green bars indicate the cross-attention weights for benign code areas.

## 5.3 Does our proposed vulnerability mask help highlight vulnerable code areas during vulnerability repair?

To investigate whether our proposed vulnerability mask (VMs) technique helps the repair decoders focus more on vulnerable code areas during repair generation, we analyze the cross-attention weights between the embeddings of vulnerable functions and embeddings of generated repair patches. We compare the cross-attention weights of our VQM approach with a variant without applying VMs (i.e., *VQM w/o VM*). We only remove the VM component to produce this variant while other components remain the same as our VQM. We analyze the cross-attention weights of each approach on correctly repaired testing samples.

The results are presented in Figure 7. We find that our VQM approach (applying VMs) has higher cross-attention mean (1.03 compared with the variant's 0.75) and median (0.84 compared with the variant's 0.61) weights for vulnerable code areas than the approach without applying VMs. Moreover, the cross-attention weights for benign code areas remain low and similar to the variant. For our VQM approach, the gaps between the cross-attention weights of the vulnerable code areas and benign code areas are 0.68 (1.03-0.35) and 0.51 (0.84-0.33) for mean and median, which is bigger than the variant's 0.39 (0.75-0.36) and 0.28 (0.61-0.33). These results confirm that our VMs can help our vulnerability queries in decoders focus more on vulnerable code areas and have better cross-attention contrast between vulnerable and benign code areas.

To demonstrate that our VMs can help decoders' cross-attention focus more on vulnerable code areas, we visualize the cross-attention maps of the two approaches, applying VMs (i.e., VQM) and without applying VMs. We visualize two vulnerable functions that are correctly repaired by both approaches in our testing set. We present the cross-attention visualization in Figure 8, where axis X is embeddings of the repair patches (i.e., Vulnerability Queries) while axis Y is embeddings of the vulnerable functions. Our approach with VMs has higher attention weights than the one without using VMs for the multi-scope (7 vulnerable code areas - the right part) vulnerable function. The visualization aligns with our analysis presented in Figure 7. This implies our VMs can

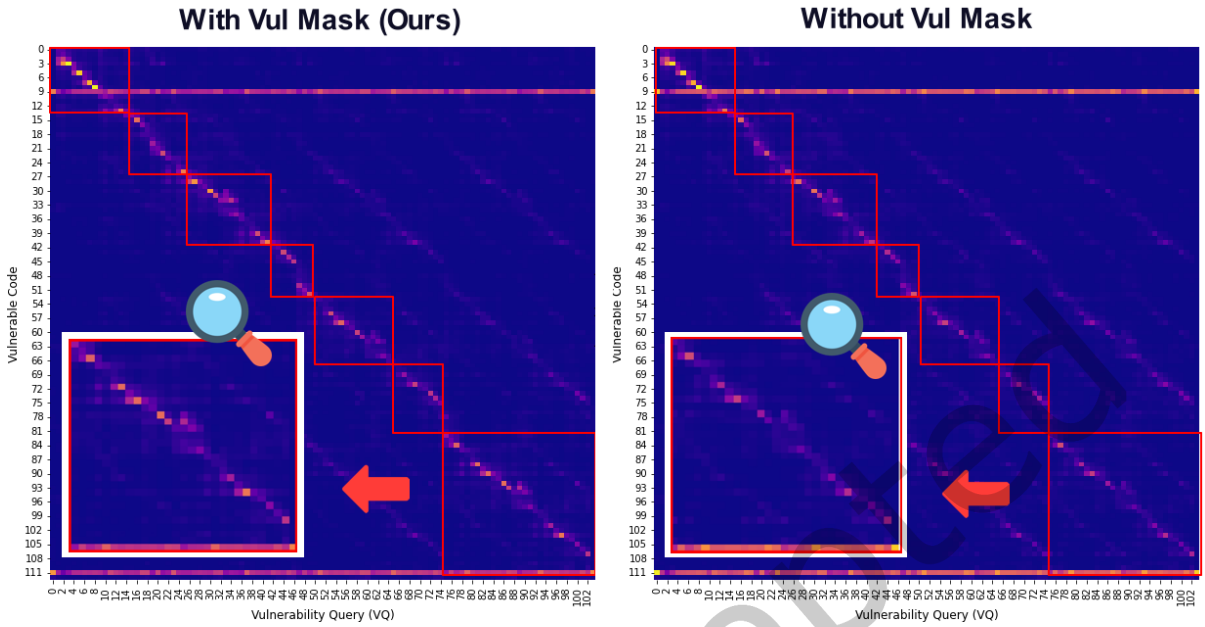


Fig. 8. The visualization of the cross-attention weights between embeddings of generated repair patches (i.e., Vulnerability Queries - axis X) and embeddings of vulnerable functions (axis Y). The ground-truth vulnerable code areas are highlighted in red boxes. It can be seen that the cross-attention map is more highly activated for vulnerable code areas when applying vulnerability masks (VMs). In other words, our VMs help better distinguish the vulnerable and benign code areas.

help decoders focus more on vulnerable code areas when generating their corresponding repair codes through vulnerability queries, which may lead to better repair accuracy as demonstrated in our RQ1.

Table 6. (Discussion Results) The effectiveness of applying our vulnerability masks to a decoder-only transformer architecture.

| Methods | Beam=1 | Beam=3 | Beam=5 |
|---|---|---|---|
| GPT2-CSRC with Vulnerability Masks | **14.86** | **26.51** | **31.93** |
| GPT2-CSRC | 11.93 | 19.27 | 24.77 |

## 5.4 Can our proposed vulnerability mask be applied to decoder-only transformer architectures?

As our proposed vulnerability masks (VMs) are expressly designed to seamlessly integrate with the self-attention mechanism intrinsic to transformer models, the feasibility of their application to decoder-only transformers becomes evident. To explore the potential enhancement our VMs might confer upon decoder-only models, we proceed to integrate them with GPT2-CSRC [Pearce et al. 2023], one of the prominent decoder-only program repair approaches. Notably, this integration involves applying our VMs to the self-attention mechanisms of the decoders, a strategy detailed in Section 2.3.3.

The experimental results are presented in Table 6. Evidently, the integration of our vulnerability masks (VMs) consistently yields performance enhancements in the context of the GPT2-CSRC approach. Noteworthy improvements are discernible across various beam widths, notably augmenting performance from 12% to 15% for $beam = 1$, from 19% to 27% for $beam = 3$, and from 25% to 32% for $beam = 5$. These results highlight the effectiveness of our proposed masking technique for the transformer's self-attention mechanism.

## 6 A USER STUDY OF AI-GENERATED VULNERABILITY REPAIRS

In addition to the performance evaluation of our approach in RQ1 and RQ2, we conducted a user study with 71 practitioners with software security backgrounds to evaluate the usefulness of AI-generated vulnerability repairs. We answered the following research question:

**(RQ3) Are AI-generated vulnerability repairs perceived as useful by software developers?** As we have comprehensively evaluated the performance in RQ1 and RQ2, it is essential to delve into the practical utility of AI-generated repairs for software practitioners, a vital aspect yet to be explored. To bridge this gap in knowledge, we diligently tackle RQ3 by conducting a user study tailored to assess the perceptions of security-aware software developers regarding AI-generated vulnerability repairs. This investigation seeks to unveil the practicality and real-world implications of our approach.

Following [Kitchenham and Pfleeger 2008], we conduct our study according to the following steps: (1) design and develop a survey, (2) recruit and select participants, and (3) verify data and analyze data. We explain the details of each step below.

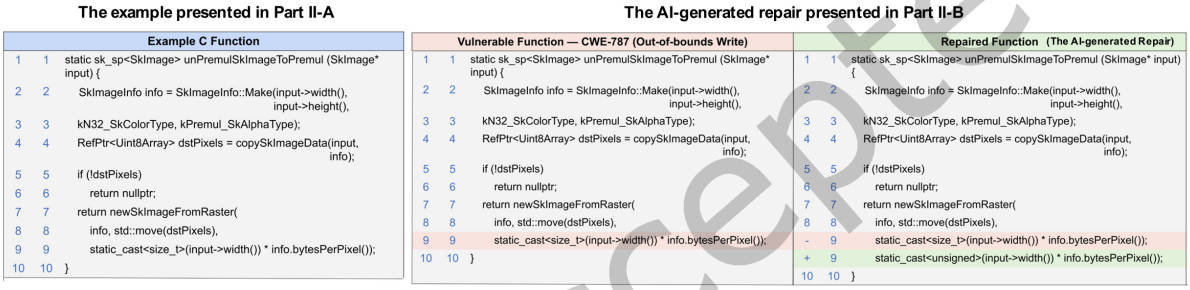


Fig. 9. The left part shows the example vulnerable C function presented in Part II-A in our user study while the right part shows the example of AI-generated repair presented in Part II-B in our user study.

### 6.1 Survey Design

**Step 1 – Design and development of the survey:** We designed our survey as a cross-sectional study where participants provided their responses at one fixed point in time. The survey consists of 8 closed-ended questions and 3 open-ended questions. For closed-ended questions, we use multiple-choice questions and a Likert scale from 1 to 5. Our survey consists of two parts: preliminary questions and participants' perceptions of AI-generated software vulnerability repairs.

***Part I: Demographics.*** The survey commences with a query, "(D1) What is your role in your software development team?", to ensure that our survey captures responses from the intended target participants. Subsequently, the survey features a demographics question, "(D2) What is the level of your professional experience?", aimed at ensuring a diverse distribution of responses across software practitioners with varying degrees of professional experience.

***Part II-A: Manual Vulnerability Repair.*** To simulate a realistic vulnerability analysis scenario, we presented an example vulnerable C function to the participants as depicted in the left part of Figure 9. We then asked the participants to examine whether the function is vulnerable and propose a fix if required. Notably, we prepared ten different examples of vulnerable C functions which were spread equally to groups of participants to ensure that our survey is not biased toward a specific vulnerable function.

Precisely, four inquiries were presented to the participants, commencing with "(Q1) Do you think the C function presented in Figure 1 is a vulnerable function or not?"; followed by "(Q2) Which line of code do you think is

vulnerable?"; then "(Q3) Please suggest a fix to patch the vulnerable line."; and concluded with "(Q4) How long did it take for you to identify whether the function is vulnerable and propose a fix to the vulnerable function (if required)?".

***Part II-B: Participants' Perception of AI-generated Vulnerability Repairs.*** As illustrated by the experimental results presented in Section 4, our approach consistently attains the highest perfect repair accuracy, surpassing other baseline vulnerability repair methods by a significant margin. Given this demonstrated superiority, we selected our VQM approach as the representative AI-generated vulnerability repair method for this user study.

To assess the participants' perception regarding AI-generated vulnerability repairs, we presented the repairs generated by our VQM approach as shown in the right part of Figure 9.

Precisely, five inquiries were presented to the participants, "(Q5) Do you think the AI-generated vulnerability repair by our approach is correct or not?"; followed by "(Q6) How do you perceive the usefulness of AI-generated vulnerability repairs? "; then "(Q7) Please justify your answer to Q6."; then "(Q8) Would you consider adopting AI-generated vulnerability repair techniques if they are integrated into your software development IDEs (e.g., VSCode) for free with no conditions?"; and concluded with "(Q9) What is your expectation of AI-generated vulnerability repairs and how can we improve them?"

We employed Google Forms as the platform for our online survey administration. Each participant was greeted with a comprehensive introductory statement upon accessing the landing page. This statement elucidated the study's objectives, rationale for participant selection, potential advantages and risks, and the commitment to safeguarding confidentiality. The survey was designed to be succinct, with an estimated completion time of around 15 minutes, and ensured complete anonymity for all respondents. Importantly, our survey underwent a rigorous evaluation process and received ethical approval from the Monash University Human Research Ethics Committee (MUHREC ID: 40251).

**Step 2: Recruit and select participants:** We recruited developers who have software engineering and/or software security expertise through LinkedIn and Facebook platforms. We received hundreds of Expressions of Interest in one week. We then randomly split our target audience into ten groups, where each group was given one distinct example vulnerable function while all of the questions were identical across all groups (as described in Part II-A). Finally, we obtained a total of 82 responses in one week.

**Step 3: Verify data and analyze data:** To ascertain the completeness of survey responses, particularly regarding open-ended questions, we conducted a thorough manual review. We filtered out 11 invalid responses (e.g., should any of the open-ended questions remain unanswered or if the responses are incomprehensible) out of a total of 82 responses. Thus, we included the remaining 71 responses for analysis. Closed-ended responses were quantitatively analyzed and presented using Likert scales through stacked bar plots. Additionally, we performed an in-depth manual analysis of open-ended question responses to gain a better understanding of participants' insights.

## 6.2 Survey Results

**Part I: Demographics**. Figure 10 presents the overall respondent demographic. In terms of the profession of the participants, 25% ($\frac{18}{71}$) of them are security analysts, 9% ($\frac{6}{71}$) of them are security researchers, 38% ($\frac{27}{71}$) of them are full-stack software engineers, while the other 28% ($\frac{20}{71}$) are software quality assurance engineers and software project managers. In terms of the level of their professional experience, 27% ($\frac{19}{71}$) of them have less than 5 years of experience, 49% ($\frac{35}{71}$) have 6-10 years of experience, 21% ($\frac{15}{71}$) have 11-20 years of experience, while the other 3% ($\frac{2}{71}$) has more than 20 years of experience.

**Part II-A: Manual Vulnerability Repair**. Figure 11 summarizes the answers to (Q1)-(Q4) regarding manual vulnerability repair from the participants. In particular, 85% ($\frac{60}{71}$) of them can correctly identify that the given function is vulnerable, 41% ($\frac{29}{71}$) of them can correctly locate the vulnerable statement, while only 1% ($\frac{1}{71}$) of them can suggest the correct repair. **Our findings highlight the complexity of vulnerability repair in contrast**

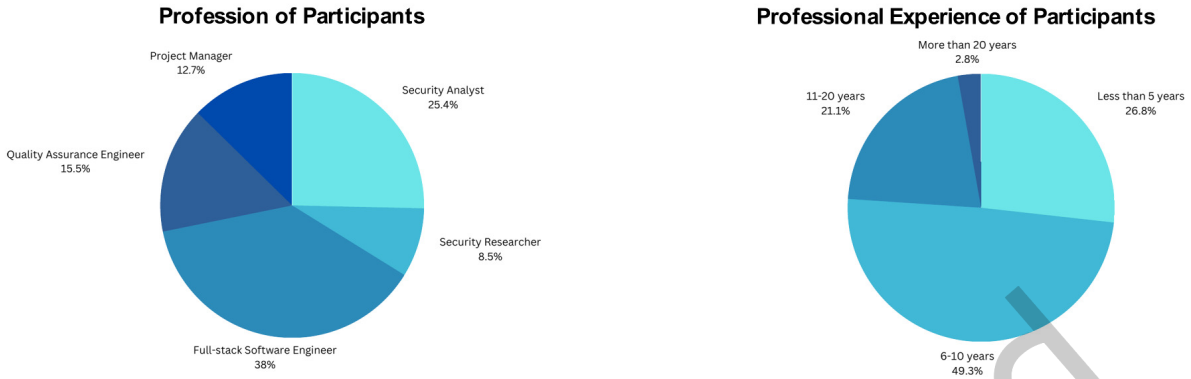


Fig. 10. The demographics of our survey participants in terms of their profession and professional experience.

**to the identification and localization of vulnerabilities.** A significant majority, accounting for over 87% of the participants, encountered challenges when tasked with proposing vulnerability repairs. Additionally, among those participants who did attempt to provide repair suggestions, most of them were incorrect.

In the context of identifying, locating, and suggesting repairs for vulnerabilities, 38% ($\frac{27}{71}$) spent 1-5 minutes, 30% ($\frac{21}{71}$) took 6-10 minutes, and 32% ($\frac{23}{71}$) required over 10 minutes to complete these tasks. **This highlights the time-consuming nature of vulnerability repair, with 62% of participants requiring over 10 minutes to address a single vulnerable function containing 5-15 statements.**

**Part II-B: Participants' Perception of AI-generated Vulnerability Repairs**. Figure 12 summarizes the answers to (Q5)-(Q9) regarding participants' perception of AI-generated vulnerability repairs provided by our approach. **As presented in (Q5) results, the majority of participants demonstrated the ability to accurately assess the correctness of AI-generated vulnerability repairs, with 85% ($\frac{60}{71}$) providing correct evaluations.** However, 11% ($\frac{8}{71}$) of participants failed to correctly identify the accuracy of the repairs, while 4% ($\frac{3}{71}$) expressed uncertainty regarding the correctness of the AI-generated repairs.

Participants have actively engaged in the manual vulnerability repair process encompassing questions (Q1) to (Q4). In (Q5), we simulate the vulnerability repair workflow involving AI tools, wherein users are required to evaluate the correctness of AI-generated repairs before implementing them. The outcomes of (Q5) provide further evidence of the participants' proficiency in assessing AI-generated repairs. Hence, we proceed to evaluate their perspective on the usefulness of AI-generated vulnerability repairs, drawing from their professional expertise in vulnerability repair.

**(Q6) How do you perceive the usefulness of AI-generated vulnerability repairs?**

**Findings**. **86% of the participants perceived that the vulnerability repairs generated by our approach are useful** due to various reasons stated in (Q7):

- **Efficiency & Productivity** – R8 (a security analyst with 6-10 years of experience): *The AI systems can quickly and efficiently scan large code bases for vulnerabilities and can also identify vulnerabilities that human developers may miss*; R56 (a security analyst with 6-10 years of experience): *AI-generated vulnerability repairs can enhance the efficiency and effectiveness of security teams by automating the identification and resolution of vulnerabilities, allowing them to focus on more complex security tasks and reducing the time required to patch potential weaknesses*; R22 (a full-stack software engineer with 6-10 years of experience): *Its automation and data analysis saves time and effort, boosting productivity. It also speeds up scientific discovery*

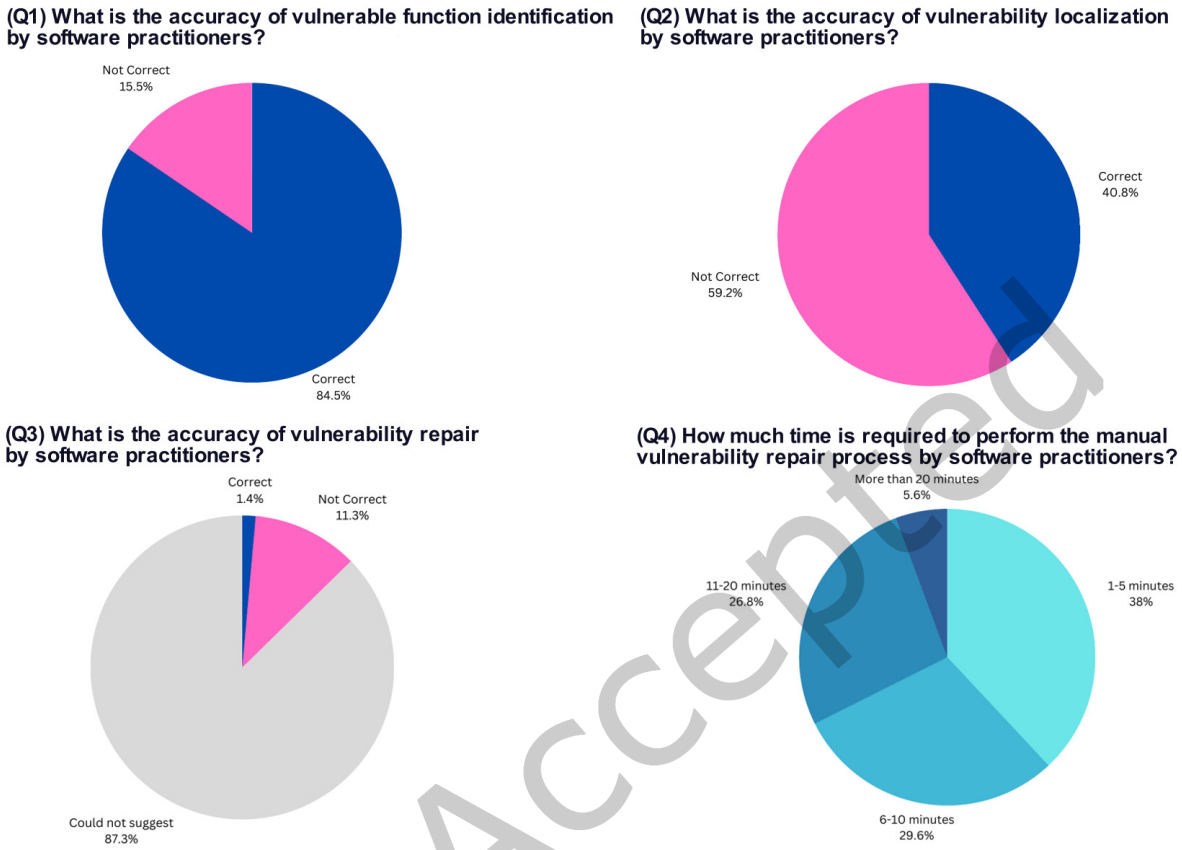


Fig. 11. (Survey Results) A summary of the survey questions (i.e., Part II-A: Q1-Q4) and the results obtained from 71 participants.



*and development in various fields*; and R12 (a full-stack software engineer with 6-10 years of experience): *Throughout my seven years of experience, AI-generated vulnerability repairs have been very useful as they help a lot. It helps to reduce the backlog of vulnerabilities that need to be fixed, and it can also help to ensure that vulnerabilities are fixed quickly.*

- **Timeliness & Scalability** – R29 (a full-stack software engineer with less than 5 years of experience): *AI can rapidly identify and respond to vulnerabilities, which is critical in a constantly evolving threat landscape* and R41 (a security researcher with less than 5 years of experience): *AI can handle large and complex systems, making it an effective tool for identifying and addressing vulnerabilities at scale, which may be challenging for human teams.*
- **Applicability** – R18 (a security analyst with 6-10 years of experience): *The repaired function is a good one, and it shows and indicates some compatibility with the vulnerable functions. In this way, adopting AI-generated vulnerability repair plays an important role in ensuring proper functioning for greater results outcomes* and R20 (a full-stack software engineer with 11-15 years of experience): *Useful in the sense that it allows for more successful and quick corrections. Secondly, it provides a fast learning system for developers.*

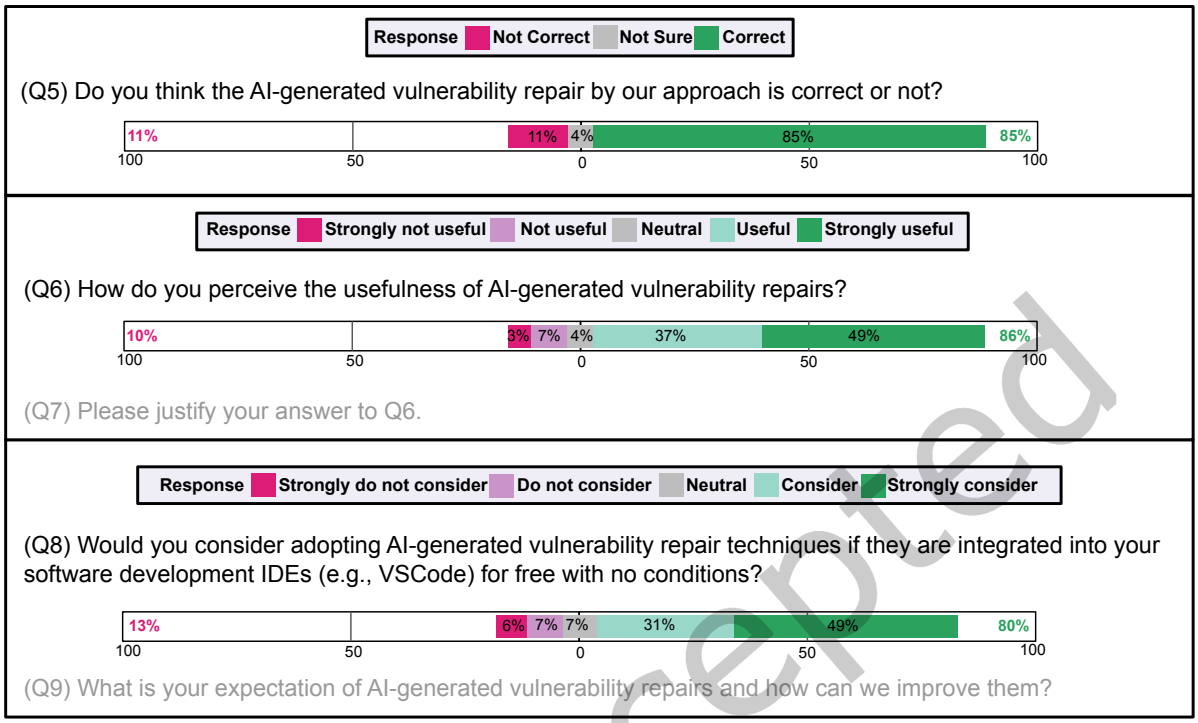


Fig. 12. (Survey Results) A summary of the survey questions (i.e., Part II-B: Q5-Q9) and the results obtained from 71 participants.



On the other hand, we observed that over 12% ($\frac{9}{71}$) of participants expressed apprehensions about human involvement or potential ethical considerations, despite recognizing the usefulness of AI-generated repairs:

- R32 (a security analyst with 6-10 years of experience): *In summary, AI-generated vulnerability repairs can be a valuable aid in the security process, but they should be used alongside human expertise and thorough testing to ensure the effectiveness and security of the repairs*;
- R33 (a security analyst with 6-10 years of experience): *However, it's important to exercise caution with AI-generated repairs. While AI algorithms can suggest potential fixes, they may not always produce the most effective or secure solutions. Human expertise is still necessary to validate and test the proposed repairs, ensuring that they don't introduce new vulnerabilities or have unintended consequences*;
- R40 (a security analyst with 6-10 years of experience): *AI-generated vulnerability repairs can be a valuable tool in cybersecurity, offering speed and scalability. However, they should be part of a broader security strategy that includes human expertise, ongoing monitoring, and ethical considerations*;
- R43 (a full-stack software engineer with 6-10 years of experience): *AI-generated vulnerability repairs have the potential to be highly useful in enhancing cybersecurity, especially in terms of speed. However, Human security professionals should provide oversight, review AI-generated repairs, and make critical decisions.*

**(Q8) Would you consider adopting AI-generated vulnerability repair techniques if they are integrated into your software development IDEs (e.g., VSCode) for free with no conditions?**

**Findings**. **80% of the participants expressed a willingness to embrace AI-generated vulnerability repair techniques if they are readily available and free of charge.** Furthermore, the participants' expectations regarding AI-generated vulnerability repairs, as indicated in their responses to (Q9), can be summarized as follows:

- **Accuracy** – R4 (a security analyst with 6-10 years of experience): *My expectation of AI-generated vulnerability repairs is that they will become more advanced and effective in detecting and patching vulnerabilities in software. To improve them, we can focus on enhancing the accuracy and speed of vulnerability detection algorithms, ensuring compatibility with different software development environments, and continuously updating the AI models with the latest threat intelligence* and R5 (a security analyst with 6-10 years of experience): *Continuous learning and adaptation: AI models should be regularly updated and fine-tuned based on feedback and new security insights to enhance their accuracy and effectiveness.*
- **Data Integrity** – R69 (a security analyst with 6-10 years of experience): *To improve the quality of AI-generated repairs, I think it will be important to have robust datasets that cover a wide range of vulnerabilities and potential solutions.*
- **Availability** – R28 (a security analyst with 6-10 years of experience): *It needs to be integrated into more reachable software.*
- **Explainability** – R33 (a security analyst with 6-10 years of experience): *Transparency and explainability: Providing clear explanations of how AI-generated repairs are generated can help developers understand and trust the suggested fixes*; R42 (a quality assurance engineer with less than 5 years of experience): *AI-generated vulnerability repairs should provide clear explanations for their decisions, allowing human operators to understand and trust the recommendations.*; and R53 (a full-stack software engineer with more than 20 years of experience): *In my opinion, AI-generated vulnerability repairs have a lot of potential, but there are some challenges that need to be addressed before they can be widely used and trusted. One challenge is ensuring the accuracy of the repairs. Another challenge is the need for explainability and transparency. Currently, many AI systems are "black boxes," meaning that it's difficult to understand how they arrive at their decisions. I think improving explainability and transparency will be key to building trust in the technology.*

**Summary**. Our survey study with 71 software practitioners provides valuable insights into the perception of AI-generated vulnerability repairs. In particular, 86% ($\frac{61}{71}$) of the participants found these repairs useful attributing their value to increased efficiency, productivity, timeliness, scalability, and broad applicability. However, experienced security analysts voiced concerns regarding the potential ethical implications and the need for human oversight. Furthermore, participants expressed expectations for improvement, emphasizing the importance of enhancing repair model accuracy, data integrity, method availability (e.g., integrating it into common software development IDEs), and the transparency and explainability of AI models. These findings underscore the potential and importance of AI-generated vulnerability repairs in the software security landscape while highlighting areas for further development and refinement.

## 7 RELATED WORK

Machine learning (ML)-based techniques have been proposed to automate software engineering-related tasks such as agile planning [Fu and Tantithamthavorn 2022a], code review [Hong et al. 2022b,a; Liu et al. 2022; Pornprasit et al. 2023; Thongtanunam et al. 2022], code completion [Takerngsaksiri et al. 2022], defect prediction [Pornprasit et al. 2021; Pornprasit and Tantithamthavorn 2021, 2022], and test case generation [Alagarsamy et al. 2023]. In particular, ML-based vulnerability prediction approaches have also been proposed to help security analysts predict vulnerabilities [Fu et al. 2023a; Fu and Tantithamthavorn 2022b; Nguyen et al. 2020, 2019, 2022b,a], explain their vulnerability types [Fu et al. 2023b], estimate their severity [Fu et al. 2023c], and recommend repair patches [Chen et al. 2022; Fu et al. 2022].

In this paper, we focus on the **Automated Vulnerability Repair (AVR)** task that uses machine learning models to generate repair patches for vulnerable C/C++ functions. In particular, our AVR task shares similarities with the widely recognized Automated Program Repair (APR) task, yet it stands apart through two distinct aspects. Firstly, the AVR task is notably more domain-specific, directed towards addressing vulnerabilities rather than general defects. Secondly, instead of generating complete repaired functions as output, the AVR task requires models to generate repair patches that exclusively address the vulnerable code regions within vulnerable functions. This design curtails output length and alleviates the model's burden of generating extensive sequences, thereby optimizing the repair process. It is worth noting that the general program repair shares a similar nature to vulnerability repair where defective functions only consist of a few defective code statements that need to be repaired. Thus, our AVR configuration and the methodology of VQM have the potential for adaptation to tackle the APR task that focuses on general bug fixing. It is important to acknowledge that vulnerability repair resides within the larger domain of program repair. Nevertheless, in this paper, our focus remains dedicated to the specialized domain of vulnerability repair.

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. In addition, 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. 2021a]) to repair C/C++ programs. Mashhadi and 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. On the other hand, DLFix [Li et al. 2020] and CURE [Jiang et al. 2021] were proposed to generate vulnerability repairs that satisfy test cases. Thus, complete repaired functions are required to train and evaluate models and the problem statement is different from ours described in Section 2.2. 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.

Previous approaches mainly focus on leveraging seq2seq models for the AVR task. In particular, transformer-based methods have achieved promising performance. As illustrated in Section 2.2, a vulnerable function usually consists of only a few vulnerable code areas that cause the vulnerability. Nevertheless, existing transformer-based approaches lack a mechanism to capture those vulnerable code areas during the repair generation. Thus, we propose a mechanism to help decoders focus more on vulnerable code areas during the repair. Specifically, we extend the VIT-based approaches for object detection (e.g., DeTR [Carion et al. 2020]) and build our own vulnerability repair approach with vulnerability queries and masks to guide decoders to focus more on vulnerable code areas during vulnerability repairs.

## 8 THREATS TO VALIDITY

Similar to other empirical studies related to deep learning models, there are various threats to the validity of our results and conclusions.

*Threats to internal validity* relate to the stochastic gradient descent process to update network weights for deep learning models during training. To mitigate this threat, we explicitly set the random seeds to ensure reproducibility and report the hyperparameter settings in the replication package to support future replication

studies. In addition, we repeated our main experiment five times with different random seeds to ensure the soundness of our experimental results and findings.

We acknowledge another internal threat to the validity of our experiments arising from the assumption that our AI-based vulnerability repairs operate on confirmed vulnerable functions. While this assumption simplifies our experimental setup, it may not fully represent the complexity of real-world scenarios where identifying vulnerabilities is an integral part of the process. However, the vulnerability detection process is beyond the scope of this paper. Thus, it is important to consider this limitation when interpreting the results of our experiments. Finally, it is paramount for future research endeavors to develop an end-to-end pipeline that encompasses vulnerability detection.

*Threats to external validity* relate to whether our VQM approach can be generalized to other vulnerabilities and projects not included in our studied dataset, and programming languages other than C/C++. Our approach is evaluated on the dataset provided by Chen et al. [2022] consisting of CVEFixes [Bhandari et al. 2021] and Big-Vul [Fan et al. 2020] vulnerability corpus. While our studied dataset includes various vulnerabilities written in C/C++ across different software projects, our VQM approach does not necessarily generalize to other data and programming languages. Since our approach is not programming language-specific, it could be trained on other vulnerabilities written in programming languages other than C/C++ or from other projects without any modification of our approach. Thus, other datasets can be explored in future work.

## 9 CONCLUSION

In this paper, inspired by VIT-based object detection approaches in the computer vision domain, we have introduced a new AVR method named VQM 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 (VQs) and their corresponding vulnerable code areas and their corresponding repairs through the cross-attention mechanism. To strengthen such cross-matchings and guide decoders to pay more attention to vulnerable code areas, we propose to learn a vulnerability mask (VMs) and incorporate it into the cross-attention. Additionally, we apply our VMs in the self-attention of encoders to guide our model to focus more on vulnerable code areas when learning the embeddings of a vulnerable function. Through an extensive evaluation of 5,417 real-world vulnerabilities, our experiment confirms the advancement of our approach over all of the baseline AVR approaches. Moreover, the additional analysis of our experimental results highlights the applicability of our VQM approach, which can accurately repair common dangerous vulnerabilities. Last but not least, our survey study with 71 software practitioners highlights the significance and usefulness of AI-generated vulnerability repairs in the realm of software security.

## 10 ACKNOWLEDGMENTS

C. Tantithamthavorn was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100941).

## REFERENCES

Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3Test: Assertion-Augmented Automated Test Case Generation. *arXiv preprint arXiv:2302.10352* (2023).

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

Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. 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*. 30–39.

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

Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering (TSE)* (2022). https://doi.org/10.1109/TSE.2019.2940179

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 (TSE)* 47, 9 (2019), 1943–1959.

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

Cppcheck. [n. d.]. A tool for static C/C++ code analysis. https://cppcheck.sourceforge.io/.

CSRC. 2020. Definition of Software Vulnerability. https://csrc.nist.gov/glossary/term/software_vulnerability.

CWE. 2022. 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

Siddhartha Shankar Das, Edoardo Serra, Mahantesh Halappanavar, Alex Pothen, and Ehab Al-Shaer. 2021. V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities. In *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 1–12.

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

Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2022. VELVET: a noVel Ensemble Learning approach to automatically locate VulnErable sTatements. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 959–970.

Edgescan. 2022. 2022 Vulnerability Statistic Report. https://www.edgescan.com/2022-vulnerability-statistics-report-lp/.

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

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. In *the Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1536–1547.

Michael Fu, Trung Le, Van Nguyen, Chakkrit Tantithamthavorn, and Dinh Phung. 2023a. Learning to Quantize Vulnerability Patterns and Match to Locate Statement-Level Vulnerabilities. *arXiv preprint arXiv:2306.06109* (2023).

Michael Fu, Van Nguyen, Chakkrit Kla Tantithamthavorn, Trung Le, and Dinh Phung. 2023b. VulExplainer: A Transformer-based Hierarchical Distillation for Explaining Vulnerability Types. *IEEE Transactions on Software Engineering* (2023).

Michael Fu and Chakkrit Tantithamthavorn. 2022a. GPT2SP: A transformer-based agile story point estimation approach. *IEEE Transactions on Software Engineering* 49, 2 (2022), 611–625.

Michael Fu and Chakkrit Tantithamthavorn. 2022b. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *the Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. 2023c. AIBugHunter: A Practical Tool for Predicting, Classifying and Repairing Software Vulnerabilities. *arXiv preprint arXiv:2305.16615* (2023).

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Phung Dinh. 2022. 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)*.

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

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

Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. 2022b. Commentfinder: a simpler, faster, more accurate code review comments recommendation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 507–519.

Yang Hong, Chakkrit Kla Tantithamthavorn, and Patanamon Pick Thongtanunam. 2022a. Where should I look at? Recommending lines that reviewers should pay attention to. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1034–1045.

ImageMagick. 2016. ImageMagick. https://github.com/ADVAN-ELAA-8QM-PRC1/platform-external-ImageMagick/commit/d8ab7f046587f2e9f734b687ba7e6e10147c294b.

Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*. Springer, 63–92.

Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 602–614.

Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. 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*. 292–303.

Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, Patanamon Thongtanunam, and Li Li. 2022. Autoupdate: Automatically recommend code updates for android apps. *arXiv preprint arXiv:2209.07048* (2022).

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. 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*. 101–114.

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

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

Van Nguyen, Trung Le, Olivier De Vel, Paul Montague, John Grundy, and Dinh Phung. 2020. Dual-Component Deep Domain Adaptation: A New Approach for Cross Project Software Vulnerability Detection. *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2020).

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

Van Nguyen, Trung Le, Tue Le, Khanh Nguyen, Olivier DeVel, Paul Montague, Lizhen Qu, and Dinh Phung. 2019. Deep Domain Adaptation for Vulnerable Code Function Identification. In *The International Joint Conference on Neural Networks (IJCNN)*.

Van Nguyen, Trung Le, Chakkrit Tantithamthavorn, John Grundy, Hung Nguyen, Seyit Camtepe, Paul Quirk, and Dinh Phung. 2022b. An Information-Theoretic and Contrastive Learning-based Approach for Identifying Code Statements Causing Software Vulnerability. *arXiv preprint arXiv:2209.10414* (2022).

Van Nguyen, Trung Le, Chakkrit Tantithamthavorn, John Grundy, Hung Nguyen, and Dinh Phung. 2022a. Cross Project Software Vulnerability Detection via Domain Adaptation and Max-Margin Principle. *arXiv preprint arXiv:2209.10406* (2022).

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. 2021. Pyexplainer: Explaining the predictions of just-in-time defect models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 407–418.

Chanathip Pornprasit, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Chunyang Chen. 2023. D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 296–307.

Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. 2021. JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 369–379.

Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. 2022. Deeplinedp: Towards a deep learning approach for line-level defect prediction. *IEEE Transactions on Software Engineering* 49, 1 (2022), 84–98.

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

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *the Proceedings of the Association for Computational Linguistics (ACL)*. 1715–1725.

Wannita Takerngsaksiri, Chakkrit Tantithamthavorn, and Yuan-Fang Li. 2022. Syntax-Aware On-the-Fly Code Completion. *arXiv preprint arXiv:2211.04673* (2022).

Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. Autotransform: Automated code transformation to support modern code review process. In *Proceedings of the 44th International Conference on Software Engineering*. 237–248.

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

Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2020. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 1–9.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021a. 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)*. 8696–8708.

Yingming Wang, Xiangyu Zhang, Tong Yang, and Jian Sun. 2021b. Anchor DETR: Query Design for Transformer-Based Object Detection. In *arXiv preprint arXiv:2109.07107*.

Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. 2020. Deformable DETR: Deformable Transformers for End-to-End Object Detection. In *the Proceedings of the International Conference on Learning Representations (ICLR)*.