2025 International Conference on Intelligent Computing
July 26-29, Ningbo, China
https://www.ic-icc.cn/2025/index.php

Enhancing Vulnerability-Fixing Commit Classification: The Synergy of User-Guided and LLM

Yaning Zheng[1][0009-0001-1232-4489] Honglin Zhuang [1(✉)] Dongxia Wang[1], Huayang Cao[1], and Cheng Qian [1]

[1] National Key Laboratory of Science and Technology on Information System Security, Beijing, China
Ynzheng37@163.com, zhlxsjl@163.com, wdxpaper@126.com, chystudy@foxmail.com, qiancheng@nudt.edu.cn,

Abstract. With the increasing complexity of software development environments, identifying and fixing vulnerabilities has become a key aspect of software maintenance. One way to improve the efficiency and effectiveness of vulnerability-fixing is to classify vulnerability-fixing commits. However, the existing vulnerability-fixing classification methods are limited to code language, code length, commit dataset, ambiguous and domain specialized commits, which leads to low precision. In this paper, we propose a user-guided classification method for vulnerability-fixing commits. For ambiguous and domain specialized commits, we incorporate human involvement and timely intervention in the process of fine-tuning the BERT model. Furthermore, a large language model (LLM) is employed to address the challenges posed by the variant code language and length. Experiment results show that our approach significantly improves the performance of commit classification. The user-guided BERT message classifier accuracy increases by 2～5% compared with baseline methods after 10 iterations of human participation. Based on the TensorFlow dataset, the patch classifier using LLM outperforms HERMES by 11.6% in terms of F1-score. In summary, our overall classification which combined the results of message classifier and patch classifier outperforms the HERMES by 14.6% and VulCurator by 5.6%.
Keywords: commit classification, user-guided, LLM.

1 Introduction

As the world's largest open-source project hosting platform, GitHub has 100+ million developers, 4+ million organizations, and 420+ million repositories so far [1]. Commits in GitHub, which record changes in code, documentation, or other assets, play a significant role in software maintenance and development [2]. As software systems grow rapidly in complexity, the likelihood of encountering bugs also grows. Consequently, identifying and fixing these bugs has become a critical aspect of software maintenance.

Detecting vulnerability-fixing commits aids in analyzing root causes and repair strategies for vulnerabilities, which in turn enables the development of enhanced testing frameworks, improved coding practices, and more efficient debugging processes.

However, it is not enough to only analyze the commit text information [3,4]. Commit messages can be ambiguous, incomplete, or misleading. For example, some messages contain keywords like "CVE" but only describe textual changes, while others lack sufficient detail, such as "minor fix" or "updated dependencies" [5]. Moreover, silent patches, which intentionally conceal their vulnerability-fixing purpose to prevent exploitation by attackers, further complicate detection [6,7]. Therefore, it is very necessary to add the analysis of code-change in the vulnerability-fixing classification problem. Existing work [8,9] also proves the necessity of adding code-change to commit classification.

Despite the incorporation of code-change analysis in certain existing methods, significant challenges persist. First, BERT-based classification models encounter difficulties with **code language and length constraints** [8,10], making it difficult to generalize across different languages or process lengthy code changes. Second, **ambiguous and domain-specific commits** require a level of expertise and contextual understanding that current automated systems often lack. These limitations reduce the effectiveness of existing approaches in accurately identifying vulnerability-fixing commits.

The emergence of large language model (LLM) provides a new way to solve this issue. However, directly using LLM to process commit messages and code changes simultaneously presents several challenges. Our experiments have shown that LLM exhibits limited capacity to manage complex, lengthy contextual information. When both commit messages and code changes are input simultaneously, the accuracy drops by 34% compared to analyzing code changes alone. In the application of LLM, processing excessive information in a single stage or running multiple stages presents two significant challenges: first, the understanding of the information may be inadequate [11,12], and second, it can lead to substantial resource consumption during the token calculation process [13,14].

To address these challenges, we propose an innovative user-guided classification framework that integrates the strengths of both small and large models. This framework leverages human expertise and incorporates human feedback into the fine-tuning of a BERT-based model, enabling it to more effectively handle ambiguous and domain-specific commit messages. Additionally, we utilize LLMs to overcome limitations related to code language diversity and input length, providing a robust solution for processing complex code changes. By combining the results from small and large models, our approach significantly conserves resources while enhancing prediction accuracy.

To evaluate the efficacy of our approach on the message classifier, we use the dataset [15] which contains 9269 commit messages of C/C++ language to train the BERT model. The results show that, after 10 rounds of user-guided refinement, resampling and smoothing methods improve the accuracy by 2.2% compared with no participation. To avoid overfitting, we validate the trained classifier on five real-world datasets [16-18], and the accuracy, precision, recall, and F1-score are significantly improved. For code changes, the patch classifier leveraging LLM (Qwen), which is tested on the TensorFlow [10] dataset, has an accuracy of 85% and is 11.6% higher than HERMES [19] in F1. By integrating the user-guided message classifier and the LLM-based patch classifier, our method achieved state-of-the-art performance, surpassing HERMES by 14.6% and VulCurator [10] by 5.6% in terms of F1 score.

In this paper, we make the following contributions:

- We propose a user-guided BERT framework to improve vulnerability-fixing commit message classification performance.
- Our proposed LLM-aided vulnerability-fixing code-change classifying method outperforms HERMES by 11.6% in F1-score, based on the TensorFlow dataset.
- By combining user-guided and LLM-based approaches, our method achieves superior performance compared to HERMES and VulCurator.
- We offer a reproduction package to facilitate the reproduction and future research at https://zenodo.org/records/14189550.

The rest of the paper is organized as follows: Section II describes our motivation; Section III describes our methodology; Section IV describes the experiments and results; Section V introduces the related work; Section VI concludes the paper.

2 Motivation

Figure 1 presents a detailed example of a message description associated with a vulnerability-fixing commit, specifically related to CVE-2019-12904. This commit message has been repeatedly misclassified by machine learning algorithms [10,20] and LLM (Qwen) as not vulnerability-fixing. Upon closer examination of the content within the message, it becomes evident that it contains numerous proper nouns and exhibits a high level of technical jargon, making it challenging for automated systems to accurately interpret its relevance. "Move look-up table to .data section and unshare between processes." This statement alone involves specific terminology such as "look-up table," ".data section," and "unshare between processes," all of which require domain-specific knowledge to fully comprehend. The presence of these terms, along with other similar phrases throughout the message, underscores the complexity involved in understanding the nuances of software development documentation.

```
GCM: move look-up table to .data section and unshare between processes

* cipher/cipher-gcm.c (ATTR_ALIGNED_64): New.
(gcmR): Move to 'gcm_table' structure.
(gcm_table): New structure for look-up table with counters before and
after.
(gcmR): New macro.
(prefetch_table): Handle input with length not multiple of 256.
(do_prefetch_tables): Modify pre- and post-table counters to unshare
look-up table pages between processes.
--

GnuPG-bug-id: 4541
Signed-off-by: Jussi Kivilinna <jussi.kivilinna@iki.fi>
```

Fig. 1. CVE-2019-12904 commit message.

As a result, there exists a clear need for human intervention when analyzing such messages. While machine learning models can certainly provide valuable insights into large datasets, they often struggle with highly specialized language and context-dependent information. In cases like this one, only a skilled developer or expert reviewer would be able to accurately assess whether the changes described constitute a genuine bug fix or merely represent routine maintenance activities.

Figure 2 illustrates a vulnerability-fixing patch derived from CVE-2018-7999, which notably includes both code modifications and alterations to text files. This dual nature of the patch poses significant challenges for existing models [10,19,21] that are primarily designed to analyze source code. Specifically, the incorporation of textual changes, such as comments and documentation, introduces essential context for understanding the rationale behind the code revisions. However, these elements often do not conform to the structural patterns expected by traditional code-focused analysis methods.

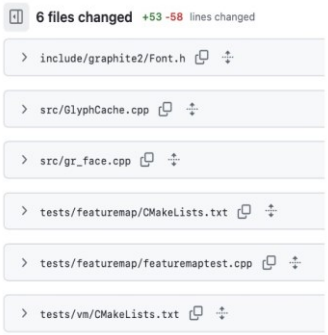


Fig. 2. CVE-2018-7999 patch.

Furthermore, the composition of this patch surpasses the input limitations of certain machine learning frameworks, such as BERT, which are constrained to processing sequences of up to 512 tokens. The mixed-language aspect of the patch, combined with the substantial amount of data it contains, exacerbates the difficulty in effectively capturing the comprehensive scope of the modifications using conventional approaches.

Given these challenges, there is a compelling argument for the adoption of LLM that is capable of handling extensive and varied inputs. These advanced models can potentially address the intricacies present in complex patches more accurately. Consequently, utilizing such models could significantly enhance our capacity to identify meaningful patterns and derive valuable insights from a wide array of software artifacts, ultimately contributing to improved software security and reliability.

3 Methodology

We aim to classify whether a commit is related to vulnerability-fixing or not. Figure 3 illustrates our proposed vulnerability-fixing commit classification framework, where the input consists of the commit text description and its code changes, and the output is

the classification result of the commit. If a commit doesn't involve vulnerability fixing, it outputs "no"; otherwise, it outputs "yes".

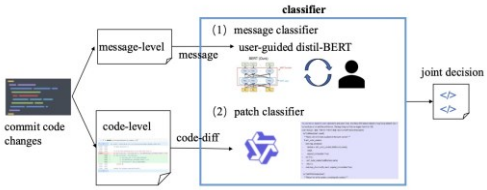


Fig. 3. The overview of vulnerability-fixing commit classifier.

Specifically, our proposed framework consists of three components:

User-Guided Message Classifier: A DistilBERT-based classifier that has been fine-tuned iteratively with human feedback to identify commit messages related to vulnerability fixes. We add human involvement to enhance the model's predictions for ambiguous or challenging cases.

LLM-Aided Patch Classifier: We opt for Qwen-max-longcontext (2000 token) [22] to evaluate the relevance of code changes to vulnerability-fixing. LLMs are effective in handling diverse programming languages and lengthy contexts.

Joint Decision: A stacking-based meta-classifier that combines the outputs of the message and patch classifiers to produce the final judgment.

3.1 Message Classifier

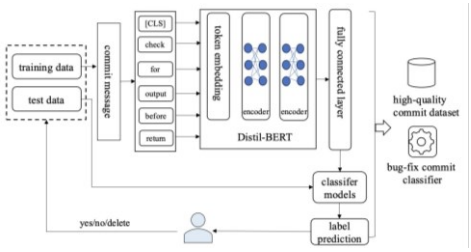


Fig. 4. The framework of user-guided fine-tuning BERT model.

User-guided BERT framework

Initial Training: As shown in Figure 4, A base BERT classification model is trained on an existing labeled dataset. This model provides initial predictions for vulnerability-fixing classification.

Confidence Assessment and Sample Selection: During the testing phase, the model outputs a confidence score (the predicted probability of being a vulnerability-fixing commit message) for each sample. Given that this is a binary classification problem, we set a confidence threshold of 0.5. If the model's confidence in a particular sample falls below this threshold, it indicates that the prediction for this sample is unreliable, marking it as a high-uncertainty sample. These uncertain samples are then added to a review queue and submitted for human verification to enhance the quality of data labeling.

Human-Machine Interaction Feedback Loop: Human feedback is primarily applied to low-confidence samples (0-0.5), which typically fall into two overlapping categories: domain-specific descriptions and ambiguous expressions. These samples represent cases where the classifier struggles to resolve the underlying semantics or contextual nuances due to a lack of prior knowledge or insufficient clarity in the input data. Upon human review of uncertain samples, three types of feedback outcomes are provided:

-Accept Model Prediction: If the human reviewer confirms the accuracy of the model's prediction, the label of the sample remains unchanged, and the confidence score is updated to reinforce the model's correct judgment.

-Label Correction: If the human reviewer identifies an error in the model's prediction, the sample label is corrected. This corrected label is then used to improve the model's handling of similar samples during subsequent training.

-Sample Removal: Remove samples that are too ambiguous or lack sufficient information for meaningful classification. (1) The description is very ambiguous, for example, the commit message of CVE-2017-14166 is: "Do something sensible for empty strings to make fuzzers happy." (2) The description does not contain any information, such as "...". Removed samples are excluded from both training and testing sets to avoid biases that could unfairly enhance performance metrics.

Model Update and Iterative Training: The updated dataset, incorporating human feedback, is reused for further training and testing. Over successive iterations, the model adapts to challenging cases, reducing its dependence on human intervention.

The user-guided updating message classifier

When a human confirms that the prediction is correct, we can treat the sample as a "high confidence" sample, and by increasing the weight or adjusting the learning rate of the model on this sample, the model can increase the prediction confidence when it encounters similar samples in the future, thereby reducing the manual dependence on similar samples in the future.

We use sample resampling or confidence smoothing update techniques to update the dataset. Both techniques are not always applied simultaneously. Resampling is used to reinforce the importance of high-confidence samples, while confidence smoothing adjusts model outputs incrementally to avoid drastic fluctuations. The choice of technique depends on the specific needs of the training phase. For example, resampling may be prioritized in earlier iterations to address rare patterns, while confidence smoothing is more effective in stabilizing predictions during later stages.

Resampling: A sample is added to the training set multiple times to strengthen the model's ability to predict this sample and similar samples. By learning the same example multiple times, the model will become more sensitive to its features, thus increasing its confidence in similar examples.

Confidence Smoothing Update: We directly adjust the confidence score of the model output to perform a "smoothing" update using a weighted average for a specific sample. Assuming that the original confidence is C', the confidence after manual confirmation can be updated as:$C = \alpha \cdot 1 + (1 - \alpha) \cdot C'$. Where α is a small weight that

indicates the strength of the confidence smoothing. Through this method, the confidence score of the model on the sample will be improved, and the drastic fluctuation of the confidence score is avoided.

3.2 Patch Classifier

LLMs possess strong contextual understanding capability and extensive applicability across programming languages. The adaptability of LLMs to different programming languages also ensures that LLM-based evaluation works across different code syntaxes and structures.

To effectively leverage LLMs' contextual understanding capability, we craft a task-specific prompt to evaluate the relevance of code changes to the corresponding vulnerability-fixing, as depicted in Figure 5. This prompt consists of three components:

(1) System prompt [23]: The LLM acts as an expert, analyzing code bugs and their corresponding fixes.

(2) Answer prompt: The LLMs evaluate the relevance of code changes to vulnerability-fixing in the file and output "YES" or "NO" indicating whether the code change is related to vulnerability-fixing. Given a confidence score within 0-100.

(3) Code Change: The specific code changes made in the file.

System prompt	You are now an expert in code vulnerability and patch fixes.
Answer prompt	Are these diffs function feature related or vulnerability-fixing related? Give the result yes or no and the confidence. The vulnerability-fixing confidence ranges from 0 to 100.

Code change	@@ -749,10 +749,10 @@ class ControlFlowContext(object): def ExitResult(self, result):
	"""Make a list of tensors available in the outer context."""
	if self. outer context:
	- nest.map structure(
	- lambda x: self. outer context.AddName(x.name),
	- result,
	- expand composites=True)
	+ def fn(x):
	+ self. outer context.AddName(x.name)
	+ return x
	+ nest.map structure(fn, result, expand composites=True)
	def GetWhileContext(self):
	"""Return the while context containing this context."""

Fig. 5. A sample prompt for LLMs evaluating the relevance of a code change to the vulnerability-fixing.

3.3 Joint Decision

We use two classifiers, each focusing on a distinct aspect of the commit. Individually, each classifier can capture information about one aspect of a commit. We enhance classification performance by combining the outputs of the message classifier and the patch classifier. Through this fusion method, we effectively combine the features of message and patch, and can more robustly cope with the performance differences of each model in different situations.

Similar to previous studies [19], we employ a stacking-based method [24]. Stacking methods use a meta-classifier that learns how to combine the outputs of each base classifier to produce a more accurate final prediction. Specifically, we obtain the confidence of whether a commit is vulnerability-fixing from the message classifier and the patch classifier. We then convert the confidence into probabilities to process this information uniformly in the stacking process.

The stacking-based joint decision method can be divided into five implementation steps. The processing procedures are described below according to the characteristics of base classifiers:

Step 1: Base Classifier Output Acquisition.

- Text Information Classifier: Directly outputs probability values through softmax:

$$P_t(x) = f_t(x) \in [0,1] \tag{1}$$

Where f_t denotes the trained Distil-BERT model, outputting probabilities via softmax.

- Patch Classifier: Obtains raw confidence scores through LLM prompt engineering:

$$S_p(x) = LLM(Code\ Change\ Description) \in \{0,1, \dots ,100\} \tag{2}$$

Step 2: Confidence Standardization: Apply linear scaling only to the patch classifier.

$$C_p(x) = \frac{S_p(x)}{100} \Rightarrow C_p(x) \in [0,1] \tag{3}$$

Step 3: Meta-Feature Construction: Combine heterogeneous outputs into a two-dimensional feature vector.

$$F(x) = \left[P_t(x), C_p(x)\right] \in R^2 \tag{4}$$

Step 4: Meta-Classifier Training.

- Define a regularized logistic regression model:

$$g\big(F(x)\big) = \sigma\big(w_0 + w_1 P_t + w_2 C_p\big) \tag{5}$$

Where $\sigma(z) = 1/(1 + e^{-z})$ is the sigmoid function.

- Optimization objective (cross-entropy loss + L2 regularization):

$$\mathcal{L} = -\sum_{i=1}^{N} [y_i \ln g_i + (1 - y_i) \ln(1 - g_i)] + \lambda \big||w|\big|_2^2 \tag{6}$$

- Parameter learning process: Update via gradient descent:

$$w_j = w_j - \eta \left(\sum_{i=1}^{N} (g_i - y_i) F_{i,j} + 2\lambda w_j \right) \tag{7}$$

where η is the learning rate, and $F_{i,j}$ denotes the $j - th$ feature of the $i - th$ sample.

Step 5: Joint Decision

$$\hat{y}(x) = \begin{cases} 1 \ \ if g\big(F(x)\big) \geq \tau \\ 0 \ \ otherwise \end{cases}, \tau \in (0,1) \tag{8}$$

The threshold τ is determined by maximizing the F1-score on the validation set.

Calculation Example: 1. Input code commit x: "A patch fixing an SQL injection vulnerability"; 2. Text classifier analyzes commit message "Fix parameterized query vulnerability" $\rightarrow P_t = 0.95$; 3. LLM evaluates code changes $\rightarrow$ outputs score 90 $\rightarrow C_p = 0.90$;4. Meta-feature construction:$F(x) = [0.95,0.90]$; 5. Learned parameters: $w_0 = -0.2, w_1 = 1.1, w_2 = 0.9$;6. Linear combination: $-0.2 + 1.1 \times 0.95 + 0.9 \times 0.90 = 1.655$;7. Sigmoid transformation: $1/(1 + e^{-1.655}) = 0.839$;8. Decision: $0.839 > \tau(0.6)\hat{y}(x) = 1$.

4 Evaluation

Our experiments are driven by these research questions:

RQ1: How effective is the user-guided approach for optimizing the dataset and improving the vulnerability-fixing classification model?

RQ2: How efficient is a large language model in handling code changes?

RQ3: How efficient is the method proposed in this paper compared to previous work?

4.1 Experimental Setting

Dataset

To evaluate the proposed method, we utilize multiple datasets tailored for different components of the framework.

-**User-Guided Message Classifier**: We use the C/C++ language part of the dataset proposed by Reis et al. [15], which comes from 370 open-source projects. The number of data used for fine-tuning is 9269, including 3013 positive data and 6256 negative data. In addition, we validate the trained classifier on five real-world datasets [16-18].

-**LLM-Aided Patch Classifier and Overall Method Validation**: We use the TensorFlow dataset introduced by Nguyen et al. [10] which is specifically designed for vulnerability-fixing classification. Unlike the Reis dataset, the TensorFlow dataset includes both commit messages and corresponding code-change information, which is essential for patch-level analysis and evaluating the overall method.

Fine-tuning BERT model

We split the dataset into 80%, 10%, and 10% as train set, validation set, and test set. We use the distil-BERT model to classify commit messages. Distil-BERT is a lightweight variant of the BERT model. Compared with traditional BERT, distill-BERT runs 60% faster while maintaining 97% performance, but only has 60% parameters [25]. We add a dense layer with 2 neurons and sigmoid activation at the end. We trained on the original dataset with batches 8 and 16, and the results are shown in Figure 6. By comparing loss and accuracy, the results with batch 16 are slightly better than those with batch 8. It is optimal when the epoch is 5. Similarly, we used a learning rate of (1e-5, 5e-5), which is optimal at 2e-5. Therefore, we trained the model for 5 epochs with a learning rate of 2e-5, and a batch size of 16.

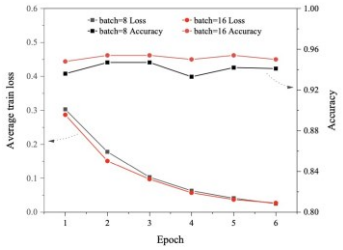


Fig. 6. Loss and Accuracy over Epochs.

User selection and label judgment standard

To ensure the accuracy of manual verification, users participating in this user bootstrap process should meet the following criteria: (1) **Software development experience:** Users should have at least two years of practical programming or software development experience to ensure their understanding of code changes and commits. (2) **Familiarity with version control systems:** Users should be familiar with a version control system such as Git, especially with the semantics of commit history and code changes. (3) **Understand the definition and characteristics of vulnerability-fixing**:

Users should have a grasp of common vulnerability-fixing patterns and be able to recognize different types of commits. (4) **Have participated in the bug tracking and fix process**: Users should have participated in at least one complete software bug tracking, fix process, and be able to identify typical vulnerability-fixing commits during development. To prevent subjective bias, the criteria defined in Table 1 should first be followed during manual confirmation. Second, double audit is used for samples with very low confidence, requiring at least two users to independently confirm the label, and then the label is updated after consensus.

Table 1. Standards for manual processing of labels.

Judgment	Standard
Accept	(1) Explicit vulnerability-fixing description: Contains obvious bug fix keywords or statements (e.g.,"fix bug", "resolve issue", "correct error", etc.). (2) Have a direct impact: The code change section has a direct impact on the program logic or functionality, fixing known problems with specific functionality or modules. (3) The actual logic of the code change is consistent with the commit description. (4) Commits follow the common vulnerability-fixing pattern.
Reject	(1) Lack of explicit vulnerability-fixing instructions. (2) The code changes are only enhancements, performance improvements, code refactoring, or documentation updates, not vulnerability-fixinges. (3) Description does not match code change.
Delete	(1) Commits are too vague or short to judge the actual purpose of the commit. (2) Commits are empty or contain no actual code changes.

4.2 Experiment Results

RQ1-Effectiveness of User-Guided Approach

We conducted 10 rounds of experiments. The dataset is updated in each round, and the training, validation, and test sets are generated randomly. In each round, users reviewed the samples identified by the model as having high uncertainty. As shown in Figure 7, when giving feedback to people for verification, we provided URL references for negative data, and CVE information for positive data in addition to the URL. Multi-source information enables users to process data more efficiently and reduces reliance on a single source.

Analysis of Feedback Rounds: Figure 8 shows the trend in validation and intervention over 10 rounds. Early rounds required more user interventions, while later rounds showed a gradual decline, indicating the model's increasing ability to classify samples independently. However, after 8 rounds, performance improvements became marginal, as shown in Figure 10. This suggests diminishing returns after sufficient iterations. Over-intervention in later rounds could lead to overfitting or dependence on specific patterns, which we address by capping the number of iterations. In each round of validation, no more than ten minutes were spent manually. However, this assessment is subjective.

Input Text:	fix stack buffer overflow with old curl curl _ easy _ getinfo expects a long for curlinfo _ activesocket, but curl _ socket _ t is an int, which was causing a stack buffer overflow and crash.
True Label:	[0.0, 1.0]
Predicted Probabilities:	[0.9972237348556519, 0.002813258673995733]
Predicted Labels:	[1, 0]
URL:	https://github.com/Cisco-Talos/clamav-devel/co..
Do you accept this prediction? (y/n/d):	**y**

Fig7. A user-guided example: the input text represents the commit message. After the model makes a prediction, it will provide predicted probabilities and predicted labels, along with a URL for the user to reference in order to make a judgment.

Analysis of Resampling and Smoothing: Figure 9 illustrates the effect of different resampling counts and smoothing factors on the model performance metrics after one round of user guidance. In Figure 9 (a), when the count of resampling is 0, the four metrics of the model are at a low level. When the count of resampling increases to 10, the metrics improve significantly, especially the improvement of accuracy and precision is more significant. However, as the count of resampling increases, the model performance starts to level off with slight fluctuations. This result indicates that moderate resampling can effectively improve the classification performance of the model. Still, excessive resampling does not further improve the performance and may lead to overfitting of the model or increased dependence on the training data. In this experiment, the model performs best when the count of resampling was 10. Figure 9 (b) examines the effect of varying smoothing factors on the performance of the model. With the increase of smoothing factor, the accuracy of the model is gradually improved, but the changes of precision, recall, and F1 are more complex, and some metrics fluctuate or decline. When the smoothing factor is low, such as 0.05, precision and recall are low, while accuracy is relatively high. As the smoothing factor increases to 0.10, the accuracy reaches the highest value, but the other indicators do not show significant improvement. After further increasing the smoothing factor to 0.20, the accuracy remained at a high level, but the precision and F1 decreased slightly.

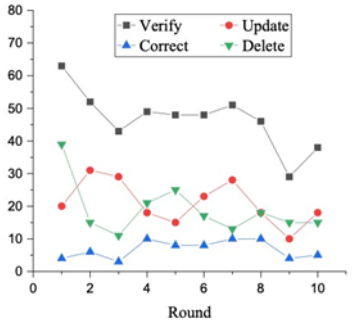
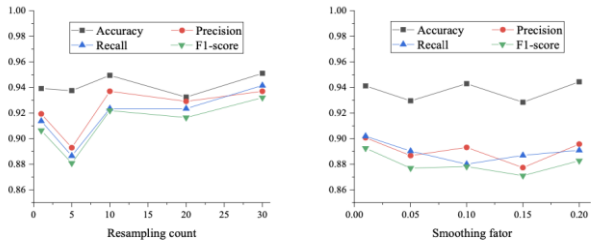


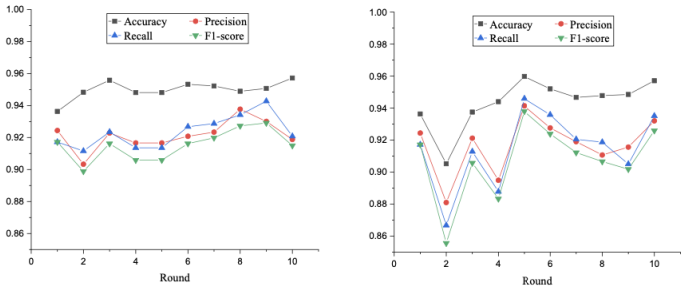
Fig. 8. The number of samples to be validated for each round of user guidance, after ten rounds of interaction. Update indicates the number of samples to update the confidence. Correct indicates the number of sample labels changed, which is the wrong sample label. Delete indicates the number of samples deleted.

According to the analysis in Figure 9, we chose the resampling number as 10 and the smoothing factor as 0.01 to conduct 10 rounds of user-guided vulnerability-fixing commit message classification experiments, and the results are shown in Figure 10. As the number of user-guided rounds increases, the resampling and smoothing update methods show an upward trend in general. In smoothing, the performance of the model fluctuates a lot in the initial rounds, especially the recall and F1. This fluctuation is due to the uncertainty of labels caused by the smooth update of confidence, which affects the training of the model. Resampling improves model performance more consistently.



(a) Varying resampling count (b) Varying smoothing factor

Fig. 9. The effect of varying resampling counts and varying smoothing factors, after one round of user-guided.



(a) Results of ten rounds user-guided with Resampling count=10

(b) Results of ten rounds user-guided with smoothing factor=0.01

Fig. 10. The performance of the message classification model after ten rounds of user-guided resampling and confidence smoothing updating.

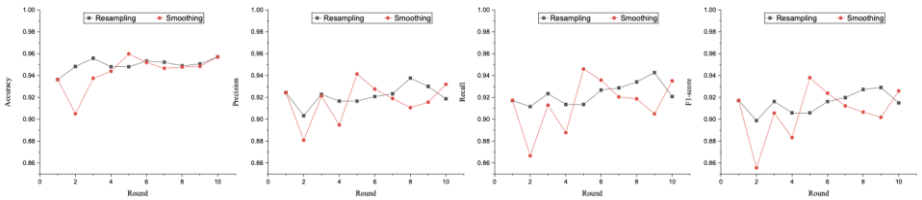


Fig. 11. Comparison results of resampling and smoothing in four dimensions of accuracy, precision, recall, F1 after ten rounds of user guided.

Figure 11 shows the comparison between resampling and smoothing in terms of accuracy, precision, recall and F1. In the user-guided scenario, resampling is generally better than smoothing update in various indicators, especially in accuracy and F1. However, resampling causes large fluctuations in the number of original datasets.

In general, the overall accuracy of the model is on the rise, and the tenth round has increased by 2.6% compared with the first round. To avoid overfitting to the same dataset, we additionally selected five real-world datasets and directly tested the model with user-guided fine-tuning. As shown in Table 2, the models which used resampling after 5 rounds and 10 rounds have higher accuracy than the original model, with a maximum increase of 5%.

Analysis of Noisy Labels: In the process of user-guided, we found the Reis dataset [15] has a label error rate of 14%, while the TensorFlow dataset [10] has a rate of 9.4%. The label error rate refers to the proportion of samples incorrectly labeled by automated methods. These errors were later corrected by human reviewers. According to the statistics of the wrong labels, we found that 95% came from negative data, and the reason was analyzed, which was because during data collection, positive data could be pointed to through other information sources, while the process of collecting and labeling negative data was relatively simple. In addition, we find that 60% of the samples with wrong prediction by the model have clear CVE URL pointing, but their commit message description are hardly vulnerability-fixing related.

Table 2. Results on real-world datasets for the user-guided fine-tuned model.

Dataset(number)	Model	Accuracy	Precision	Recall	F1
Zafar et al. [16] (324)	distilBERT	69.14%	56.14%	29.91%	39.02%
	User-guided5	70.06%	58.33%	32.71%	41.92%
	User-guided10	**72.22%**	**64.91%**	**34.58%**	**45.12%**
Zafar et al. [16] (1831)	distilBERT	66.90%	40.21%	21.31%	27.86%
	User-guided5	67.12%	42.04%	25.50%	31.75%
	User-guided10	**68.49%**	**46.17%**	**30.78%**	**36.94%**
Liven et al. [17] (1149)	distilBERT	58.40%	72.92%	7.00%	12.77%
	User-guided5	59.44%	81.48%	8.80%	15.88%
	User-guided10	**59.79%**	**80.65%**	**10.00%**	**17.79%**
Berger et al. [18] (374)	distilBERT	65.24%	69.23%	12.86%	21.69%
	User-guided5	66.84%	76.67%	16.43%	27.06%
	User-guided10	**69.25%**	**85.71%**	**21.43%**	**34.29%**
Berger et al. [18] (subset) (270)	distilBERT	71.48%	84.21%	17.78%	29.36%
	User-guided5	72.96%	86.96%	22.22%	35.40%
	User-guided10	**74.44%**	**88.89%**	**26.67%**	**41.03%**

RQ2-Effectiveness of LLM

Table 3 shows a comparison of the performance of three models on the task of classifying vulnerability-fixing code changes. The SVM and BERT results come from reproducing the papers [19] and [10].

Table 3. Comparison of the performance of models on the task of classifying vulnerability-fixing code changes.

Model	Mixed-form	Accuracy	Precision	Recall	F1
SVM	no	0.82	0.74	0.66	0.69
BERT	no	0.96	0.94	0.78	0.85
LLM (message+code-change)	yes	0.51	0.82	0.51	0.51
LLM (code-change)	yes	**0.85**	**0.63**	**1.00**	**0.77**

The BERT model achieves the highest accuracy (0.96) and F1 (0.85), with a precision of 0.94 and a recall of 0.78. This strong performance can be attributed to the fact that BERT was specifically fine-tuned on the dataset used for this task, allowing it to learn task-specific features and effectively balance precision and recall.

LLM (message+code-change) and LLM (code-change) indicate that the same prompt was input into the LLM, but with different levels of commit. The results show that the performance of LLM (code-change) is better. Although LLM (message+code-change) theoretically combines two sources of information, the noise in the message component may weaken the clear signal provided by code-change due to the uneven importance and reliability of message and code-change. When LLM focuses solely on code-change, it can fully leverage its ability to understand the logic and semantics of the code. The Recall of LLM (code-change) reaches 1.0, indicating that it can completely encompass vulnerability-fixing commits and mitigate the risk of false negatives. Although the Precision of LLM (code-change) is low, it can be enhanced by combining with message classifier to achieve a more balanced overall performance.

RQ3-Effectiveness of Our Method

The "message" column in Table 4 shows that our user-guided message classifier outperforms HERMES by 5.7% and VulCurator by 13.6%. The "ensemble" column in Table 4, which represents the final results, shows that our method achieves 14.6% improvement over HERMES and 5.6% improvement over VulCurator.

Table 4. F1 score of Previous work and our methodology on TensorFlow dataset.

Baseline	Method	Message	Patch	Ensemble
HERMES[19]	SVM	0.87	0.69	0.82
VulCurator[10]	RoBERT+CodeBERT	0.81	0.85	0.89
LLM(only)	Qwen	0.83	0.77	0.51
Our Method	**User-guided-BERT+LLM**	**0.92**	**0.77**	**0.94**

The F1-score of the LLM in commit message classification is 0.83, which is lower than HERMES (0.87) and our method (0.92). Commit messages are often succinct and lack informative language, which poses a challenge for LLMs that rely on contextual understanding. Although LLMs have advantages in processing natural language, they struggle to accurately capture signals related to vulnerability fixes without proper training or fine-tuning. Our work excels due to the improved message classification enabled by the user-guided approach and the scalability of LLM. The message classifier con-

centrates on domain-specific and ambiguity in commit messages, while the LLM focuses on the logical and semantic analysis of code changes. By employing a stacking method that combines the strengths of both modules, we achieve a final F1-score of 0.94. Additionally, our approach demonstrates enhanced language independence and adaptability.

5 Related work

The research on commit classification is mainly divided into two types. One [8,9,26-29] is to follow Swanson [30] and divide commit into three categories: "Corrective", "Perfective" and "Adaptive". Another category is vulnerability-fixing-related commits [10,16,19,21,31,32]. Existing studies have automatically classified commits into respective categories, ranging from searching indicative keywords to complex machine learning models based on a wide range of features, with limited success.

Using static keywords or any other embedding method that does not consider the context of each word in the commit message will lead to a misclassification of ambiguous commit messages, that is, messages containing words that can be used in commit messages belonging to different maintenance categories. The context was well captured using BERT, but some of the content had to be discarded due to BERT's 512 token length limit.

Most of the existing methods use BERT model in processing text and code. Different from previous work, this paper uses distil-BERT model in processing text and LLM in processing code. Compared with the BERT model, LLM breaks through the limitation of the code language and input length. It features a greater number of parameters, enhanced generative capabilities, and superior advantages in understanding and processing code.

6 Conclusion

We propose a user-guided classification method for vulnerability-fixing commits. We use stacking method to combine the message classifier based on user-guided distil-BERT and the patch classifier based on LLM. We use the existing dataset to train the model, and through the user-guided method, verify the wrong labels in the dataset, and iteratively optimize the dataset and the model. Our model achieves an F1 score of 94%, outperforming HERMES by 14.6% and VulCurator by 5.6%. Our work improves analysis techniques that provide better insight and visibility into the development process and reduce the likelihood of errors.

References

1. GitHub.
2. Tjaša Heričko and Boštjan Šumak. Commit Classification Into Software Maintenance Activities: A Systematic Literature Review. In 2023 IEEE 47th Annual Computers, Software,

and Applications Conference (COMPSAC), pages 1646–1651, June 2023. ISSN: 0730-3157.
3. Tjaša Heričko, Boštjan Šumak, and Sašo Karakatič. Commit-Level Software Change Intent Classification Using a Pre-Trained Transformer-Based Code Model. Mathematics, 12(7):1012, January 2024.
4. Tjaša Heričko. Automatic data-driven software change identification via code representation learning. In Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, EASE '23, page 319–323, New York, NY, USA, 2023. Association for Computing Machinery.
5. Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 1, pages 134–144, May 2015.
6. Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E. Hassan. Finding A Needle in a Haystack: Automated Mining of Silent Vulnerability Fixes. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 705–716, November 2021.
7. Fei Zuo, Xin Zhang, Yuqi Song, Junghwan Rhee, and Jicheng Fu. Commit Message Can Help: Security Patch Detection in Open Source Software via Transformer. In 2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA), pages 345–351, May 2023.
8. Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. Information and Software Technology, 135:106566, 2021.
9. Jian Yi David Lee and Hai Leong Chieu. Co-Training for Commit Classification. In WNUT, 2021.
10. Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Xuan-Bach D. Le, and David Lo. VulCurator: A vulnerability- fixing commit detector. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, pages 1726–1730, New York, NY, USA, November 2022. Association for Computing Machinery.
11. Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. Gsmsymbolic: Understanding the limitations of mathematical reasoning in large language models, 2024.
12. Hengxing Cai, Xiaochen Cai, Junhan Chang, Sihang Li, Lin Yao, Changxin Wang, Zhifeng Gao, Hongshuai Wang, Yongge Li, Mujie Lin, Shuwen Yang, Jiankun Wang, Mingjun Xu, Jin Huang, Xi Fang, Jiaxi Zhuang, Yuqi Yin, Yaqi Li, Changhong Chen, Zheng Cheng, Zifeng Zhao, Linfeng Zhang, and Guolin Ke. Sciassess: Benchmarking llm proficiency in scientific literature analysis, 2024.
13. Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: E!cient long-context llm inference with retrieval and streaming heads, 2024.
14. Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware llm reasoning, 2024.
15. Sofia Reis and Rui Abreu. A ground-truth dataset of real security patches, October 2021.
16. Sarim Zafar, Muhammad Zubair Malik, and Gursimran Singh Walia. Towards Standardizing and Improving Classification of Bug-Fix Commits. In 2019 ACM/IEEE International

Symposium on Empirical Software Engineering and Measurement (ESEM), pages 1–6, September 2019.
17. Stanislav Levin and Amiram Yehudai. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE, page 97–106, New York, NY, USA, 2017. Association for Computing Machinery.
18. Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality: A reproduction study. ACM Trans. Program. Lang. Syst., 41(4), oct 2019.
19. Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E. Santosa, Asankhaya Sharma, and Ming Yi Ang. HERMES: Using Commit-Issue Linking to Detect Vulnerability-Fixing Commits. In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 51–62, March 2022.
20. Muhammad Usman Sarwar, Sarim Zafar, Mohamed Wiem Mkaouer, Gursimran Singh Walia, and Muhammad Zubair Malik. Multi-label classification of commit messages using transfer learning. 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pages 37–42, 2020.
21. Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E. Hassan, Xuan-Bach D. Le, and David Lo. Multi-Granularity Detector for Vulnerability Fixes. IEEE Transactions on Software Engineering, 49(8):4035–4057, August 2023.
22. Alibaba cloud. “tongyi qianwen”. https://qianwen.aliyun.com/.
23. Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. Reposvul: A repository-level high-quality vulnerability dataset. In 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pages 472–483, 2024.
24. David H. Wolpert. Stacked generalization. Neural Networks, 5(2):241–259, 1992.
25. Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. ArXiv, abs/1910.01108, 2019.
26. Muhammad Usman Sarwar, Sarim Zafar, Mohamed Wiem Mkaouer, Gursimran Singh Walia, and Muhammad Zubair Malik. Multi-label Classification of Commit Messages using Transfer Learning. In 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pages 37–42, October 2020.
27. Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages using multi-label active learning. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC ’19, pages 1760–1767, New York, NY, USA, April 2019. Association for Computing Machinery.
28. Jiajun Tong, Zhixiao Wang, and Xiaobin Rui. Boosting Commit Classification with Contrastive Learning, August 2023.
29. Yasin Sazid, Sharmista Kuri, Kazi Ahmed, and Abdus Satter. Commit Classification into Maintenance Activities Using In-Context Learning Capabilities of Large Language Models. In Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering, pages 506–512, Angers, France, 2024. SCITEPRESS – Science and Technology Publications.
30. E. Burton Swanson. The dimensions of maintenance. In Proceedings of the 2nd International Conference on Software Engineering, ICSE ’76, page 492–497, Washington, DC, USA, 1976. IEEE Computer Society Press.

31. Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2A: A dataset built for AI-Based vulnerability detection methods using di"erential analysis. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 111–120, 2021.
32. Antonino Sabetta and Michele Bezzi. A Practical Approach to the Automatic Classification of Security-Relevant Commits. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 579–582, September 2018.