# Not Every Patch is an Island: LLM-Enhanced Identification of Multiple Vulnerability Patches

Yi Song[1], Dongchen Xie[2†], Lin Xu[2†], He Zhang[1], Chunying Zhou[1], Xiaoyuan Xie[1*]
[1]*School of Computer Science, Wuhan University*, Wuhan, China
[2]*School of Cyber Science and Engineering, Wuhan University*, Wuhan, China
{yisong, xiedongchen, xulin_xl, zhanghe, zcy9838, xxie}@whu.edu.cn

***Abstract*—For a vulnerability reported as an item of platforms such as CVE or NVD, software maintainers need to submit patches (in the form of *code commit*) to fix it, which is often performed silently for the sake of keeping products' reputation or avoiding malicious attacks. But such a silent practice keeps patches hidden from affected downstream software maintainers, thus they have to identify patches in a large corpus of code commits manually, i.e., silent vulnerability patch identification (SVPI). Existing techniques in this field were often developed under the assumption that a vulnerability is matched to one patch, thus output a ranking list that simply reflects the similarity between one individual patch and the vulnerability. However, previous research has demonstrated that many vulnerabilities correspond to more than one patch in practice, this phenomenon largely threatens the effectiveness of existing SVPI techniques because they typically ignore the correlation between patches. In this paper, we propose SHIP, a Silent vulnerability patcH Identification approach suited for multiPle-patch scenarios, to make patches corresponding to a vulnerability no longer isolated islands. For a vulnerability item, we first obtain several highly-relevant code commits by measuring heuristic features, and then employ a large language model (i.e., DeepSeek-V3) to predict both the link between a code commit and the vulnerability as well as the link between a pair of code commits, and thus deliver candidate groups each containing one or more code commits that could be patches of the vulnerability. Finally, we perform the max-pooling strategy on the features of code commit(s) contained in each candidate group to determine the ranking of groups, the Top-1 group will be output. The experimental results demonstrate the promise of SHIP: on the benchmark consisting of 4,631 vulnerability items, it can achieve 84.30%, 59.14%, and 69.51% of Recall, Precision, and F1-Score, respectively, outperforming the state-of-the-art SVPI technique by 37.54%, 28.71%, and 32.35%, respectively.**

***Index Terms*—Security patches, Software vulnerability, Patch identification, Large language models**

## I. Introduction

Open-source software communities bring together a large number of developers and maintainers, and their collaboration forms a huge ecosystem, providing stakeholders with a friendly and efficient platform [1], [2]. As with a double-edged sword, the harm of software vulnerabilities will be greatly amplified in such an ecosystem. Specifically, once a vulnerability is introduced into a project, downstream projects that invoke the project could be also threatened by the vulnerability [3], [4]. For the sake of maintaining the reputation of products or avoiding malicious attacks, upstream project owners typically submit patches (in the form of *code commit*) silently without notifying the maintainers of the downstream projects that could be affected [1], [4]–[6]. As a consequence of this, patches are often buried within a bundle of code commits that are not relevant to the vulnerability, so that the maintainers of the downstream projects have to carry out manual identification to perform the fix [7]. This silent vulnerability patch identification (SVPI) problem has attracted considerable attention from both academia and industry [5], [8]–[10], because not finding silent patches and fixing vulnerable software in time can cause serious damage. For example, Equifax, a famous US credit rating agency, did not apply a security patch for a vulnerability in Apache Struts 2 (CVE-2017-5638 [11]) in a timely manner, causing the personal information of nearly half of Americans to be leaked due to an attack. An investigation found that the patch for the vulnerability was released two months before the attack, but was hidden in many non-security code commits [12]. The typical pipeline of SVPI involves three steps. First, determining candidate code commits of the given vulnerability and then defining and extracting their features. Second, further representing the vulnerability and candidate code commits to make them have a comparable form by embedding their features in a higher-dimensional space. Third, training a model to predict code commits' relevance to the vulnerability based on the representation and producing a final ranking list [9], [10], [13], [14]. Following this principle, many SVPI techniques emerged and have been demonstrated to be effective [1], [9], [10], [13], [15], achieving high recall values and making the vulnerability patch stand out from candidate code commits.

Despite the progress of this field, current techniques are typically under the assumption of "one-to-one" relationship, that is, a vulnerability matches only one patch [9], [14]. Therefore, they consider the similarity between each individual code commit and the given vulnerability, and rank candidate code commits based on this similarity. The effectiveness of these SVPI approaches is typically evaluated by Recall@$r$, i.e., an approach can order the patch in the Top-$r$ position of the ranking list on how many vulnerability items. However, much research has pointed out that such "one-to-one" relationships

† Co-first authors.
∗ Corresponding author.
This work was partially supported by the National Key R&D Program of China (No. 2024YFF0908003), the National Natural Science Foundation of China (No. 62472326), CCF-Zhipu Large Model Innovation Fund (No. CCF-Zhipu202408).

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00087

are difficult to hold in open-source ecosystems in the real world. For example, the empirical study carried out in the reference [15] reveals that multiple patches are developed for about 41% of the vulnerabilities, which means that nearly half of the vulnerabilities are associated with multiple patches. Hommersom et al.'s work supports this opinion, stating that a vulnerability might be addressed by a set of code commits that represent the fix together [1]. Similarly, Bhandari et al. observe a phenomenon that a patch referenced to a vulnerability in current public platforms like CVE (Common Vulnerabilities and Exposures) [16] or NVD (National Vulnerability Database) [17] could not be complete as it can still leave unaddressed issues, motivating an important future topic that analyzing consecutive patches [18]. Thus, we can conclude that "one-to-many" relationships between vulnerabilities and patches are very common, they account for a considerable proportion of the open-source vulnerability library [19]–[23].

The "one-to-many" relationship between vulnerabilities and patches can largely threaten the effectiveness of existing SVPI techniques. This is because for a vulnerability associated with multiple patches, if we follow traditional SVPI principles to mainly focus on the similarity between the given vulnerability and each candidate code commit, only those code commits that are directly associated with the vulnerability (i.e., have explicit feature similarities with it) can be identified. However, apart from these easy-to-track patches, there could be many other patches fixing the vulnerability that are difficult to link directly to the vulnerability and thus buried [9], [10], [13]–[15]. The causes of such hard-to-track patches can be varied, we explain these causes in detail by giving examples in Section II-B. Furthermore, it is important to note that for an SVPI technique that needs to recommend multiple patches that may be related to one another, producing a ranking list of code commits is not the best practice. This is because this form fragments the interrelationship between patches and necessitates that maintainers look for patches from the list by themselves, which raises labor costs. Consequently, a more hassle-free output form is required.

To tackle the above challenges, in this paper, we propose SHIP, a Silent vulnerability patcH Identification approach suited for multiPle-patch scenarios, to make patches corresponding to a vulnerability no longer isolated islands. The intuition of SHIP is that an effective SVPI technique should take into account the internal relevance between code commits, not just focusing on the relevance between the given vulnerability and code commits. Specifically, for a given vulnerability, SHIP first utilizes four categories of rule-based features and semantic features to measure the relevance between the vulnerability and candidate code commits preliminarily, and filters a number of most relevant code commits into further identification. Then, we feed each filtered code commit into a large language model (LLM) DeepSeek-V3 [24] and design a prompt to predict whether any two code commits aim to fix the same vulnerability. Apart from the prediction result, we also extract heuristic and semantic features between two code commits to determine their relevance. Third, code commits that have high relevance will be connected and thus delivering candidate groups of code commits. Later, we generate feature vectors for every code commit in a group and perform the max-pooling strategy to integrate them into a whole one, which will be used to determine the relevance between this group and the given vulnerability. Finally, the most (Top-1) relevant group will be regarded as the *patch group* and delivered to maintainers.

In the experiment, we evaluate SHIP in both single-patch and multiple-patch scenarios considering that in practice the number of patches linked to a vulnerability is not known in advance. We construct a benchmark of 4,631 vulnerabilities including 3,071 single-patch ones and 1,560 multi-patch ones with $2 \sim 23$ patches. Experimental results demonstrate the promise of SHIP: It can achieve 84.30% of Recall, 59.14% of Precision, and 69.51% of F1-Score, outperforming the state-of-the-art (SOTA) technique in this field by 37.54%, 28.71%, and 32.35%, respectively.

The main contributions of this paper are as follows:

- **A novel approach**. We propose SHIP, an automated silent vulnerability patch identification approach that can well handle multi-patch scenarios. By taking into account the relevance between code commits and producing a patch group instead of a ranking list, SHIP achieves a considerable improvement compared with SOTA techniques.
- **A comprehensive evaluation**. To better measure SHIP's capability of identifying patches, we put it into a large corpus of real-world vulnerabilities and candidate code commits, and adopt well-recognized metrics to reveal the strong effectiveness of SHIP in practical environment.
- **A public code and data repository**. To facilitate any future intention of replication or reproduction, we release experimental code as well as the 4,631 vulnerability items and the links to their real patches[1], which can be used as a benchmark in future research.

## II. Background

### A. Silent Vulnerability Patch Identification

Open-source software encourages developers to share their source code, and thus stakeholders can invoke previous projects to avoid reinventing the wheel. This paradigm expands the scope of the harm of software vulnerabilities from a single project to the entire community, because the security of a software system will be bound to all the upstream software it calls [25]. However, when upstream software maintainers fix the vulnerability by submitting a code commit to the repository, they often do not explicitly mention any security-related words in the message of code commits, nor notify downstream stakeholders who call the software [1], [4]–[6], to maintain the reputation of the product or prevent attackers from analyzing the patch code and exploiting it [26], [27]. The cause of silent vulnerability patches may also be responsible for the Coordinated Vulnerability Disclosure principle, which requires that before publicly disclosing a vulnerability, time should be reserved for the owners of the affected software to

[1]https://github.com/SHIP-Repo/Repo.

TABLE I
RANKING OF PATCHES PRODUCED BY EXISTING TECHNIQUES

| Techniques | Patch① | Patch② | Patch③ | Total Effort |
|---|---|---|---|---|
| PatchScout | 36 | 371 | 425 | 425 |
| PatchFinder | 136* | 842* | 1127* | 100* |
| PromVPat | 2 | 13 | 9 | 13 |
| VCMatch | 2 | 16 | 9 | 16 |

* In PatchFinder, only code commits that ranked in the Top-100 during the initial phase will be identified further. The three patches were ranked $136^{\mathrm{th}}$, $842^{\mathrm{nd}}$, and $1127^{\mathrm{th}}$ in the initial phase, hence, the value of Total Effort will be 100.

respond [28]–[30]. As a result, patches are often submitted to the code repository much before the vulnerability is disclosed, so it can be neglected to attach a link (links) to the previous patch(es) when disclosing the vulnerability [31]. Silent vulnerability patches prevent compromised software from being fixed in a timely manner, greatly increasing its risk of being attacked [5], [9], [10], [13].

### B. Limitation of Current Methods: Multiple Patch Scenarios

The origin of multiple patches includes but is not limited to the following situations. The first situation is that a vulnerability involves different components and each of them is addressed by one patch respectively (for example, the description of CVE-2014-5273 [32] specifies five vulnerable locations and thus corresponds to five patches). The second situation is that the patch introduces an additional bug or vulnerability, thus one or more supplementary patches are needed to perform further fixes (for example, CVE-2019-12108 [33] corresponds to two patches, in which the latter *Commit 86030db* fixes the error induced by the former *Commit 13585f1*). The third situation is that a vulnerability has complex code logic or trigger paths, thus the initial patch could not be enough to fully fix it (for example, CVE-2020-25781 [34] corresponds to two patches, the latter *Commit 9de20c0* calls the functions modified by the former *Commit 5595c90*). The fourth situation is that a better vulnerability fixing scheme is found than the initial patch (for example, CVE-2015-2080 [35] corresponds to two patches according to the reference [3], in which the latter simplifies the original behavior). In addition to the aforementioned cases, there are many other scenarios that can cause multiple patches. These multi-patch scenarios account for a considerable portion of vulnerability management in open-source communities [1], [15], [18]–[22] and therefore need to be addressed urgently. However, existing SVPI methods may have low effectiveness in multi-patch scenarios because they typically only consider the similarity between vulnerabilities and code commits, and their output is typically a ranking list in which interrelationship between patches could be broken up.

### C. Motivating Example

CVE-2021-25932 [36] records a vulnerability caused by improper validation checks on user input, allowing attackers to inject an arbitrary script into the database. This vulnerability requires three patches to be fixed collaboratively, namely Patch① (*8a97e68* [37]), Patch② (*eb08b5e* [38]), and Patch③ (*f3ebfa3* [39]), and each of the three targets an aspect of fixing it (i.e., the first situation in Section II-B). To check existing SVPI techniques' effectiveness in dealing with this vulnerability, we extract three patches (positive samples) along with additional 1,500 code commits that are not relevant to the fixing of this vulnerability (negative samples) from the repository where the vulnerability resides. We select four SVPI techniques, namely PatchScout [13], PatchFinder [10], PromVPat [14], and VCMatch [9], because they have been demonstrated to be highly-promising and their solutions are from diverse perspectives (e.g., rule-based, deep learning-based, and LLM-based) thus can be representatives. The result is given in Table I. For example, PatchScout ranks Patch①, Patch②, and Patch③ at the positions $36^{\mathrm{th}}$, $371^{\mathrm{st}}$, and $425^{\mathrm{th}}$, respectively, meaning that downstream software maintainers need to inspect at least 425 code commits to find all patches using the ranking list-form output (i.e., Total Effort). The Total Effort values of PatchFinder, PromVPat, and VCMatch are 100, 13, and 16, respectively, which are not effective enough. This example double-confirms the two limitations we mentioned earlier, i.e., the interrelationship between code commits is not considered, and the output is in the form of a ranking list. For an effective SVPI technique, it is expected to consider both the relationship between vulnerabilities and code commits and the interrelationship between code commits, and directly recommend a patch group that contains all the identified patches for the vulnerability.

## III. APPROACH

In this section, we propose SHIP, a silent vulnerability patch identification approach suited for multi-patch scenarios, to alleviate the two main challenges. We depict its workflow in Figure 1 and summarize the three phases as follows:

- **Phase-1 - Initial Ranking.** For a given vulnerability, we use various basic features (involving lines of codes, identity, location, and token) as well as semantic features to extract its relevance to candidate code commits. A small part of highly-relevant code commits will be delivered to the next phase to be further checked.
- **Phase-2 - Commits Linkage Prediction.** Predicting the relevance score of any pair of highly-relevant code commits to quantitatively reflect whether these two are responsible for the same vulnerability. LLM-generated features, as well as rule-based and semantic features, participate in the prediction process.
- **Phase-3 - Commit Group Forming and Ranking.** Labeling the pairs of code commits whose relevance score exceeds the preset threshold as *relevant*, and constructing maximum connected subgraphs from all *relevant* pairs. For each subgraph (i.e., commit group), performing max-pooling to the numeric vectors of the code commits it contains to produce a proxy vector. Measuring the relevance between all proxy vectors and the vulnerability, the commit group with the highest-relevant proxy vector will be determined as the *patch group* and output.

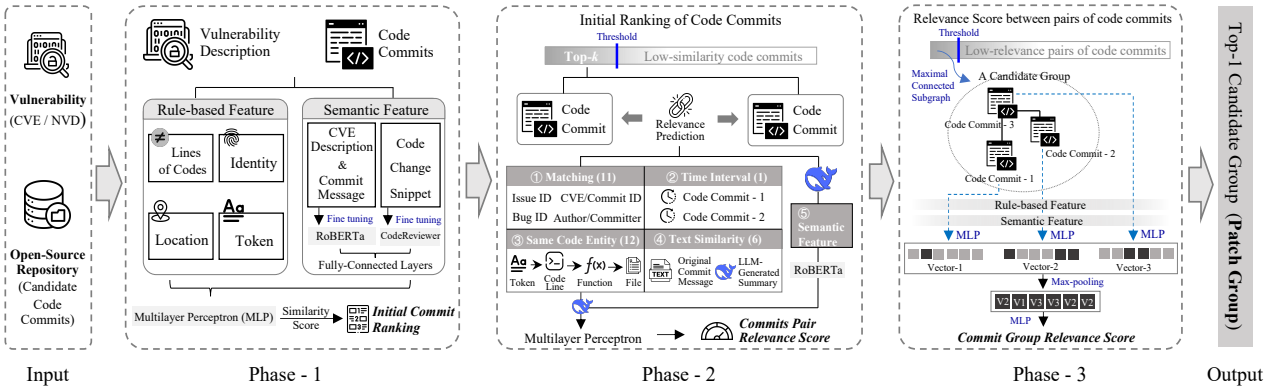


Fig. 1. The overview of SHIP

### A. Initial Ranking

For a vulnerability that needs to match security patches, all code commits in its repository are candidates and thus should be considered. According to our preliminary investigation and previous research [6], [28], [40]–[42], a code repository often has a large number of code commits, most of which are daily updates or performance improvements, and security patches are only a small part of them. To provide the whole identification process for a more precise context and save computation costs, we refine the search scope by initially ranking candidate code commits and selecting highly-relevant ones, those code commits that are regarded as not highly-relevant will be excluded from the following procedures.

Many pioneering SVPI techniques have been proposed and demonstrated to be effective in pinpointing the similarity between candidate code commits and a vulnerability, most of which assume that a vulnerability corresponds to a single patch [9], [14], [43]. The initial ranking phase of SHIP aims to filter out some candidate code commits to facilitate further identification, the mission of this phase is just to narrow down the search space and provide a refined scope for downstream multi-patch identification. Therefore, we follow the strategies of PatchScout [13] and VCMatch [9], two well-known existing SVPI techniques, to complete this phase. These two techniques define a series of handcrafted features that can reflect the similarity between a given vulnerability and a code commit, involving the aspects of lines of codes, identity, location, and token. According to our investigation on the design of these features, we make some adjustments when implementing them in SHIP. Specifically, we first merge the feature "URL Num" into the features "Bug Num" and "Issue Num" respectively, by extra taking the numbers of Bug IDs and Issue IDs mentioned in URLs into account. Secondly, we add two new features, "CWE Match" and "Issue Match", to observe whether the code commit refers to the CWE/Issue ID on the NVD page of the vulnerability. Besides, we remove the "Patch Likelihood" feature because we will specifically evaluate the probability of a code commit to be a patch for the given vulnerability in Phase-2. The adjusted features are given in our repository [44]. As such, we get a 38-dimensional vector for each candidate code commit.

In addition to the rule-based features mentioned above, we also extract semantic features between vulnerabilities and code commits from the aspects of textual description and code. Specifically, we input the description of the vulnerability and commit messages of code commits to RoBERTa [45], [46], a well-known pretrained model that has been applied to many natural language processing tasks and has been shown to achieve promising results compared to other popular models [47], [48], and perform fine-tuning. Besides, we also input the code change part of code commits to CodeReviewer [49], [50], a pretrained model that is tailored specifically for understanding the difference before and after code editing, and perform fine-tuning. As such, text information from both the vulnerability side and the code commit side, as well as code snippets before and after modification from the code commit side, will be embedded into 1,024 and 786-dimensional vectors respectively based on the characteristics of the used models. We further feed the two vectors into fully-connected layers to make them have a unique length (i.e., 32 dimensions), and splice them into a 64-dimensional vector.

Finally, the rule-based feature (38-dimensional vector) and semantic feature (64-dimensional vector) will be merged and used to train a Multilayer Perceptron (MLP) [51], in which the merged vectors of candidate code commits and their labels (i.e., whether a code commit is the patch for the vulnerability or not) serve as training data. The trained MLP can output the similarity scores of unseen code commits to the given vulnerability, and thus an initial ranking list of code commits can be produced, as shown in the "Phase-1" part of Figure 1.

### B. Commits Linkage Prediction

In this phase, SHIP intends to extract the interrelationship between code commits, to find potential multiple patches for the vulnerability, as shown in the "Phase-2" part of Figure 1. The intuition of this strategy is that if more than one patch fixes a vulnerability collaboratively, there could be only one or a few of them that can be linked to the vulnerability explicitly, and

TABLE II
FEATURES DESIGNED IN SHIP TO PREDICTING THE RELEVANCE OF TWO CODE COMMITS

| Category | Feature | Meaning |
|---|---|---|
| Matching | Issue Num 1 | The number of Issue-IDs in the message of code commit 1. |
| | Issue Num 2 | The number of Issue-IDs in the message of code commit 2. |
| | Issue Match | Whether the message of a code commit mentions the Issue-ID in the other code commit. |
| | Bug Num 1 | The number of Bug-IDs in the message of code commit 1. |
| | Bug Num 2 | The number of Bug-IDs in the message of code commit 2. |
| | Bug Match | Whether the message of a code commit mentions the Bug-ID in the other code commit. |
| | CVE Num 1 | The number of CVE-IDs in the message of code commit 1. |
| | CVE Num 2 | The number of CVE-IDs in the message of code commit 2. |
| | CVE Match | Whether the message of a code commit mentions the CVE-ID in the other code commit. |
| | Commit-ID Match | Whether the message of a code commit mentions the Commit-ID of the other code commit. |
| | Committer/Author Match | Whether the two code commits have the same committer/author. |
| Time Interval | Time Interval | Time interval between the two code commits. |
| Same Code Entity | Same File Num | The number of files modified in both code commit 1 and code commit 2. |
| | Same File Ratio | Same File Num / The least number of modified files of the two code commits. |
| | Same Function Num | The number of functions modified in both code commit 1 and code commit 2. |
| | Same Function Ratio | Same Function Num / The least number of modified functions in the two code commits. |
| | Same Function-related Num | The number of functions called and defined in both code commit 1 and code commit 2. |
| | Same Function-related Ratio | Same Function-related Num / The least number of called and defined functions of the two code commits. |
| | Same Modified Line (Reverse) Num | The number of the modified code lines with the same content in two code commits (a commit adds and the other deletes) |
| | Same Modified Line (Reverse) Ratio | Same Modified Line (Reverse) Num / The least number of modified code lines of the two code commits |
| | Same Modified Line (Equivalent) Num | The number of the modified code lines with the same content in two code commits (both commits add or both delete) |
| | Same Modified Line (Equivalent) Ratio | Same Modified Line (Equivalent) Num / The least number of modified code lines of the two code commits |
| | Same Code Token Num | The number of the same code tokens between code commit 1 and code commit 2. |
| | Same Code Token Ratio | Same Code Token Num/ The least number of code tokens in the two code commits. |
| Text Similarity | Msg TF-IDF Similarity | Cosine similarity between the TF-IDF vectors of the messages of code commit 1 and code commit 2. |
| | Same Msg Token Num | The number of the same tokens between the messages of code commit 1 and code commit 2. |
| | Same Msg Token Ratio | Same Msg Token Num / The least number of tokens in the two commit messages. |
| | LLMText TF-IDF Similarity | Cosine similarity between the TF-IDF vectors of the LLM-generated summaries of code commit 1 and code commit 2. |
| | Same LLMText Token Num | The number of the same tokens between the LLM-generated summaries of code commit 1 and code commit 2. |
| | Same LLMText Token Ratio | Same LLMText Token Num / The least number of tokens in the LLM-generated summaries of code commit 1 and code commit 2. |

other patches need to first establish a link to the patch(es) that has an explicit relationship with the vulnerability before they can establish a link to the vulnerability. Therefore, the linkage between code commits is worth analyzing because it can show whether these two work together to fix the vulnerability, and thus can serve as the clue of finding missing patches which could be ignored by existing SVPI techniques.

We first take the Top-$k$ candidate code commits from the initial ranking list of code commits (the output of Phase-1) to save cost and to facilitate the effectiveness and efficiency of the training process. Then we pair each of the $k$ code commits and design rule-based features and semantic features to predict the relevance score of the $C_2^k$ code commit pairs. We introduce the details of the two feature classes and the process of fusing them to train the model as follows.

***Rule-based Features***. We design 30 features in total involving the aspects of *Matching* (11), *Time Interval* (1), *Same Code Entity* (12), and *Text Similarity* (6). The details of these features are elaborated in Table II. In particular, the 11 features in the *Matching* category evaluate the extent to which the issue, bug, and CVE mentioned in the two code commits are relevant (note that mentioning a CVE/Bug/Issue-ID in a commit cannot guarantee a necessary linkage to that CVE/Bug/Issue, since there are diverse reasons for commits to contain them), whether a code commit mentions the other code commit, and whether the two code commits are authored or committed by the same account. The one feature in the *Time Interval* category evaluates how long it takes for two code commits to be submitted. The intuition behind this feature is that previous research has shown that the interval between the time a vulnerability is disclosed by CVE and the time a vulnerability-fixing commit is created in the code repository is an informative indicator in SVPI [13], thus we naturally conjecture that the time interval between two code commits can also be helpful in predicting the relevance between them. The 12 features in the *Same Code Entity* category compare the code change part of the two code commits at the granularity of files, functions, lines, and tokens. The six features in the *Text Similarity* category measure the cosine similarity between the TF-IDF vectors [52], [53] of the two code commits' messages, and count the number and ratio of the same tokens between the two messages. To enrich code commits' information beyond off-the-shelf sources (i.e., commit messages and code changes), we employ a large language model, DeepSeek-V3, to further extract code commits' characteristics from code changes. Specifically, we send the code change part of each of the two code commits to DeepSeek-V3 along with the designed prompt (the prompt used in this paper is released in our repository [54]), ask the model to analyze the code and thus produce a summary of what the code change does. The LLM-generated summaries will be transformed into TF-IDF vectors and then be analyzed in the same way as original commits' message.

***Semantic Features.*** Apart from the rule-based features, SHIP extracts semantic features of a pair of code commits as well to analyze the relevance between them. A pair of code commits' original commit messages and LLM-generated summaries will be spliced, starting with [CLS] and separated by [SEP]. Then, it will be fed to a pretrained model RoBERTa and a fully-connected layer in order, as shown at the bottom of Figure 2. As such, the text information from four different sources can be integrated into a vector with the length of 32.

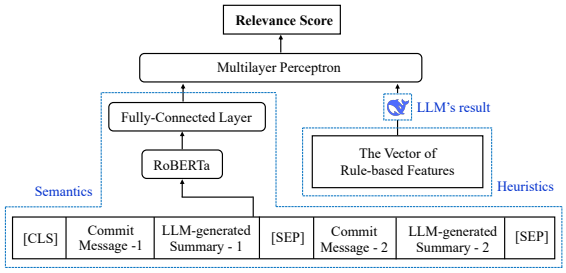


Fig. 2. Prediction of the relevance score between two code commits

***Feature Fusion.*** The rule-based features and the semantic features aim to reflect the relevance between a pair of code commits from the perspectives of heuristics and deep semantics respectively. As a reminder, the possibility of a code commit to be the patch of the given vulnerability can also contribute to the interrelationship between code commits, because if two code commits are predicted to be patches for the same vulnerability, they should obviously have high relevance. Hence, we input the vulnerability's CVE-ID and description as well as a code commit's message and code change to DeepSeek-V3, ask it to predict whether this code commit is the patch for that vulnerability and output *yes*, *no*, or *unknown*. The LLM's result, the vector of the rule-based features originated from Table II, and the vector of the semantics features produced by the fully-connected layer, will be together fed to an MLP comprising two fully-connected layers and an activation function ReLU [55] in between. The structure of the model is shown in Figure 2, whose production is the relevance score ($0 \sim 1$) of a code commit pair.

The reason for using the large language model in this phase lies in the inherent gap between CVE descriptions and code commits, which typically involve different dimensions. LLMs have an understanding of both natural languages and code changes, thus can grasp the semantic essence of texts and code snippets at a deeper level.

### C. Commit Group Forming and Ranking

***Commit Group Forming***. Having the relevance scores of all pairs of code commits, we determine whether two code commits should be connected or not and thus form code commit groups. The input of this phase is the Top-$k$ code commits $C = \{c_1, c_2, ..., c_k\}$ obtained by filtering in Phase-1 as well as the relevance scores of pairs of code commits $R \in \mathbb{R}^{k \times k}$, where $R_{ij}$ $(i, j = 1, 2, ..., k)$ is the relevance score of the code commits $c_i$ and $c_j$, it is obvious that $R_{ij} \in [0, 1]$ and $R_{ij} = R_{ji}$ according to the description of Phase-2. Based on $C$ and $R$, we first construct an undirected graph $G_\theta = (C, E_\theta)$, where $E_\theta = \{(c_i, c_j) | R_{ij} \geq \theta, i \neq j\}$ and $\theta$ is a parameter used to adjust the acceptance of two code commits being able to be connected. The higher the value of $\theta$, the more difficult it is for two code commits to be connected, the smaller the size of the commit group formed, and vice versa. As such, $G_\theta$ is an undirected graph containing one or more maximal connected subgraphs $MCS = \{MCS_1, MCS_2, ..., MCS_p\}$, in which each $MCS$ contains one or more code commits based on the value of $\theta$. It can be found that $MCS_i \cap MCS_j = \varnothing$ $(i, j = 1, 2, ..., p$ and $i \neq j)$ because a code commit cannot appear in two different $MCS$ simultaneously, and $MCS_1 \cup MCS_2 \cup ... \cup MCS_p = C$.

TABLE III
FURTHER FEATURES TO EMBEDDING CODE COMMITS IN AN $MCS$

| Feature | Meaning |
|---|---|
| LLMText TF-IDF Similarity | Cosine similarity between the TF-IDF vectors of the vulnerability description and LLM-generated summary for code change |
| Same LLMText-VulDesc Token Number | The number of the same tokens between the vulnerability description and LLM-generated summary for code change |
| Same LLMText-VulDesc Token Ratio | Same LLMText-VulDesc Token Number / The number of tokens in the vulnerability description |
| Max/Sum/Mean/Var | The Maximal/Sum/Mean/Variance of the frequencies for all Same LLMText-VulDesc Tokens |

***Commit Group Ranking***. To predict the relevance between an $MCS$ and the vulnerability, we need to build an integral representation for each $MCS$ instead of treating it as separate code commits. We design a twofold strategy to achieve this goal. First, we convert each code commit in an $MCS$ to a numerical vector reflecting its relevance to the given vulnerability, which is based on the extraction of rule-based and semantic features. For rule-based features, we continue using the features in Phase-1 because the mission here is actually the same as that of Phase-1. But in this phase, candidate code commits have been greatly filtered out, and thus a more sophisticated solution is affordable. Specifically, considering that vulnerability items identify themselves in the form of textual description information, we employ DeepSeek-V3 to translate the code change part of code commits into a summary (the same form as vulnerability descriptions), and resultingly define the features in Table III to replace all the features involving code changes in Phase-1. Based on these rule-based features, we can get a numerical vector for each code commit in an $MCS$. For semantic features, we send three sources of information, the vulnerability description, the code commit's message, and the LLM-generated summary for code changes, to a RoBERTa model and a fully-connected layer successively. As such, we can obtain another numerical vector for each code commit in an $MCS$. The above two vectors reflecting rule-based features and semantic features are then input to an MLP to be embedded into a fixed-length vector. Second, we merge all fixed-length vectors of code commits in an $MCS$ into a single one to represent this $MCS$. We utilize the max-pooling operation to perform the merging process, as shown in the "Phase-3" part of Figure 1. The intuition behind this strategy is that within a formed $MCS$, it is possible that only a few patches exhibit explicit connections with the target vulnerability. While other code commits in the same group are also patches, it could be difficult for them to be linked to the vulnerability directly, but they can be linked to the patch that has explicit connections with the vulnerability (for example, Patch-A can be directly linked and Patch-B is the subsequent patch or repair of Patch-A). That is to say, for an $MCS$, taking

the average of the relevance scores of its member patches is inappropriate, because some patches with weak linkages to the vulnerability do not necessarily descend the overall relevance score of this $MCS$. Consequently, the part of each fixed-length vector showing the highest relevance to the vulnerability needs to be selected as the representative of the $MCS$. The max-pooling strategy actually aims to identify such a representative.

### D. Running Example

We give a running example (CVE-2023-50723 [56]) to exemplify the three phases of SHIP. The complete ranking lists and relevance scores of the three phases of this running example can be found in our repository [57].

**Phase-1 - Initial Ranking.** SHIP obtains candidate code commits in the repository where the vulnerability resided (XWiki [58]), extracts rule-based and semantic features for candidate code commits, and then uses the trained model to predict their relevance to the vulnerability. In the output ranking list, *0f367aa*, *1157c1e*, *749f6ae*, and *bd82be9* (i.e., the four patches of this vulnerability) are ranked at the $5^{\text{th}}$, $8^{\text{th}}$, $22^{\text{nd}}$, and $1^{\text{st}}$ positions, respectively.

**Phase-2 - Commits Linkage Prediction.** Top-$k$ candidate code commits in the above ranking list (the value of $k$ is taken as 50 here as an example) will be input to this phase to predict their interrelationship. Among the $C_2^{50}$ pairs of code commits, the relevance scores of "*0f367aa-1157c1e*", "*0f367aa-749f6ae*", "*0f367aa-bd82be9*", "*1157c1e-749f6ae*", "*1157c1e-bd82be9*", and "*749f6ae-bd82be9*" are 1.00, 1.00, 0.98, 1.00, 1.00, and 1.00, respectively.

**Phase-3 - Commit Group Forming and Ranking.** The parameter $\theta$ that determines whether two code commits can be connected is taken as 0.9 as an example (the impact of $\theta$ on SHIP's effectiveness will be investigated in Section V-A). As such, code commits *0f367aa*, *1157c1e*, *749f6ae*, and *bd82be9* will form an $MCS$ because no other code commit has a relevance score greater than or equal to 0.9 with any of them. SHIP then produces proxy vectors for all $MCS$s and gets their relevance scores with the vulnerability. The $MCS$ containing and only containing the aforementioned four code commits (i.e., four real patches) is ranked at the first position, which will be determined as the patch group and output.

## IV. Experimental Setup

### A. Research Questions (RQ)

- **RQ1: The impact of the threshold of code commits' relevance score.** When forming code commit groups, a threshold determines whether two commits can be connected. How does this parameter impact SHIP?
- **RQ2: The comparison between SHIP and SOTA techniques.** We compare the effectiveness of SHIP with PatchScout, PatchFinder, PromVPat and VCMatch, in both single-patch and multi-patch scenarios.
- **RQ3: The contribution of LLM and non-LLM features.** The use of DeepSeek-V3 contributes to the prediction of code commits' relevance and the ranking of code commit groups. We analyze SHIP's effectiveness when only adopting LLM features and excluding LLM features.

### B. Datasets

We measure SHIP's capability of identifying silent vulnerability patches under both multi-patch and single-patch scenarios. For multi-patch vulnerabilities, we manually analyze vulnerability items in mainstream platforms such as NVD [17] and Snyk [59], extracting those that have more than one patch[2]. Besides, we also analyze previous works that release well-structured datasets in the field of SVPI (e.g., the references [3], [7], [10], [15], [27], [60]) to check if they contained vulnerability items with multi-patch. In total, we get 1,560 vulnerability items with $2 \sim 23$ patches. And we also randomly select twice as many single-patch vulnerability items as multi-patch ones (i.e., 3,120) from the datasets of the aforementioned works, because single-patch vulnerabilities occupy a non-trivial part in the real world. This choice strikes a balance between simulating the real distribution of multi-patch and single-patch vulnerabilities as much as possible and ensuring the training effectiveness of SHIP.

After removing some items for which patch details (e.g., code changes) can not be accessed, we get a benchmark containing 1,560 multi-patch and 3,071 single-patch vulnerability items. Apart from patches (positive samples), we randomly select 1,500 code commits that are not relevant to the vulnerability (negative samples) for each of the 4,631 items from the repository where the vulnerability is located (if available negative samples are less than 1,500, we take all of them).

### C. Metrics

As we mentioned above, silent vulnerability patch identification tasks in multi-patch scenarios expect a group-form output, that is, all patches related to a vulnerability are packaged together and thus downstream maintainers can receive an out-of-the-box result (group-form outputs are also suited for single-patch scenarios because a patch group can contain only one patch as well). SHIP meets this expectation and thus needs to be evaluated in a slightly different way from existing SVPI methods. Specifically, traditional methods typically deliver a ranking list-form output, that is, a ranking list of code commits sorted from high to low in terms of their relevance to the vulnerability, therefore, Recall [61] can be directly obtained by measuring in how many vulnerability items the patch can enter the Top-$r$ positions in the ranking. To make the evaluation of Recall available for SHIP, we give the definition of Recall as follows. Let $V = \{v_1, v_2, ..., v_N\}$ be all vulnerability items, for each item $v_i$ $(i = 1, 2, ..., N)$, $P_i \subseteq U_i$ is the set of real patches of $v_i$, ($U_i$ is candidate code commits), and $\hat{O}_i \subseteq U_i$ is the patch group output by SHIP. We use Formula 1 and Formula 2 to define Recall:

[2]Document-based links (such as vendor advisories) are sometimes labeled as "Patch" as well, these are duplicates of real patches and should therefore be excluded to ensure an accurate evaluation.

$$\text{Recall} = \frac{1}{N}\sum_{i=1}^{N}\delta_i, \tag{1}$$

$$\delta_i = \begin{cases} 1, & \text{if } P_i \subseteq \hat{O}_i \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Besides, we use Precision [62] as a complement to evaluate SHIP, as shown in Formula 3 and Formula 4:

$$\text{Precision} = \frac{1}{N}\sum_{i=1}^{N}\gamma_i, \tag{3}$$

$$\gamma_i = \begin{cases} 1, & \text{if } \hat{O}_i \subseteq P_i \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

We also use F1-score [63] to integrate Recall and Precision, as shown in Formula 5:

$$\text{F1-score} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5}$$

Notice that existing SVPI techniques, which typically deliver a ranking list of code commits as the outcome and thus use Recall@$r$ or Precision@$r$ as metrics, are not designed for the multi-patch scenario. To make these methods applicable to the multi-patch scenario, a step that determines how many top patches on the list should be linked to the CVE must be implicitly included. In our experiment, we apply the scale of the SHIP-recommended patch group (i.e., $r$) to these traditional methods to perform evaluation. We believe this is one of the fairest solutions for the experiment. As a reminder, it is not reasonable to directly use the ground-truth number on either the baselines or our method, since it can never be known in advance.

### D. Parameters and Environments

We randomly split our dataset in the proportion of 8 : 1 : 1 (the training set : the validation set : the test set) to follow the commonly-adopted strategy and keep the same settings as the four baseline techniques. In the training process, the learning rate is set to $1e$-4, the max iteration and the batch size are 20 and 48 respectively, and the loss function is CrossEntropyLoss. The training and test processes are performed on a server equipped with 36 Intel(R) Xeon(R) Gold 6254 CPUs with 3.10GHz and 100GB of memory as well as 3 NVIDIA RTX A6000 GPUs.

## V. Result and Analysis

### A. RQ1:The impact of the threshold of code commits' relevance score

To quantitatively measure the impact of the threshold $\theta$ that determines whether two code commits can be connected on SHIP, we set $\theta$ from 0.1 to 1.0 with increments of 0.1, and observe the change on the validation set. Specifically, we get 501 "patch-patch" pairs and 33,108 "patch-non patch" pairs of the vulnerability items in the validation set. We find that for "patch-non patch" pairs, setting the value of $\theta$ to 1.0 can obtain the best result, because this value can make most of the "patch-non patch" pairs correctly identified (with only 9 exceptions). And the value 0.9 is the second best (with 444 exceptions). We also find that for 86.63% of the "patch-patch" pairs, their predicted relevance scores are less than 1.0, so setting the value of $\theta$ to 1.0 is not a proper choice. Considering these two aspects comprehensively, $\theta$ = 0.9 can be a balanced choice for SHIP, which can facilitate the identification of patches related to the same vulnerability while excluding non-relevant ones. To more comprehensively illustrate this trend, we also give the result on the test set in Table IV and Figure 3.

TABLE IV
Recall and Precision values under different $\theta$

| $\theta$ | Recall | | | Precision | | |
|---|---|---|---|---|---|---|
| | All | Single | Multi | All | Single | Multi |
| **0.1** | 86.24% | 94.48% | 73.25% | 27.31% | 30.52% | 21.02% |
| **0.2** | 86.45% | 93.51% | 72.61% | 34.19% | 37.66% | 27.39% |
| **0.3** | 86.24% | 93.18% | 72.61% | 39.35% | 43.83% | 29.94% |
| **0.4** | 85.81% | 92.86% | 71.97% | 43.66% | 48.38% | 34.39% |
| **0.5** | 86.02% | 93.18% | 71.97% | 47.53% | 52.60% | 37.58% |
| **0.6** | 85.81% | 92.86% | 71.97% | 50.32% | 56.17% | 38.85% |
| **0.7** | 85.38% | 92.53% | 71.34% | 51.83% | 57.79% | 40.13% |
| **0.8** | 84.73% | 92.53% | 69.43% | 55.27% | 61.69% | 42.68% |
| **0.9** | 84.30% | 92.86% | 67.52% | 59.14% | 65.58% | 46.50% |
| **1.0** | 62.37% | 90.58% | 7.01% | 83.87% | 89.29% | 73.25% |

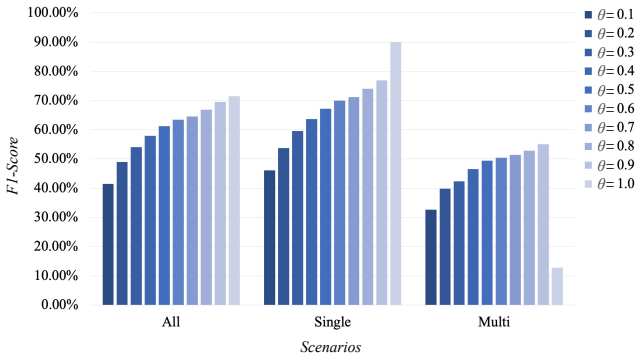


Fig. 3. F1-Score values under different $\theta$

Table IV provides the values of Recall and Precision of SHIP under different values of $\theta$ (0.1, 0.2, ..., 1.0) from three perspectives: "All" means that all vulnerability items are considered, while "Single" and "Multi" mean that only single-patch or multi-patch vulnerability items are considered, respectively (these three abbreviations also apply to the following tables and figures). It can be seen that when $\theta$ is set to 0.1 $\sim$ 0.9, the values of Recall are basically stable (84.30% $\sim$ 86.45% for "All", 92.53% $\sim$ 94.48% for "Single", and 67.52% $\sim$ 73.25% for "Multi"). But when $\theta$ is set to 1, the value of Recall drops to 62.37% for "All" and significantly drops to 7.01% for "Multi", and is basically unchanged (i,e., 90.58%) for "Single". The reason for this phenomenon can be found in the functionality of the threshold $\theta$: the higher the value of $\theta$, the smaller the generated patch group, and thus the lower the probability that the group covers all real

TABLE V
RECALL AND PRECISION VALUES OF SHIP AND BASELINES

| Techniques | Recall | | | Precision | | |
|---|---|---|---|---|---|---|
| | All | Single | Multi | All | Single | Multi |
| **PatchScout** | 26.88% | 36.69% | 7.64% | 19.52% | 28.52% | 4.46% |
| **PatchFinder** | 38.06% | 46.75% | 21.19% | 24.52% | 29.22% | 15.29% |
| **PromVPat** | 61.29% | 71.43% | 41.40% | 45.95% | 57.79% | 26.11% |
| **VCMatch** | 51.61% | 59.09% | 36.94% | 35.95% | 44.87% | 21.02% |
| **SHIP** | **84.30%** | **92.86%** | **67.52%** | **59.14%** | **65.58%** | **46.50%** |

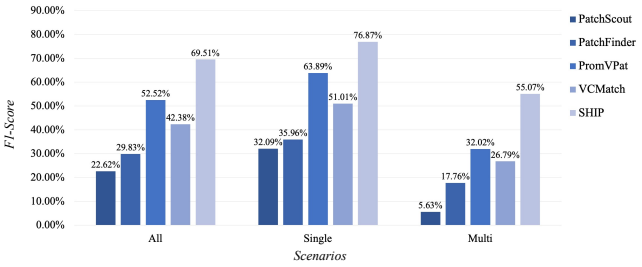


Fig. 4. F1-Score values of SHIP and baselines

patches when a vulnerability is related to more than one patch. The impact of a smaller patch group on single-patch vulnerability items is limited, because the ground truth of this scenario is just one patch. Let's move on to the analyses for Precision. As the value of $\theta$ increases, the values of Precision increase monotonically, whether in column "All", "Single", or "Multi". This result can be easily understood because Precision requires the predicted results to be as correct as possible rather than expecting to find all positive samples. Exactly for this reason, when $\theta$ is set to 1, the values of Precision have a significant improvement: from 59.14% to 83.87%, from 65.58% to 89.29%, and from 46.50% to 73.25%, for "All", "Single", and "Multi", respectively, because the highest value of $\theta$ causes a patch to be connected only to the patch with the highest relevance with it thus delivers the smallest group.

Figure 3 gives the trend of F1-Score values. With the increase of $\theta$, the values of F1-Score in columns "All" and "Single" increase monotonically, from 41.48% to 71.54% and from 46.14% to 89.93%, respectively. While in column "Multi", the value of F1-Score increases monotonically when $\theta$ is set from 0.1 to 0.9 (from 32.67% to 55.07%), but it drops to 12.80% when $\theta$ is set to 1.0, this can be attributed to the drop in Recall (the cell "Recall-Multi-1.0" in Table IV is 7.01%).

### B. RQ2: The comparison between SHIP and SOTA techniques

To evaluate the competitiveness of SHIP, we choose PatchScout [13], PatchFinder [10], PromVPat [14], and VCMatch [9] as baseline techniques, because these four have been demonstrated to be highly-effective and their designs cover mainstream solutions, e.g., rule-based, deep learning-based, and LLM-based tactics. The result is given in Table V and Figure 4. It can be seen that SHIP outperforms the four baseline techniques significantly, regardless of the perspective of Recall, Precision, or F1-Score.

Specifically, in the scenarios of "All", "Single", and "Multi", the values of Recall of SHIP are 84.30%, 92.86%, and 67.52% respectively, which are 37.54% , 30.00%, and 63.09% higher than those of PromVPat respectively (We calculate the proportion of growth by division instead of delta). A similar trend can be observed in terms of Precision: for "All", "Single", and "Multi", the Precision values of SHIP are 59.14%, 65.58%, and 46.50% respectively, which are 28.71%, 13.48%, and 78.09% higher than those of PromVPat respectively. As for F1-Score, SHIP still achieves superior results: for "All", "Single", and "Multi", the F1-Score values of SHIP are 69.51%, 76.87%, and 55.07% respectively, which are 32.35%, 20.32%, and 71.99% higher than those of PromVPat respectively. The detailed results of the other three baselines can be found in Table V and Figure 4.

TABLE VI
EFFECTIVENESS OF SHIP AND BASELINES IN MULTI-PATCH SCENARIOS

| | | 2 Patches | 3 Patches | 4 Patches | 5 Patches | 5+ Patches |
|---|---|---|---|---|---|---|
| **Recall** | PatchScout | 7.87% | 6.45% | 6.67% | 14.29% | 6.67% |
| | PatchFinder | 28.09% | 9.68% | 20.00% | 14.29% | 6.67% |
| | PromVPat | 42.70% | 38.71% | **66.67%** | 14.29% | 26.67% |
| | VCMatch | 40.45% | 32.26% | 46.67% | 14.29% | 26.67% |
| | SHIP | **67.42%** | **74.19%** | **66.67%** | **57.14%** | **60.00%** |
| **Precision** | PatchScout | 2.25% | 6.45% | 6.67% | 14.29% | 6.67% |
| | PatchFinder | 17.98% | 12.90% | 20.00% | 14.29% | 0.00% |
| | PromVPat | 26.97% | 16.13% | **60.00%** | 14.29% | 13.33% |
| | VCMatch | 19.10% | 22.58% | 46.67% | 14.29% | 6.67% |
| | SHIP | **39.33%** | **64.52%** | 53.33% | **42.86%** | **46.67%** |
| **F1-Score** | PatchScout | 3.50% | 6.45% | 6.67% | 14.29% | 6.67% |
| | PatchFinder | 21.93% | 11.06% | 20.00% | 14.29% | 0.00% |
| | PromVPat | 33.06% | 22.77% | **63.16%** | 14.29% | 17.78% |
| | VCMatch | 25.95% | 26.57% | 46.67% | 14.29% | 10.67% |
| | SHIP | **49.68%** | **69.02%** | 59.26% | **48.98%** | **52.50%** |

SHIP is designed as an SVPI approach that can well handle multi-patch vulnerabilities, thus we split scenario "Multi" into five sub-scenarios, i.e., "2 Patches", "3 Patches", "4 Patches", "5 Patches", and "5+ Patches" (vulnerabilities that are related to two, three, four, five, and more than five patches, respectively), to further observe SHIP's effectiveness in a finer-grained perspective. The result is given in Table VI: SHIP still outperforms all baselines regardless of how many patches a vulnerability corresponds to (with only two exceptions where PromVPat's Precision value in "4 Patches": 60.00% is higher than that of SHIP: 53.33%, and PromVPat's F1-Score value in "4 Patches": 63.16% is higher than that of SHIP: 59.26%).

The effectiveness of the four baseline techniques in our experiment is lower than that in their original experiments, we conjecture that this is because they are mainly designed to deal with single-patch vulnerabilities, and therefore do not pay more attention to analyzing the intrinsic relationships between code commits. The experimental data supports our speculation, as their decreased effectiveness is mainly dragged down by their performance in the multi-patch scenario.

### C. RQ3: The contribution of LLM and non-LLM features

SHIP employs a large language model, DeepSeek-V3, to enhance the processes of predicting code commits' relevance (in Phase-2) and ranking code commits groups (in Phase-3). We further explore how different features (LLM features and non-LLM features) of SHIP contribute to its effectiveness. For this goal, we first ablate all the features contributed by LLM: in Phase-2, the LLM-related features in the rule-based features,

TABLE VII
COMPARISON BETWEEN SHIP AND ITS TWO VARIANTS

| | | All | Single | Multi |
|---|---|---|---|---|
| **Recall** | SHIP$_{non\text{-}LLM}$ | 68.60% | 82.20% | 43.59% |
| | SHIP$_{LLM}$ | 66.45% | 79.87% | 40.13% |
| | SHIP | **84.30%** | **92.86%** | **67.52%** |
| **Precision** | SHIP$_{non\text{-}LLM}$ | 51.83% | 59.80% | 37.66% |
| | SHIP$_{LLM}$ | **65.59%** | **73.38%** | **50.32%** |
| | SHIP | 59.14% | 65.58% | 46.50% |
| **F1-Score** | SHIP$_{non\text{-}LLM}$ | 59.05% | 69.23% | 40.41% |
| | SHIP$_{LLM}$ | 66.02% | 76.49% | 44.65% |
| | SHIP | **69.51%** | **76.87%** | **55.07%** |

the LLM-generated summary in the semantic features, and the LLM's output will be ablated. And in Phase-3, the LLM-related features in the rule-based features and the LLM-generated summary in the semantic features will be ablated. As such, we can form a variant of SHIP, **SHIP**$_{non\text{-}LLM}$. In contrast, we also retain all the aforementioned features contributed by LLM and ablate the other features that have nothing to do with LLM in Phase-2 and Phase-3, to form another variant, **SHIP**$_{LLM}$. The result of these two variants versus the original SHIP is given in Table VII.

Let us first focus on the comparison between SHIP and SHIP$_{non\text{-}LLM}$. It can be seen that the latter is worse than the former regardless of scenarios ("All", "Single", and "Multi") and metrics (Recall, Precision, and F1-Score), this result emphasizes the importance of LLM features in SVPI tasks because ablating them causes a significant decrease in effectiveness. Notice that such LLM features' importance is even more prominent in multi-patch vulnerabilities, because the values of the three metrics of SHIP and SHIP$_{non\text{-}LLM}$ differ more in column "Multi" than those in columns "All" and "Single" in Table VII. Then we focus on the comparison between SHIP and SHIP$_{LLM}$. There is an identical trend that SHIP dominates SHIP$_{LLM}$ in terms of Recall and F1-Score, indicating that non-LLM features (such as heuristic features directly extracted from the description of vulnerabilities as well as the commit message and code changes of code commits) also play an essential role. While in terms of Precision, SHIP$_{LLM}$ performs better than SHIP in scenarios "All" (65.59% vs. 59.14%), "Single" (73.38% vs. 65.58%), and "Multi" (50.32% vs. 46.50%). We examine the data and find that the code commits' relevance scores produced by SHIP$_{LLM}$ were generally lower, making it more difficult to connect code commits. This will result in smaller groups of code commits, which is conducive to obtaining higher Precision values. Nonetheless, the original SHIP achieves better performance with regard to Recall and F1-Score, which are more valuable metrics in real-world SVPI tasks.

## VI. DISCUSSION

In the experiment, we find the phenomenon that the length of vulnerability description may affect the effectiveness of SHIP. We discuss under different description lengths, how many vulnerability items SHIP can achieve perfect results for (i.e., the recommended patch group contains all real patches without introducing irrelevant code commits). The result is given in Figure 5. We can see that when the description length is between 30 and 50 tokens, the proportion of perfect items is the highest, and a too short (e.g., less than 30 tokens) or too long (e.g., more than 50 tokens) description will harm SHIP's effectiveness. We conjecture that a too short description may usually miss some key information, while a too long description may usually contain redundant information, which makes it difficult for the model to extract core features of vulnerabilities and thus hinders the subsequent identification of its patch(es).

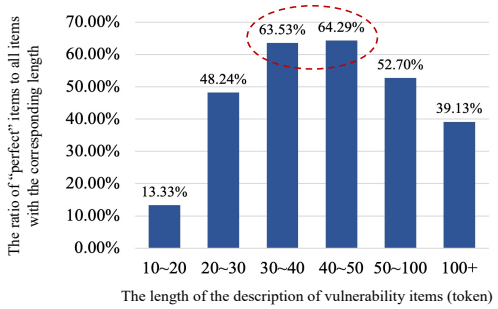


Fig. 5. SHIP's performance under different lengths of vulnerability description

Principal engineer of Threat Detection & Response at Cisco (a well-known computer security enterprise) investigated the length of vulnerability descriptions in open-source communities and found that from 2001 to 2025, the average length each year was basically between 30 $\sim$ 50 words (except 2024) [64]. This result indicates that writing moderate-length descriptions has become a common practice in real-world communities, which is precisely the scenario where SHIP is most likely to achieve good results.

## VII. THREAT AND VALIDITY

Our experiment is subject to several threats to validity. First, we collect more than 4,000 real-world vulnerability items from well-known platforms and high-quality datasets of previous works. Although this allows us to have higher confidence in terms of the generalization ability of SHIP, these could still not be enough to represent different kinds of vulnerabilities. Second, though the evaluation metrics we used are well-recognized and have been widely adopted, there could be other potential metrics that are suitable for evaluating the group-form output in multi-patch scenarios. In the future, we plan to build a larger-scale benchmark and search for or design better metrics to further evaluate SHIP.

## VIII. RELATED WORK

The research in the field of silent vulnerability patch identification mainly focuses on two aspects, feature extraction for the vulnerability and code commits, and model training for predicting the similarity between them.

A lot of studies propose diverse solutions for feature extraction. For instance, Tan et al. extract four types of features [13],

Wang et al. define 36 statistical features between a vulnerability and a code commit and 64 deep textual features [9], to measure the similarity between code commits and a vulnerability, and Zhang et al. further optimize the aforementioned features to retain 26 statistical ones [14]. Although these works extract abundant features from diverse aspects, they overlook the interrelationship between code commits and thus may encounter limitations in multi-patch scenarios.

Many researchers dedicate their effort to training a prediction model. For example, Nguyen et al. use three deep learning models to analyze commit messages, issue reports, and code changes respectively, and aggregate these models' outputs to form a final ranking list of code commits [8]. Liu et al. point out that too many non-patch code commits of a vulnerability would affect the identification efficiency, thus they first perform lexical and semantic matching to select a part of code commits, and then fine-tune a pretrained model with labeled data to enable it to re-rank the selected code commits [10]. Though these models have shown promising results in SVPI tasks, their ranking list-form outputs are not an appropriate choice to facilitate real-world tasks especially when dealing with multi-patch vulnerabilities.

## IX. Conclusion

Silent vulnerability patch identification is highly-important in software security and quality assurance, multi-patch vulnerabilities are very common in this task. This paper proposes SHIP, a novel SVPI approach that can perform well especially in multi-patch scenarios. The main innovation of SHIP is twofold: Taking the interrelationship between code commits into account, and directly producing a patch group instead of a ranking list. Experimental results show the promise of SHIP: It exceeds the SOTA technique by 37.54%, 28.71%, and 32.35% in terms of Recall, Precision, and F1-Score, respectively.

In the future, we will further explore the linkage between vulnerabilities and patches from a deeper insight by presenting finer-grained features and designing more effective models. A more comprehensive experiment is also to be considered.

## References


[1] D. Hommersom, A. Sabetta, B. Coppola, D. D. Nucci, and D. A. Tamburri, "Automated mapping of vulnerability advisories onto their fix commits in open source repositories," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–28, 2024.

[2] L. Wang, J. Wu, Y. Wu, Z. Rui, T. Luo, S. Qu, and M. Yang, "Intelligent perception for vulnerability threats in open-source software supply chain," *Ruan Jian Xue Bao/Journal of Software*, vol. 36, no. 2, pp. 511–536, 02 2025.

[3] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem," in *2023 IEEE/ACM 45th International Conference on Software Engineering*, 2023, pp. 1046–1058.

[4] M. Han, L. Wang, J. Chang, B. Li, and C. Zhang, "Learning graph-based patch representations for identifying and assessing silent vulnerability fixes," in *2024 IEEE 35th International Symposium on Software Reliability Engineering*, 2024, pp. 120–131.

[5] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, "Graphspd: Graph-based security patch detection with enriched code semantics," in *2023 IEEE Symposium on Security and Privacy*, 2023, pp. 2409–2426.

[6] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "SPI: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–27, 2021.

[7] J. Yu, Y. Chen, D. Tang, X. Liu, X. Wang, C. Wu, and H. Tang, "LLM-enhanced software patch localization," *arXiv preprint arXiv:2409.06816*, 2024.

[8] T. G. Nguyen, T. Le-Cong, H. J. Kang, X.-B. D. Le, and D. 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*, 2022, pp. 1726–1730.

[9] S. Wang, Y. Zhang, L. Bao, X. Xia, and M. Wu, "Vcmatch: a ranking-based approach for automatic security patches localization for oss vulnerabilities," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2022, pp. 589–600.

[10] K. Li, J. Zhang, S. Chen, H. Liu, Y. Liu, and Y. Chen, "Patchfinder: A two-phase approach to security patch tracing for disclosed vulnerabilities in open-source software," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 590–602.

[11] National Vulnerability Database. (2016) CVE-2017-5638. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-5638

[12] WIRED. (2017, Sep.) Equifax officially has no excuse: A patch that would have prevented the devastating equifax breach had been available for months. [Online]. Available: https://www.wired.com/story/equifax-breach-no-excuse/

[13] X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, and M. Yang, "Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3282–3299.

[14] J. Zhang, X. Hu, L. Bao, X. Xia, and S. Li, "Dual prompt-based few-shot learning for automated vulnerability patch localization," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2024, pp. 940–951.

[15] C. Xu, B. Chen, C. Lu, K. Huang, X. Peng, and Y. Liu, "Tracking patches for open source software vulnerabilities," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 860–871.

[16] The MITRE Corporation. (1999) Common vulnerabilities and exposures. [Online]. Available: https://cve.mitre.org

[17] US-CERT Security Operations Center. (2005) National vulnerability database. [Online]. Available: https://nvd.nist.gov

[18] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021, pp. 30–39.

[19] R. Lin, Y. Fu, W. Yi, J. Yang, J. Cao, Z. Dong, F. Xie, and H. Li, "Vulnerabilities and security patches detection in oss: A survey," *ACM Computing Surveys*, vol. 57, no. 1, pp. 1–37, 2024.

[20] X. Tan, Y. Zhang, J. Cao, K. Sun, M. Zhang, and M. Yang, "Understanding the practice of security patch management across multiple branches in oss projects," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 767–777.

[21] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[22] S. Woo, E. Choi, and H. Lee, "A large-scale analysis of the effectiveness of publicly reported security patches," *Computers & Security*, vol. 148, p. 104181, 2025.

[23] M. Jiang, J. Jiang, T. Wu, Z. Ma, X. Luo, and Y. Zhou, "Understanding vulnerability inducing commits of the linux kernel," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–28, 2024.

[24] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[25] B. Wu, S. Liu, R. Feng, X. Xie, J. Siow, and S.-W. Lin, "Enhancing security patch identification by capturing structures in commits," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[26] T. Chen, L. Li, T. Qian, J. Liu, W. Yang, D. Li, G. Liang, Q. Wang, and T. Xie, "Compvpd: Iteratively identifying vulnerability patches based on human validation results with a precise context," *arXiv preprint arXiv:2310.02530*, 2023.

[27] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting '0-day' vulnerability: An empirical study of secret security patch in oss," in

*2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2019, pp. 485–492.
[28] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. 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*, 2021, pp. 705–716.
[29] A. D. Householder, G. Wassermann, A. Manion, and C. King, "The cert guide to coordinated vulnerability disclosure," *Software Engineering Institute, Pittsburgh, PA*, 2017.
[30] ISO/IEC, "ISO/IEC 29147:2018: Information technology - security techniques - vulnerability disclosure," 2018. [Online]. Available: https://www.iso.org/standard/72311.html
[31] A. D. Sawadogo, T. F. Bissyandé, N. Moha, K. Allix, J. Klein, L. Li, and Y. L. Traon, "Learning to catch security patches," *arXiv preprint arXiv:2001.09148*, 2020.
[32] National Vulnerability Database. (2014) CVE-2014-5273. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2014-5273
[33] ——. (2019) CVE-2019-12108. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-12108
[34] ——. (2020) CVE-2020-25781. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2020-25781
[35] ——. (2015) CVE-2015-2080. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2015-2080
[36] ——. (2021) CVE-2021-25932. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-25932
[37] Github Repo: OpenNMS. (2021) Commit 8a97e68. [Online]. Available: https://github.com/OpenNMS/opennms/commit/8a97e6869d6e49da18b208c837438ace80049c01
[38] ——. (2021) Commit eb08b5e. [Online]. Available: https://github.com/OpenNMS/opennms/commit/eb08b5ed4c5548f3e941a1f0d0363ae4439fa98c
[39] ——. (2021) Commit f3ebfa3. [Online]. Available: https://github.com/OpenNMS/opennms/commit/f3ebfa3da5352b4d57f238b54c6db315ad99f10e
[40] R. Cabrera Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2vec: Learning distributed representations of code changes," *SN computer science*, vol. 2, no. 3, p. 150, 2021.
[41] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 914–919.
[42] X. He, S. Wang, P. Feng, X. Wang, S. Sun, Q. Li, and K. Sun, "Bingo: Identifying security patches in binary code with graph representation learning," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1186–1199.
[43] Y. Yang, L. Bo, Y. Wei, X. Wu, and X. Sun, "Patch-locator: A ranking-based approach of security patch localization for oss vulnerabilities," *Journal of Chinese Computer Systems*, vol. 45, 2024.
[44] Public Repository. (2025) Rule-based features used in Phase-1 of SHIP. https://github.com/SHIP-Repo/Repo/blob/main/Phase-1 Initial Ranking/Phase1-Features.pdf.
[51] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.
[45] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.
[46] J. Lee and K. Toutanova, "Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, vol. 3, no. 8, 2018.
[47] J. Lan, L. Gong, J. Zhang, and H. Zhang, "Btlink: automatic link recovery between issues and commits based on pre-trained bert model," *Empirical Software Engineering*, vol. 28, no. 4, p. 103, 2023.
[48] D. Cortiz, "Exploring transformers in emotion recognition: a comparison of bert, distillbert, roberta, xlnet and electra," *arXiv preprint arXiv:2104.02041*, 2021.
[49] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.
[50] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.
[52] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.
[53] A. I. Kadhim, "Term weighting for feature extraction on twitter: A comparison between bm25 and tf-idf," in *2019 International Conference on Advanced Science and Engineering*, 2019, pp. 124–128.
[54] Public Repository. (2025) Prompt designed for LLM in SHIP. [Online]. Available: https://github.com/SHIP-Repo/Repo/blob/main/Prompt.md
[55] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323.
[56] National Vulnerability Database. (2023) CVE-2023-50723. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-50723
[57] Public Repository. (2025) Complete version of the running example. https://github.com/SHIP-Repo/Repo/tree/main/Running example.
[58] Github. (2025) Github repo: XWiki-platform. [Online]. Available: https://github.com/xwiki/xwiki-platform
[59] Snyk Company. (2015) Snyk vulnerability database. [Online]. Available: https://security.snyk.io
[60] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*, 2019, pp. 383–387.
[61] C. D. Manning, *An introduction to information retrieval*, 2009.
[62] A. Gunawardana and G. Shani, "A survey of accuracy evaluation metrics of recommendation tasks." *Journal of Machine Learning Research*, vol. 10, no. 12, pp. 2935–2962, 2009.
[63] G. Naidu, T. Zuva, and E. M. Sibanda, "A review of evaluation metrics in machine learning algorithms," in *Computer science on-line conference*, 2023, pp. 15–25.
[64] J. Gamblin. (2025, March) Does a CVE's description length matter? average description length of CVEs by year from 1988 to 2025. https://github.com/SHIP-Repo/Repo/blob/main/Link to Post.pdf.