FixSeeker: An Empirical Study and Graph-based Approach for Detecting Silent Vulnerability Fixes in Open Source Software

YIRAN CHENG, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, China
TING ZHANG*, Monash University, Australia
LWIN KHIN SHAR, Singapore Management University, Singapore
ZHE LANG, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, China
DAVID LO, Singapore Management University, Singapore
SHICHAO LV*, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, China
DONGLIANG FANG, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, China
ZHIQIANG SHI, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, China
LIMIN SUN, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, China

Open source software (OSS) vulnerabilities pose significant security risks to downstream applications. While vulnerability databases provide valuable information for mitigation, many security patches are released *silently* in new commits of OSS repositories without explicit indications of their security impact. This makes it challenging for software maintainers and users to detect and address these vulnerabilities. Several approaches exist for detecting vulnerability-fixing commits (VFCs), but most of these approaches leverage commit messages, which would miss *silent* VFCs. On the other hand, there are some approaches for detecting silent

*Ting Zhang and Shichao Lv are the corresponding authors.

Authors' addresses: Yiran Cheng, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, chengyiran@iie.ac.cn; Ting Zhang, Monash University, Melbourne, Australia, ting.zhang@monash.edu; Lwin Khin Shar, Singapore Management University, Singapore, Singapore, lkshar@smu.edu.sg; Zhe Lang, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, langzhe@iie.ac.cn; David Lo, Singapore Management University, Singapore, Singapore, davidlo@smu.edu.sg; Shichao Lv, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, lvshichao@iie.ac.cn; Dongliang Fang, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, fangdongliang@iie.ac.cn; Zhiqiang Shi, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, shizhiqiang@iie.ac.cn; Limin Sun, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, sunlimin@iie.ac.cn.

ACM XXXX-XXXX/2025/4-ART
https://doi.org/XXXXXXX.XXXXXXX

VFCs based on code change patterns, but they often fail to characterize vulnerability fix patterns, thereby lacking effectiveness. For example, some approaches analyze each hunk in known VFCs, in isolation, to learn vulnerability fix patterns; but vulnerability fixes are often associated with multiple hunks, in which case correlations of code changes across those hunks are essential for characterizing the vulnerability fixes.

To address these problems, we first conduct a large-scale empirical study on 11,900 VFCs across six programming languages, in which we found that over 70% of VFCs involve multiple hunks with various types of correlations. Based on our findings, we propose FIXSEEKER, a graph-based approach that extracts the various correlations between code changes at the hunk level to detect silent vulnerability fixes. Our evaluation demonstrates that FIXSEEKER outperforms state-of-the-art approaches across multiple programming languages, achieving a high F1-score of 0.818 on average in balanced datasets and consistently improving F2-score, AUC-ROC, and AUC-PR scores by 10.6%, 5.3%, and 10.3% on imbalanced datasets compared to the best baseline method. Our evaluation also indicates the generality of FIXSEEKER across different vulnerability types and repository sizes.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Vulnerability-fixing commit, Open source software, Graph learning

ACM Reference Format:
Yiran Cheng, Ting Zhang, Lwin Khin Shar, Zhe Lang, David Lo, Shichao Lv, Dongliang Fang, Zhiqiang Shi, and Limin Sun. 2025. FIXSEEKER: An Empirical Study and Graph-based Approach for Detecting Silent Vulnerability Fixes in Open Source Software. 1, 1 (April 2025), 31 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

With the increasing adoption of open-source software (OSS) components, vulnerabilities embedded within them can propagate to a vast number of downstream applications [70**?**]. One example is the critical OpenSSL vulnerability (CVE-2022-3602 [8]) discovered in 2022, which affected numerous organizations and systems that rely on this library. This vulnerability in X.509 certificate verification could allow attackers to trigger a buffer overflow during certificate parsing, potentially leading to remote code execution. Such incidents highlight how vulnerabilities in widely used OSS can have far-reaching consequences across the software ecosystem.

While vulnerability databases like the National Vulnerability Database (NVD) provide valuable information for mitigation, many security patches are released silently by making new commits in OSS repositories without explicit indications about vulnerability fixes and their security impacts. This practice, known as *silent fixing*, follows coordinated vulnerability disclosure guidelines that recommend avoiding any reference to a security-related nature in commit messages [31, 66, 67, 76]. Silent fixing is not merely about secrecy; it also serves a crucial function in the security ecosystem by providing a time window for downstream users and maintainers to update their software before the vulnerability becomes widely known to potential attackers. [42, 49]. This balance between transparency and protection is critical, as publicly disclosing vulnerabilities too quickly can significantly increase the risk of exploitation before patches can be widely deployed.

However, the delay between patch availability and vulnerability disclosure creates a critical security challenge. Without explicit security indicators, downstream projects and users may fail to recognize the security implications of these changes, potentially delaying critical updates. For instance, a critical remote code execution vulnerability (CVE-2017-1000251 [5]) in the Linux kernel's Bluetooth implementation was silently fixed in August 2009, but not disclosed until September 2017, leaving users exposed to potential attacks during the eight-year period. This creates a pressing need for automated methods that can identify silent security fixes based solely on code changes, helping downstream maintainers prioritize security-relevant updates without relying on explicit vulnerability disclosures.

To address these problems, automated approaches for identifying vulnerability-fixing commits (VFCs) have been proposed. Many approaches [23, 52, 55] rely on textual analysis of commit messages or other metadata that contain security-related terms. While this can be effective for explicitly labeled security patches, it misses "silent" security fixes where developers intentionally omit security-related terms. Furthermore, commit messages are often unclear, making it difficult to comprehend the commit's intention [13, 24]. There are some approaches [53, 76] that attempt to address the silent security fix problem by learning patterns solely from code changes to identify VFCs. VulFixMiner [76] extracts features from the diff between commits, while Midas [53] employs multiple feature extractors at different code granularities. These code-based approaches show promise, but they face limitations in handling the structural complexity of vulnerability fixes. Many VFCs are composed of multiple diff hunks [66], where a diff hunk represents a contiguous group of modified (i.e., added and removed) lines in a single file. The changes within hunks may be correlated in terms of fixing the vulnerability. VulFixMiner and Midas both treat code changes as basic sequential structures, failing to consider the semantic relationships and dependencies between different hunks of code change, which are crucial for machine learning-based approaches to learn the vulnerability fix patterns adequately. Therefore, to effectively model such semantic relationships in VFC detection, we first need to understand their characteristics. However, the challenge is that *it remains unclear what types of relationships exist between the different hunks of VFCs.*

Empirical Study. To address this challenge, we conduct an empirical study on a large-scale dataset containing 11,900 VFCs across six popular programming languages. Our analysis reveals that the majority of VFCs (over 70% across all languages) involve multiple hunks. Through manual analysis, we observe that there are *four* main types of correlations between hunks in multi-hunk VFCs: *Caller-Callee Dependency*, *Data Flow Dependency*, *Control Dependency*, and *Pattern Replication*. These correlations exist in 92.7% of the examined multi-hunk VFCs. Furthermore, we find that multi-hunk VFCs are more prevalent in fixing severe vulnerabilities, with 68.5% of multi-hunk VFCs addressing "critical" and "high" security vulnerabilities, compared to 57.3% for single-hunk VFCs. These findings highlight the importance of considering inter-hunk correlations in VFC detection approaches.

Our Approach. Based on the findings from our empirical study, we propose FIXSEEKER, a graph-based approach for detecting silent VFCs in OSS repositories. At a high level, FIXSEEKER is a novel, synergistic combination of the use of code property graphs (CPGs) capturing the relationships among hunks, static analysis of CPGs extracting features that characterize vulnerability fix patterns based on our empirical findings, and a graph-based learning approach to capture the features and classify VFCs effectively. More specifically, FIXSEEKER consists of five steps. First, given a commit, FIXSEEKER retrieves pre-commit and post-commit codes and preprocesses both versions, then generates CPGs for them. Second, it conducts static analysis in CPGs to extract the hunks' explicit correlation, containing the caller-callee dependency, data flow dependency, and control dependency. It uses code mapping with the Levenshtein algorithm to detect implicit correlations (pattern replication). Third, it constructs Hunk Correlation Graphs (HCGs) for both pre-commit and post-commit states, then merges them into a unified CommitHCG. Then, it embeds nodes with CodeBERT [25] and encodes edges as 4-D binary vectors to convert the CommitHCG into a numeric format. Finally, CommitGNN uses Graph Convolutional Networks (GCNs) with an edge attention mechanism to learn the embedded CommitHCG. This model, trained with a weighted loss function to address the class imbalance, classifies whether a commit fixes a vulnerability.

Evaluation. To evaluate FIXSEEKER, we use two benchmark datasets: a balanced dataset with a 1:1 ratio of VFCs to non-VFCs, and an imbalanced dataset with a 1:50 ratio of VFCs to non-VFCs. The balanced dataset allows us to assess the model's performance under ideal conditions, while the imbalanced dataset reflects the natural distribution in real-world scenarios where VFCs are rare. Our datasets cover four programming languages: C/C++, Java, Python, and PHP, with a total of 9,983 VFCs across 2,094 open-source projects.

To demonstrate the effectiveness of FIXSEEKER, we compare it with five state-of-the-art (SOTA) VFC detection approaches, namely PatchRNN [68], VFFINDER [51], VulFixMiner [76], Midas [53], and GRAPE [28], on both datasets. The results show that: i) FIXSEEKER outperforms the baselines in most metrics across different languages, achieving the highest F1-scores for four languages respectively on balanced datasets; ii) On imbalanced datasets, FIXSEEKER consistently demonstrates superior performance, improving the F2-score by 10.6%, AUC-ROC by 5.3%, AUC-PR by 10.3% on average compared to the best baseline method. In terms of detecting the capability of different vulnerability types, we find that FIXSEEKER performs strongly in memory-related vulnerabilities, achieving F2 exceeding 0.9 for these types. Moreover, to evaluate the efficiency of FIXSEEKER, we conduct an overhead analysis across 8 popular repositories of varying sizes. The results show that our CommitHCG generation process is efficient, achieving average generation time per commit ranging from 16.63 seconds to 24.39 seconds.

Contribution. This work makes the following contributions.

- We conduct a large-scale empirical study to investigate the multi-hunk VFCs and the correlations within different hunks.
- Based on our empirical study, we propose a novel graph-based automated approach, named FIXSEEKER, to detect silent vulnerability fixes, supporting cross-language scenarios.
- We conduct extensive experiments to demonstrate the effectiveness, generality, and practical usefulness of FIXSEEKER.

The rest of the paper is organized as follows. Section 2 introduces the preliminaries and a motivating example. Section 3 presents our empirical study on multi-hunk VFCs. Motivated by the findings of the empirical study, Section 4 details our proposed approach FIXSEEKER. Section 5 presents the experimental setup and evaluation results. Section 6 discusses our approach's practicality, limitations, and future work. Section 7 reviews related work and Section 8 concludes the paper.

2 PRELIMINARIES AND MOTIVATION

2.1 Definition

VFCs and non-VFCs. The commits of OSS record the changes between two different source code versions. In our work, we define a commit as a single-purpose Git commit. We consider a commit as a vulnerability-fixing commit (VFC) if it fixes a vulnerability belonging to any Common Weakness Enumeration (CWE) type. Conversely, we define a non-vulnerability-fixing commit (non-VFC) as any commit that does not address a security vulnerability, including those that fix functional bugs, add new features, or perform code refactoring.

Diff Hunk. In the context of Git and the unified diff format [11], a hunk is a single, contiguous section of a diff that represents a group of changes (additions, deletions, or modifications) to a file, along with surrounding unchanged lines for context. Each hunk begins with a header specifying the range of affected lines in the original and modified files (e.g., `@@ -S, O, +S', N @@`), where `S` is the starting line number in the original file, `O` is the number of lines affected in the original file, `S'` is the starting line number in the modified file, and `N` is the number of lines affected in the modified file. The header is followed by the changed lines, marked with `-` for deletion and `+` for additions,

alongside contextual unchanged lines. A single file's diff may contain multiple hunks, depending on the number of distinct groups of contiguous changed and unchanged lines [14]. Formally, a diff hunk can be represented as a tuple $H = (-S, O, +S', N)$, capturing the line ranges and changes in both file versions. As shown in Listing 1, the commit has three diff hunks in the modified file `iwl-agn-sta.c`, the corresponding value of $(-S, O, +S', N)$ in the first hunk is (-35,9,+35,12). Diff hunks are fundamental units in understanding and analyzing code changes, as they provide a granular view of the modifications made in a commit. In the context of VFCs, analyzing hunks and their relationships can offer insights into the nature and extent of security-related code changes.

CPG. A CPG is a unified graph representation that combines multiple program analysis graph types to capture different aspects of code semantics. Formally, a CPG can be defined as $G = (V, E)$, where V is a set of nodes representing program elements (e.g., functions, variables, operations), E is a set of directed edges connecting these nodes. CPG integrates the nodes and edges from three fundamental program analysis graphs: Abstract Syntax Tree (AST) graph representing the syntactic structure, Control Flow Graph (CFG) capturing control dependencies, and Data Flow Graph (DFG) representing data dependencies.

2.2 Motivating Example

To motivate our work, we present a vulnerability fix in the Linux kernel's wireless driver that demonstrates how multiple hunks collaborate to address a security vulnerability, highlighting the importance of understanding hunk relationships in vulnerability fixes. In this case, shown in Listing 1, three semantically related hunks collectively fix a potential out-of-bounds vulnerability in the `iwl_sta_ucode_activate` function. The first hunk (Lines 9-16) modifies the function signature from `void` to `int` and adds a bounds check `if (sta_id >= IWLAGN_STATION_COUNT)` to validate the station ID parameter. The second hunk (Line 24) adds a `return 0` statement to handle the success case properly. These two hunks together transform the function to enforce proper bounds checking and return a status indication. The third hunk (Lines 32-34) adapts the caller side by replacing the direct function call with proper error handling of the newly added return value. These hunks exhibit a caller-callee dependency: modifying the callee function's signature and behavior (first two hunks) necessitates corresponding changes in the caller's code (third hunk).

Current approaches like Midas [53], although extracting features at multiple granularities including commit, file, hunk, and line levels, focus primarily on analyzing each hunk independently when designing their feature extractors. When applied to this case, it processes the addition of bounds checking (Lines 13-15) and the modification of return value handling (Lines 32-34) as separate features, failing to capture their inherent caller-callee relationship that is crucial for understanding how the complete vulnerability fix works through coordinated error handling modifications across multiple functions. If these semantic relationships between hunks could be explicitly modeled and learned, the model would be better equipped to recognize such coordinated security fixes.

This example reveals a critical challenge in VFC detection: vulnerability fixes often involve multiple coordinated changes (hunks) that are semantically related but syntactically dispersed. By modeling these inter-hunk relationships in a graph structure, we can learn richer embeddings from graph representation that incorporate both the content of individual hunks and their relationships with other hunks, leading to more accurate detection of vulnerability fixes. However, to effectively model these semantic relationships in VFC detection, we first need to understand what types of relationships exist between different hunks of VFCs. This observation motivates us to conduct a large-scale empirical study on the relationships between different hunks in VFCs, aiming to understand the patterns and types of inter-hunk correlations contributing to security fixes.

```
diff --git a/drivers/net/wireless/iwlwifi/iwl-agn-sta.c b/drivers/net/wireless/iwlwifi/iwl-agn-
  sta.c
index 7353826095f110..8d4353a42568f7 100644
--- a/drivers/net/wireless/iwlwifi/iwl-agn-sta.c
+++ b/drivers/net/wireless/iwlwifi/iwl-agn-sta.c
-------------------------------------------------------------------------------------------Hunk1
@@ -35,9 +35,12 @@
 #include "iwl-trans.h"

 /* priv->shrd->sta_lock must be held */
-static void iwl_sta_ucode_activate(struct iwl_priv priv, u8 sta_id)
+static int iwl_sta_ucode_activate(struct iwl_priv priv, u8 sta_id)
 {
+  if (sta_id >= IWLAGN_STATION_COUNT) {
+   IWL_ERR(priv, "invalid sta_id %u", sta_id);
+   return -EINVAL;
+}
   if (!(priv->stations[sta_id].used & IWL_STA_DRIVER_ACTIVE))
     IWL_ERR(priv, "ACTIVATE a non DRIVER active station id %u "
       "addr %pM\n",
-------------------------------------------------------------------------------------------Hunk2
@@ -53,6 +56,7 @@ static void iwl_sta_ucode_activate(struct iwl_priv *priv, u8 sta_id)
     IWL_DEBUG_ASSOC(priv, "Added STA id %u addr %pM to uCode\n",
         sta_id, priv->stations[sta_id].sta.sta.addr);
   }
+ return 0;
 }

 static int iwl_process_add_sta_resp(struct iwl_priv *priv,
-------------------------------------------------------------------------------------------Hunk3
@@ -77,8 +81,7 @@ static int iwl_process_add_sta_resp(struct iwl_priv *priv,
   switch (pkt->u.add_sta.status) {
   case ADD_STA_SUCCESS_MSK:
     IWL_DEBUG_INFO(priv, "REPLY_ADD_STA PASSED\n");
-  iwl_sta_ucode_activate(priv, sta_id);
-  ret = 0;
+  ret = iwl_sta_ucode_activate(priv, sta_id);
     break;
   case ADD_STA_NO_ROOM_IN_TABLE:
     IWL_ERR(priv, "Adding station %d failed, no room in table.\n",
```

Listing 1. Motivating Example of CVE-2017-14169.

3 AN EMPIRICAL STUDY

We design an empirical study to understand the characteristics of VFC hunks for OSS vulnerabilities in databases. We aim to answer the following research questions (RQs).

- **RQ1: Multi-hunk Prevalence:** What proportion of VFCs contain multiple code hunks? (Section 3.2)
- **RQ2: Hunk Correlation Analysis:** What types of correlations exist between different hunks in multi-hunk VFCs? (Section 3.3)

- **RQ3: Severity and Vulnerability Type Analysis:** How does vulnerability severity relate to multi-hunk VFCs? (Section 3.4)

3.1 Data Preparation

We selected six popular PLs as our analysis subjects: C/C++, Java, Python, PHP, JavaScript, and Golang. We extracted all CVEs related to the six languages disclosed by September 26, 2024 from NVD [10]. For each vulnerability entry in NVD, we analyze its reference section, which contains links to various resources. From these references, we identify GitHub commit URLs by filtering for links that contain "github.com" and "commit". These commits typically represent the fixes for the reported vulnerabilities. A commit may contain code changes in various types of files, but we focus only on .c, .cpp, .java, .py, .php, .js, and .go files. Therefore, we filtered the dataset by removing the commits that only contained modification files unrelated to the target languages. Finally, we collected 5,168 commits in C/C++, 786 commits in Java, 1,147 commits in Python, 3,157 commits in PHP, 1,039 commits in JavaScript, and 603 commits in Golang, across 2,832 open-source projects. As shown in Table 1, C/C++ projects contributed the most VFCs. We observed that some commits include modifications to test files. However, since the modifications of test files are typically intended to verify the effectiveness of vulnerability fixes rather than containing the actual vulnerability fix code, we ignore them in our analysis.

Table 1. Dataset Distribution across Different Programming Languages (PLs)

PL	#VFC	#File	#Hunk	#Project
C/C++	5,168	13,148	35,723	720
Java	786	4,281	9,817	260
Python	1,147	4,990	12,163	385
PHP	3,157	27,907	60,162	729
JavaScript	1,039	8,267	16,857	527
Golang	603	3,695	9,756	211
Overall	**11,900**	**62,288**	**144,478**	**2,832**

3.2 Multi-hunk Prevalence (RQ1)

Examining the complexity of modifications, we categorized VFCs into single-hunk and multi-hunk changes. Figure 1 illustrates that across all languages, the number of multi-hunk modifications significantly exceeds single-hunk modifications, indicating that most vulnerability fixes involve multiple code changes. Specifically, in C/C++, we observed 3,406 multi-hunk VFCs compared to 1,762 single-hunk VFCs (a ratio of approximately 2:1). This pattern is even more pronounced in other languages, particularly in Python and Golang, where multi-hunk VFCs are seven times more common than single-hunk VFCs, and in Java with 656 multi-hunk VFCs versus 130 single-hunk VFCs (a ratio of 5:1).

Finding#1: Across all studied PLs, multi-hunk fixes significantly outnumber single-hunk fixes. This is particularly evident in Python and Golang, where multi-hunk fixes exceed single-hunk fixes by over 7 times.

3.3 Hunk Correlation Analysis (RQ2)

To investigate this RQ, we conduct a manual analysis of sample VFCs. We considered a 95% confidence level with a 5% margin of error, resulting in a sample size of 416 VFCs out of the 11,900

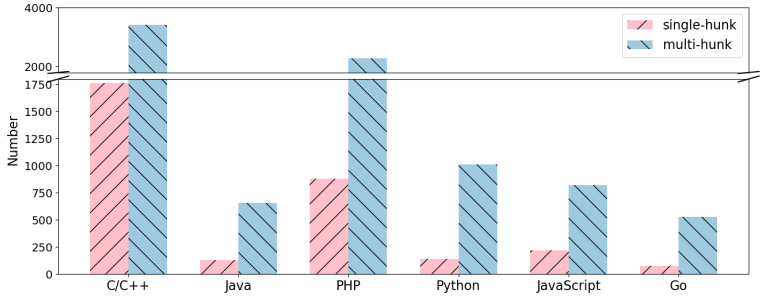


Fig. 1. The Number of Multi-hunk and Single-hunk VFCs across Different Programming Languages

total VFCs in the dataset. To ensure a thorough understanding of hunk correlations, we conducted a structured manual analysis process across C/C++ and Java projects. For C/C++, we focused on 316 multi-hunk VFCs from three widely-used OSS projects (Linux, FFmpeg, and ImageMagick), which provide a representative sample of complex and well-maintained codebases with established security practices. For Java, we analyzed 100 VFCs randomly selected from a diverse set of projects to capture a broader range of project patterns and contexts. This complementary sampling allowed us to balance depth (C/C++ projects) with breadth (Java projects).

To ensure the reliability of our findings, the first and second authors independently analyzed the same set of multi-hunk VFCs without relying on predefined categories. Our manual analysis followed an open-card sorting procedure [62]. This bottom-up method, widely adopted in software engineering research [16, 61] allows researchers to empirically derive taxonomies by freely clustering concepts based on observed patterns. Following the independent analysis phase, the authors compared their findings, resolved divergent interpretations through face-to-face discussions, and reached a consensus on the final set of correlation types. We observed that the hunks in 92.7% commits have correlations in terms of Caller-Callee Dependency, Data Flow Dependency, Control Dependency, and Pattern Replication, as shown in Table 2. Figure 2 illustrates the above four types of correlation. We use arrows to indicate the starting points of different hunks. The four types of correlations are explained below.

Caller-Callee Dependency. The hunk in function A has code changes (i.e., function addition and modification), and the call-related hunk of A's caller also undergoes code changes. In the VFC of CVE-2017-7645 [6] in Figure 2, the second hunk calls the function `nfs_request_too_big` in the first hunk. This type of correlation accounts for 28.80% of commits in C/C++, and 67% in Java projects.

Control Dependency. The code in hunk B has a control dependency on hunk A. When the code in hunk A changes, hunk B changes accordingly. In the VFC of CVE-2015-5283 [2] in Figure 2, the second hunk has a control dependency on the first hunk because of the statement `goto err_ctl_sock_init`. This type of correlation is most prevalent in C/C++ projects at 30.7%, and in Java projects is 21%.

Data Flow Dependency. The code in hunk B uses the data defined or assigned in hunk A. In the VFC of CVE-2018-20784 [7] in Figure 2, the second hunk uses the pointer variables `cfs_rq` and `pos` from the first hunk. This type of correlation is particularly prominent in Java projects, accounting for 45% of commits, while it represents 42.1% in C/C++ projects.

Pattern Replication. The code in hunk A and hunk B have highly similar code change patterns. In the VFC of CVE-2015-8543 [3] in Figure 2, the two hunks both add two conditional checks for

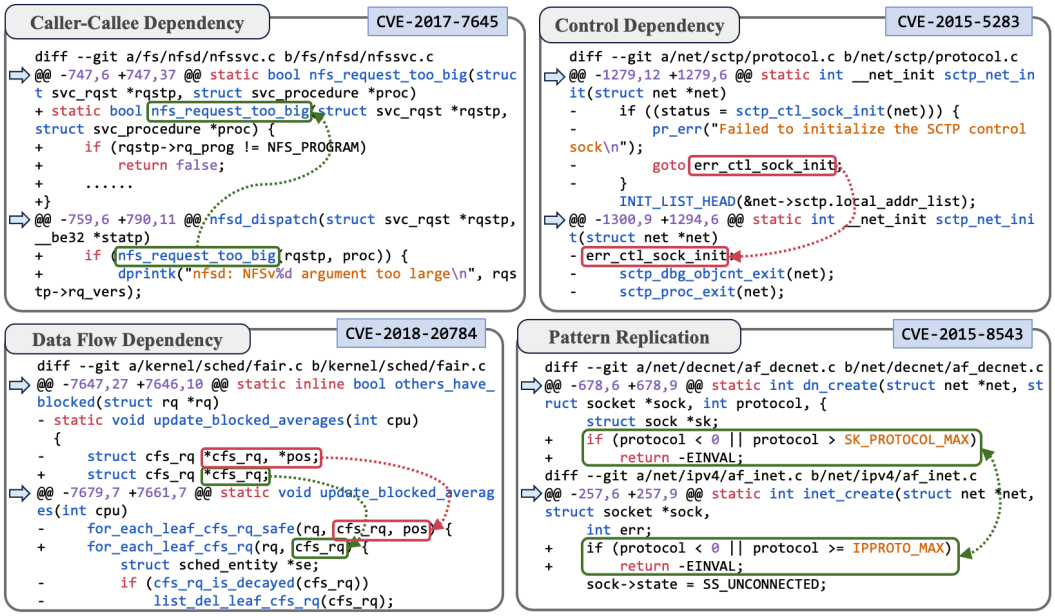


Fig. 2. Examples for Different Hunk Correlations

variable `protocol`. This type of correlation is most common in Java projects, accounting for 50% of commits, while it represents 41.5% in C/C++ projects.

The first three are explicit correlations, and the last one is an implicit correlation. We observed that many commits contain multiple types of hunk correlations (correlation "Hybrid" in Table 2). The proportion of such commits in the two programming languages is 35.8%, and 59%, respectively.

In addition to the four main correlations, our manual analysis also revealed some less common correlation patterns. For instance, we observe code relocation correlations[1], where code is removed from one hunk and added elsewhere. Another example is the correlation of removing the class extension[2], where a class inheritance is removed along with the corresponding overridden methods. Although these additional correlation patterns provide interesting insight into fixing patterns, they appear much less frequently in our dataset compared to the four main types. Therefore, our automated approach focuses on extracting and modeling the four primary correlation types identified above, which cover most inter-hunk relationships in vulnerability fixes.

Finding#2: In our manual analysis of 416 multi-hunk VFCs, we identified four main correlation types: Caller-Callee Dependency, Data Flow dependency, Control Dependency, and Pattern Replication. The correlations exist in 92.7% of these examined samples. Many commits contain multiple correlation types.

3.4 Severity Analysis (RQ3)

We collected the severity of the vulnerabilities from NVD corresponding to the VFCs. NVD uses Common Vulnerability Scoring System (CVSS) to rate vulnerabilities on a scale of 0 to 10, the severity levels are Low (0-3.9), Medium (4.0-6.9), High (7.0-8.9) and Critical (9.0-10.0). We found that 96 CVEs (0.8% of total CVEs) did not have assigned CVSS severity scores from the NVD. We

[1]https://github.com/torvalds/linux/commit/28f5a8a7c033cbf3e32277f4cc9c6afd74f05300
[2]https://github.com/jenkinsci/subversion-plugin/commit/25f6afb

Table 2. Distribution of Different Hunk Correlations in Open-source Software

Type	Correlation	C/C++	Java
Explicit	Caller–Callee Dependency	28.8%	67.0%
	Control Dependency	30.7%	21.0%
	Data Flow Dependency	42.1%	45.0%
Implicit	Pattern Replication	41.5%	50.0%
Hybrid	–	35.8%	59.0%

excluded these CVEs to ensure the accuracy of our findings. Specifically, 5,962 (68.5%) multi-hunk VFCs fix the vulnerability with severity rated "Critical" and "High", while for single-hunk VFCs, the number is 1,831 (57.3%). This suggests that multi-hunk VFCs are more inclined to fix more severe vulnerabilities.

Finding#3: 68.5% of multi-hunk VFCs fix vulnerabilities of "critical" and "high" severity, compared to 57.3% for single-hunk VFCs.

4 OUR APPROACH

In this section, we propose an automated approach, FIXSEEKER, for detecting VFCs in OSS repositories, without relying on commit messages. The motivation of FIXSEEKER is based on the findings from our empirical study that VFCs often address the vulnerability by changing multiple associated code hunks, exhibiting four types of correlation patterns. We leverage code graphs as they characterize dependencies among code changes and hunks. We employ graph neural networks (GNNs) as they can naturally learn complex patterns in graphs. In the following part, we first provide an overview of our approach. We then explain the steps involved in detail, using a running example (Figure 4).

Figure 3 presents an overview of FIXSEEKER. FIXSEEKER takes a commit as its input and outputs whether the commit is for vulnerability-fixing or not. FIXSEEKER works in five steps. 1) First, it retrieves pre-commit and post-commit code via git reset, preprocesses functions, and generates CPGs for both versions. 2) It then proceeds to extract the hunk correlations, conducting the static analysis in CPGs to analyze explicit correlations while using code mapping to identify implicit correlations, creating Hunk Correlation Graphs (HCGs). 3) The Merging step combines pre-commit and post-commit HCGs into a unified CommitHCG. 4) Then, it embeds nodes with CodeBERT and encodes edges as 4-D binary vectors. 5) Finally, we propose a GNN model, CommitGNN, which processes the embedded CommitHCG through GCN layers with edge attention mechanisms and a weighted loss function, and then predicts whether the corresponding commit fixes a vulnerability.

4.1 Hunk Correlation Graph Construction

A commit is composed of code changes in a group of files. A hunk represents a set of sequentially concatenating lines of code changes within a file. We decompose the diff code of commit C into a set of hunks: $C = \{H_1, H_2, ..., H_n\}$. Since a hunk contains both pre-commit and post-commit code, we analyze the correlation of hunk code in pre-commit and post-commit states separately, denoted as H_{pre} and H_{post}, which are assigned the same id.

4.1.1 Source Code Preprocessing (Step 1). Given a commit ID, we can use git reset to roll back and obtain the source code before and after the commit. By diffing the source code before and after the commit, we can identify the changed files, denoted as F_{pre} and F_{post}. We first identify all functions and their respective scopes (defined by start and end line numbers) in both F_{pre} and F_{post} using

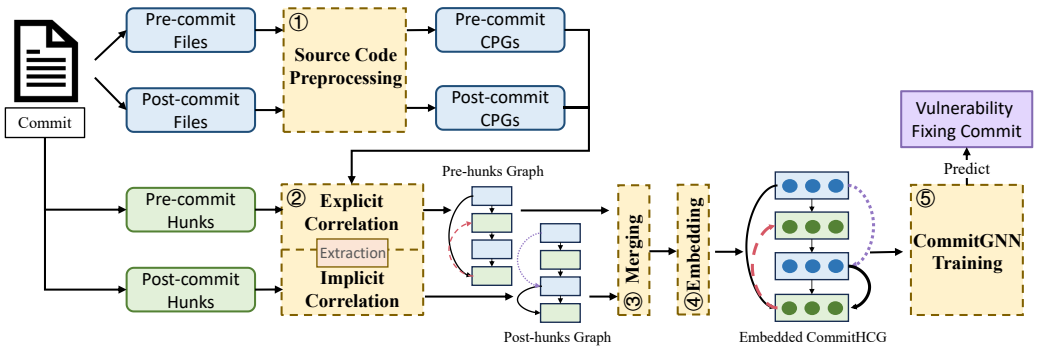


Fig. 3. The Workflow of FIXSEEKER

Joern Parser [9]. The diff file provides range information for each code change hunk (e.g., Line 2 of CVE-2017-7645's VFC in Figure 2), enabling us to locate the corresponding functions of hunks, denoted as Fc_{pre} and Fc_{post} via scope comparison. To focus more specifically on the functions affected by code changes, we remove patch-irrelevant functions in the files. Subsequently, we utilize the Joern parser again to generate CPGs for F_{pre} and F_{post} separately, yielding CPG_{pre} and CPG_{post} respectively. Note that if the VFC contains only a single hunk, we can skip the hunk correlation extraction (Step 2) and merging process (Step 3) since there are no inter-hunk relationships to analyze.

4.1.2 Multi-hunk Correlation Extraction (Step 2). As we mentioned in Section 3.3, the correlation in different hunks can be categorized into explicit and implicit correlations. Explicit correlation includes caller-callee, data flow, and control dependencies between hunks. Implicit correlation refers to hunk diffs that have similar code change patterns (pattern replication).

1) *Caller-callee dependency extraction.* Given a CPG, we traverse the functions in the CPG to obtain the caller and callee for each function, forming a function graph with two sets: (N, E). N is a set of nodes represented by function *ids*, where *id* is the unique identifier assigned to each function by the CPG. E is a set of directed edges represented by 2-tuples $(id1, id2)$, where $id1$ is the caller and $id2$ is the callee. If there is a call operation to the callee within the hunks of the caller, we add an edge between these two hunks, represented as $(H_{id1}, H_{id2}, type)$, $type \in \{CALL\}$.

2) *Data flow and control dependency extraction.* CPG consists of nodes and edges. Nodes are represented by *id*, *code*, and *line number*. The *code* refers to a statement component in the source code, while the *line number* indicates the line where this statement is located. Edges represent relations between the nodes, represented as $(id1, id2, type)$. There are three types of edges: AST, Data Dependency (DD), and Control Dependency (CD). We only consider relation DD and CD, since AST relations exist only among multiple components within a single statement, which is irrelevant to correlations between hunks. We employ scope comparison to determine whether *ids* with relations belong to different hunks, if so, we add an edge between these two hunks, represented as $(H_{id1}, H_{id2}, type)$, $type \in \{DD, CD\}$.

3) *Pattern replication extraction.* To determine whether similar modification patterns have been applied across different hunks, we employ a code similarity analysis. First, we preprocess the hunk code through normalization and abstraction techniques. During normalization, we conduct whitespace standardization, comment removal, and identifier standardization to establish consistent formatting. In the abstraction phase, as shown in Algorithm 1, we replace all variable names with standardized placeholders (e.g., VAR1, VAR2) and replace all argument

names with placeholders (e.g., ARG1, ARG2), while preserving their usage context. For each pair of hunks (H_{id1}, H_{id2}), we calculate the normalized Levenshtein distance: $NLD(H_{id1}, H_{id2}) = 1 - LD(H_{id1}, H_{id2})/max(len(H_{id1}), len(H_{id2}))$, where $LD(H_{id1}, H_{id2})$ is the Levenshtein distance between the code sequences of H_{id1} and H_{id2}. To ensure cross-hunk comparability, we normalize the distance by the maximum hunk length $len(H)$. This yields a normalized similarity score $NLD \in [0, 1]$, where a higher value indicates a closer modification pattern. We set a similarity threshold $\theta = 0.8$ following previous work [39]. If $NLD(H_{id1}, H_{id2}) > \theta$, we consider the hunks to have a similar modification pattern and add an edge between these two hunks, represented as $(H_{id1}, H_{id2}, type)$, where $type \in \{SIM\}$.

Algorithm 1 Code Abstraction Phase in Pattern Replication Extraction

```
function CodeAbstraction(code)
    keywords ← {Built-in function names such as "print"}
    varMap ← ∅, argMap ← ∅, varCount ← 1, argCount ← 1
                                              ▷ Extract function definitions and arguments
    for each function definition (funcName, args) in code do
        if funcName ∉ keywords then
            varMap[funcName] ← "FUNC" + varCount++
        end if
        for each arg in args do
            if arg ∉ keywords then
                argMap[arg] ← "ARG" + argCount++
            end if
        end for
    end for
                                                  ▷ Extract variable assignments
    for each variable assignment var in code do
        if var ∉ argMap ∧ var ∉ varMap then
            varMap[var] ← "VAR" + varCount++
        end if
    end for
                                                  ▷ Apply replacements
    replacements ← Sort(varMap ∪ argMap) by key length descending
    for each (name, placeholder) in replacements do
        code ← ReplaceAll(code, name, placeholder)
    end for
    return code
end function
```

4.1.3 Merging into CommitHCG (Step 3). For each pair of pre-commit and post-commit hunks, we merge the corresponding two HCGs into a unified graph structure called CommitHCG. We first merge the H_{pre} and H_{post} with the same *id* as a unified hunk node. Then, we transform the source and end nodes of pre-commit and post-commit HCG edges into the unified hunk node. In this way, we obtain a CommitHCG depicted by two sets (N', E'), where N' is the set of unified hunk nodes and $E' = E_{pre} \cup E_{post}$, and E' is represented with 3-tuples $(H_{id1}, H_{id2}, type)$, $type \in \{CALL, DD, CD, SIM\}$.

4.1.4 Running Example. We use the commit `410dd3` in the Linux Kernel as the running example. Figure 4 shows the code diff of the commit. In step 1, FixSeeker identifies the modified files (`fs/isofs/inode.c`) through the `diff -git [a/file] [b/file]` markers. Then it separates the

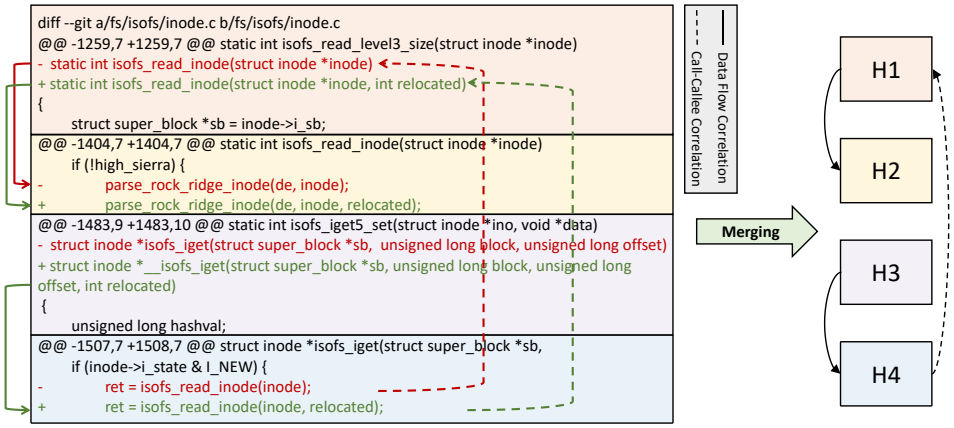


Fig. 4. A Running Example of Hunk Correlation Graph Construction

four modified hunks within each file using the `@@[-s,o,+s',n]@@` markers. FIXSEEKER obtains the F_{pre} and F_{post} of the file using `git reset` to roll back the changes in the repository. The Joern parser is used to determine the scope of all functions in both versions. By comparing the hunk ranges (s,o) and (s',n) with all the function scopes, FIXSEEKER locates the changed function `isofs_read_inode` with one parameter and `isofs_iget` in F_{pre}, `isofs_read_inode` with two parameters and `__isofs_iget` in F_{post}. It then removes the other unchanged (patch-unrelated) functions from F_{pre} and F_{post}. Finally, FIXSEEKER generates the corresponding CPGs for the refined versions: CPG_{pre} and CPG_{post}.

In step 2, FIXSEEKER extracts the correlations between hunks. In the caller-callee dependency extraction, it identifies function calls to `isofs_read_inode` in `isofs_iget` of F_{pre} and `__isofs_iget` of F_{post}, creating *CALL* edges between these hunks. For data flow dependency extraction, FIXSEEKER analyzes the CPGs and finds that within the `isofs_read_inode` function, it finds *DD* edges between the first hunk and the second hunk. In `__isofs_iget` function, it creates *DD* edges between the third hunk and the fourth hunk because of the dependency of variable `relocated`. For pattern replication extraction, it tokenizes the code changes in each hunk and calculates the normalized Levenshtein distance between them but does not find similar code changes.

In step 3, FIXSEEKER merges the pre-commit and post-commit HCGs into a unified CommitHCG. The resulting graph contains four nodes representing the modified hunks, with three edges representing the identified correlations (*CALL*, *DD*).

4.2 Graph Learning

After constructing the CommitHCG, we employ graph learning techniques to detect VFCs. This process involves three main steps: graph embedding, where we transform node and edge information into numerical vectors; CommitGNN model design, which leverages GNN with an edge attention mechanism to capture the complex relationships within the CommitHCG; and model training, where we address class imbalance issues and optimize the model's performance.

4.2.1 Graph Embedding (Step 4). To input CommitHCGs into a GNN-based model, the node and edge attributes in the graphs should be embedded as numeric vectors. Node attributes represent the Hunk code snippet in each node, while edge attributes represent the correlations between nodes.

1) *Node embedding.* A graph node is a pair of pre-commit (removed) and post-commit (added) codes within a hunk code snippet. FIXSEEKER fine-tunes CodeBERT [25] as code embedding models

for representing hunk code snippets in the graph nodes. In the representation, we consider the joint input format of CodeBERT as follows:

$$[CLS] < -code > [SEP] < +code > [EOS]$$

where [CLS] token is placed at the beginning of the input sequence and is used to capture a representation of the entire input for classification tasks. [SEP] token separates different segments (removed and added parts) within the input. [EOS] marks the end of the input sequence. <-code> represents removed codes, where each statement begins with the '-' symbol. <+code> represents added codes, where each statement begins with the '+' symbol.

2) *Edge embedding.* Edge embedding is used to reflect the relations between two nodes. The edge types in CommitHCGs involve four types of relations: CALL, CD, DD, and SIM. Since the edges of CALL, CD, and DD types are directed, we design the SIM type edges as bidirectional. Additionally, multiple edges may exist between two nodes. Therefore, the edge embedding is designed as a 4-dimensional binary vector, with each of the 4 bits indicating whether there exists any CALL, CD, DD, or SIM edge between the current two nodes, respectively. If two hunk nodes have CD and SIM relations, the edge embedding vector will be (0, 1, 0, 1).

4.2.2 CommitGNN Model (Step 5). Unlike conventional models focusing on matrix data, GNNs can directly capture complex node relations and dependencies in graph-structured data. In this section, we introduce our CommitGNN model to address the VFC detection problems. As illustrated in Figure 5, our model consists of three parts: graph input, graph learning, and graph classification.

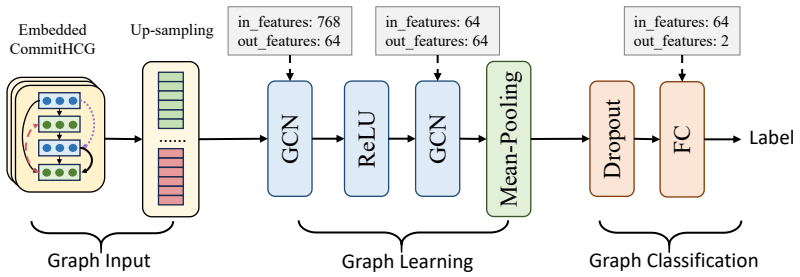


Fig. 5. The Design of CommitGNN

1) *Graph input.* The input graph to our model is CommitHCG ($\mathcal{G}$), consisting of nodes $\mathcal{V} = \{H_1, H_2, ..., H_n\}$ and edges $\mathcal{E}$. Node features are represented by a matrix with dimensions ($|\mathcal{V}|$, 768), where each node has a 768-dimensional feature vector generated by CodeBERT. Edge connections are denoted by an index matrix of size (2, $|\mathcal{E}|$), with each column representing an edge's connected nodes. Since it is a directed graph, the first index of the column indicates the source node, and the second index represents the target node. Edge attributes are captured in a matrix sized ($|\mathcal{E}|$, 4), providing a 4-dimensional feature vector for each edge. Here, $|\mathcal{V}|$ and $|\mathcal{E}|$ denote the total number of nodes and edges in $\mathcal{G}$, respectively.

For commits with only a single hunk that results in a graph with one node and no edges ($|\mathcal{V}| = 1$, $|\mathcal{E}| = 0$), the edge index matrix and edge attribute matrix will be empty, and the node feature matrix will contain one 768-dimensional vector representing the single hunk's features. To avoid the problem of the model tending to learn towards the majority class (non-vulnerability fixing commits), we up-sample the labeled graphs from the minority class until the graphs in both the training and validation sets are balanced across different classes. Only then do we begin the entire training process.

2) *Graph learning*. In this work, we choose GCNs [40] as the component of basic GNN convolutional layers. To learn complex graph structures and edge attributes better, we introduce an edge attention mechanism in GCN. Each layer of the graph convolutional network performs the following operations: weighted adjacency matrix calculation, edge attention coefficient calculation and feature aggregation. Specifically, for each edge type r, the model calculates the normalized adjacency matrix $\hat{A}_r$, then computes the attention coefficient α_{ij}^r for each edge from node i to j of type r, weighting the influence of node features. Finally, for each node i, we weight and aggregate the features of all neighboring nodes to produce a new feature representation.

We set two GCN layers and use ReLU [27] in the middle to help the model better capture information from different types of nodes and edges in heterogeneous graphs. Then, we employ mean-pooling to aggregate the node features and obtain the graph representation.

3) *Graph classification.* Following the graph learning phase, we leverage a dropout layer [63] as a regularization method to mitigate overfitting during model training. To determine if a commit is fixing-related, we employ a fully connected (FC) layer. This layer transforms the learned graph feature representation into a 2-channel output. Each channel corresponds to the probability of the commit being a non-VFC or VFC, respectively.

To address the class imbalance problem in VFC detection [76], we adopted a weighted loss function strategy. This approach assigns different weights to samples from different classes, making the model more attentive to the minority class (i.e., VFC). Specifically, the weighted cross-entropy loss function is defined as follows: $L_w = -\frac{1}{N}\sum_{i=1}^{N} w_{y_i}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$, where N is the total number of samples, y_i is the true label of the i-th sample, $\hat{y}_i$ is the model's predicted probability for the i-th sample, and w_{y_i} is the weight corresponding to the class of sample i. We set weights w_0 and w_1, typically with $w_0 + w_1 = 2$, and inversely proportional to the sample ratio. For example, if the ratio of positive to negative samples is 1:9, we can set $w_1 = 1.8$ and $w_0 = 0.2$.

5 EXPERIMENTAL EVALUATION

5.1 Implementation

Our implementation consists of approximately 2K lines of code in Python and Scala. To retrieve and preprocess pre-commit and post-commit code versions, we implement a Python script to analyze git diffs. This parser extracts relevant code changes and prepares them for further processing. To generate CPGs for both pre-commit and post-commit versions, we extend the Joern parser [9] using Scala scripts. The hunk correlation analysis, including caller-callee, data flow, and control dependencies extraction, was implemented using Python scripts. To detect code change pattern replication, we utilize the Levenshtein distance [73] to implement the similarity computation.

The core of our CommitGNN model was built using PyTorch 1.11.0, leveraging its efficient tensor computations and deep learning capabilities. We extend this with PyTorch Geometric 2.5.3 to support our graph-based learning approach. For code tokenization and embedding, we use the RobertaTokenizer [48] provided by the HuggingFace Transformers library, which is optimized for CodeBERT. Our weighted loss function is also implemented in PyTorch. Each hunk node is represented by a 768-dimensional feature vector, which is determined by the pre-defined hidden layer size of the CodeBERT [25]. The GNN architecture comprises two GCN layers interleaved with ReLU activation functions. Following the experimental setup in prior vulnerability-related research [64, 76], we set the learning rate to 1e-4 and employ the Adam optimizer for model training.

5.2 Experimental Setup

Research Questions. We design our evaluation to answer the following four research questions.

- **RQ4: Detection Effectiveness:** How is the effectiveness of FIXSEEKER in detecting VFCs, compared to current SOTA methods? (Section 5.3)
- **RQ5: Vulnerability Coverage:** How does FIXSEEKER perform in detecting different types of vulnerabilities? (Section 5.4)
- **RQ6: Feature Analysis:** What is the relative contribution of different correlation features to FIXSEEKER's detection accuracy? (Section 5.5)
- **RQ7: Computational Efficiency:** How efficiently does FIXSEEKER generate CommitHCGs, and how well does it scale across repositories of varying sizes and commit complexities? (Section 5.6)

Dataset. As the datasets used by previous works only contained Java and Python projects, we built a new dataset containing vulnerabilities across four PLs: C/C++, Java, Python, and PHP to answer the RQs. We additionally considered C/C++ and PHP because they contributed the largest portions of VFCs in our empirical study. These languages are well-supported by the Joern parser, ensuring high-quality CPG generation and analysis. In Section 3, we constructed the VFC dataset. Specifically, for each CVE in the NVD database, we identified GitHub commit URLs with a patch tag from the reference sections, and the project that corresponds to the commit was included in our dataset. To address the issue of duplicate patches that might be applied across different branches or versions of the same software, we performed a deduplication process. First, we remove commit-specific identifiers (hash values) within the diff code. Then, we converted the diff code into hashlib digests and used these digests to identify exact duplicates. Through the cleaning process, we removed 2.68% VFCs.

We initially considered the other commits from the development history of 2,094 open-source projects in the VFC dataset as potential non-VFC candidates. However, among these commits, some contain security-related keywords in their messages as identified by Zhou et al. [77], suggesting they might be silent VFCs that have not been explicitly documented. Since manually verifying the security nature of each suspicious commit would be impractical, we chose to exclude these commits from our non-VFC dataset to maintain dataset quality and avoid potential contamination from unconfirmed commits.

To evaluate our model from the perspectives of both balanced and imbalanced class classification, we randomly sampled from the non-VFC set to obtain a balanced dataset with a 1:1 ratio of VFC:non-VFC and an imbalanced dataset with a 1:50 ratio of VFC:non-VFC. The imbalanced ratio aligns with previous studies [47, 72] focused on vulnerability detection in highly skewed datasets. The balanced dataset assesses model performance under ideal conditions, and the imbalanced dataset reflects the natural distribution in real-world scenarios. The non-VFC set is sampled by project, using the proportion of project non-VFCs in the total number of non-VFCs as the sampling ratio. After constructing the dataset, we split it into training, validation, and test sets following the standard approach. We first divide the dataset into a training+validation set (80%) and a test set (20%). Then, we further split the training and validation sets at an 80%/20% ratio. We note that the training and validation sets undergo up-sampling using the Synthetic Minority Over-sampling Technique (SMOTE) [19] to reduce the imbalanced nature of the data. Specifically, SMOTE generates synthetic samples for the minority class (VFCs) by interpolating between existing VFC samples and their k-nearest neighbors rather than simply duplicating existing samples. The detailed number of the datasets is shown in Table 3.

Figure 6 presents the CDF and PDF of the lines of code (LOC) changed per commit for both VFCs and non-VFCs. As shown in the CDF, VFCs are generally smaller in scale: in the medium range (10–100 LOC), the VFC curve lies to the left of the non-VFC curve, indicating that vulnerability fixes tend to reach lower LOC thresholds earlier. In contrast, extremely large commits (>500 LOC) are more common in non-VFCs. The PDF further confirms this trend. VFCs are slightly more

Table 3. The Distribution of Balanced and Imbalanced Dataset

	Balanced				Imbalanced			
	C/C++	Java	Python	PHP	C/C++	Java	Python	PHP
#VFC	5,139	735	1,131	2,978	5,139	735	1,131	2,978
#non-VFC	5,139	735	1,131	2,978	256,950	36,750	56,550	148,900

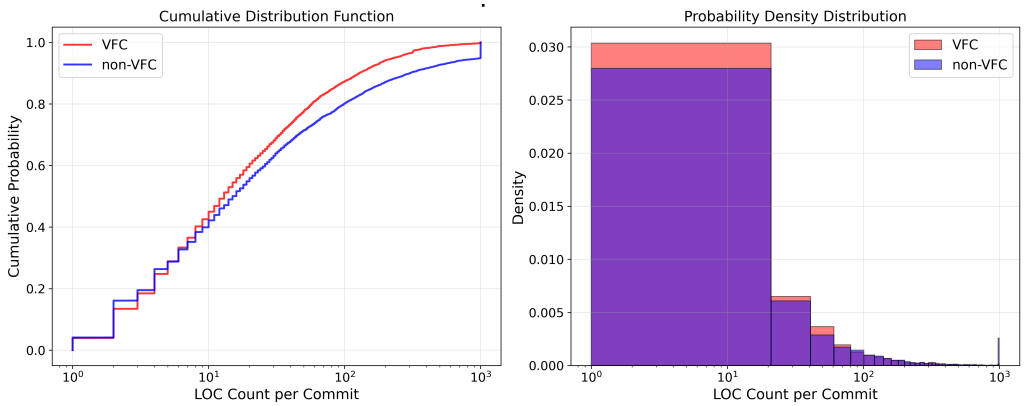


Fig. 6. Distribution of Lines of Code per Commit for VFCs and Non-VFCs

concentrated in small changes (1–10 LOC), reflecting their localized nature, while non-VFCs show higher probability in large modifications (>100 LOC), consistent with feature development or large-scale refactoring. Overall, the results suggest that VFCs are typically more focused and precise than non-VFCs.

However, it is important to observe the substantial overlap between the two distributions. The PDF shows that while VFCs have a slightly higher concentration in the 1–10 LOC range, a vast majority of non-VFCs also fall within this same scale. In the balanced dataset environment, a random forest classifier trained solely on the LOC feature achieved an F1-score of only 0.43. The result demonstrates that using LOC as a standalone feature is insufficient for predicting VFCs, necessitating the use of deeper semantic and structural analysis as provided by FIXSEEKER.

Metrics. For both balanced and imbalanced datasets, we report $Precision = \frac{|identified_v \cap correct_v|}{|identified_v|}$ and $Recall = \frac{|identified_v \cap correct_v|}{|correct_v|}$, where $|identified_v \cap correct_v|$ represents the number of VFCs correctly identified by the model, $|identified_v|$ represents the total number of commits identified as VFCs by the model and $|correct_v|$ represents the total number of actual VFCs. In the balanced dataset, we use $F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$ to measure the trade-off between correctly identifying VFCs and avoiding false positives. In the imbalanced dataset, in addition to the above two metrics, we use four metrics: F2-score, AUC-ROC, AUC-PR, and effort-aware metrics CostEffort@L to measure the effectiveness of VFC detection, which are used by SOTA works [53, 68, 76]. $F2 = (1 + 2^2) \times \frac{Precision \times Recall}{2^2 \times Precision + Recall}$, it weights recall higher than precision, which is appropriate for security contexts where missing vulnerability fixes (false negatives) is generally more costly than false alarms [60].

AUC-ROC is an area under the Receiver Operating Characteristic (ROC) curve [29]. It is a metric for binary classification models, representing the ability to distinguish between classes across all possible thresholds. The score is mathematically defined as: AUC-ROC $= \frac{\sum_{i \in \text{positiveClass}} \text{rank}_i - \frac{M(1+M)}{2}}{M \times N}$,

where rank_i is the rank of the i-th positive sample in the ranked list of predictions, M is the number of positive examples, and N is the number of negative samples. AUC-ROC values range from 0 to 1, with 0.5 representing random guessing and 1 indicating perfect classification. AUC benefits from its insensitivity to class imbalance and its ability to provide a comprehensive performance measure without requiring a specific classification threshold.

AUC-PR [22] is an area under the Precision-Recall (PR) curve. It is a metric for binary classification models, particularly useful for imbalanced datasets, representing the trade-off between precision and recall across various classification thresholds. The AUC-PR score is mathematically defined as: AUC-PR $= \int_0^1 P(R)\, dR$, where P is precision, R is recall, and the integral is computed over all possible recall values. AUC-PR values range from 0 to 1, with higher values indicating better performance. Unlike AUC-ROC, AUC-PR focuses on the performance of the positive class and is not affected by the large number of true negatives, making it more informative in identifying positive instances accurately.

CostEffort@L measures the proportion of VFCs that the model can detect when inspecting L% of the total lines of code. First, commits are ranked from high to low based on the probabilities predicted by the model, then the number of actual VFCs within the top L% of code lines is counted. This metric can be represented as: (number of VFCs detected in the top L% of code lines) / (total number of VFCs). The higher the value of CostEffort@L, the better the model's performance. We considered CostEffort under multiple L values (i.e., 5%and 20%) to evaluate the model's performance under different workload constraints, which are standard benchmarks in the field of software engineering for assessing bug prediction and code review effort [37, 38].

Baseline. We compare our method with the following five baselines:

(1) **PatchRNN** [68] is a deep learning system for identifying security patches, using both commit messages and diff code as features, employing TextRNN [41] for commit messages and a twin RNN for diff code. Because our task scenario only considers the code changes in commits, we remove the commit messages feature extraction and training from PatchRNN.
(2) **VFFINDER** [51] is a graph-based approach that captures code structural changes using annotated ASTs for silent vulnerability fix identification. It utilizes a graph attention network (GAT) to extract features from ASTs and distinguish VFC from non-VFC. However, VFFINDER is currently limited to C/C++ projects.
(3) **VulFixMiner** [76] is a Transformer-based model for detecting silent vulnerability fixes in code commits. It uses CodeBERT [25] to extract semantic features from file-level code changes, then aggregates them to the commit level.
(4) **Midas** [53] is a multi-granularity deep learning model for detecting VFCs. It extracts features at commit, file, hunk, and line levels, using CodeBERT for code change representation and various neural networks for feature extraction. A neural classifier makes the final prediction.
(5) **GRAPE** [28] is a graph-based patch representation approach that identifies and assesses silent vulnerability fixes by merging CPGs of vulnerable and fixed programs, and using a graph convolutional neural network (GCN) to capture structural information.

5.3 Detection Effectiveness (RQ4)

5.3.1 Overall Results. As illustrated in Table 4, the FIXSEEKER system demonstrates strong performance across multiple PLs. On the balanced datasets, FIXSEEKER achieves the highest F1-score for C/C++ (0.818), Java (0.813), Python (0.811), and PHP (0.829). FIXSEEKER maintains robust performance on the imbalanced datasets with F2 scores of 0.708, 0.759, 0.735, and 0.749 for C/C++, Java, Python, and PHP, respectively. These F scores, though lower than those on balanced datasets due to the inherent challenge of class imbalance, still outperform baseline approaches. While maintaining

Table 4. Performance of FIXSEEKER and Baselines on Different Programming Language Projects

PL	Method	Balanced Dataset			Imbalanced Dataset						
		Pr	Re	F1	Pr	Re	F2	AUC-ROC	AUC-PR	CostEffort @5	CostEffort @20
C/C++	PatchRNN	0.698	0.818	0.753	0.325	0.684	0.560	0.736	0.326	0.318	0.648
	VFFINDER	0.766	0.759	0.762	0.331	0.654	0.547	0.725	0.319	0.338	0.624
	VulFixMiner	0.783	0.752	0.767	0.329	0.824	0.633	0.759	0.516	0.575	0.730
	Midas	0.769	**0.855**	0.810	0.347	0.837	0.653	0.804	0.586	0.525	0.818
	GRAPE	0.784	0.826	0.804	0.362	0.786	0.637	0.781	0.596	0.505	0.753
	FIXSEEKER	**0.794**	0.842	**0.818**	**0.419**	**0.856**	**0.708**	**0.833**	**0.624**	**0.617**	**0.830**
Java	PatchRNN	0.768	0.754	0.761	0.274	0.614	0.492	0.602	0.412	0.357	0.654
	VulFixMiner	**0.815**	0.779	0.797	0.693	0.406	0.443	0.757	0.610	0.472	0.780
	Midas	0.787	0.811	0.799	0.436	0.704	0.627	0.788	0.594	0.459	0.783
	GRAPE	0.794	0.809	0.801	0.458	0.703	0.635	0.759	0.567	0.477	0.764
	FIXSEEKER	0.796	**0.830**	**0.813**	**0.767**	**0.757**	**0.759**	**0.818**	**0.630**	**0.505**	**0.835**
Python	PatchRNN	0.702	0.788	0.743	0.219	0.731	0.498	0.675	0.289	0.278	0.565
	VulFixMiner	0.711	0.853	0.776	**0.680**	0.709	0.703	0.795	0.525	0.472	0.787
	Midas	0.734	**0.897**	0.807	0.640	0.726	0.707	0.804	0.572	0.441	0.805
	GRAPE	0.713	0.892	0.793	0.489	0.761	0.685	0.813	0.599	0.372	0.724
	FIXSEEKER	**0.741**	0.895	**0.811**	0.622	**0.770**	**0.735**	**0.851**	**0.672**	**0.528**	**0.863**
PHP	PatchRNN	0.625	0.731	0.674	**0.676**	0.559	0.579	0.785	0.628	0.341	0.578
	VulFixMiner	0.676	0.806	0.735	0.589	0.714	0.685	0.735	0.520	0.503	0.723
	Midas	0.745	0.871	0.803	0.473	0.767	0.682	0.748	0.566	**0.596**	0.779
	GRAPE	0.722	0.851	0.781	0.637	0.731	0.710	0.737	0.554	0.417	0.729
	FIXSEEKER	**0.767**	**0.903**	**0.829**	0.566	**0.815**	**0.749**	**0.809**	**0.634**	0.585	**0.826**
Overall	PatchRNN	0.698	0.773	0.733	0.374	0.647	0.532	0.700	0.414	0.324	0.611
	VulFixMiner	0.746	0.798	0.769	0.573	0.663	0.616	0.762	0.543	0.506	0.755
	Midas	0.759	0.859	0.805	0.474	0.759	0.667	0.786	0.580	0.505	0.796
	GRAPE	0.753	0.845	0.795	0.487	0.745	0.667	0.773	0.579	0.443	0.743
	FIXSEEKER	**0.775**	**0.868**	**0.818**	**0.593**	**0.800**	**0.738**	**0.828**	**0.640**	**0.559**	**0.839**

high recall values, FIXSEEKER achieves reasonable precision values (0.419, 0.767, 0.622, and 0.566) in the imbalanced dataset. This indicates that our approach does not achieve high recall by simply predicting most cases as positive, but rather learns patterns of vulnerability fixes. Java shows the most balanced performance, and C/C++ prioritizes recall over precision, which may be appropriate given the critical nature of memory-related vulnerabilities common in C/C++.

Additionally, FIXSEEKER consistently outperforms other methods in terms of AUC-ROC and AUC-PR metrics. For instance, in the Python dataset, FIXSEEKER achieves an AUC-ROC of 0.851 and an AUC-PR of 0.672. It is important to note that the performances on balanced and imbalanced datasets are not directly comparable due to their different data distributions, reflecting various real-world scenarios of vulnerability detection tasks.

5.3.2 Comparison with RNN and Graph-based Approaches. We compare FIXSEEKER with PatchRNN and VFFINDER on our ground-truth dataset by applying the same training and test set splitting. Because VFFINDER is limited to C/C++ projects, we only compare the results of VFFINDER and FIXSEEKER in the C/C++ dataset in Table 4. FIXSEEKER significantly outperforms PatchRNN with 11.6% higher F1 in the balanced dataset and 38.7% higher F2, 18.3% higher AUC-ROC in the imbalanced dataset on average. The main advantage of FIXSEEKER over PatchRNN stems from its

ability to capture complex hunk relationships through graph structures. While PatchRNN processes code changes as sequential data, it struggles to model the intricate dependencies between different hunks, such as caller-callee relationships and data flow dependencies that are naturally represented in our graph structure.

Although both Fixseeker and VFFINDER utilize GNNs, Fixseeker demonstrates superior performance with 7.3% higher F1-score and 29.4% higher F2-score in the balanced and imbalanced datasets, respectively. This improvement can be attributed to two key factors: (1) Feature extraction: While VFFINDER focuses on structural changes at the AST level, Fixseeker extracts richer features by analyzing the relationships between different hunks, capturing both syntactic and semantic dependencies that are crucial for vulnerability fix identification. (2) Network architecture: Our edge-aware attention mechanism in GCNs shows better performance than VFFINDER's GAT architecture in capturing hunk-level correlations. This design allows the model to learn different attention weights for different types of hunk relationships during feature aggregation. In contrast, VFFINDER's GAT treats all structural relationships uniformly when computing attention.

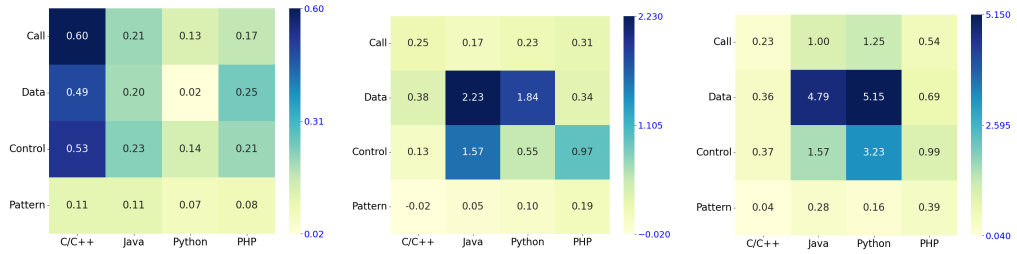
5.3.3 Comparison with SOTA VFC Detection Approaches. We compare Fixseeker with baseline works over datasets for C/C++, Java, Python, and PHP by applying the same training and test set splitting. The experimental results are summarized in Table 4. On the balanced dataset, Fixseeker outperforms other approaches in most metrics across different PLs. For the C/C++ and Python data, Fixseeker achieves the highest F1 and precision, though Midas has the highest recall. On the imbalanced dataset, Fixseeker demonstrates consistently superior performance across all evaluation metrics. Specifically, Fixseeker improves the F2-score by 10.6% to 19.8% compared to the SOTA VFC detection methods. For example, on the Java dataset, Fixseeker achieves an F2 of 0.759, outperforming Midas (0.627) and GRAPE (0.635). While Midas shows competitive performance in AUC-ROC, Fixseeker maintains a notable advantage in AUC-PR scores, improving Midas by 10.34% on average across four languages.

Due to the extreme imbalance between non-VFC and VFC in OSS (VFCs typically account for a small percentage), false positives matter more than false negatives. Therefore, our method aims to reduce the false positive rate while maintaining a high recall. The consistently high F2-scores of Fixseeker across languages (e.g., 0.759 for Java, 0.749 for PHP) demonstrate its effectiveness in handling this imbalance, outperforming other approaches.

5.3.4 Effort-aware Performance. In real-world scenarios, it is crucial to detect VFCs efficiently with minimal inspection effort. To evaluate this aspect, we use CostEffort@5 and CostEffort@20 metrics, which measure the percentage of VFCs detected when inspecting 5% and 20% of the total lines of code, respectively. When performing Fixseeker on the imbalanced datasets, Table 4 shows high CostEffort@5 and CostEffort@20 scores across languages, indicating that Fixseeker can detect a high percentage of VFCs with minimal inspection effort. For example, in C/C++, Fixseeker achieves a CostEffort@5 of 0.617, CostEffort@20 of 0.830, outperforming the SOTA works.

5.4 Vulnerability Coverage (RQ5)

As a large-scale real-world dataset, our dataset allows us to evaluate system performance across different vulnerability types. Across the VFCs of four languages, we identify their Top 3 most common CWEs separately, ensuring to capture the most prevalent vulnerability types in different languages. Table 5 lists the vulnerability types and their corresponding F2-score of Fixseeker. F2-score assesses Fixseeker's performance on different vulnerability types when working with imbalanced data. Through analyzing the results for each type, we draw two important conclusions:



(a) F1-Score Reduction on Balanced Dataset (%)

(b) AUC-ROC Reduction on Imbalanced Dataset (%)

(c) AUC-PR Reduction on Imbalanced Dataset (%)

Fig. 7. Correlation Importance Across Programming Languages: Performance Degradation when Removed (%)

1) FIXSEEKER excels at detecting certain types of vulnerability. This variation reflects the uniqueness of code patterns in different types of vulnerability fixes. For example: Memory-related vulnerabilities (i.e., CWE-119, CWE-125, and CWE-787) generally have high detection rates, with F2 exceeding 0.9. This might be due to these vulnerabilities often involving explicit memory operations and boundary check patterns that our model can easily recognize. Path Traversal (CWE-22) issue, despite having fewer samples, still shows good detection results with F2 of 0.843. This indicates our model's ability to generalize well to vulnerabilities with specific patterns.

2) Some vulnerability types show relatively lower detection performance, reflecting the complexity of their fix patterns. Specifically: Cross-site scripting (CWE-79) and improper input validation (CWE-20) have relatively lower F2 score of 0.713 and 0.714, respectively. This may be because fixes for these vulnerabilities often involve diverse context handling and complex validation logic, making pattern recognition more difficult. For example, a cross-site scripting fix might require understanding both the input sanitization code and how the sanitized data is used across different functions and files, while an input validation fix often involves multiple validation checks distributed across different program paths. Future work could explore incorporating program analysis techniques that can better track such cross-function and cross-file dependencies.

Table 5. Performance of FIXSEEKER over Different Vulnerability Types

CWE ID	Description	Language (Rank)	F2
CWE-125	Out-of-bounds Read	C/C++ (*Top1*)	0.947
CWE-79	Cross-Site Scripting	Java (*Top1*), PHP (*Top1*), Python (*Top3*)	0.713
CWE-22	Path Traversal	Python (*Top1*), Java (*Top2*), PHP (*Top3*)	0.843
CWE-787	Out-of-bounds Write	C/C++ (*Top2*)	0.947
CWE-20	Improper Input Validation	Python (*Top2*)	0.714
CWE-89	SQL Injection	PHP (*Top2*)	0.743
CWE-119	Memory Buffer Restriction	C/C++ (*Top3*)	0.925
CWE-611	XML External Entity Restriction	Java (*Top3*)	0.753

5.5 Feature Analysis (RQ6)

5.5.1 Ablation Study (all correlations). To understand the contribution of edge features to FIXSEEKER's performance, we conducted an ablation study comparing models with and without these correlation edges. Table 6 presents the results of this experiment, showing both the performance metrics with and without edge features. This analysis was performed in four programming languages using

Table 6. Impact of Edge Features on VFC Detection Performance

PL	Config	Balanced Dataset			Imbalanced Dataset			
		Pr	Re	F1	Re	F2	AUC-ROC	AUC-PR
C/C++	w *edge feature*	0.794	0.842	0.818	0.856	0.708	0.833	0.624
	w/o *edge feature*	0.621 (↓)	0.874 (↑)	0.726 (↓)	0.789 (↓)	0.650 (↓)	0.712 (↓)	0.413 (↓)
Java	w *edge feature*	0.796	0.830	0.813	0.757	0.759	0.818	0.630
	w/o *edge feature*	0.614 (↓)	0.786 (↓)	0.689 (↓)	0.656 (↓)	0.639 (↓)	0.680 (↓)	0.410 (↓)
Python	w *edge feature*	0.741	0.895	0.811	0.770	0.735	0.851	0.672
	w/o *edge feature*	0.619 (↓)	0.882 (↓)	0.729 (↓)	0.722 (↓)	0.688 (↓)	0.719 (↓)	0.440 (↓)
PHP	w *edge feature*	0.767	0.903	0.829	0.815	0.749	0.809	0.634
	w/o *edge feature*	0.641 (↓)	0.895 (↓)	0.745 (↓)	0.759 (↓)	0.655 (↓)	0.694 (↓)	0.466 (↓)

balanced and unbalanced datasets to ensure robust conclusions. For the balanced dataset, removing edge features caused F1-score reductions ranging from 10.1% to 15.3% across different languages. The impact was even more pronounced in the challenging scenario of an imbalanced data set, where the AUC-PR values decreased substantially, from 26.5% to 34.9%. Particularly notable is the effect on Java's F2-score, which decreased by 15.8% when edge features were removed, highlighting the importance of capturing inter-hunk correlations for accurate vulnerability fix detection in this language.

5.5.2 Ablation Study (each correlation). Furthermore, to thoroughly understand the contribution of each correlation type to FIXSEEKER's performance, we conducted an ablation study by removing one type of correlation edge at a time from the CommitHCG and measuring the resulting performance degradation across key metrics. The importance of each correlation type is quantified as the percentage decrease in performance when that correlation is removed:

$$Importance_{type} = (Metric_{full} - Metric_{ablated})/Metric_{full} * 100\%$$

Figure 7 presents our findings across four programming languages (C/C++, Java, Python, and PHP) for three key metrics: F1-score on the balanced dataset, AUC-ROC, and AUC-PR on the imbalanced dataset. The results reveal several insights. The Caller-Callee Dependency shows the highest importance for C/C++, with a 0.60% reduction in the F1-score when removed, which corresponds to the fact that Caller-Callee Dependency appears more frequently in C/C++ VFCs than in non-VFCs (37.3% vs 29.0% in our balanced dataset). Data Flow Dependency demonstrates particularly strong importance in Java and Python, with performance reductions of 2.23% and 1.84% in AUC-ROC, and even more reductions of 4.79% and 5.15% in AUC-PR, respectively. The result corresponds to the fact that data flow dependencies appear more frequently in Java and Python VFCs than in other languages. Control Dependency exhibits moderate importance across all languages, with a notable impact on Python's AUC-PR (3.23% reduction). Pattern Replication generally shows the lowest importance among the four correlation types, though it still contributes positively to the model's performance in most cases.

5.6 Computational Efficiency (RQ7)

As illustrated in Table 7, we evaluated the overhead of CommitHCG generation across eight repositories. The size of these repositories ranges from 34MB to 1.5GB. The average number of nodes in the generated CommitHCGs ranges from 2.79 to 18.43, while the average number of edges varies from 2.21 to 28.62. This suggests that the complexity of the generated graphs is manageable

and appropriate for our model. The main overhead of our approach is in constructing CommitHCGs for given commits. The last column shows that the average time to generate a CommitHCG ranges from 16.63s to 24.39s across different repositories. Notably, the generation time does not directly correlate with repository size or number of commits but appears more related to the complexity of the changes (as indicated by the number of hunks and average nodes/edges).

For large-scale projects such as the Linux Kernel [1] and TensorFlow [4], which contain millions of lines of code and very long commit histories, FIXSEEKER demonstrates comparable overheads with average processing times of 21.80s and 21.58s per commit, respectively. This result highlights that the scalability of FIXSEEKER is not hindered by overall repository size, but rather depends on the inherent complexity of code changes. In practice, the overhead can be further mitigated through parallelization or distributed deployment, making FIXSEEKER feasible for real-world, large-scale projects.

Table 7. Overhead of CommitHCG Generation

Repo	Size	#Commit	#Hunk	Avg. Node	Avg. Edge	Avg. Time
ImageMagick	213MB	179	699	3.91	5.22	24.39s
Gpac	199MB	137	533	3.89	4.72	16.63s
FFmpeg	516MB	121	337	2.79	2.21	16.92s
Tcpdump	34MB	117	1014	8.67	28.62	23.06s
CPython	646MB	87	890	9.18	5.52	19.67s
XWiki	784MB	80	1474	18.43	8.69	20.58s
Linux Kernel	~1.2GB	358	5442	15.20	10.47	21.80s
TensorFlow	~1.5GB	233	3217	13.81	11.64	21.58s

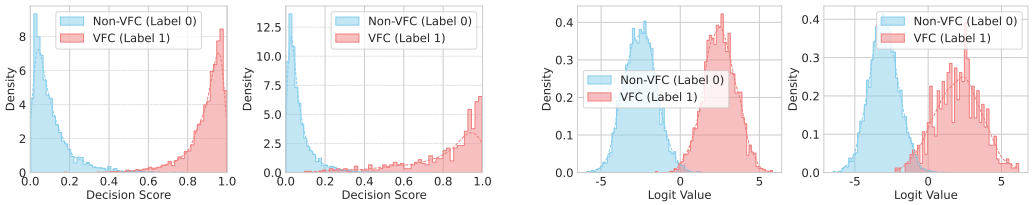
6 DISCUSSION

6.1 Practicality

Due to the prevalence of silent vulnerability fixes in open-source software, traditional methods relying on commit messages or explicit security labeling may fail to identify critical security patches. FIXSEEKER addresses this challenge by focusing solely on code diffs and leveraging advanced hunk correlation analysis. This approach allows FIXSEEKER to outperform existing methods, particularly in detecting complex, multi-file vulnerability fixes that may not be apparent through surface-level analysis. FIXSEEKER's language-agnostic nature, focusing on structural and semantic code changes rather than language-specific features, makes it applicable across various programming languages. This versatility allows security teams and developers to maintain consistent security practices across diverse technology stacks. Additionally, FIXSEEKER can prioritize code reviews for potentially security-relevant commits, reducing vulnerability windows.

6.2 Analysis of Performance on Imbalanced Datasets

Our evaluation indicates that the performance improvement of FIXSEEKER over baselines is larger on imbalanced datasets compared to balanced ones. As shown in Table 4, FIXSEEKER outperforms the best baseline by 1.6% in F1-score on the balanced dataset, whereas the gain increases substantially to 10.6% in F2-score and 10.3% in AUC-PR on the imbalanced dataset. To understand this phenomenon, we analyze the results through three perspectives: theoretical limitations of baselines, quantitative internal decision boundaries, and qualitative case studies.



(a) Decision Score Distribution Histograms (Left: Balanced Data; Right: Imbalanced Data)

(b) Logit Value Distribution Histograms (Left: Balanced Data; Right: Imbalanced Data)

Fig. 8. Visualization of Model Discriminative Power and Calibration

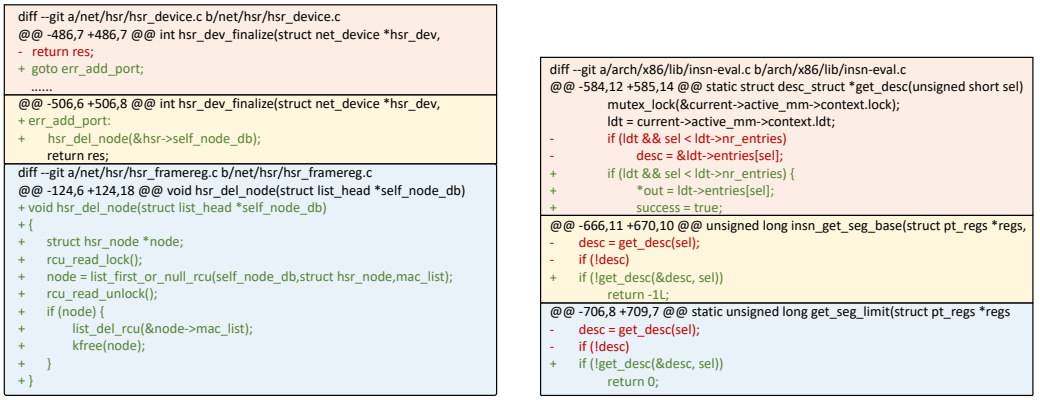
6.2.1 Theoretical Implications. Baseline approaches, such as VulFixMiner and Midas, treat code hunks primarily as independent or sequential units relying on aggregated representations. When the positive class (VFCs) is underrepresented, these methods often fail to capture sparse but critical inter-hunk dependencies that characterize rare vulnerability-fixing patterns. Consequently, the learned features tend to be dominated by the majority class (non-VFCs), reducing discriminative power in high-skew scenarios.

In contrast, the edge attention mechanism in FIXSEEKER explicitly models multi-relational relationships between hunks. This design enables the model to amplify structural signals that are otherwise buried in noise when positive samples are sparse. Our ablation study supports this: removing edge features results in a larger performance drop on imbalanced datasets (AUC-PR decreases by 26.5%–34.9%) compared to balanced datasets (F1 decreases by 10.1%–15.3%). This suggests that the core advantage of FIXSEEKER lies in its ability to reconstruct the global logic of a fix through graph structures, providing superior resilience against skewed data distributions.

6.2.2 Quantitative Evidence. To validate the theoretical analysis, we examined the internal decision boundaries of the model. Standard classifiers trained on imbalanced data often exhibit logits collapsing toward the decision boundary due to majority class bias [45]. As illustrated in Figure 8a, FIXSEEKER maintains separability between VFC (Label 1) and Non-VFC (Label 0) samples. In the imbalanced scenario, while the density peak for Non-VFC samples is higher, the minority class scores concentrate near 1.0. The Logit Distribution (Figure 8b) confirms this separation; the mean Logit for the minority class remains distinct from the zero boundary. This indicates that the model resists the prediction bias often observed in baseline approaches and maintains confidence calibration despite class imbalance.

6.2.3 Qualitative Analysis. To illustrate FIXSEEKER 's advantages, we examine VFC cases detected solely by our model. We highlight two representative scenarios:

1) Multi-file correlations. FIXSEEKER specifically analyzes the correlations between hunks, making it more adept at detecting related fixes across multiple files. The commit in Figure 9a shows changes in two files `hsr_device.c` and `hsr_framereg.c`. In `hsr_device.c`, there is a modification in the error handling logic, replacing a simple return res; with goto `err_add_port;`. This change is correlated with the addition of the `hsr_del_node()` function call. The `hsr_del_node()` function is then implemented in `hsr_framereg.c`, where it's defined with proper error handling and memory management. Fixseeker's ability to analyze relationships between these hunks across different files allows it to recognize this as a coordinated fix for a potential resource leak or error-handling vulnerability. This is often missed by approaches that only consider adjacent hunks or line-level change features.



(a) VFC of CVE-2019-16995 (b) VFC of CVE-2019-13233

Fig. 9. The Cases only Detected by FIXSEEKER

2) Synchronized changes across multiple functions. This involves simultaneously modifying the definition of an interface function and multiple places where this function is called. The changes shown in Figure 9b involve the modification of the `get_desc` function and its usage in two other functions. In the `get_desc` function, the logic is altered to use a pointer out and a success flag, potentially improving error handling. Correspondingly, the calls to `get_desc` in `insn_get_seg_base` and `get_seg_limit` functions are modified to adapt to this new interface. The new version passes &desc as an argument and checks the return value of the function. This coordinated change across multiple functions demonstrates a consistent modification pattern that addresses a potential vulnerability in how descriptor information is retrieved and handled. FIXSEEKER's ability to analyze similar code change patterns between different hunks allows it to recognize this as a coordinated fix, which is overlooked by baselines.

6.3 Vulnerability Repair and General Bug Repair

While both vulnerability fixes and general bug fixes aim to correct defects in software code, our research highlights several key differences between these two repair tasks. Vulnerability fixes exhibit distinct patterns that warrant specialized approaches for their detection and analysis.

When comparing our VFC dataset of empirical study with the widely-used general bug repair benchmark Defects4J [36], we observe that only 40.9% of bug fixing commits (BFCs) involve multiple hunks, in stark contrast to over 70% of vulnerability fixes. More importantly, nearly half (47%) of multi-hunk BFCs lack any of the four correlation types that we identified, whereas these correlations are present in 92.7% of multi-hunk vulnerability fixes. Meanwhile, hybrid correlations account for only 9.2% in the Defects4J dataset, whereas in our VFC dataset, they account for 35.8% in C/C++ and 59.0% in Java, respectively. This suggests that vulnerability fixes typically require more interrelated changes across different code hunks—reflecting the complex nature of security vulnerabilities that often span multiple execution paths.

These structural differences are further corroborated by the limited effectiveness of general bug repair techniques when applied to vulnerabilities. For instance, state-of-the-art automated program repair tools like PraPR [26], which employ templates designed for general bugs, struggle to address real-world vulnerabilities due to the absence of security-specific templates [44]. Recent empirical studies have demonstrated that vulnerabilities differ fundamentally from general bugs in both root causes and repair requirements [17, 18]. The studies explain that while general bugs typically

involve violations of functional logic, vulnerabilities are often tied to specific CWEs, which require specialized security-critical ingredients missing from general APR templates. Meanwhile, APR4Vul found that existing APR tools only generate end-to-end test patches for about 20% of vulnerabilities. On average, about 73% of the end-to-end test patches generated eliminated vulnerabilities, but only 44% of patches fixed vulnerabilities while maintaining program functionality. This underscores the importance of specialized approaches for vulnerability detection and repair.

6.4 Limitation and Future Work

Dataset. As mentioned in the dataset description in Section 5, we used regular expressions to filter out many suspicious commits, as we could not definitively determine whether they were VFCs. The filtering process, while necessary for creating a reliable dataset, potentially excludes a significant number of actual VFCs. In future work, we plan to leverage manual verification to confirm whether these suspicious commits are vulnerability-related. This effort will help expand and enrich our dataset, potentially improving the model's ability to detect a wider range of vulnerability-fixing patterns.

Pattern Replication Detection. There are four clone types based on clone pair similarity [57, 74]. Our current approach to detecting pattern replications focuses primarily on Type-I (exact duplicates) and Type-II clones (syntactically similar with renamed variables), which our empirical study identified as the most common in vulnerability fixes. However, we acknowledge limitations in detecting Type-III clones (structurally similar with added or removed statements) and Type-IV clones (functionally similar but syntactically different). Type-III clones require more sophisticated structural matching algorithms that can handle partial similarities, while Type-IV clones would need semantic analysis beyond our current capabilities. Future work could explore integrating advanced clone detection techniques, such as LLM-based semantic similarity measures, to capture more complex pattern replications.

Isolating Security Fixes. While our empirical study found that mixed changes in VFCs are relatively rare (7 cases in 416 VFCs), completely separating security fixes from other changes can be challenging in some cases, particularly with large commits. To minimize the inclusion of mixed commits, we observed that *test* file changes are often included in commits to verify the effectiveness of vulnerability fixes. Since these changes don't represent the actual fixing patterns, we removed *test* files from our analysis. Meanwhile, we reviewed commit messages and filtered out commits containing terms like "refactor" or "refactoring" that might introduce non-security-related modifications. Moreover, in our dataset, each vulnerability fix commit is assigned to a specific CVE for tracing and validation. This data quality issue was also discussed by prior works [12, 59], and remains an open challenge in the field of vulnerability analysis. Currently, there is no definitive method for perfectly isolating security fixes from mixed-purpose commits. In future work, we plan to develop more sophisticated techniques to isolate security-relevant portions of commits, potentially using program slicing or more advanced semantic analysis to identify the minimal changes required to address a specific vulnerability.

Tool Efficiency. A significant portion of Fixseeker's processing time is spent on generating and analyzing CPGs using the static analysis tool, Joern parser. While this step is crucial for extracting rich semantic information from code changes, it can be time-consuming, especially for large commits. Our future work will focus on optimizing this process, possibly by exploring parallelizing CPG generation or developing more efficient static analysis methods tailored to vulnerability detection, ensuring Fixseeker can operate effectively in real-time within fast-paced development workflows.

7 RELATED WORK

Empirical Studies on Vulnerability Fixing. To investigate the correlations between hunks in VFCs, we conducted an empirical study using a large-scale dataset. Many research studies [15, 20, 33, 34, 42, 46, 50, 65, 71] have investigated vulnerability fixing (security patches) from other perspectives. Xu et al. [71] conducted an empirical study to understand the quality and characteristics of patches for OSS vulnerabilities in two industrial vulnerability databases. Li et al. [42] found that security fixes are smaller than non-security ones, vulnerabilities often exist for years, and a significant percentage of patches are flawed or incomplete. Emanuele et al. [33] examined vulnerabilities' introduction and removal and found that vulnerabilities often result from multiple commits and remain unfixed for over a year. Chinthanet et al. [20] examine delays in adopting security fixes in npm packages. It found that fixing releases often include unrelated changes. Tan et al. [65] found that 80% of CVE-Branch pairs are unpatched, posing significant risks. Patch porting is time-consuming, averaging 40.46 days. Some studies [66, 67] aim to investigate the secret security patch in open-source software. Wang et al. [67] developed a machine learning tool to distinguish security from non-security patches, analyzing three SSL libraries to highlight the need for better patch management. Wang et al. [66] analyzed over 4,700 known security patches, and discovered 12 secret patches in SSL libraries. Different from these, our work is the first to systematically study the correlations between different code hunks within vulnerability fixes. Our empirical findings motivate the design of FIXSEEKER.

Silent Vulnerability Fixing Detection. Some works have similar task objectives to FIXSEEKER that focus on detecting silent vulnerability fixes, which only consider code diffs [28, 53, 69, 75, 76]. VulFixMiner [76] uses a Transformer-based model, which automatically extracts semantic meaning from commit-level code changes to identify silent vulnerability fixes in open-source software. Midas [53] proposed a multigranularity model to detect VFCs in open-source software. It analyzes code changes at different levels and uses an ensemble approach, improving detection accuracy and efficiency. GRAPE [28] proposes an innovative graph-based representation that merges the CPGs to detect vulnerability fixing. EarlyVulnFix [69] automatically identifies silent taint-style vulnerability fixes by analyzing the relationship between newly introduced code and sinks through data flow graphs, covering both sanity check fixes and permission list fixes. CoLeFunDa [75] leverages a contrastive learner to learn function-level code change representations from augmented fix data, creating a pre-trained encoder that is subsequently fine-tuned for three downstream security tasks.

Patch Detection with the help of Textual Information. Many other approaches have been proposed to detect VFCs with the help of information in commit messages and security issues. Sabetta et al. [58] treat code changes as natural language documents, using document classification methods to identify security-relevant commits in open-source software repositories. Hermes [55] proposed a commit-issue link recovery technique to infer the potential missing link, incorporating the information from issue trackers to boost the VFC classifier. Vulcurator [52] is a preliminary work that combines commit messages, code changes and issue reports using deep learning for VFC classification. As its code change analysis component later evolved into VulFixMiner [76], we chose to compare our approach with the more mature VulFixMiner in our evaluation. VFCFinder [23] used NL-PL models to generate the top-five ranked set of VFCs for a given security advisory. PatchFinder [43] designs a model that takes the CVE description, commit messages, and code changes as input, then generates a ranked list of commits for the given CVE, indicating the likelihood of each commit being the relevant patch. Unlike previous works that rely heavily on commit messages and issue reports, or treat code changes as flat sequences or isolated units, our work is the first to model the complex relationships between different code hunks using a graph-based approach. By capturing these inter-hunk correlations through our CommitHCG

representation, we can better understand the scope and impact of vulnerability fixes, improving detection accuracy.

Vulnerability-related Task with Graph Representation. FIXSEEKER employs a GNN model to learn from a graph where hunks serve as nodes and correlations as edges, aiding in VFC classification. Many other methods [21, 30, 32, 54, 56] also use graph learning to address vulnerability-related tasks. LineVD [30] combined GNNs and transformers to analyze code dependencies and tokens for statement-level vulnerability detection in software. Qiu et al. [56] introduced a heterogeneous GNN framework for predicting specific vulnerability types in code. Chu et al. [21] proposed a novel counterfactual explainer for GNN-based vulnerability detection. It identifies minimal code graph changes that alter predictions, helping to understand and fix vulnerabilities. ReGVD [54] is a language-independent GNN model for vulnerability detection. It combines token sequences with pre-trained language models to analyze code vulnerabilities. Islam et al. [35] leverage a vulnerability graph that integrates sequential flow and data flow, then utilize GNNs with poacher flow edges to capture long-range dependencies in the code, enabling effective vulnerability classification.

8 CONCLUSION

In this work, we first conducted a large-scale empirical study to understand the characteristics of VFCs across six popular programming languages, revealing the prevalence and importance of multi-hunk correlations in VFCs. We then proposed FIXSEEKER, a graph-based approach to detect silent vulnerability fixes in open-source software. Our comprehensive evaluation has demonstrated the effectiveness, efficiency, and generality of FIXSEEKER across multiple programming languages and various vulnerability types. We have shown that FIXSEEKER outperforms state-of-the-art approaches on different metrics. The data and the implementation of FIXSEEKER are publicly available at https://github.com/Veronica-L/Fixseeker_main.

9 ACKNOWLEDGMENT

This research was supported by the National Natural Science Foundation of China (Grant No. 61702504), the National Research Foundation, Singapore, and the Smart Nation Group under the Smart Nation Group's Translational R&D Grant (Award No. TRANS2026-TGC01). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Natural Science Foundation of China, National Research Foundation, Singapore or the Smart Nation Group.

REFERENCES

[1] 1991. Linux Kernel - Home. https://www.kernel.org/.
[2] 2015. CVE-2015-5283. https://github.com/torvalds/linux/commit/8e2d61e0aed2b7c4ecb35844fe07e0b2b762dee4.
[3] 2015. CVE-2015-8543. https://github.com/torvalds/linux/commit/79462ad02e861803b3840cc782248c7359451cd9.
[4] 2015. Tensorflow - Home. https://github.com/tensorflow/tensorflow.
[5] 2017. cve-2017-1000251. https://nvd.nist.gov/vuln/detail/cve-2017-1000251.
[6] 2017. CVE-2017-7645. https://nvd.nist.gov/vuln/detail/CVE-2017-7645.
[7] 2018. CVE-2018-20784. https://github.com/torvalds/linux/commit/c40f7d74c741a907cfaeb73a7697081881c497d0.
[8] 2022. CVE-2022-3602. https://nvd.nist.gov/vuln/detail/CVE-2022-3602.
[9] 2024. Joern - Home. https://joern.io/.
[10] 2024. NVD - Home. https://nvd.nist.gov/.
[11] 2025. The Unified Format of Diff Hunk. https://en.wikipedia.org/wiki/Diff#Unified_format.
[12] Jafar Akhoundali, Sajad Rahim Nouri, Kristian Rietveld, and Olga Gadyatskaya. 2024. MoreFixes: A large-scale dataset of CVE fix commits mined through enhanced repository discovery. In Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering. 42–51.
[13] Dimah Al-Fraihat, Yousef Sharrab, Abdel-Rahman Al-Ghuwairi, Nour Sbaih, and Ayman Qahmash. 2024. Detecting refactoring type of software commit messages based on ensemble machine learning algorithms. Scientific Reports 14,

1 (2024), 21367.
[14] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. 2008. What's a typical commit? A characterization of open source software repositories. In 2008 16th IEEE international conference on program comprehension. IEEE, 182–191.
[15] Gábor Antal, Márton Keleti, and Péter Hegedűs. 2020. Exploring the security awareness of the python and javascript open source communities. In Proceedings of the 17th International Conference on Mining Software Repositories. 16–20.
[16] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In 2013 35th International Conference on Software Engineering (ICSE). IEEE, 712–721.
[17] Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. 2024. APR4Vul: an empirical study of automatic program repair techniques on real-world Java vulnerabilities. Empirical software engineering 29, 1 (2024), 18.
[18] Gerardo Canfora, Andrea Di Sorbo, Sara Forootani, Matias Martinez, and Corrado A Visaggio. 2022. Patchworking: Exploring the code changes induced by vulnerability fixing activities. Information and Software Technology 142 (2022), 106745.
[19] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. Journal of artificial intelligence research 16 (2002), 321–357.
[20] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. Empirical Software Engineering 26 (2021), 1–28.
[21] Zhaoyang Chu, Yao Wan, Qian Li, Yang Wu, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2024. Graph Neural Networks for Vulnerability Detection: A Counterfactual Explanation. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 389–401.
[22] Jesse Davis and Mark Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In Proceedings of the 23rd international conference on Machine learning. 233–240.
[23] Trevor Dunlap, Elizabeth Lin, William Enck, and Bradley Reaves. 2024. VFCFinder: Pairing Security Advisories and Patches. In Proceedings of the 19th ACM Asia Conference on Computer and Communications Security. 1128–1142.
[24] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2019. Confusion in code reviews: Reasons, impacts, and coping strategies. In 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). IEEE, 49–60.
[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020). Association for Computational Linguistics, 1536–1547.
[26] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 19–30.
[27] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In Proceedings of the fourteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings, 315–323.
[28] Mei Han, Lulu Wang, Jianming Chang, Bixin Li, and Chunguang Zhang. 2024. Learning Graph-based Patch Representations for Identifying and Assessing Silent Vulnerability Fixes. In 2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 120–131.
[29] James A Hanley and Barbara J McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology 143, 1 (1982), 29–36.
[30] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In Proceedings of the 19th international conference on mining software repositories. 596–607.
[31] Allend D Householder, Garret Wassermann, Ant Manion, and Chris King. 2017. The CERT R Guide to Coordinated Vulnerability Disclosure. Special Report. Technical Report. CMU/SEI-2017-SR-022, CERT Division, Carnegie Mellon University. Available ....
[32] Yutao Hu, Suyuan Wang, Wenke Li, Junru Peng, Yueming Wu, Deqing Zou, and Hai Jin. 2023. Interpreters for GNN-based vulnerability detection: Are we there yet?. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 1407–1419.
[33] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. The secret life of software vulnerabilities: A large-scale empirical study. IEEE Transactions on Software Engineering 49, 1 (2022), 44–63.
[34] Ivana Clairine Irsan, Ratnadira Widyasari, Ting Zhang, Huihui Huang, Ferdian Thung, Yikun Li, Lwin Khin Shar, Eng Lieh Ouh, Hong Jin Kang, and David Lo. 2026. Revisiting Vulnerability Patch Identification on Data in the Wild.

arXiv preprint arXiv:2603.17266 (2026).
[35] Nafis Tanveer Islam, Gonzalo De La Torre Parra, Dylan Manuel, Elias Bou-Harb, and Peyman Najafirad. 2023. An unbiased transformer source code learning with semantic vulnerability graph. In 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P). IEEE, 144–159.
[36] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 international symposium on software testing and analysis. 437–440.
[37] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E Hassan. 2010. Revisiting common bug prediction findings using effort-aware models. In 2010 IEEE international conference on software maintenance. IEEE, 1–10.
[38] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering 39, 6 (2012), 757–773.
[39] Oscar Karnalim, Setia Budi, Hapnes Toba, and Mike Joy. 2019. Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation. Informatics in Education 18, 2 (2019), 321–344.
[40] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.
[41] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent convolutional neural networks for text classification. In Proceedings of the AAAI conference on artificial intelligence, Vol. 29.
[42] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2201–2215.
[43] Kaixuan Li, Jian Zhang, Sen Chen, Han Liu, Yang Liu, and Yixiang Chen. 2024. 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. 590–602.
[44] Bo Lin, Shangwen Wang, Liqian Chen, and Xiaoguang Mao. 2025. There are More Fish in the Sea: Automated Vulnerability Repair via Binary Templates. Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (2025), 1–11.
[45] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In Proceedings of the IEEE international conference on computer vision. 2980–2988.
[46] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 1547–1559.
[47] Shigang Liu, Guanjun Lin, Qing-Long Han, Sheng Wen, Jun Zhang, and Yang Xiang. 2019. DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection. IEEE Transactions on Fuzzy Systems 28, 7 (2019), 1329–1343.
[48] Yinhan Liu. 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 (2019).
[49] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The attack of the clones: A study of the impact of shared code on vulnerability patching. In 2015 IEEE symposium on security and privacy. IEEE, 692–708.
[50] Huu Hung Nguyen, Duc Manh Tran, Yiran Cheng, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Shar Lwin Khin, Ouh Eng Lieh, Ting Zhang, and David Lo. 2025. Mapping nvd records to their vfcs: How hard is it? arXiv preprint arXiv:2506.09702 (2025).
[51] Son Nguyen, Thanh Trong Vu, and Hieu Dinh Vo. 2023. VFFINDER: A Graph-based Approach for Automated Silent Vulnerability-Fix Identification. In 2023 15th International Conference on Knowledge and Systems Engineering (KSE). IEEE, 1–6.
[52] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Xuan-Bach D Le, and David Lo. 2022. 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. 1726–1730.
[53] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E Hassan, et al. 2023. Multi-granularity detector for vulnerability fixes. IEEE Transactions on Software Engineering 49, 8 (2023), 4035–4057.
[54] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. ReGVD: Revisiting graph neural networks for vulnerability detection. In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. 178–182.
[55] Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E Santosa, Asankhaya Sharma, and Ming Yi Ang. 2022. Hermes: Using commit-issue linking to detect vulnerability-fixing commits. In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 51–62.

[56] Fangcheng Qiu, Zhongxin Liu, Xing Hu, Xin Xia, Gang Chen, and Xinyu Wang. 2024. Vulnerability detection via multiple-graph-based code representation. IEEE Transactions on Software Engineering (2024).

[57] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of computer programming 74, 7 (2009), 470–495.

[58] Antonino Sabetta and Michele Bezzi. 2018. A practical approach to the automatic classification of security-relevant commits. In 2018 IEEE International conference on software maintenance and evolution (ICSME). IEEE, 579–582.

[59] Antonino Sabetta, Serena Elisa Ponta, Rocio Cabrera Lozoya, Michele Bezzi, Tommaso Sacchetti, Matteo Greco, Gergő Balogh, Péter Hegedűs, Rudolf Ferenc, Ranindya Paramitha, et al. 2024. Known vulnerabilities of open source projects: Where are the fixes? IEEE Security & Privacy 22, 2 (2024), 49–59.

[60] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. IEEE Transactions on Software Engineering 40, 10 (2014), 993–1006.

[61] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 248–259.

[62] Donna Spencer. 2009. Card sorting: Designing usable categories. Rosenfeld Media.

[63] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research 15, 1 (2014), 1929–1958.

[64] Jiamou Sun, Zhenchang Xing, Qinghua Lu, Xiwei Xu, Liming Zhu, Thong Hoang, and Dehai Zhao. 2023. Silent vulnerable dependency alert prediction with vulnerability key aspect explanation. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 970–982.

[65] Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. 2022. Understanding the practice of security patch management across multiple branches in oss projects. In Proceedings of the ACM Web Conference 2022. 767–777.

[66] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2019. 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 (DSN). IEEE, 485–492.

[67] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2020. An empirical study of secret security patch in open source software. Adaptive Autonomous Secure Cyber Systems (2020), 269–289.

[68] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM). IEEE, 595–600.

[69] Zhongzhen Wen, Jiayuan Zhou, Minxue Pan, Shaohua Wang, Xing Hu, Tongtong Xu, Tian Zhang, and Xuandong Li. 2024. Silent Taint-Style Vulnerability Fixes Identification. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 428–439.

[70] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 1046–1058.

[71] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. 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. 860–871.

[72] Jiao Yin, MingJian Tang, Jinli Cao, Hua Wang, Mingshan You, and Yongzheng Lin. 2022. Vulnerability exploitation time prediction: an integrated framework for dynamic imbalanced learning. World Wide Web (2022), 1–23.

[73] Li Yujian and Liu Bo. 2007. A normalized Levenshtein distance metric. IEEE transactions on pattern analysis and machine intelligence 29, 6 (2007), 1091–1095.

[74] Zixian Zhang and Takfarinas Saber. 2024. Assessing the code clone detection capability of large language models. In 2024 4th International Conference on Code Quality (ICCQ). IEEE, 75–83.

[75] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E Hassan. 2023. Colefunda: Explainable silent vulnerability fix identification. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2565–2577.

[76] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 705–716.

[77] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In Proceedings of the 2017 11th joint meeting on foundations of software engineering. 914–919.