Repository-Level Graph Representation Learning for Enhanced Security Patch Detection

Xin-Cheng Wen[1], Zirui Lin[1], Cuiyun Gao[1,2*], Hongyu Zhang[3], Yong Wang[4], Qing Liao[1,2]
[1] Harbin Institute of Technology, Shenzhen, China
[2] Peng Cheng Laboratory, Shenzhen, China
[3] Chongqing University, Chongqing, China
[4] Anhui Polytechnic University, Anhui, China
xiamenwxc@foxmail.com, 210110128@stu.hit.edu.cn, gaocuiyun@hit.edu.cn,
hyzhang@cqu.edu.cn, yongwang@ahpu.edu.cn, liaoqing@hit.edu.cn

Abstract—**Software vendors often silently release security patches without providing sufficient advisories (e.g., Common Vulnerabilities and Exposures) or delayed updates via resources (e.g., National Vulnerability Database). Therefore, it has become crucial to detect these security patches to ensure secure software maintenance. However, existing methods face the following challenges: (1) They primarily focus on the information within the patches themselves, overlooking the complex dependencies in the repository. (2) Security patches typically involve multiple functions and files, increasing the difficulty in well learning the representations. To alleviate the above challenges, this paper proposes a *Repo*sitory-level Security Patch Detection framework named *RepoSPD*, which comprises three key components: 1) a repository-level graph construction, RepoCPG, which represents software patches by merging pre-patch and post-patch source code at the repository level; 2) a structure-aware patch representation, which fuses the graph and sequence branch and aims at comprehending the relationship among multiple code changes; 3) progressive learning, which facilitates the model in balancing semantic and structural information. To evaluate RepoSPD, we employ two widely-used datasets in security patch detection: SPI-DB and PatchDB. We further extend these datasets to the repository level, incorporating a total of 20,238 and 28,781 versions of repository in C/C++ programming languages, respectively, denoted as SPI-DB* and PatchDB*. We compare RepoSPD with six existing security patch detection methods and five static tools. Our experimental results demonstrate that RepoSPD outperforms the state-of-the-art baseline, with improvements of 11.90%, and 3.10% in terms of accuracy on the two datasets, respectively. These results underscore the effectiveness of RepoSPD in detecting security patches. Furthermore, RepoSPD can detect 151 security patches, which outperforms the best-performing baseline by 21.36% with respect to accuracy.**

I. Introduction

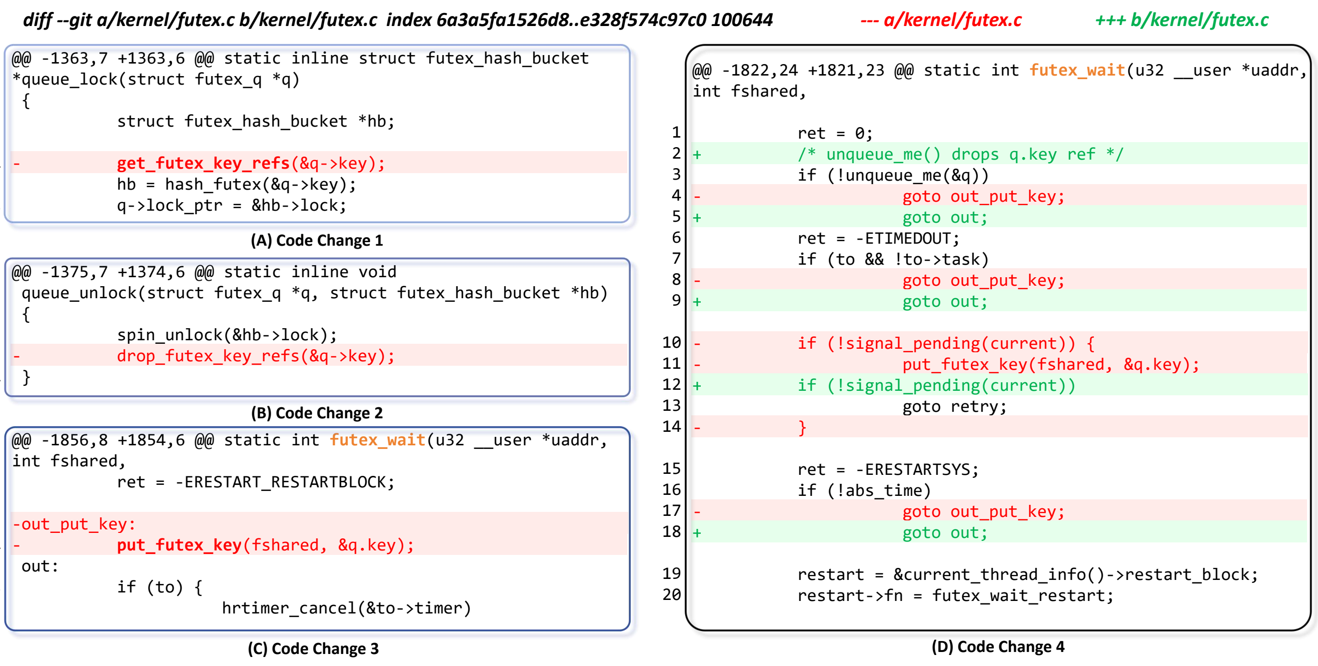
In recent years, the increasing number and diversity of vulnerabilities [1], [2] in Open-Source Software (OSS) have presented significant challenges to software security, posing substantial risks to society [3], [4]. According to the Synopsys [5] report in 2024, 84% of codebases contain at least one open-source vulnerability, and 91% of these codebases include components that are outdated by ten or more versions [6]. There is a critical need for the timely detection of software security patches to mitigate attacks [7]. However, the management of security patches is often subjective by managers [7], [8], leading software vendors to release security updates without sufficient publicity [9]. This practice of silently releasing patches complicates the identification and remediation processes, as users or administrators are frequently overwhelmed by the growing number of patches [10], which often results in delayed software updates and vulnerability reports. The existing study has revealed that over 82% of user-submitted software vulnerability reports are filed more than 30 days after the initial detection [11]. For example, CVE-2024-24919 [12], an information disclosure vulnerability, was first disclosed on May 24th, 2024. However, threat actors had begun exploiting this vulnerability as early as April 30th, targeting over 51 IP addresses [13]. Consequently, more than a hundred thousand users, including those in banks, federal agencies, and large enterprises, faced significant exposure risks. Therefore, it is imperative for both users and developers to automatically distinguish security patches from other updates and prioritize those that directly address security vulnerabilities.

Deep Learning (DL)-based methods have achieved great success in identifying security patches as they can reduce the dependence on the quality of commit messages in patches and offer a broader spectrum of capabilities in detecting various security patches [14]–[16]. Current methods can be categorized into sequence-based and graph-based approaches. Sequence-based methods process the sequential inputs of all code changes in a patch and then utilize DL models to determine whether a code commit fixes a vulnerability. For example, PatchRNN [17] uses both commit messages and code changes as input and then employs the Recurrent Neural Network (RNN) [18] to deal with the input sequentially. Graph-based methods convert the code changes from a code commit into a graph structure, incorporating control flow [19] or data flow dependencies [20], and then use Graph Neural Networks (GNNs) [21] or serialize the graph structure to identify security patches. For example, GraphSPD [22] proposes a PatchCPG and employs a Graph Convolutional Network (GCN) [23] to detect security patches.

However, existing methods face the following challenges:

* Corresponding author.

DOI 10.1109/ICSE55347.2025.00121



```
diff --git a/kernel/futex.c b/kernel/futex.c  index 6a3a5fa1526d8..e328f574c97c0 100644        --- a/kernel/futex.c        +++ b/kernel/futex.c
```

```
  @@ -1363,7 +1363,6 @@ static inline struct futex_hash_bucket
  *queue_lock(struct futex_q *q)
1  {
2           struct futex_hash_bucket *hb;
3
4 -         get_futex_key_refs(&q->key);
5           hb = hash_futex(&q->key);
6           q->lock_ptr = &hb->lock;
```

(A) Code Change 1

```
  @@ -1375,7 +1374,6 @@ static inline void
   queue_unlock(struct futex_q *q, struct futex_hash_bucket *hb)
1  {
2           spin_unlock(&hb->lock);
3 -         drop_futex_key_refs(&q->key);
4  }
```

(B) Code Change 2

```
  @@ -1856,8 +1854,6 @@ static int futex_wait(u32 __user *uaddr,
  int fshared,
1           ret = -ERESTART_RESTARTBLOCK;
2
3 -out_put_key:
4 -         put_futex_key(fshared, &q.key);
5  out:
6           if (to) {
7                   hrtimer_cancel(&to->timer)
```

(C) Code Change 3

```
   @@ -1822,24 +1821,23 @@ static int futex_wait(u32 __user *uaddr,
   int fshared,

1           ret = 0;
2  +        /* unqueue_me() drops q.key ref */
3           if (!unqueue_me(&q))
4  -                goto out_put_key;
5  +                goto out;
6           ret = -ETIMEDOUT;
7           if (to && !to->task)
8  -                goto out_put_key;
9  +                goto out;

10 -        if (!signal_pending(current)) {
11 -                put_futex_key(fshared, &q.key);
12 +        if (!signal_pending(current))
13                  goto retry;
14 -        }

15          ret = -ERESTARTSYS;
16          if (!abs_time)
17 -                goto out_put_key;
18 +                goto out;

19          restart = &current_thread_info()->restart_block;
20          restart->fn = futex_wait_restart;
```

(D) Code Change 4

Fig. 1: The part of security patch that fixes a buffer overflow vulnerability (i.e., CWE-119 [24]). The red and green lines represent the before-fixed code (pre-patch) and after-fixed (post-patch), respectively.

(1) Lack of consideration of the comprehensive contexts at the repository level. The previous methods [17], [22] primarily focus on the information within the security patches, overlooking the complex dependencies in the repository. However, patches typically have complex dependencies in the repository, such as the call function to invoke functions in the patches, which are necessary for identifying a security patch. For instance, as shown in Fig. 1, the patch addresses a buffer overflow vulnerability identified as CVE-2014-0205 [25]. The function `futex_wait()` (shaded in orange in Fig. 1(C) and (D)) fails to properly manage a specific reference count during requeue operations, which may lead to trigger the use-after-free or system crash via the `queue_lock()` and `queue_unlock()` functions (in Fig. 1(A) and (B), respectively). Moreover, the provided patch does not include repository-level dependency about the `get_futex_key_refs` (Line 4 in Fig. 1(A)) and `put_futex_key` (Line 3 in Fig. 1(B), respectively). Therefore, we cannot determine if this patch is a valid security patch due to insufficient contextual information. **(2) Hard to learn the patch representation due to the complex relationships among multiple code changes.** Security patches generally encompass multiple functions and files, increasing the difficulty in learning the representations. For instance, the given commit involves five functions and seven code changes (Fig. 1 shows three functions and four code changes). These code changes do not represent a sequential relationship (i.e., they are not linearly related, such as Fig. 1(C) and Fig. 1(D) simultaneously modifying `futex_wait()` function, shaded in orange). This complexity can limit the existing models' capability to learn representations among multiple code changes.

Our work. To alleviate the above challenges, we propose a ***Repo****sitory-level* ***S****ecurity* ***P****atch* ***D****etection framework named* ***RepoSPD***, which comprises three key components: 1) a novel graph structure, called RepoCPG, aims at extracting comprehensive contexts at the repository level by merging pre-patch and post-patch source code and retaining code changes semantics within the patches; 2) a structure-aware patch representation, which fuses the graph-based and sequence-based representations, aiming at comprehending the relationship among multiple code changes from the structure and semantics perspective, respectively; 3) progressive learning, which aims at facilitating the model in balancing structural and semantic information. Additionally, we extend the SPI-DB [26] and PatchDB [27] datasets to incorporate a total of 20,238 and 28,781 versions of repositories, respectively, denoted as SPI-DB* and PatchDB*.

To evaluate RepoSPD, we compare RepoSPD with five existing security patch detection baselines and five static vulnerability detection methods. The experimental results show that RepoSPD outperforms the state-of-the-art security patch detection approaches, with improvements of 11.90%, and 3.10% in terms of accuracy and F1 score, respectively. Furthermore, RepoSPD detects 151 security patches with an accuracy of 78.65%, which achieves a substantial improvement of 21.36% over the static-analysis-based baselines. These results demonstrate the effectiveness of RepoSPD in identifying security patches.

Contributions. The major contributions of this paper are summarized as follows:

1) To the best of our knowledge, we are the first to propose the repository-level patch CPG, which integrates the repository-level information and retains code change semantics within the security patches.
2) We propose RepoSPD, a repository-level security patch detection framework for capturing patch patterns from both structure and semantics perspectives by fusing the graph-based and sequence-based representations.
3) We curate the SPI-DB* and PatchDB* datasets, and perform an extensive evaluation. The results demonstrate the effectiveness of RepoSPD in security patch detection.

II. Background

A. Code Property Graph

Code Property Graphs (CPGs) [28] are widely used in software vulnerability-related tasks [29]–[31]. They merge Abstract Syntax Trees (ASTs) [32], Control Flow Graphs (CFGs) [19], and Program Dependence Graphs (PDGs) [20] to obtain a joint graph. The PDG [33] is divided into the Control Dependency Graph (CDG) and Data Dependency Graph (DDG), which represent control and data dependencies, respectively. For example, AMPLE [34] explored graph simplification on CPGs and used multi-edge-based GNNs for vulnerability detection. GraphSPD [22] introduced PatchGPG, applying CPGs to software security patch detection.

Although CPG encompasses comprehensive structural and semantic information about the source code, the current approaches are limited in their ability to extract repository-level dependencies, focusing primarily on files corresponding to patches. In this paper, we extend CPGs to the repository level, incorporating more semantic and structural information to comprehend multiple code changes.

B. Repository Context in Code-related Tasks

Incorporating repository-level context for code-related tasks has been a significant challenge. These tasks introduce numerous reasoning challenges based on real software engineering subtasks [35]–[38], such as identifying relevant code, recognizing cross-file dependencies, and understanding repository-specific symbols and conventions. For example, Liu et al. present RepoBench [39], a benchmark specifically designed for evaluating repository-level code completion. Similarly, Wen et al. propose VulEval [40], which integrates repository-level context for vulnerability detection.

However, previous studies on identifying security patches have still relied on file- and patch-level contexts. This limitation presents challenges in real-world software production, as it is crucial for developers to be aware of other files within the repository during programming. In this paper, we further enrich the widely-used SPI-DB and PatchDB datasets at the repository level, which highlights the future directions at the repository level security patch detection.

III. Proposed Framework

We provide an overview of RepoSPD workflow in Fig. 2. RepoSPD mainly consists of three components: (A) RepoCPG construction, (B) structure-aware patch representation, and (C) progressive learning.

Algorithm 1 RepoCPG Construction

```
Input  : Pre_code: $Code\_pre$, Post_code: $Code\_post$, Repo_Function: $Repo\_func$
Output : RepoCPG, $RepoCPG$
1  Function $RepoCPG\ Construction$:
2      // Generating the Pre-Patch and Post-Patch CPGs
3      $Pre\_CPG(N\_pre, E\_pre) \leftarrow Code\_pre$
4      $Post\_CPG(N\_post, E\_post) \leftarrow Code\_post$
5      // Fuse the Pre_CPG and Post_CPG to obtain the MergeCPG.
6      for $v \leftarrow N\_pre, N\_post, E\_pre, E\_post$ do
7          if $v \in V\_pre$ and $v \in V\_post$ then
8              $v$.type $\leftarrow$ fuse
9          else
10             if $v \in V\_pre$ then
11                 $v$.type $\leftarrow$ pre
12             end
13             if $v \in V\_post$ then
14                 $v$.type $\leftarrow$ post
15             end
16         end
17     end
18     $Merge\_CPG \leftarrow Construct\_MergeCPG(Pre\_CPG, Post\_CPG)$
19     // Integrating the repository-level dependency.
20     if $node.type \in Call\_graph$ then
21         // Adopt static tool to extract $Function\_name$ in the repository
22         if $Function\_name \in Repo\_func$ then
23             $Call\_func \leftarrow Repo\_func[i]$
24             // Construct the $Call\_CPG$ of $Call\_func$
25             $Call\_CPG \leftarrow Construct\_Call\_CPG(Call\_func)$
26             // Find the $root\_node$ of $Call\_CPG$
27             Add Edge ($node, root\_node$) and mark the $root\_node$ as $R$
28             // Update Merge_CPG
29             $RepoCPG \leftarrow$ Update($Merge\_CPG, Call\_CPG, R$)
30         end
31     end
32     // Slicing code changes in repository-level.
33     if $R \in$ Code Change then
34         // Deleted-based Repository Slicing
35         $RepoCPG\_Deleted \leftarrow$ Delete_Slice($RepoCPG$)
36         // Added-based Repository Slicing
37         $RepoCPG\_Added \leftarrow$ Add_Slice($RepoCPG$)
38     end
39     $RepoCPG \leftarrow RepoCPG\_Deleted \cup RepoCPG\_Added$
40     return $RepoCPG$
```

A. RepoCPG Construction

The purpose of the *Repository-level CPG (RepoCPG)* construction is to extract comprehensive contexts at the repository level and retain code change semantics within the patches, which consists of the following three steps:

(1) Generating MergeCPG: It aims to establish connections before (pre-patch) and after (post-patch) the patch. However, when multiple code changes occur within a single file, they often lack structural connection to each other due to the patch only containing three lines of code within the code changes [22]. Therefore, we initially employ the *commit-id* in conjunction with the *git reset* command to precisely revert to the specific versions of the repository, thereby retrieving the entire files both pre-patch and post-patch, rather than merely using the code changes. As shown in the Algorithm 1 (Lines 2-4), we generate the CPG for each version of the file to construct pre-patch and post-patch CPGs, respectively.

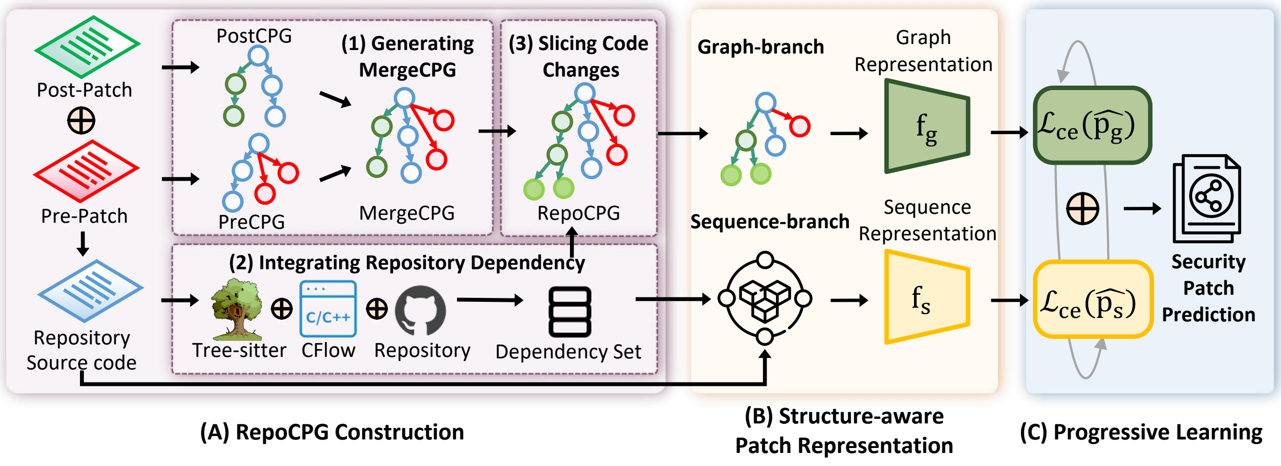


Fig. 2: The overview of RepoSPD.

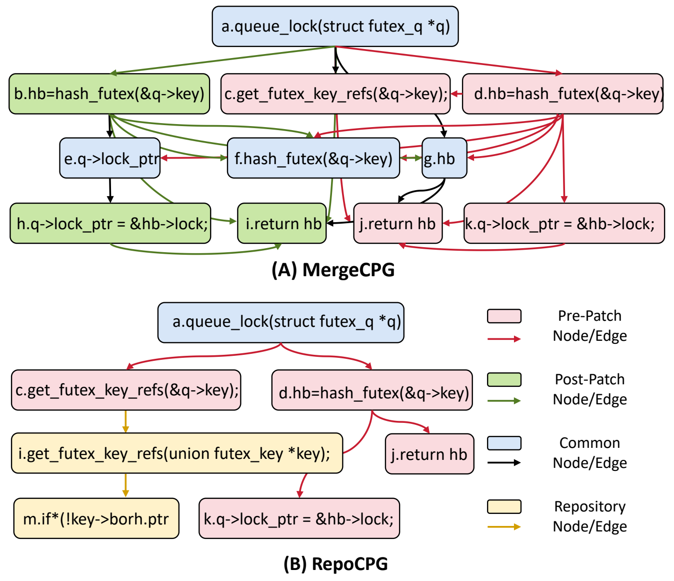


Fig. 3: An example of MergeCPG and RepoCPG construction for the code change 1 in Fig. 1 (A). Due to the large size of the RepoCPG, we ignore some nodes and edges here.

Subsequently, we integrate the pre-patch and post-patch CPGs to establish a unified graph, designated as MergeCPG (Lines 5-18). During the integration process, we retain *common nodes* that are common across both the pre-patch and post-patch versions, such as nodes a, e, f, and g, as shown in Fig. 3 (A). Nodes that exist exclusively either before or after the patch, termed *changed nodes*, are also merged; for example, node c in Fig. 3 (A) appears only in the pre-patch version. We then identify and label nodes that are connected to the *changed nodes*. For instance, nodes d, j, and k, which connect with the *changed node* c, are labeled as *pre-patch nodes*. Correspondingly, nodes b, h, and i, which exist in the post-patch version, are labeled as *post-patch nodes*. By merging the *pre-patch nodes*, *post-patch nodes*, and *common nodes*, we construct the MergeCPG, which effectively preserves the semantics of the original source code and builds the relationship between the pre-patch and post-patch.

(2) Integrating the Repository-level Dependency: We then integrate the repository-level dependency (Lines 19-31 in Algorithm 1) to obtain more contextual information. We utilize the *commit-id* to collect the corresponding specific versions of the repository. We employ the Tree-sitter [41] to traverse the repository and extract the call graph [42] (Line 21). We annotate all function-level call dependencies (i.e., identified as related dependencies) and construct the repository set $Repo_{func}$ for each patch. Subsequently, we select the *change code* in MergeCPG and utilize the Cflow [43], to extract dependency elements and further construct the RepoCPG (Lines 23-29). Specifically, since each node in the MergeCPG contains detailed node information, we identify potential call relationships $Call_{func}$, mark the target nodes, and retrieves within the repository set $Repo_{func}$. For example, as shown in Fig. 3 (B) if a dependency is identified in repository (such as node c), we build an edge from the target node c to the root node i of the function. We then generate the RepoCPG and mark node types (node i and m) according to its affiliation with the node version (*pre-patch nodes*, *post-patch nodes*, or *common nodes*). We integrate MergeCPG with repository-level dependencies to construct RepoCPG.

(3) Slicing Code Changes in Repository-level: RepoSPD further generates a slicing RepoCPG from the original RepoCPG, which includes two steps: Deleted-based repository slicing and Added-based repository slicing. **Deleted-based repository slicing** targets the statements deleted in the pre-patch CPG. For instance, Line 4 in Fig. 1 (A) is a deleted statement in the pre-patch, which calls the `get_futex_key_refs` function. We retain the context retrieved from the repository and incorporate it with the pre-patch statement into RepoCPG (as discussed in Section IIIA step (2)). Other context lines that have no dependency on

the deleted statement are excluded. **Added-based repository slicing** focuses on the statements added in the post-patch CPG. For example, in Fig. 1 (D), Line 10 is an added statement in the post-patch, which calls the `signal_pending` function. We retain the context and incorporate it with the post-patch statement into RepoSPD. Both deleted-based and added-based repository slicing are conducted according to the types of edges within the CDG and DDG, where each node represents a statement. All retained nodes in RepoCPG are directly dependent on the code changes in the patch. Fig. 3 (B) shows an example of RepoCPG after code change slicing at the repository level.

B. Structure-aware Patch Representation

In this component, we elaborate on the proposed structure-aware patch representation, which involves the graph branch for capturing the structural information of RepoCPG, and combines the sequence branch for further enhancing the security patch representations.

(1) Graph branch: The graph branch learns vulnerability patterns from semantic and structural information. The semantic information is exhibited by each node embedding in the RepoCPG, while the structural information is achieved by the graph structure by diverse relationships between node and edges. The content of each node in RepoCPG can be a statement node in CDG/DDG or token node in AST. We use UniXcoder [44] to initialize the representations of each node in the RepoCPG. We generate the node vector $h_i^{(0)}$ for each node i in the RepoCPG, which is calculated as follows:

$$h_i^{(0)} = H^0(N_i) + H^L(N_i) \tag{1}$$

where H^0 and H^L denote the first and last layer embedding of UniXCoder [44], respectively.

We then use the Graph Attention Networks (GAT) [45], which aims at capturing the local structural information of the RepoCPG. Specifically, two types of edge roles and four-bit vectors are defined: version (*pre-patch*, *post-patch*, or *common*), and functional (*CFG*, *DDG*, or *AST*). It is noteworthy that repository-level edges are not listed separately; instead, they encompass both version information and functional relationships. Owing to the distinct roles, it is impractical to apply a uniform set of weights across the entire model for learning the graph representation. Consequently, we construct four subgraphs (i.e., four GAT layers) to cater to four-bit vectors. Within this framework, a GAT layer computes a new set of node embedding by leveraging the input node features along with the attention coefficients that have been learned. For each subgraph, the normalized attention coefficients between nodes i and j are calculated using the following formula:

$$\alpha_{ij}^{(l)}[k] = \text{softmax}\left(\text{LeakyReLU}\left(a^{(l)}\left[Wh_i^{(l-1)} \parallel Wh_j^{(l-1)}\right]\right)\right), \tag{2}$$

where $a^{(l)}$ and W denote the learnable vector and weight matrix, respectively. $\parallel$, softmax, and LeakyReLU denote the concatenation operation, softmax function, and activation function, respectively. k is the index of the subgraph. The node embedding $h^{(l)}[k]$ of subgraph k in layer l is computed as follows:

$$h_i^{(l)}[k] = \sigma\left(\sum_{j \in N(i)} \alpha_{ij}^{(l)}[k] W[k] h_j^{(l-1)}\right), \tag{3}$$

where $N(i)$ and σ denote the set of neighboring nodes of node i and the activation function, respectively. The different roles of edges and the structure-level information in the RepoCPG will be aggregated to the whole graph feature $h_i^{(l)}$, which can be formulated as:

$$h_i^{(l)} = \parallel_{k=1}^{K} h_i^{(l)}[k] \tag{4}$$

where K represents the total number of subgraphs. Since the graph feature $h_i^{(l)}$ only provides individual attention for each subgraph, we further employ another GAT layer to learn the structural information from the whole RepoCPG. Finally, we calculate the representation $h_i^{(l)}$ through a pooling layer to obtain the graph-branch representation f_g.

(2) Sequence branch: We also integrate a sequence branch specifically designed to analyze the code changes within a patch. A typical patch includes several lines of contextual code, but it also often encompasses a substantial amount of extraneous data, such as line numbers and diff markers (e.g., "@@ string1 @@"). Such information can potentially mislead the model during the learning process. To mitigate this issue, the sequence branch is configured to selectively retain only the lines of code changes, along with the version information (either pre-patch or post-patch), to enhance semantic learning. We exclude non-critical elements such as index lines, filenames, and location indicators. Specifically, we fine-tune [46] the UniXcoder [44] and obtain the sequential representation f_s for the sequence branch.

C. Progressive Learning

As shown in Fig. 2(C), we introduce progressive learning that systematically alternates focus between the graph and sequence branches by modulating the respective model weights. This approach is necessitated by the inherent differences in modalities and input characteristic between the graph and sequence branches. Due to these differences, the learning rates and shared parameters between the branches vary substantially, posing challenges for capturing patch-related patterns. Therefore, we propose a method to progressively learn the graph branch and sequence branch, by selectively freezing the model weights of branches during the training process.

More concretely, the feature vectors f_g and f_s will be sent into the classifiers W_g and W_s respectively and the outputs will be integrated together. The predicted output is formulated as:

$$p = \frac{W_g^\top f_g + W_s^\top f_s}{2} \tag{5}$$

Then, progressive learning is to initially train the sequence model, leveraging the domain knowledge provided by the pre-trained model. Subsequently, the focus shifts to the more complex graph structure, ensuring comprehensive learning

TABLE I: Statistics of the SPI-DB* and PatchDB* datasets.

Dataset	Set	#Patch	#File	#Version of Repo
SPI-DB*	Train	16,454	64,580	16,296
	Valid	2,028	7,994	2,024
	Test	2,000	7,649	1,996
	All	20,482	80,223	20,238
PatchDB*	Train	23,229	82,913	23,047
	Valid	2,907	10,349	2,903
	Test	2,906	9,996	2,900
	All	29,042	103,258	28,781

across different data representations. The progressive learning loss of RepoSPD is illustrated as:

$$\mathcal{L} = T\mathcal{L}_{ce}(\hat{p_g}, y) + (1-T)\mathcal{L}_{ce}(\hat{p_s}, y), T = \begin{cases} 0 & \text{if } E \geq \frac{E_{max}}{2} \\ 1 & \text{if } E < \frac{E_{max}}{2} \end{cases} \tag{6}$$

where $\mathcal{L}_{ce}$ and y denote the cross-entropy loss function and the label of data, respectively. $\hat{p_g} = W_g^\top f_g$ and $\hat{p_g} = W_s^\top f_s$ are the predicted output of graph and sequence branch, respectively. T denotes the shifting parameter and E denotes the current training epoch.

IV. Experimental setup

A. Research Questions

In this section, we evaluate the effectiveness of RepoSPD by comparing it with the state-of-the-art security patch detection approaches and focus on the following four Research Questions (RQs):

RQ1: How effective is RepoSPD compared with existing security patch detection approaches?

RQ2: How effective is RepoSPD in security patch detection compared with the static analysis-based approaches?

RQ3: How effective is RepoSPD over patches with different vulnerability types?

RQ4: What is the influence of different components of RepoSPD on the performance for identifying security patches?

B. Datasets

1) Data Source: To address the proposed RQs, we select two widely-used datasets as the raw data: SPI-DB [26] and PatchDB [27]. Specifically, SPI-DB collects patches from two major C/C++ datasets, FFMPeg and Qemu, encompassing 25k patches, of which 10k have been classified as security-related. PatchDB compiles data from 348 open-source repositories, containing over 36k code snippets, approximately 12k identified as security patches.

2) Data Process: To evaluate the security patches at the repository level, we further collect the versions of the repository source code and extract the dependency at the repository level via three steps: (1) We initially select repositories from which complete source code and commit logs can be retrieved via GitHub. This process resulted in the selection of 20,482 patches from SPI-DB and 29,042 from PatchDB, as detailed in Table I. (2) For each identified patch, we use the corresponding commit ID to collect specific code versions at the repository level. This effort has led to the collection of 20,238 versions from repositories (i.e., #Version of Repo in Table I) in SPI-DB and 28,781 from PatchDB. (3) Finally, we traverse the entire repository by Tree-sitter [41] tool to parse the function-level dependencies. Subsequently, we employ the CFlow [43] to further extract dependency elements. As detailed in Table I, we extract the repository dependencies from 80,223 and 103,258 files to construct the SPI-DB* and PatchDB*, respectively.

3) Data Split: Following the previous work [22], we split the datasets into disjoint training, validation, and test sets in a ratio of 8:1:1, as shown in the Table I. We use the training set to train the models, use the validation set for selecting best-performance models, and evaluate the performance in the test set.

C. Baselines

1) Comparison on Security Patch Detection Approaches: To address RQ1, we evaluate the effectiveness of RepoSPD by comparing the following six security patch detection approaches. We categorize these baselines into three groups: **Supervised-based methods:** We utilize PatchRNN and GraphSPD for this category. PatchRNN [17] employs an RNN-based model that processes only source code as input, while GraphSPD [22] introduces PatchCPG and leverages a GNN to learn structural information. These methods are widely recognized and frequently adopted as baselines in recent studies. **Pretrained model-based methods:** We select three prominent pre-trained models: CodeBERT [47], CodeT5 [48], and UniXcoder [44]. These models use code changes (i.e., patches) as input and are further fine-tuned for the downstream task of security patch detection. **LLM-based methods:** Due to resource constraints, we construct the prompt and utilize Llama3-70b [49] to assess the performance of LLMs in security patch detection.

2) Comparison on Static Analysis Approaches: In RQ2, beyond directly identifying security patches, a common approach involves utilizing static analysis tools to detect security patches [22]. This method entails detecting vulnerabilities in pre-patch code snippets and verifying their absence in post-patch code snippets. In this paper, we select five widely used baselines: Cppcheck [50], RATS [51], Semgrep [52], Flawfinder [53], and VUDDY [54]. These methods employ predefined rules and patterns to identify potential vulnerabilities in source code at the repository level. Therefore, they can detect vulnerabilities in the vulnerable version (i.e., pre-patch) and should not identify such vulnerabilities if they have been successfully patched (i.e., post-patch).

D. Evaluation Metrics

We choose the following three metrics to evaluate the performance of RepoSPD.

- **Accuracy:** $Accuracy = \frac{TP+TN}{TP+TN+FN+FP}$. It reflects the percentage of the samples which are correctly classified among samples. TP and TN denote the counts of true positive and true negative samples, respectively. $TP + TN + FN + FP$ represents the total number of samples.
- **F1 score:** $F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$. F1 is the harmonic mean of precision and recall.
- **False Positive Rate (FPR):** $FPR = \frac{FP}{FP+TN}$. It measures the proportion of negative samples that are erroneously classified as positive.

E. Implementation Details

For all the baselines except Llama3-70b, we directly use the publicly available source code and hyper-parameters released by the paper. For Llama3-70b, we use the Ollama [55] framework and evaluate them on our server.

To ensure the fairness of all experiments, we consistently apply the same data across all approaches. We utilize Joern [56] to generate distinct CPGs [28] for both pre-patch and post-patch code. Additionally, Scala is employed to construct the MergeCPG. Then, we use the Tree-sitter [41] and Cflow [43] to analyze and construct the RepoCPG. Tree-sitter is utilized to extract function-level source code by traversing the repository, thereby facilitating subsequent analysis. It is a static analysis tool, which can quickly parse and analyze code structures across repositories [57]. Cflow is employed to extract dependencies at the repository level, focusing on the flow of function calls [40]. For the graph node embedding, we leverage the pre-trained model UniXcoder [44], utilizing its tokenization and model weight to obtain initial node vectors. In the sequence branch, we fine-tune the UniXcoder model with a learning rate of 2×10^{-5}.

For the graph branch, we use a learning rate of 5×10^{-5} and set the GAT with 2 heads. The training process spans 10 epochs with a batch size of 4. We conduct experiments to determine the suitable hyper-parameters. Due to space limitations, the experimental results of PatchDB* are shown in GitHub. All experiments are conducted on a server equipped with NVIDIA GeForce RTX 3090 GPU and CUDA 11.3.

V. Experimental Results

A. RQ1: RepoSPD VS. Security Patch Detection Approaches

To answer RQ1, we compare the three types of security patch detection approaches, including supervised-based, pre-trained model-based, and LLM-based methods. Table II shows the experimental results of each baseline of accuracy, F1 score, and FP rate metrics.

1) Overall Results: The experimental results presented in Table II demonstrate that RepoSPD consistently outperforms all baseline methods on the SPI-DB* and PatchDB* datasets across all evaluated metrics. Specifically, RepoSPD achieves an accuracy of up to 74.55%, and an F1 score of 68.98% on the SPI-DB* dataset. Furthermore, RepoSPD exhibits a higher accuracy of 83.35% and a reduced FP rate of 6.65% on the PatchDB* dataset. This enhanced performance in PatchDB* may be attributed to the larger training data compared to SPI-DB*.

TABLE II: Experimental results of RepoSPD and the security patch detection baselines on the SPIDB* and PatchDB* datasets. Texts in bold represent the best performance of the best methods in each metric.

Dataset	Method	Accuracy↑	F1 Score↑	FP Rate↓
SPIDB*	GraphSPD	59.40	59.11	22.08
	PatchRNN	57.95	60.46	43.97
	CodeBERT	65.61	61.40	32.56
	CodeT5	66.62	61.87	30.31
	UniXCoder	66.27	60.26	28.27
	Llama3-70b	56.35	42.98	26.38
	RepoSPD	**74.55***	**68.98**	**14.67**
PatchDB*	GraphSPD	71.42	48.67	16.52
	PatchRNN	70.15	30.45	8.23
	CodeBERT	78.27	65.27	16.56
	CodeT5	80.84	58.52	9.28
	UniXCoder	80.70	64.34	8.68
	Llama3-70b	74.65	55.84	15.43
	RepoSPD	**83.35***	**69.13**	**6.65**

2) RepoSPD VS. Supervised-based Methods: The results summarized in Table II indicate that RepoSPD enhances performance metrics across both datasets when compared to all other supervised-based methods. Specifically, RepoSPD achieves average improvements of 14.22% in accuracy, and 20.61% in F1 score. GraphSPD outperforms TwinRNN, primarily because GraphSPD integrates the dependency within the patch to provide structural information. Furthermore, RepoSPD incorporates repository-level information to construct RepoCPG, which contributes to its superior performance.

3) RepoSPD VS. Pre-trained Model-based Methods: Table II reveals that pre-trained model-based methods generally surpass those supervised-based methods. Specifically, CodeBERT, CodeT5, and UniXcoder achieve an average performance of 66.17% and 61.18% in terms of the accuracy and F1 score on the SPI-DB* dataset. Despite these results, these methods still behave worse than RepoSPD across all metrics. Particularly, RepoSPD outperforms the state-of-the-art baseline, with relative improvements of 7.50% of accuracy, and 8.70% of F1 score on average across the SPI-DB* and PatchDB* datasets. The primary factor contributing to this performance gap is that pre-trained model-based methods focus on semantic information within code changes. However, they do not adequately address the lack of structural information at the repository level, which is crucial for comprehensive analysis. In contrast, RepoSPD effectively integrates both semantic and structural information, thereby enhancing its overall performance in security patch detection. Furthermore, we also conduct the statistical significance tests between RepoSPD and the best-performing methods, CodeT5. RepoSPD surpasses CodeT5 in both PatchDB* and SPI-DB* at the 0.05 significance level, with p-values of 1.15E-2 and 5.46E-7, respectively. These results show that the RepoSPD significantly outperforms other baselines.

TABLE III: The experimental results of RepoSPD and the static vulnerability detection baselines on the PatchDB* dataset. Texts in bold represent the best performance of the best methods in each metric.

Method	**#Vul$_{\text{pre-patch}}$**	**#Vul$_{\text{post-patch}}$**	**#Security Patch↑**	**Accuarcy(%)↑**
Cppcheck	31	31	0	0.00
RATS	109	110	0	0.00
Semgrep	16	20	0	0.00
Flawfinder	148	150	5	2.60
VUDDY	131	59	110	57.29
RepoSPD	-	-	**151**	**78.65**

4) RepoSPD VS. LLM-based Methods: Due to resource constraints, we only select Llama3-70b for evaluating the performance of LLMs in security patch detection. As indicated in Table II, RepoSPD surpasses the performance of Llama3-70b in all datasets and metrics, despite the generally robust capabilities of LLM-based methods across various domains. There are two primary reasons that may lead Llama3-70b to failing to identify security patches. Firstly, the single prompt may result in a lack of domain-specific knowledge in security patch detection, thereby rendering the detection of complex vulnerability types particularly challenging. Secondly, the model struggles with the lack of repository-level dependencies between code changes at the repository level, which often leads to an inability to make definitive judgments.

Answer to RQ1: In comparison with the security patch detection approaches, RepoSPD achieves the best performance across all evaluated performance metrics, with improvements of 7.50% of accuracy, and 8.70% of F1 score on average across the SPI-DB* and PatchDB* datasets.

B. RQ2: RepoSPD VS. Static Analysis Approaches

To evaluate the effectiveness of RepoSPD in identifying security patches, we also compare with widely-used static analysis-based approaches. In this paper, we select five static analysis tools and utilize a dataset comprising 192 security patches. Specifically, to maintain consistency with GraphSPD [22], the criteria of the selection include: (1) The patches without associated CVEs are removed since they are hard to be verified. (2) The non-security patches are excluded from the dataset. (3) The patches are selected only from the test set to prevent data leakage between the training and valid sets. Based on the selection criteria, the Patch-DB* comprises only 192 patches.

Table III presents the number of vulnerabilities detected by static analysis-based techniques in both pre-patch and post-patch code. As demonstrated in Table III, the results show that RepoSPD surpasses all baselines, detecting an additional 41 security patches and achieving a 21.36% improvement in terms of accuracy. The static analysis tools such as Cppcheck, RATS, and Semgrep fail to detect any security patches, indicating their potential limitation in accurately identifying identify security patches. For instance, Cppcheck identifies 31 vulnerabilities in the pre-patch versions of the code. However, it also identifies the same vulnerabilities in the post-patch versions, indicating that it fails to recognize any of the applied security patches.

Among the static analysis-based approaches, VUDDY exhibits superior detection performance, identifying 131 vulnerabilities in pre-patch code and 59 in post-patch code. Among the vulnerabilities detected in the post-patch code, 38 are not associated with the same patches as the 131 vulnerabilities identified in the pre-patch code. This discrepancy highlights the challenges in accurately classifying patches, with VUDDY correctly identifying 110 patches as secure patches.

In summary, despite the widespread use of existing static analysis-based methods in practice, they exhibit a notable deficiency in detecting security patches. This limitation underscores the value of RepoSPD, which demonstrates a robust capability to identify security patches in the repository-level, thereby enhancing its practical utility.

Answer to RQ2: In comparison with the static analysis-based approaches, RepoSPD surpasses all baselines, detecting an additional 41 security patches and achieving a 21.36% improvement in terms of accuracy.

C. RQ3: Effectiveness of Different Types of Patches in RepoSPD

To evaluate the effectiveness of RepoSPD in identifying different types of security patches, we utilize the same dataset as in RQ2. This dataset encompasses 28 types of Common Weakness Enumerations (CWE) [58] across 74 projects. The proportion and type of vulnerability are presented in Table IV. It is important to note that this dataset exclusively contains security patches and does not include non-security patches.

Overall, we observe that RepoSPD is effective across all ten types of vulnerabilities analyzed, achieving an average accuracy of 81.11%. We can observe the following findings: (1) The RepoSPD shows excellent performance at identifying vulnerabilities associated with a higher number of security patches, such as Buffer Overflow, Resource Leakage, and Numeric Error, with accuracy performance of 87.88%, 82.14%, and 91.67%, respectively. In addition, it also reveals that RepoSPD with a relatively low number of security patches can perform well in some cases, such as untrusted data and race conditions. (2) The security patches that are frequently misclassified typically pertain to vulnerabilities stemming from inadequate verification processes, such as improper input validation and improper access control. These cases often do not present overt errors but rather require the understanding of multiple dependencies to identify vulnerabilities.

TABLE IV: The security patch detection performance of RepoSPD over different vulnerability types.

Vulnerability Type	CWE-ID	Ratio(%)	Acc(%)↑
Buffer Overflow	119, 125, 787	34.38	87.88
Double Free / Use After Free	415, 416	3.65	85.71
Injection	74, 77, 94	1.56	66.67
Improper Input Validation	20, 22	9.38	61.11
Resource Leakage	200, 399, 400	14.58	82.14
Numeric Error	189, 190, 369	12.50	91.67
NULL Pointer Dereference	476	6.77	69.23
Improper Access Control	264, 284	9.38	66.67
Untrusted Data	502	0.52	100.00
Race Condition	362	1.04	100.00
Other Vulnerabilities	16, 17, 19, 59, 254, 310, 426	6.25	41.67

Answer to RQ3: RepoSPD shows excellent performance in identifying vulnerabilities, especially with a higher number of security patches. The security patches that are frequently misclassified typically pertain to vulnerabilities stemming from inadequate verification processes.

D. RQ4: Effectiveness of Different Components in RepoSPD

In this section, we explore the impact of different components of RepoSPD including the RepoCPG construction (i.e., w/o RepoCPG), the structure-aware patch representation (i.e., w/o sequence and w/o graph), and progressive learning (i.e., w/o progressive). The experimental results are shown in Table V.

1) RepoCPG Construction: To explore the effect of the RepoCPG, we deploy one variant (i.e., w/o RepoCPG) by only using the PatchCPG proposed by GraphSPD. As shown in Table V, the RepoCPG can improve the performance of RepoSPD on all datasets. Specifically, incorporating the repository-level information to construct the RepoCPG leads to the average drop of 5.12% in accuracy, and 5.88% F1 score on two datasets. Especially on the SPI-DB* dataset, RepoCPG boosts the performance by 9.80% for accuracy, and 11.32% for F1 score, respectively. In addition, We attribute the relatively small improvement on the PatchDB* dataset to the higher number of dependencies, which is approximately twice that of SPI-DB*. The complex inter-dependencies may hinder the model's learning process to some extent.

2) Structure-aware Patch Representation: To investigate the impact of the structure-aware patch representation, we construct two experimental variants for comparative analysis: (1) a variant exclusively utilizes the graph-based branch (i.e., w/o sequence), and (2) another solely employs the sequence-based branch (i.e., w/o graph), which verify the effectiveness of capturing semantic and structural information for detecting security patches, respectively.

Our findings indicate a consistent performance decline across two datasets when the variants operate independently. Specifically, the variant using only the graph branch exhibits a decrease of 8.00%, while the variant relying solely on the sequence branch shows a decrease of 5.47% in terms of accuracy. This demonstrates that the sequence branch exerts more influence on RepoSPD. Furthermore, the graph branch notably reduces the FP rate, with the decrease of 13.60% and 2.03% in the SPI-DB* and PatchDB* datasets, respectively. This underscores the importance of structural information to increase the performance of security patch detection. We can achieve that both the graph and sequence branches contribute to the overall performance of RepoSPD, each playing a crucial role in security patch detection.

TABLE V: The experimental results of RepoSPD in PatchDB* and SPIDB* datasets when removing the RepoCPG (i.e., w/o RepoCPG), removing the sequence branch (i.e., w/o sequence), removing the graph branch (i.e., w/o graph) and removing the progressive learning(i.e., w/o progressive) in three metrics.

Dataset	Variant	Accuracy↑	F1 score↑	FP Rate↓
SPIDB*	w/o RepoCPG	64.75	57.66	24.82
	w/o sequence	69.60	62.33	17.99
	w/o graph	66.27	60.26	28.27
	w/o progressive	72.25	66.91	18.45
	w/o change order	67.45	62.65	25.92
	RepoSPD	74.55	68.98	14.67
PatchDB*	w/o RepoCPG	82.91	68.69	7.41
	w/o sequence	72.31	31.49	4.86
	w/o graph	80.70	64.34	8.68
	w/o progressive	80.94	65.55	9.38
	w/o change order	81.66	68.25	10.62
	RepoSPD	83.35	69.13	6.65

3) Progressive Learning: To understand the effect of progressive learning, we also implement two variants for comparative analysis: (1) a variant of RepoSPD without the progressive learning component (i.e., w/o progressive), thereby requiring the model to learn weights simultaneously across all components. (2) another variant first trains the graph representation and then refines it into a sequence representation (i.e., changes order). Specifically, the performance consistently shows an average decrease of 2.36% in accuracy, and 2.83% in F1 score across SPI-DB* and PatchDB* datasets without a progressive learning component. It also demonstrate that RepoSPD performs well across two datasets, with an average improvement of 4.40% in accuracy and 3.61% in F1 score, while reducing the FPR by 7.61%. These results underscore the different branches have different learning strategies across various branches of the RepoSPD, with each branch learning unique discriminative representations. It allows for staged optimization and more targeted learning of different branches, which in turn enhances the overall capabilities of detecting security patches.

Answer to RQ4: All components, including RepoCPG construction, structure-aware patch representation, and progressive learning, enhance the performance of RepoSPD.

TABLE VI: The experimental results between RepoSPD and CodeT5 on false negativese.

Dataset	Baseline	Precision	Recall	F1 Score
PatchDB*	CodeT5	73.62	58.52	65.21
	RepoSPD	**80.18**	**60.76**	**69.13**
SPI-DB*	CodeT5	61.17	**62.59**	61.87
	RepoSPD	**78.07**	61.79	**68.98**

TABLE VII: Time cost between RepoSPD and CodeT5 per epoch training and inference time.

Baseline	Time\Dataset	PatchDB*	SPI-DB*
CodeT5	Train time (s)	1573.56	1171.95
	Inference time (s)	99.02	90.27
RepoSPD	Train time (s)	420.45	436.70
	Inference time (s)	37.20	27.38

VI. Discussion

A. Why does RepoSPD Work?

We identify the advantages of RepoSPD, which can explain its effectiveness in security patch detection.

(1) Incorporating repository-level information to help the security patch detection. We propose RepoCPG to integrate repository-level dependencies, which enhances the performance of security patch detection. As illustrated in Fig. 4(A), the example is drawn from a resource leakage vulnerability in the Linux kernel (i.e., CVE-2016-5243 [59]). Specifically, the function `tipc_nl_compat_link_dump` in net/tipc/netlink_compat.c fails to copy a particular string (Line 5), thereby allowing local users to access sensitive data from the kernel stack memory. This issue is addressed in Lines 6-7 through the use of `nla_strlcp`. However, `nla_strlcp` is not defined within the patch itself. RepoSPD effectively incorporates `nla_strlcp` as a dependency into RepoCPG, and accurately identifies the security patch.

(2) Effectively capturing both structural and sequential information among multiple code changes. We propose the structure-aware patch representation to comprehend the relationships among multiple code changes. As illustrated in Fig. 4 (C), the `_TIFFmalloc` call at Line 5 of the pre-patch code implicitly assumes successful memory allocation, which effectively acts as an implicit assertion check. This assumption may lead to buffer overflows in release mode, particularly when handling atypical tile sizes. Furthermore, the patch includes a total of 11 code changes. RepoSPD can extract the structural dependency between Line 12 and Line 18, which collaboratively changes the function `fpDiff`. Additionally, RepoSPD captures the semantic information, revealing that Line 4 and Line 18 utilize identical statements to address the vulnerabilities. Based on the structural and semantic information, RepoSPD effectively detects the security patch.

B. Impact of Repository-level Context on False Negatives

We also perform experiments on the impact of repository-level context on false negatives and the results are listed in Table VI. The experimental results demonstrate that RepoSPD outperforms the current state-of-the-art baseline, CodeT5, in 5 out of 6 cases. Specifically, RepoSPD achieves improvements of 6.56% in precision and 2.24% in recall on the PatchDB* dataset. The sole exception is in the recall metric in SPI-DB*, where CodeT5 surpasses RepoSPD by 0.8%. It may be attributed to the additional context, which in some cases, has a minor impact on the rate of false negatives.

C. Training and Inference Time

We have conducted a detailed analysis of the training and inference time costs per epoch for both RepoSPD and CodeT5, as presented in Table VII. The results indicate that RepoSPD requires, only 857.15 seconds for training per epoch and 64.58 seconds for inference in the test set. In comparison, the best-performing baseline, CodeT5, requires 2745.51 seconds for training and 189.29 seconds for inference. The reason is that the graph branch is more efficient than the sequence branch (pre-trained model).

D. Threats and Limitations

We have identified the following major threats and limitations:

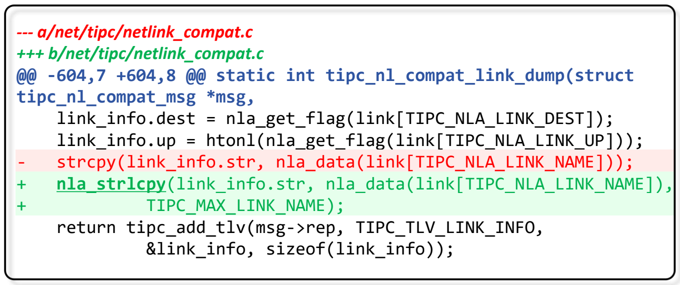
Constraints of Data Collection. To facilitate the identification of security patches at the repository level, we meticulously extracted 20,238 and 28,781 versions of repositories from platforms such as GitHub to reconstruct SPI-DB* and PatchDB*. Despite our extensive efforts to crawl the largest known repositories, some repositories listed in the National Vulnerability Database (NVD) [61] remained inaccessible. In our future work, we aim to expand our collection of security patches.

Generalizability on Other Programming Languages. In this paper, we construct the RepoCPG using Joern, Tree-sitter, and Cflow, specifically tailored for C/C++. While our experimental evaluations are focused on C/C++, the RepoSPD can be extended to support other programming languages by integrating code analyzers that accommodate the respective syntax and structural paradigms. In future work, we plan to explore the applicability and effectiveness of RepoSPD across wider programming languages, such as Java and Python, thereby broadening its applicability for security patch detection.

Restrictions of Dependency Extraction in Repository. We do not use the entire dependency graph in the repository due to the size of the RepoCPG and the limitations inherent in static analysis techniques. While this approach is generally sufficient, it fails to encompass all scenarios, such as multi-level function calls within the repository. A potential solution to enhance the coverage could involve the development of additional rules and the application of more sophisticated slicing methods to construct the RepoCPG.

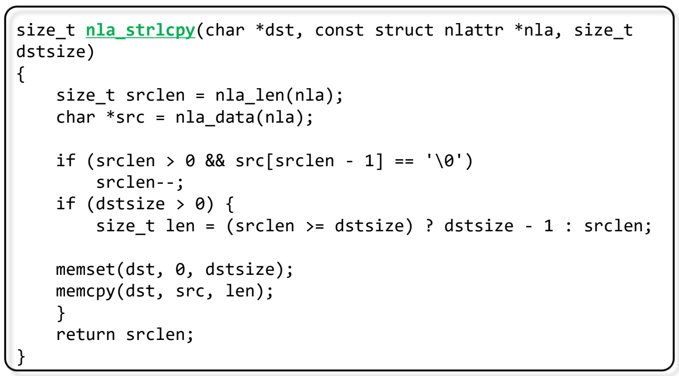
VII. Related Work

Security patch detection is crucial for enabling users to identify and apply updates addressing vulnerabilities on



```
--- a/net/tipc/netlink_compat.c
+++ b/net/tipc/netlink_compat.c
@@ -604,7 +604,8 @@ static int tipc_nl_compat_link_dump(struct
tipc_nl_compat_msg *msg,
    link_info.dest = nla_get_flag(link[TIPC_NLA_LINK_DEST]);
    link_info.up = htonl(nla_get_flag(link[TIPC_NLA_LINK_UP]));
-   strcpy(link_info.str, nla_data(link[TIPC_NLA_LINK_NAME]));
+   nla_strlcpy(link_info.str, nla_data(link[TIPC_NLA_LINK_NAME]),
+            TIPC_MAX_LINK_NAME);
    return tipc_add_tlv(msg->rep, TIPC_TLV_LINK_INFO,
            &link_info, sizeof(link_info));
```

(A) An example of security patch (CVE-2016-5243).

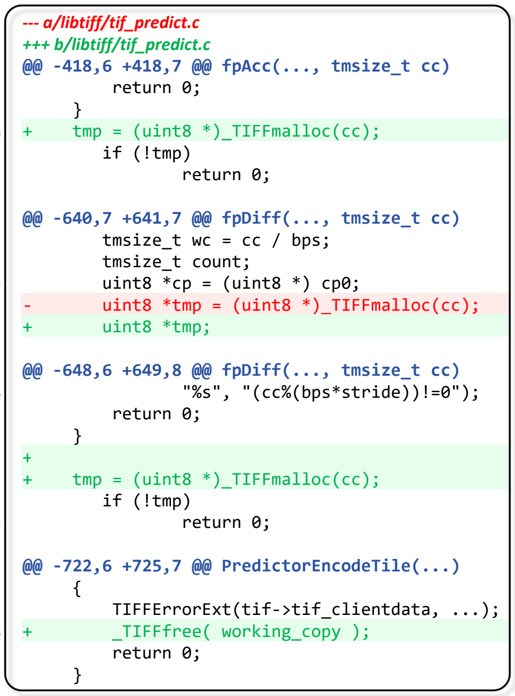


```
size_t nla_strlcpy(char *dst, const struct nlattr *nla, size_t
dstsize)
{
    size_t srclen = nla_len(nla);
    char *src = nla_data(nla);

    if (srclen > 0 && src[srclen - 1] == '\0')
        srclen--;
    if (dstsize > 0) {
        size_t len = (srclen >= dstsize) ? dstsize - 1 : srclen;

    memset(dst, 0, dstsize);
    memcpy(dst, src, len);
    }
    return srclen;
}
```

(B) A dependency extracted from CVE-2016-5243.



```
--- a/libtiff/tif_predict.c
+++ b/libtiff/tif_predict.c
@@ -418,6 +418,7 @@ fpAcc(..., tmsize_t cc)
        return 0;
    }
+   tmp = (uint8 *)_TIFFmalloc(cc);
        if (!tmp)
                return 0;

@@ -640,7 +641,7 @@ fpDiff(..., tmsize_t cc)
        tmsize_t wc = cc / bps;
        tmsize_t count;
        uint8 *cp = (uint8 *) cp0;
-       uint8 *tmp = (uint8 *)_TIFFmalloc(cc);
+       uint8 *tmp;

@@ -648,6 +649,8 @@ fpDiff(..., tmsize_t cc)
                "%s", "(cc%(bps*stride))!=0");
        return 0;
    }
+
+   tmp = (uint8 *)_TIFFmalloc(cc);
        if (!tmp)
                return 0;

@@ -722,6 +725,7 @@ PredictorEncodeTile(...)
    {
        TIFFErrorExt(tif->tif_clientdata, ...);
+       _TIFFfree( working_copy );
        return 0;
    }
```

(C) An example of security patch (CVE-2016-9535).

Fig. 4: (A) and (C) represent the security patch for resource leakage vulnerability (CVE-2016-5243 [59]) and buffer overflow vulnerability (CVE-2016-9535 [60]), respectively. Fig. (B) is the dependency (i.e., repository information) extracted from CVE-2016-5243 to construct RepoCPG.

time [62]. It is important in OSS, where it is recommended to address vulnerabilities silently until they are publicly disclosed [7]. Initially, security patch detection primarily employed rule-based [63], [64] and traditional machine learning techniques [65]–[69]. For example, Li et al. [9] conducted an empirical study on security patches. Wu et al. [64] developed symbolic rules to characterize security patches. VCCFinder [70] utilizes SVM to identify potential vulnerabilities in open-source projects. Subsequent research has incorporated DL-based methods for security patch detection [26], [71]–[73]. Zuo et al. [74] highlighted the role of commit messages in detecting security patches and introduced a Transformer-based approach. PatchRNN [17] integrates both source code and commit messages to improve performance for identifying security patches. GraphSPD [22] proposes the PatchCPG and employs a graph-based approach for security detection. VulFixMiner [75] extracts added and removed code from commits, utilizing CodeBERT to identify security patches in Java and Python.

Despite these advancements, existing methods do not consider the repository-level dependency and struggle to comprehend the relationship among multiple code changes within a patch. To address these challenges, we propose a repository-level security patch detection framework RepoSPD. We also extend both SPI-DB* and PatchDB* to the repository level.

VIII. Conclusion

In this paper, we propose a repository-level security patch detection framework named RepoSPD, which comprises the RepoCPG construction to incorporate repository-level dependency, a structure-aware patch representation for comprehending the relationship among multiple code changes, and progressive learning to facilitate the model in balancing semantic and structural information. We further extend two widely-used datasets SPI-DB and PatchDB to the repository level, incorporating a total of 20,238 and 28,781 versions of repository in C/C++ programming languages, respectively (denoted as SPI-DB* and PatchDB*). Compared with the state-of-the-art approaches, the experimental results underscore the effectiveness of RepoSPD for security patch detection.

Our source code and detailed experimental results are available at: *https://github.com/Xin-Cheng-Wen/RepoSPD*.

Acknowledgment

This research is supported by National Key R&D Program of China (No. 2022YFB3103900), National Natural Science Foundation of China under project (No. 62472126), Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), Shenzhen-Hong Kong Jointly Funded Project (Category A, No. SGDX20230116091246007), Shenzhen Basic Research (General Project No. JCYJ20220531095214031), Shenzhen International Science and Technology Cooperation Project (No. GJHZ20220913143008015), the Major Key Project of PCL (Grant No. PCL2024A05), CCF-Huawei Populus Grove Fund and the University Synergy Innovation Program of Anhui Province (Grant No. GXXT-2022-047).

REFERENCES

[1] Statista, “Number of common it security vulnerabilities and exposures (cves) worldwide from 2009 to 2024 ytd,” 2024. [Online]. Available: https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/

[2] X. Wen, C. Gao, F. Luo, H. Wang, G. Li, and Q. Liao, “LIVABLE: exploring long-tailed classification of software vulnerability types,” *IEEE Trans. Software Eng.*, vol. 50, no. 6, pp. 1325–1339, 2024.

[3] R. Telang and S. Wattal, “An empirical analysis of the impact of software vulnerability announcements on firm stock price,” *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 544–557, 2007.

[4] X. Wen, X. Wang, C. Gao, S. Wang, Y. Liu, and Z. Gu, “When less is enough: Positive and unlabeled learning model for vulnerability detection,” in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 345–357.

[5] Synopsys. (2024). [Online]. Available: https://www.synopsys.com/

[6] Synopsys. (2024) 2024 open source security and risk analysis report. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2024.pdf

[7] X. Wang, K. Sun, A. L. Batcheller, and S. Jajodia, “Detecting ”0-day” vulnerability: An empirical study of secret security patch in OSS,” in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 2019, pp. 485–492.

[8] A. TamjidYamcholo and A. Toloie Eshlaghy, “Subjectivity reduction of qualitative approach in information security risk analysis,” *Journal of System Management*, vol. 8, no. 1, pp. 145–166, 2022.

[9] F. Li and V. Paxson, “A large-scale empirical study of security patches,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2201–2215.

[10] S. Mirhosseini and C. Parnin, “Can automated pull requests encourage software developers to upgrade out-of-date dependencies?” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 84–94.

[11] F. Thung, D. Lo, L. Jiang, Lucia, F. Rahman, and P. T. Devanbu, “When would this bug get reported?” in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 420–429.

[12] CVE-2024-24919. (2024) Cve-2024-24919 detail. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-24919

[13] T. stack. (2024) Check point vulnerability far worse than thought – exploited in wild since april. [Online]. Available: https://www.thestack.technology/check-point-vulnerability-cve-2024-24919/

[14] X. Cheng, X. Nie, N. Li, H. W. Z. Zheng, and Y. Sui, “How about bug-triggering paths?-understanding and characterizing learning-based vulnerability detectors.” IEEE, 2022.

[15] Z. Zhang, C. Luo, B. Zhang, H. Jiang, and B. Zhang, “Multi-task framework of precipitation nowcasting,” *CAAI Trans. Intell. Technol.*, vol. 8, no. 4, pp. 1350–1363, 2023.

[16] F. Liu, Z. Zheng, Y. Shi, Y. Tong, and Y. Zhang, “A survey on federated learning: a perspective from multi-party computation,” *Frontiers Comput. Sci.*, vol. 18, no. 3, p. 181336, 2024.

[17] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun, and F. Geck, “Patchrnn: A deep learning-based system for security patch identification,” in *2021 IEEE Military Communications Conference, MILCOM 2021, San Diego, CA, USA, November 29 - Dec. 2, 2021*. IEEE, 2021, pp. 595–600.

[18] J. L. Elman, “Finding structure in time,” *Cogn. Sci.*, vol. 14, no. 2, pp. 179–211, 1990.

[19] X. Huo, M. Li, and Z. Zhou, “Control flow graph embedding based on multi-instance decomposition for bug localization,” in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 4223–4230.

[20] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O’Boyle, and H. Leather, “Programl: A graph-based program representation for data flow analysis and compiler optimizations,” in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 2021, pp. 2244–2253.

[21] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, “Gated graph sequence neural networks,” in *4th International Conference on Learning Representations, ICLR 2016*, 2016.

[22] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, “Graphspd: Graph-based security patch detection with enriched code semantics,” in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2409–2426.

[23] T. N. Kipf and M. Welling, “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*. OpenReview.net, 2017.

[24] “Cwe-119: Improper restriction of operations within the bounds of a memory buffer.” [Online]. Available: https://cwe.mitre.org/data/definitions/119.html

[25] CVE-2014-0205. (2014) Cve-2014-0205 detail. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2014-0205

[26] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, “SPI: automated identification of security patches via commits,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, pp. 13:1–13:27, 2022.

[27] X. Wang, S. Wang, P. Feng, K. Sun, and S. Jajodia, “Patchdb: A large-scale security patch dataset,” in *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. IEEE, 2021, pp. 149–160.

[28] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *2014 IEEE Symposium on Security and Privacy, SP 2014*. IEEE Computer Society, 2014, pp. 590–604.

[29] X. Wang, T. Zhang, R. Wu, W. Xin, and C. Hou, “CPGVA: code property graph based vulnerability analysis by deep learning,” in *ICAIT*. IEEE, 2018, pp. 184–188.

[30] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*, 2019, pp. 10 197–10 207.

[31] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: Are we there yet?” *CoRR*, vol. abs/2009.07235, 2020.

[32] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 783–794.

[33] Y. Li, S. Wang, and T. N. Nguyen, “Vulnerability detection with fine-grained interpretations,” in *ESEC/SIGSOFT FSE*. ACM, 2021, pp. 292–303.

[34] X. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, “Vulnerability detection with graph simplification and enhanced graph representation learning,” in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2275–2286.

[35] R. Bairi, A. Sonwane, A. Kanade, V. D. C., A. Iyer, S. Parthasarathy, S. K. Rajamani, B. Ashok, and S. Shet, “Codeplan: Repository-level coding using llms and planning,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 675–698, 2024.

[36] M. Liang, X. Xie, G. Zhang, X. Zheng, P. Di, W. Jiang, H. Chen, C. Wang, and G. Fan, “REPOFUSE: repository-level code completion with fused dual context,” *CoRR*, vol. abs/2402.14323, 2024.

[37] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang, X. Che, Z. Liu, and M. Sun, “Repoagent: An llm-powered open-source framework for repository-level code documentation generation,” *CoRR*, vol. abs/2402.16667, 2024.

[38] H. N. Phan, H. N. Phan, T. N. Nguyen, and N. D. Q. Bui, “Repohyper: Better context retrieval is all you need for repository-level code completion,” *CoRR*, vol. abs/2403.06095, 2024.

[39] T. Liu, C. Xu, and J. J. McAuley, "Repobench: Benchmarking repository-level code auto-completion systems," *CoRR*, vol. abs/2306.03091, 2023.

[40] X. Wen, X. Wang, Y. Chen, R. Hu, D. Lo, and C. Gao, "Vuleval: Towards repository-level evaluation of software vulnerability detection," *CoRR*, vol. abs/2404.15596, 2024.

[41] (2023) Tree-sitter. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[42] M. Keshani, G. Gousios, and S. Proksch, "Frankenstein: fast and lightweight call graph generation for software builds," *Empir. Softw. Eng.*, vol. 29, no. 1, p. 1, 2024.

[43] S. Poznyakoff, ""GNU cflow"," 2005, https://www.gnu.org/software/cflow/.

[44] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 7212–7225.

[45] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," *CoRR*, vol. abs/1710.10903, 2017.

[46] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway, and J. Liang, "Convolutional neural networks for medical image analysis: Full training or fine tuning?" *IEEE Trans. Medical Imaging*, vol. 35, no. 5, pp. 1299–1312, 2016.

[47] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "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*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[48] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708.

[49] Meta, "Meta-llama-3-70b," 2024. [Online]. Available: https://huggingface.co/meta-llama/Meta-Llama-3-70B

[50] Cppcheck-team, ""Cppcheck"," [n.d.], http://cppcheck.sourceforge.net/.

[51] "Rough audit tool for security." [n.d.]. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security.

[52] r2c, ""Semgrep"," 2021, https://semgrep.dev.

[53] D. A. Wheeler, "Flawfinder," [n.d.]. [Online]. Available: https://dwheeler.com/flawfinder/

[54] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 595–614.

[55] "Ollama," 2024. [Online]. Available: https://ollama.com/

[56] (2021) The joern project. code property graph — joern documentation. [Online]. Available: https://docs.joern.io/code-property-graph/

[57] X. Wen, C. Gao, S. Gao, Y. Xiao, and M. R. Lyu, "SCALE: constructing structured natural language comment trees for software vulnerability detection," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 235–247.

[58] "Common weakness enumerations." [Online]. Available: https://cwe.mitre.org/

[59] CVE-2016-5243. (2016) Cve-2016-5243 detail. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016-5243

[60] CVE-2016-9535. (2016) Cve-2016-9535 detail. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016-9535

[61] "National vulnerability database," [n.d.]. [Online]. Available: https://nvd.nist.gov/

[62] K. Vaniea and Y. Rashidi, "Tales of software updates: The process of updating software," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*, J. Kaye, A. Druin, C. Lampe, D. Morris, and J. P. Hourcade, Eds. ACM, 2016, pp. 3215–3226.

[63] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 539–554.

[64] Q. Wu, Y. He, S. McCamant, and K. Lu, "Precisely characterizing security impact in a flood of patches via symbolic rule comparison," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[65] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 386–396.

[66] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu, "Automatic hot patch generation for android kernels," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 2397–2414.

[67] X. Wang, S. Wang, K. Sun, A. L. Batcheller, and S. Jajodia, "A machine learning approach to classify security patches into vulnerability types," in *8th IEEE Conference on Communications and Network Security, CNS 2020, Avignon, France, June 29 - July 1, 2020*. IEEE, 2020, pp. 1–9.

[68] M. Soto, F. Thung, C. Wong, C. Le Goues, and D. Lo, "A deeper look into bug fixes: patterns, replacements, deletions, and additions," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 512–515.

[69] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," in *TEFSE'11, Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, May 23, 2011, Waikiki, Honolulu, HI, USA*, D. Poshyvanyk, M. D. Penta, and H. H. Kagdi, Eds. ACM, 2011, pp. 31–37.

[70] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015, pp. 426–437.

[71] T. G. Nguyen, T. Le-Cong, H. J. Kang, X. D. Le, and D. Lo, "Vulcurator: a vulnerability-fixing commit detector," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 1726–1730.

[72] B. Wu, S. Liu, R. Feng, X. Xie, J. K. Siow, and S. Lin, "Enhancing security patch identification by capturing structures in commits," *CoRR*, vol. abs/2207.09022, 2022.

[73] J. Zhou, M. Pacheco, J. Chen, X. Hu, X. Xia, D. Lo, and A. E. Hassan, "Colefunda: Explainable silent vulnerability fix identification," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2565–2577.

[74] F. Zuo, X. Zhang, Y. Song, J. Rhee, and J. Fu, "Commit message can help: Security patch detection in open source software via transformer," in *21st IEEE/ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2023, Orlando, FL, USA, May 23-25, 2023*. IEEE, 2023, pp. 345–351.

[75] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding A needle in a haystack: Automated mining of silent vulnerability fixes," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 705–716.