GITPATCHDB: A LARGE-SCALE GITHUB COMMIT DATABANK FOR VULNERABILITY PATCH ANALYSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Machine learning based vulnerability detection relies on datasets that link vulnerabilities to their corresponding patches. However, existing resources such as *Common Vulnerabilities and Exposures (CVE)* often lack reliable patch references, *e.g.*, many CVE entries do not provide patch commits, and a significant share of existing commits become inaccessible due to code repository changes. To bridge this gap and better facilitate vulnerability detection, we curate GITPATCHDB, a large-scale, semantic-rich dataset that pairs CVEs with their corresponding patch commits, where each commit is formatted not only as code diffs but also as inter-procedural program slices generated through program slicing and related program analysis techniques. To leverage this semantic-rich dataset, we further propose *Contrastive Natural-language Programming-language Pre-training (CNPP)*, a novel approach that enables multimodal vulnerability patch search via contrastive learning. Extensive evaluations demonstrate that GITPATCHDB paired with CNPP achieves 95.99% top-10 accuracy in vulnerability patch search, surpassing baseline manual methods by over 8% and establishing a new state-of-the-art performance.

1 INTRODUCTION

Software vulnerabilities pose critical threats to digital infrastructures by enabling security exploits, such as unauthorized access, remote code execution, and data breaches (Tsankov et al., 2018; Li & Paxson, 2017; Li et al., 2016; Kim et al., 2017). To support vulnerability triage and mitigation, databases such as the Common Vulnerabilities and Exposures (CVE) (The MITRE Corporation, 2025) and the National Vulnerability Database (NVD) (National Institute of Standards and Technology, 2025) maintain high-level descriptions of known vulnerabilities (Frei et al., 2006). However, these databases frequently lack direct references to corresponding patch commits (Ponta et al., 2019; Nguyen & Massacci, 2013; Chaparro et al., 2017), either because such links were never recorded or because they have become inaccessible due to repository deletions, branch renames, or pull request merges (Wang et al., 2021; Fan et al., 2020). This significantly limits their utility for tasks such as vulnerability analysis, patch auditing, and forensic investigation during incident response, a limitation well documented in prior studies (Nguyen & Massacci, 2013; Chaparro et al., 2017).

The issue becomes even more pronounced in open-source software (OSS), which is increasingly central to today's software ecosystems (Ponta et al., 2020). Recent surveys show that over 96% of codebases incorporate OSS, and more than 84% contain at least one known vulnerability (MvnRepository, 2025). Alarmingly, over half of the CVEs in NVD do not include any patch references (Xu et al., 2022; Tan et al., 2021). Even when patch commits are cited, our empirical study of 193,448 CVE entries shows that only 30.99% include such links, and among these, 41.34% are non-functional [1]. This fragmentation makes it difficult to retrieve the actual code fixes for known vulnerabilities.

In this work, we aim to address the problem of linking CVEs to their corresponding source-level patches, coined as *vulnerability patch search*. Unlike prior datasets that rely on manually curated patch references or small-scale heuristics (Xu et al., 2022; Tan et al., 2021; Wang et al., 2022), we propose a large-scale and semantic-rich dataset named GITPATCHDB for vulnerability patch search based on automatic program analysis techniques, and further propose **C**ontrastive **N**atural-language **P**rogramming-language **P**re-training (CNPP), a simple and effective approach that enables

[1] See Appendix A for detailed methodology and empirical results.

multimodal vulnerability patch search via contrastive learning, with comparisons against other baselines.

Key Challenges and Insights. We identify four key challenges in *accurately* modeling the *semantics* of a patch and *efficiently* matching them across thousands of real-world codebases:

- **Patch context completeness.** Existing vulnerability patch search approaches rely on patch commit diffs, *i.e.*, added and deleted lines in the patch commit, and keyword matching such as CVE-ID strings (Tan et al., 2021; Wang et al., 2022; Xu et al., 2022), which fails to capture all contextual semantics related to vulnerability patches, *e.g.*, other code lines related to the added and deleted lines in the patch commit. *We address this challenge by applying interprocedural program slicing, which traces data dependencies backward and forward from patch lines to recover full code context (e.g., variable definitions, function calls).*
- **Patch context fidelity**. Interprocedural program slicing used in extracting patch contextual semantics constructs data dependencies for the program, which requires accurate aliasing information, *e.g.*, set of program variables pointing to the same object. Traditional approaches handle it conservatively, often over-approximating all possible aliases and thus bloating program slices (Andersen & Lee, 2005). *We address this challenge by equipping traditional pointer analysis with flow-sensitivity (algorithm 1), pruning spurious dependencies while preserving true data flows.*
- **Analysis scalability**. Interprocedural program slicing is computationally infeasible for large codebases. *We propose a novel approach named on-the-fly patch program co-analysis by combining program slicing with flow-sensitive pointer analysis on-the-fly, dynamically expanding slices from patch commit code lines and deferring analyzing irrelevant code regions (algorithm 1).*
- **Embedding scalability**. Interprocedural program slices generated from real-world commits often yield high-dimensional token sequences (often exceeding 10K tokens). *We propose* CNPP*, which encodes high-dimensional token sequences with hierarchical attention mechanisms and long short-term memory (LSTM) into fixed-length embeddings, prioritizing vulnerability-critical tokens such as bounds checks and pointer dereferences, while preserving long-range dependencies.*

Contributions. This work makes the following key contributions:

- We introduce GITPATCHDB, a large-scale and semantic-rich dataset for vulnerability patch search based on alias-aware interprocudural program slicing, that surpasses existing datasets in scale, semantic coverage, and semantic fidelity. This dataset addresses the longstanding lack of comprehensive patch data linking CVEs to source-level patches, offering the community a reliable foundation for tasks such as vulnerability patch search, vulnerability understanding, and other downstream applications in deep learning. By releasing GITPATCHDB [2][3], we aim to foster research reproducibility and support future work at the intersection of software security and machine learning.
- We propose *on-the-fly patch program co-analysis*, that combines forward/backward program slicing with flow-sensitive pointer analysis. By iteratively refining aliasing relationships and data-flow dependencies, this co-analysis analyzes the patch semantics on-the-fly, which efficiently boosts the scalability of our dataset, while avoiding the pitfalls in traditional static analysis (e.g., over-approximation of data flows) and enabling rich patch semantics.
- We further propose Contrastive Natural-language Programming-language Pre-training (CNPP), a novel approach that enables multimodal vulnerability patch search by encoding and fusing both vulnerability and patch information into embeddings for contrastive learning. Extensive evaluations demonstrate that GITPATCHDB paired with CNPP achieves high accuracy in vulnerability patch search, surpassing baseline methods and establishing a competitive performance.

2 RELATED WORK AND BACKGROUND

Research on vulnerability patch analysis falls into two categories: *vulnerability patch classification* and *vulnerability patch search*. Our proposed dataset GITPATCHDB and approach CNPP mainly belongs to vulnerability patch search.

Vulnerability patch classification. Early research such as Big-Vul (Fan et al., 2020) and PatchDB (Wang et al., 2021) focuses on binary classification, *i.e.*, determining whether a given commit constitutes a vulnerability fix. Big-Vul provides CVE-labeled snippets but lacks complete

[2]https://anonymous.4open.science/r/gitpatchdb-2EC7

[3]GITPATCHDB is licensed under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0).

diffs or commits, limiting itself to downstream retrieval tasks. PatchDB extends coverage with synthetic and wild patches, but it only supports C/C++ and does not incorporate interprocedural or alias-aware analyses. Similarly, SPI (Zhou et al., 2021) and GraphSPD (Wang et al., 2023) provide large-scale annotations but are limited to intraprocedural views, often at the function level and without alias analysis, thereby restricting semantic expressiveness. CommitBART (Liu et al., 2022) scales to millions of commits but lacks any CVE linkage, rendering it unsuitable for

Vulnerability patch search. Recent retrieval-based systems such as VCMatch (Wang et al., 2022) and PatchScout (Tan et al., 2021) rely on shallow textual features or ensemble ranking methods, and are constrained to small, manually curated datasets. Tracing-based tools (Xu et al., 2022) depend on repository links, which frequently become obsolete due to refactoring, branch renaming, or repository deletions. Moreover, none of these methods support scalable, multi-language search or leverage alias-aware, interprocedural semantics, with limitations summarized in Table 1.

Table 1: Comparisons of vulnerability patch analysis datasets.

Dataset (Problem Type)	CVE-linked Commits	Commit Diff	Program Slicing	Pointer Analysis	Languages
PatchDB Security Patch Classification	4,076	✓	✗	✗	C/C++ only
SPI Security Patch Classification	1,045	✓	✗	✗	C only
Big-Vul Security Patch Classification	3,754	✓	✗	✗	C/C++ only
GraphSPD Security Patch Classification	5,121[4]	✓	✓ Intraprocedural	✗	C/C++ only
CommitBART Commit Understanding & Generation	✗ Not CVE-linked	✓	✗	✗	Multi-lang (7)
PatchScout Vulnerability Patch Search	1,628	✓	✗	✗	Multi-lang (2)
VCMATCH Vulnerability Patch Search	1,669	✓	✗	✗	Multi-lang (3)
GitPatchDB Vulnerability Patch Search	12,629	✓	✓ Interprocedural	✓	Multi-lang (10)

In contrast, GITPATCHDB is designed to address these limitations by integrating rich semantics of both vulnerabilities and patches with automatic scalable interprocedural slicing and pointer analysis across multiple languages. Prior literature (Smaragdakis et al., 2015; Hind & Pioli, 2000) highlights the critical role of pointer analysis in capturing data and control dependencies, while code property graphs (CPGs) (Yamaguchi et al., 2014) have proven effective for interprocedural vulnerability detection. As Ponta et al. (Ponta et al., 2019) note that existing CVE-to-patch mappings suffer from link deterioration and metadata inconsistencies, limiting the reliability of existing resources, GITPATCHDB offers a semantic-rich dataset that includes CVE descriptions, commit messages, and program slices, etc., rather than unstable links. To the best of our knowledge, GITPATCHDB is the first dataset to provide end-to-end CVE-to-patch search with interprocedural semantics, complete CVE metadata, and retrieval-oriented code embeddings, filling a critical gap in vulnerability patch analysis research.

We now turn to related work in contrastive learning, the core technique underlying CNPP and essential for enabling cross-modal vulnerability patch search.

Contrastive Learning for Cross-Modal Alignment. Inspired by recent advances in multimodal learning (e.g., CLIP (Radford et al., 2021), SimCLR (Chen et al., 2020), SimCSE (Gao et al., 2021)), CNPP employs contrastive pretraining to align textual vulnerability descriptions and source-level code patches in a shared semantic space. Gui et al. (Gui et al., 2022) demonstrated the efficacy of contrastive learning for vulnerability detection, while Allamanis et al. (Allamanis et al., 2021)showed its potential for code representation. Our work extends these insights to CVE-to-patch retrieval, addressing the semantic gaps observed in previous vulnerability datasets (Zhou et al., 2019).

3 THE GITPATCHDB DATASET

3.1 GITPATCHDB OVERVIEW

We introduce GITPATCHDB, a large-scale dataset that links software vulnerability reports, e.g., CVEs, to their corresponding code patches. The dataset contains 14,575 total samples, of which the positive subset comprises 12,629 CVE-commit pairs spanning 2010-2023 and drawn from 3,071 distinct GitHub/GitLab/SVN open-source program code repositories, with its code repository size statistics summarized in Table 2.

Table 2: GITPATCHDB repository size distribution.

Size Interval (KB)	Repository Count
(0, 1,000]	636
(1,000, 5,000]	690
(5,000, 20,000]	655
(20,000, 50,000]	433
(50,000, 100,000]	268
(100,000, 250,000]	163
(250,000, 500,000]	89
(500,000, 1,000,000]	37
(1,000,000, 2,500,000]	13
(2,500,000, 5,444,607]	11

Dataset samples of GITPATCHDB comprise two complementary components: *CVE side* and *patch side*: (i) *CVE side* includes natural language descriptions of the vulnerability, *i.e.,* CVE identifier,

[4]The GraphSPD dataset does not explicitly report the total number of CVE-linked commits. The cited 5,121 commits is an upper bound by merging SPI-DB and PatchDB, which GraphSPD extends for greater diversity.

Algorithm 1: On-the-fly co-analysis of program slicing and pointer analysis.

Function EnhancedDynamicSlicing*(P)*:
Input : Program P with source files
Output : R: Pointer-aware, context-sensitive slice
`/* Step 1: Graph Construction (AST, CFG, DFG) */`
$(G_{\text{AST}}, G_{\text{CFG}}, G_{\text{DFG}}) \leftarrow \text{BuildGraphs}(P)$
`/* Initialize slicing criteria and analysis state */`
$\Delta \leftarrow \text{GetModifiedLines}(P)$ `// Patch diff lines (e.g., deleted)`
$W \leftarrow \text{MapLinesToNodes}(G_{\text{DFG}}, \Delta)$ `// Initial slice worklist`
$\Pi \leftarrow \emptyset$ `// Points-to map: Var ↦ HeapLoc`
$R \leftarrow \emptyset$, `converged` $\leftarrow$ False
while $\neg$*converged* **do**
  `/* Step 2: Forward and Backward Interprocedural Slicing */`
  $S' \leftarrow \emptyset$
  foreach $v \in W$ **do** `// v: node in the DFG from diff`
    foreach $d \in \{fwd, bwd\}$ **do** `// d: slice direction`
      $S' \leftarrow S' \cup \text{InterproceduralSlice}(G_{\text{DFG}}, v, \Pi, d)$
  `/* Step 3: Flow-Sensitive Pointer Analysis */`
  $\Pi_{\text{new}} \leftarrow \text{FlowSensitiveAndersen}(G_{\text{CFG}}, \Pi, \Delta)$
  `/* Step 4: Graph Update and Convergence Check */`
  if $\text{CFGChanged}(G_{CFG}, \Pi_{new})$ **then**
    $(G_{\text{CFG}}, G_{\text{DFG}}) \leftarrow \text{UpdateGraphs}(G_{\text{CFG}}, \Pi_{\text{new}})$
    $\text{UpdateDefUseChains}(G_{\text{DFG}}, \Pi_{\text{new}})$
    $W \leftarrow \text{GetAffectedNodes}(G_{\text{DFG}}, S')$
  if $(\Pi_{new}, S') = (\Pi, R)$ **then**
    `converged` $\leftarrow$ True; `// Converged: alias and slice unchanged`
    $R \leftarrow R \cup S'$
  else
    $\Pi \leftarrow \Pi_{\text{new}}$, $R \leftarrow R \cup S'$,
    $W \leftarrow W \cup \text{GetNewWorkItems}(G_{\text{DFG}}, \Pi_{\text{new}})$
return R

CVE detailed description, and CVE references, e.g., CWE category and CVSS severity score. (ii) *Patch side* provides the context and technical details of the fix, including commit message, commit diff that captures the exact code changes, and interprocedural program slice, which enriches the code semantics by incorporating relevant control and data dependencies across function boundaries.

3.2 Program Analyses in GitPatchDB Curation

3.2.1 Motivation

The aforementioned interprocedural slices extend the patch context, enabling more comprehensive and accurate analysis of both the vulnerability and its remediation. This requires automatic program analysis techniques of program slicing and pointer analysis, as motivated in the following:

Motivating example. CVE-2019-17498 is an integer overflow vulnerability in `libssh2` [5].

Program slicing. The commit message of CVE-2019-17498 vulnerability, *"packet.c: improve message parsing"*, obscures the nature of the bug. Thus, only commit diff rarely captures the full semantic footprint of a vulnerability. Code diffs reflect syntactic edits, but omit surrounding context such as alias-resolved variables, call chains, and implicit program flows. Applying interprocedural slicing to the deleted lines reveals that the overflow arose from parsing an attacker-controlled length field (via `_libssh2_ntohu32`), followed by an insufficient bounds check (`if (len < datalen - 13)`). The forward slice shows this unchecked value propagates into the disconnect logic, risking an out-of-bounds read. By tracing both data and control dependencies across function boundaries, the slicing reconstructs a precise narrative of what was vulnerable and how it was fixed—bridging the abstraction gap between the CVE and the source-level patch.

Pointer analysis. Traditional program slicing struggles with real-world code due to pointer aliasing and interprocedural flows (Weiser, 1981; Tip, 1994; Lhoták & Hendren, 2003; Pearce et al., 2007; Sridharan & Bodík, 2006). Slicing directly from deleted lines without pointer analysis often fails to capture indirect flows (e.g., `*p = buf`) or shared dependencies across procedure boundaries. To recover full semantic context, we embed flow-sensitive pointer analysis into an interprocedural slicing pipeline and iterate to convergence. The process alternates between pointer-aware def-use slicing and pointer resolution via a customized Andersen-style analysis.

[5] The movitvating example details can be found in Appendix B

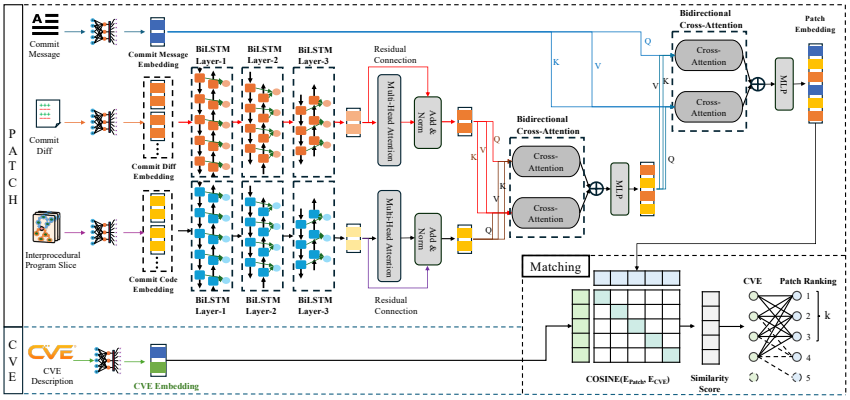

Figure 1: Overview of GITPATCHDB and downstream task CNPP.

3.2.2 ANALYSES DETAILS

Program slicing is a program analysis technique that extracts the subset of program statements that affect (backward slice) or are affected by (forward slice) a given slicing criterion such as a variable or line of code (Weiser, 1981; Tip, 1994).

Pointer analysis is another program analysis technique that analyzes the set of abstract locations each pointer variable may point to (Møller & Schwartzbach, 2023). With the knowledge of such aliasing information, it can produce useful dataflow and control flow analysis results.

We listed our on-the-fly co-analysis of program slicing and pointer analysis in Algorithm 1. In a nutshell, this process involves four stages: Step 1. On-the-Fly Graph Construction (Line 2); Step 2. Pointer-Aware Def-Use Slicing (Lines 8-11); Step 3. Flow-Sensitive Pointer Analysis (Line 12); Step 4. Convergence and fixed-point check (Lines 13-22). This algorithm offers three salient features [6]:

Interprocedural program slicing. In our work, we adopt a static, context-sensitive interprocedural slicing approach, enhanced with pointer analysis to resolve indirect dependencies. To the best of our knowledge, we are the first interprocedural slicing approach in patch analysis datasets, as listed in Table 1.

Flow-sensitive pointer analysis. Classic pointer analysis such as Andersen's pointer analysis (Andersen & Lee, 2005) is flow-insensitive, which computes over-approximated, global alias sets without tracking control-flow order and can introduce spurious def-use edges. In our work, we design a flow-sensitive pointer analysis to precisely reason about the aliasing relationships between program variables. To the best of our knowledge, we are the first pointer analysis adopted in patch analysis datasets, as listed in Table 1.

On-the-fly co-analysis of program slicing and pointer analysis. In order to analyze vulnerable programs at scale in the wild real-world cases, GITPATCHDB performs co-analysis of program slicing and pointer analysis on the current program slice on-the-fly and extends the analysis scope gradually. This design allows GITPATCHDB to scale slicing across 12,000 patches without sacrificing semantic precision or completeness.

4 METHOD

Our approach, CNPP, builds on the GITPATCHDB dataset to learn semantic alignment between vulnerability descriptions and their corresponding code patches. We address two key challenges: (1) **embedding scalability.** Patch code diffs and alias-aware program slices often exceed 10K tokens, making raw representations inefficient, noisy, and even infeasible, and (2) **cross-modal matching.** Natural language CVE descriptions must be effectively aligned with programming language written patches. We tackle these challenges via using hierarchical attention for compact, context-aware code embeddings and contrastive learning to train a shared embedding space across modalities.

4.1 CROSS-MODAL EMBEDDING

Vulnerability patch retrieval requires aligning heterogeneous information sources: natural language descriptions, commit messages, diffs, and interprocedural code slices. However, these inputs can be extremely long: diffs in our dataset exceed 819k tokens, and static slicing generates over 1.3M

[6]The algorithm details of our program analysis can be found in Appendix C.

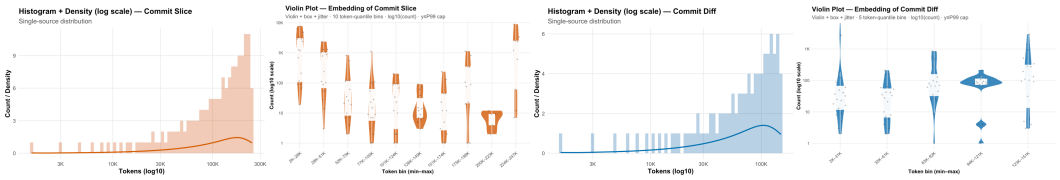

Figure 2: Token-length Distribution. From left: slice log-hist, slice violin, diffs log-hist, and diffs violin. Plots show that diffs/slices have a heavier tail and need larger context windows.

tokens. Existing large pretrained models like `text-embedding-ada-002` (OpenAI, 2023a;b) offer strong semantic encoding with up to 8192 tokens per pass, but this capacity remains insufficient for processing real-world patches at scale.

Shared latent space. We design a unified representation framework that encodes all modalities into a shared semantic space $\mathbb{R}^d$. Given an input consisting, we obtain their token-level embeddings as: a CVE description D ($\mathbf{E}_{\text{CVE}} = f_\theta(D)$), commit message M ($\mathbf{E}_{\text{M}} = f_\theta(M)$), diff D_f ($\mathbf{E}_{D_f} = \{f_\theta(d_t)\}_{t=1}^{T}$), and pointer-aware slice S ($\mathbf{E}_S = \{f_\theta(s_t)\}_{t=1}^{U}$), where $f_\theta(\cdot)$ denotes the token encoder, and T and U represent the sequence lengths of the diff and slice. These produce high-dimensional matrices $\mathbf{E}_{D_f} \in \mathbb{R}^{T\times d}$ and $\mathbf{E}_S \in \mathbb{R}^{U\times d}$, capturing fine-grained semantic signals across modalities.

Directly processing these large embeddings is computationally infeasible. To address this, we apply hierarchical sequence dimensionality reduction (see Section 4.2), which condenses these token-level representations into dense, semantic-rich vectors. This allows our method to scale to large patches while preserving essential context for cross-modal alignment. By embedding all modalities into a unified space, our framework eliminates the need for specialized encoders, enabling efficient and generalizable patch retrieval across diverse input types.

4.2 Patch Encoder

Processing vulnerability patches involving large diffs D_f and pointer-aware slices S present severe scalability challenges. Figure 2 shows that inputs are overwhelmingly long: around 95% of slices require ∼235k tokens and around 95% of diffs with ∼143k, far beyond conventional 4-32k context windows. This motivates the need for a hierarchical dimensionality reduction mechanism to distill these long sequences into fixed-size embeddings while preserving vulnerability-relevant semantics.

Hierarchical sequence dimensionality reduction. Given token-level embeddings for the diff and slice, we apply a three-layer residual-enhanced BiLSTM to reduce the sequence dimensionality:

$$\mathbf{h}_t^{(l)} = \text{BiLSTM}^{(l)}(\mathbf{h}_t^{(l-1)}, \mathbf{h}_{t-1}^{(l)}) + \mathbf{h}_t^{(l-1)}, \quad l \in \{1,2,3\}, \tag{1}$$

with $\mathbf{h}_t^{(0)} = f_\theta(x_t)$. This yields dimension-reduced hidden states for both D_f and S.

We further apply *context-gated attention* to summarize the sequence into a single vector (Bahdanau et al., 2016; Vaswani et al., 2017). This mechanism computes token-level attention scores α_t by comparing each token $\mathbf{h}_t$ with the overall sequence context $\overline{\mathbf{h}}$, defined as the mean of all token embeddings. Specifically:

$$\alpha_t = \frac{\exp(\mathbf{w}^\top \tanh(\mathbf{W}_c[\mathbf{h}_t \oplus \overline{\mathbf{h}}]))}{\sum_{t'=1}^{L}\exp(\cdot)}, \quad \overline{\mathbf{h}} = \frac{1}{L}\sum_{t=1}^{L}\mathbf{h}_t, \tag{2}$$

where $\oplus$ denotes vector concatenation. This formulation enables the model to assign higher importance to tokens that are not only locally salient but also globally relevant in the context of the entire sequence. We apply this attention mechanism to both the diff and slice sequences, resulting in fixed-size embeddings $\mathbf{z}_{D_f} \in \mathbb{R}^{1\times d}$ and $\mathbf{z}_S \in \mathbb{R}^{1\times d}$.

Bidirectional diff-slice fusion. To capture interactions between the diff and its execution context, we apply *bidirectional cross-attention* (Luong et al., 2015; Liu & Guo, 2019):

$$\mathbf{z}_{ds} = \underbrace{\text{softmax}\left(\frac{Q_d K_s^\top}{\sqrt{d_k}}\right)V_s}_{\text{Diff}\to\text{Slice}} + \underbrace{\text{softmax}\left(\frac{Q_s K_d^\top}{\sqrt{d_k}}\right)V_d}_{\text{Slice}\to\text{Diff}} \tag{3}$$

where $Q_d = \mathbf{z}_d\mathbf{W}_q^d$, $K_s, V_s = \mathbf{z}_s\mathbf{W}_k^s, \mathbf{z}_s\mathbf{W}_v^s$, and vice versa. This dual pathway models both how diffs affect surrounding slices context and how slices context validates the diff (Seo et al., 2018).

Cross-modal code-message fusion. We further refine the fused representation by applying *bidirectional cross-attention* between the fused patch embedding $\mathbf{z}_{ds} \in \mathbb{R}^{1\times d}$ and the commit message embedding $\mathbf{E}_M \in \mathbb{R}^{1\times d}$:

Table 3: Effectiveness of GITPATCHDB.

GITPATCHDB	Recall (%)														MRR
	Top 1	2	3	4	5	6	7	8	9	10	15	20	25	30	
95% CI	78.56	83.17	88.11	91.37	93.12	94.41	94.87	95.23	95.43	95.99	96.51	97.28	97.98	99.02	0.86

$$\mathbf{z}_{\text{patch}} = \text{CrossAttn}(\mathbf{z}_{ds}, \mathbf{E}_M) + \text{CrossAttn}(\mathbf{E}_M, \mathbf{z}_{ds}) \tag{4}$$

allowing mutual interaction between the patch context and the commit message, enabling the model to jointly attend over both sources of information without imposing dominance constraints.

4.3 CVE ENCODER

The CVE description is encoded simply using the pretrained ada-002 model: $\mathbf{z}_{\text{CVE}} = f_\theta(D)$. This ensures that $\mathbf{z}_{\text{CVE}}$ and $\mathbf{z}_{\text{patch}}$ are compatible in the same latent space, facilitating direct similarity computation.

4.4 CONTRASTIVE VULNERABILITY-PATCH ALIGNMENT

We train CNPP to align $\mathbf{z}_{\text{CVE}}$ and $\mathbf{z}_{\text{patch}}$ using the InfoNCE loss (van den Oord et al., 2019). Given a batch $\mathcal{B}$ of N CVE–patch pairs (D_i, P_i), we define:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{\exp\left(\frac{\langle \mathbf{z}_{\text{CVE}_i}, \mathbf{z}_{\text{patch}_i} \rangle}{\tau}\right)}{\sum_{j=1}^{N} \exp\left(\frac{\langle \mathbf{z}_{\text{CVE}_i}, \mathbf{z}_{\text{patch}_j} \rangle}{\tau}\right)} \tag{5}$$

where τ is a temperature hyperparameter and $\langle \cdot, \cdot \rangle$ denotes dot product. The objective encourages matching CVE–patch pairs to lie close in the embedding space while pushing apart non-matching ones. At inference time, we compute cosine similarity:

$$\text{sim}(\mathbf{z}_{\text{CVE}}, \mathbf{z}_{\text{patch}}) = \frac{\mathbf{z}_{\text{CVE}}^\top \mathbf{z}_{\text{patch}}}{\|\mathbf{z}_{\text{CVE}}\| \cdot \|\mathbf{z}_{\text{patch}}\|} \tag{6}$$

This enables efficient top-k retrieval for a given vulnerability. By contrastively training over multiple modalities (Chen et al., 2020; Radford et al., 2021) and enforcing scalable dimensionality reduction on long patch contexts, CNPP provides a principled and practical solution to vulnerability patch search.

5 EXPERIMENTS AND EVALUATION

5.1 IMPLEMENTATION AND EXPERIMENTAL SETUP

System and frameworks. GITPATCHDB is implemented in Scala and Python, integrating Joern [7] for multi-language intraprocedural CPG extraction. We extend it to interprocedural slicing using NetworkX [8]. Our multimodal encoder and contrastive training are implemented in PyTorch [9]. Experiments run on an Ubuntu 22.04 server with NVIDIA H100 GPUs and AMD EPYC CPUs.

Training configuration. We train our CNPP model for using the AdamW optimizer with a learning rate of 2×10^{-5}, batch size of 128, weight decay of 0.01, and gradient accumulation of 2 steps. A temperature parameter $\tau = 0.005$ is used for the InfoNCE loss, and early stopping is applied based on validation loss with a patience of 3 epochs.

Dataset composition and splits. Our total 14,575 dataset includes 12,629 CVE-linked patches. We follow a stratified 70/10/20 train/validation/test split at the patch level. To simulate realistic retrieval scenarios, we augment the test set with 1,946 randomly selected non-vulnerability (negative) patches, ensuring the model is evaluated against both true and irrelevant candidates. All code, data, and evaluation scripts are publicly available at `https://anonymous.4open.science/r/gitpatchdb-2EC7` to support full reproducibility.

5.2 RETRIEVAL EFFECTIVENESS

We evaluate CNPP on top-k patch retrieval, using CVE descriptions and pre-patch slices as queries. As shown in Table 3, GITPATCHDB achieves Accuracy@1 of 78.56% and MRR of 0.86, consistently outperforming baselines across all k (Table 4). All results are averaged over five independent runs with

[7] `https://joern.io/`
[8] `https://networkx.org/`
[9] `https://pytorch.org/`

different random seeds to account for variability due to model initialization and data sampling. We report mean performance along with 95% confidence intervals, computed as ± 1.96 times the standard error of the mean across five independent runs. We observe stable performance with low variance across all reported metrics, demonstrating the robustness of CNPP in realistic retrieval settings.

Error Analysis. We identify two primary sources of false positives: limitations in CVE descriptions and patch ambiguity. On the CVE side, vague or inconsistent descriptions in the National Vulnerability Database often lack technical precision, leading to superficial or misleading matches. On the patch side, errors are often caused by low-quality or sparse commit messages, large commits that combine security fixes with unrelated changes, or patches addressing multiple issues simultaneously. These issues hinder precise alignment and underscore the difficulty of bridging noisy natural language descriptions with complex code changes in real-world software repositories.

5.3 COMPARATIVE ANALYSIS

Table 4: Comparative analysis of GITPATCHDB.

Top N	PATCHSCOUT (manual keyword search)	VCMATCH (manual keyword search)	GITPATCHDB (automatic search)
	Accuracy (%)	Accuracy (%)	Accuracy (%)
1	57.58	63.13	**78.56**
2	60.60	74.24	**83.17**
3	66.68	77.75	**88.11**
4	71.70	81.30	**91.37**
5	78.73	84.39	**93.12**
6	81.92	87.99	**94.41**
7	83.22	89.38	**94.87**
8	84.14	90.01	**95.23**
9	86.88	90.47	**95.43**
10	87.73	92.22	**95.99**
15	88.68	94.61	**96.51**
20	89.01	96.02	**97.28**
25	89.51	97.44	**97.98**
30	90.17	98.29	**99.02**

Baselines. We compare GITPATCHDB against two state-of-the-art dataset-based approaches: PATCHSCOUT (Tan et al., 2021) and VCMATCH (Wang et al., 2022), both designed for CVE-to-patch research. We exclude TRACER (Xu et al., 2022), a web-based dynamic tracing tool that neither requires nor produces a reproducible dataset. Other works on patch classification (Liu et al., 2022; Wang et al., 2021; Fan et al., 2020; Tian et al., 2012; Wang et al., 2023; Zhou et al., 2021; 2023), patch presence testing (Bhandari et al., 2021; Dai et al., 2020), and vulnerability detection (Kim et al., 2017; Li et al., 2016) are out of scope, as they address different problem settings.

We evaluate GITPATCHDB, VCMATCH (Wang et al., 2022), and PATCHSCOUT (Tan et al., 2021) on a common test set. As shown in Table 4, GITPATCHDB achieves superior performance with Accuracy@1 of 78.56%, Accuracy@3 of 88.11%, Accuracy@10 of 95.99%, with nearly all correct patches retrieved within the top 30 results, substantially outperforming both baselines. Unlike VCMATCH and PATCHSCOUT, which rely on manual keyword engineering, GITPATCHDB operates fully automatically. We observe that baseline performance is lower than reported in their original publications, likely due to the inclusion of repositories with sparse commit messages in our test set, which is a common challenge in real-world projects like Wireshark and Apache Airflow. GITPATCHDB 's multimodal encoding of both textual and code-level inputs offers robust retrieval even when commit metadata is incomplete, providing a significant advantage over purely keyword-based approaches.

5.4 EMBEDDING ANALYSIS

Table 5: Additional baselines on GITPATCHDB with frozen encoders and learned projection heads.

Top N	Qwen3	SparseCoder	GITPATCHDB (CNPP)
1	73.68	58.33	**78.56**
2	**83.34**	69.44	83.17
3	**88.89**	73.15	88.11
4	**92.06**	78.74	91.37
5	92.95	81.23	**93.12**
6	93.48	83.33	**94.41**
7	94.33	86.55	**94.87**
8	95.13	87.96	**95.23**
9	**95.48**	87.97	95.43
10	95.95	89.09	**95.99**
15	96.39	90.12	**96.51**
20	97.04	90.33	**97.28**
25	97.25	90.31	**97.98**
30	98.61	92.48	**99.02**

In addition to our primary baselines, we evaluated recent non-OpenAI embedding models [10].

Setup. We evaluate *Qwen3-embedding* (Yang et al., 2025) and *SparseCoder* (Yang et al., 2024) as *frozen* encoders for CVE text and patch code. Lightweight modality-specific projection heads map encoder outputs into a shared space trained with a CLIP-style contrastive loss. Backbone embeddings remain fixed; only the projection heads are optimized.

Results. Without task-specific tuning, Qwen3 delivers competitive retrieval and closely approaches our supervised CNPP model; SparseCoder trails but remains strong, as listed in Table 5. These results indicate that our retrieval method is robust to the choice of embedding encoder, consistently maintaining strong performance. Note that SparseCoder underperforms due to objective mismatch (seq2seq + sparse activations) and English centric pretraining on our multilingual GITPATCHDB. Multilingual pretraining with contrastive fine-tuning should mitigate the discrepancy.

[10] Additional baselines evaluation result details can be found in Appendix E.

Table 6: Ablation study of GITPATCHDB.

Top N	Preprocessing Ablation Study (%)					Model Ablation Study (%)			GITPATCHDB (%)
	M	MD-TRU-AP	MD-CON-AP	MDC-TRU-AP	MDC-CON-AP	MLP-RDC	SUM-FUS	COS-SIM	
1	40.90 (37.66 ↓)	50.13 (28.43 ↓)	53.12 (25.44 ↓)	52.65 (25.91 ↓)	52.52 (26.04 ↓)	61.32 (17.24 ↓)	70.56 (8.00 ↓)	73.88 (4.68 ↓)	**78.56**
2	47.47 (38.66 ↓)	55.17 (28.00 ↓)	58.18 (24.99 ↓)	57.83 (25.34 ↓)	58.92 (24.25 ↓)	69.22 (13.95 ↓)	79.15 (4.02 ↓)	79.34 (3.83 ↓)	**83.17**
3	50.51 (39.66 ↓)	60.23 (27.88 ↓)	62.22 (25.89 ↓)	62.18 (25.93 ↓)	64.07 (24.04 ↓)	72.19 (15.92 ↓)	81.26 (6.85 ↓)	82.39 (5.72 ↓)	**88.11**
4	52.53 (40.66 ↓)	65.24 (26.13 ↓)	66.75 (24.62 ↓)	66.62 (24.75 ↓)	66.91 (24.46 ↓)	75.73 (15.64 ↓)	81.74 (9.63 ↓)	87.43 (3.94 ↓)	**91.37**
5	53.54 (41.66 ↓)	66.78 (26.34 ↓)	68.27 (24.85 ↓)	68.31 (24.81 ↓)	68.50 (24.62 ↓)	77.12 (16.00 ↓)	87.50 (5.62 ↓)	88.78 (4.34 ↓)	**93.12**
6	55.05 (42.66 ↓)	66.98 (27.43 ↓)	68.38 (26.03 ↓)	69.83 (24.58 ↓)	69.94 (24.47 ↓)	77.98 (16.43 ↓)	88.29 (6.12 ↓)	90.30 (4.11 ↓)	**94.41**
7	57.07 (43.66 ↓)	68.29 (26.58 ↓)	69.79 (25.08 ↓)	70.29 (24.58 ↓)	70.58 (24.29 ↓)	78.48 (16.39 ↓)	89.51 (5.36 ↓)	91.99 (2.88 ↓)	**94.87**
8	58.89 (44.66 ↓)	70.30 (24.93 ↓)	70.33 (24.90 ↓)	71.54 (23.69 ↓)	71.69 (23.54 ↓)	80.20 (15.03 ↓)	91.51 (3.72 ↓)	92.01 (3.22 ↓)	**95.23**
9	61.11 (45.66 ↓)	71.81 (23.62 ↓)	72.37 (23.06 ↓)	73.21 (22.22 ↓)	73.38 (22.05 ↓)	80.91 (14.52 ↓)	92.20 (3.23 ↓)	93.92 (1.51 ↓)	**95.43**
10	62.13 (46.66 ↓)	72.62 (23.37 ↓)	73.81 (22.18 ↓)	73.92 (22.07 ↓)	74.12 (21.87 ↓)	81.18 (14.81 ↓)	92.97 (3.02 ↓)	94.11 (1.88 ↓)	**95.99**
15	68.18 (47.66 ↓)	72.83 (23.68 ↓)	73.83 (22.68 ↓)	74.15 (22.36 ↓)	74.88 (21.63 ↓)	82.01 (14.50 ↓)	93.81 (2.70 ↓)	95.33 (1.18 ↓)	**96.51**
20	69.19 (48.66 ↓)	73.33 (23.95 ↓)	75.07 (22.21 ↓)	76.36 (20.92 ↓)	76.91 (20.37 ↓)	82.92 (14.36 ↓)	94.18 (3.10 ↓)	96.01 (1.27 ↓)	**97.28**
25	72.22 (49.66 ↓)	76.33 (21.65 ↓)	75.08 (22.90 ↓)	77.95 (20.03 ↓)	78.09 (19.89 ↓)	82.91 (15.07 ↓)	96.18 (1.80 ↓)	97.26 (0.72 ↓)	**97.98**
30	75.76 (50.66 ↓)	83.53 (15.49 ↓)	75.08 (23.94 ↓)	78.31 (20.71 ↓)	78.88 (20.14 ↓)	83.44 (15.58 ↓)	96.53 (2.49 ↓)	98.21 (0.81 ↓)	**99.02**

5.5 ABLATION ANALYSIS

Feature Ablation Analysis. To examine the effect of contextual richness on retrieval performance, we construct five model variants: (1) M: message, (2) MD-TRU-AP: message + truncated diff, (3) MD-CON-AP: message + concatenated diff, (4) MDC-TRU-AP: message + truncated diff + truncated sliced code, (5) MDC-CON-AP: message + concatenated diff + concatenated sliced code. Results indicate a monotonic performance improvement with the inclusion of additional modalities, reaching its highest with MDC-CON-AP. As listed in Table 6, results affirm that GITPATCHDB 's multimodal patch metadata offers complementary semantic context, spanning natural language and code structure and collectively enhancing retrieval accuracy.

Model Ablation Analysis. Our reference retrieval models:

- **MLP-RDC**: Replacing the residual BiLSTM encoder with a simple MLP leads to a significant **17.24%** degradation in Recall@1, underscoring the importance of sequential modeling for capturing token-level dependencies in diffs and slices.
- **SUM-FUS**: Substituting the cross-attention fusion mechanism with naive element-wise summation reduces Recall@1 by **8.00%**, validating that GitPatchDB provides sufficient multimodal diversity to benefit from structured attention-based fusion.
- **COS-SIM**: Replacing the contrastive InfoNCE objective with cosine similarity ranking incurs a minor **4.68%** drop in Recall@1, highlighting the dataset's compatibility with contrastive learning frameworks that leverage both positive and negative sample pairs.

Takeaways. These ablation studies highlight GITPATCHDB 's effectiveness in enabling controlled, systematic evaluation of multimodal retrieval systems. The feature ablation results demonstrate that GitPatchDB offers rich, layered patch metadata that benefits from multi-level contextual fusion. The model ablation demonstrates that GITPATCHDB introduces sufficient complexity to reveal the impact of architectural decisions. These findings position GITPATCHDB as a rigorous and informative benchmark for advancing research in multimodal vulnerability patch analysis.

5.6 SCALABILITY ANALYSIS

We further evaluate the scalability of GITPATCHDB by training on increasingly larger subsets of GitPatchDB. As shown in Figure 3, retrieval performance steadily improves with larger training sizes, achieving competitive accuracy with as few as 6,000 examples and continuing to improve beyond 8,000 samples. These results demonstrate GitPatchDB's capacity to fuel future advances in large-scale, data-driven vulnerability analysis.

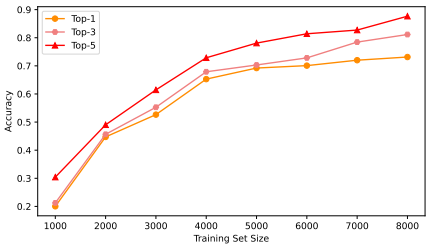

Figure 3: Scalability of GITPATCHDB.

6 CONCLUSIONS

We introduce GITPATCHDB, a large-scale, semantic-rich dataset for vulnerability patch analysis that offers a reliable foundation for linking CVEs to their patch fixes in machine learning research. Our proposed CNPP framework bridges natural language with progamming language code changes, significant outperforms prior methods and achieving state-of-the-art accuracy (95.99% accuracy). GitPatchDB's structured metadata and open availability address longstanding reproducibility barriers in vulnerability research, providing a scalable benchmark for training and evaluating machine learning models. Overall, by lowering the barrier to automatic patch analysis, GITPATCHDB advances both academic research and industrial practices in software security.

ETHICS STATEMENT

Data Collection. GITPATCHDB is built exclusively from publicly accessible security-patch commits that correspond to already disclosed CVEs across GitHub, GitLab, and SVN. No zero-day vulnerabilities or live exploits are included. All data collection complies with repository licenses and the terms of service of the respective platforms.

License. GITPATCHDB is released under the *Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA 4.0)* license, permitting non-commercial academic use.

Privacy. To safeguard privacy, personal identifiers (e.g., author names, email addresses, usernames) are removed. Only commit hashes and repository names are retained to ensure reproducibility.

User Agreement. GITPATCHDB is distributed with a *Responsible Use Policy* and a *User Agreement* that explicitly prohibit exploit development, unauthorized penetration testing, surveillance, or other harmful activities. Violations may result in permanent access revocation and potential legal action under the applicable license and jurisdictional laws.

Fairness. We acknowledge that GITPATCHDB may inherit natural representation biases from the underlying public CVE data. No behavioral, biometric, conversational, or telemetry data is collected or included.

REFERENCES

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876, 2021.

Lars Ole Andersen and Peter Lee. Program analysis and specialization for the c programming language. 2005. URL https://api.semanticscholar.org/CorpusID:20876553.

Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pp. 4945–4949. IEEE, 2016.

Guru Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 30–39, 2021.

David W Binkley and Keith Brian Gallagher. Program slicing. *Advances in computers*, 43:1–50, 1996.

Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pp. 396–407, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106285. URL https://doi.org/10.1145/3106237.3106285.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International Conference on Machine Learning*. PMLR, 2020.

Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. BScout: Direct whole patch presence test for java executables. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1147–1164. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/dai.

Debian Project. Debian: The universal operating system. https://www.debian.org, 2024.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th international conference on mining software repositories*, pp. 508–512, 2020.

Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33 (11):725–743, 2007.

Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pp. 131–138, 2006.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*, 2021.

Yi Gui, Yao Wan, Hongyu Zhang, Huifang Huang, Yulei Sui, Guandong Xu, Zhiyuan Shao, and Hai Jin. Cross-language binary-source code matching with intermediate representations. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 601–612. IEEE, 2022.

Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 232–241, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. doi: 10.1145/1134285.1134319. URL https://doi.org/10.1145/1134285.1134319.

Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pp. 226–238, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583792. doi: 10.1145/1480881.1480911. URL https://doi.org/10.1145/1480881.1480911.

Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 289–298, 2011. doi: 10.1109/CGO.2011.5764696.

Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 113–123, 2000.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 595–614, 2017. doi: 10.1109/SP.2017.62.

Sunghun Kim, Shivkumar Shivaji, and Jim Whitehead. A reflection on change classification in the era of large language models. *IEEE Transactions on Software Engineering*, 51(3):864–869, 2025. doi: 10.1109/TSE.2025.3539566.

Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In Görel Hedin (ed.), *Compiler Construction*, pp. 153–169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36579-2.

Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2201–2215, 2017.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*, pp. 201–213, 2016.

Gang Liu and Jiabao Guo. Bidirectional lstm with attention mechanism and convolutional layer for text classification. *Neurocomputing*, 337:325–338, 2019.

Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. CommitBART: A large pre-trained model for github commits. *arXiv preprint arXiv:2208.08100*, 2022.

V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, USA, 2005. USENIX Association.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

MvnRepository. Maven central repository. https://mvnrepository.com/repos/central, 2025.

Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*. May 2023. URL https://cs.au.dk/~amoeller/spa/.

National Institute of Standards and Technology. National Vulnerability Database. https://nvd.nist.gov, 2025.

Viet Hung Nguyen and Fabio Massacci. The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pp. 493–498, 2013.

OpenAI. Openai embeddings documentation, 2023a. URL https://platform.openai.com/docs/guides/embeddings.

OpenAI. Gpt-4 technical report, 2023b.

David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4–es, November 2007. ISSN 0164-0925. doi: 10.1145/1290520.1290524. URL https://doi.org/10.1145/1290520.1290524.

Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 383–387. IEEE, 2019.

Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 8748–8763. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/radford21a.html.

Red Hat, Inc. Red hat security. https://access.redhat.com/security/, 2024.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pp. 49–61, New York, NY, USA, 1995a. Association for Computing Machinery. ISBN 0897916921. doi: 10.1145/199448.199462. URL https://doi.org/10.1145/199448.199462.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pp. 49–61, New York, NY, USA, 1995b. Association for Computing Machinery. ISBN 0897916921. doi: 10.1145/199448.199462. URL https://doi.org/10.1145/199448.199462.

Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension, 2018. URL https://arxiv.org/abs/1611.01603.

Yannis Smaragdakis, George Balatsouras, et al. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pp. 387–400, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933204. doi: 10.1145/1133981.1134027. URL https://doi.org/10.1145/1133981.1134027.

Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 3282–3299, 2021.

The MITRE Corporation. Common Vulnerabilities and Exposures. https://cve.mitre.org/, 2025.

Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 386–396, 2012. doi: 10.1109/ICSE.2012.6227176.

Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 67–82, 2018.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding, 2019. URL https://arxiv.org/abs/1807.03748.

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

Shichao Wang, Yun Zhang, Liagfeng Bao, Xin Xia, and Minghui Wu. VCMatch: A ranking-based approach for automatic security patches localization for oss vulnerabilities. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 589–600, 2022. doi: 10.1109/SANER53432.2022.00076.

Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. GraphSPD: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2409–2426, 2023. doi: 10.1109/SP46215.2023.10179479.

Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. PatchDB: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 149–160, 2021. doi: 10.1109/DSN48987.2021.00030.

Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pp. 439–449. IEEE Press, 1981. ISBN 0897911466.

Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. Tracking patches for open source software vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 860–871, 2022.

Fabian Yamaguchi, Nicklaus Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, Berkeley, CA, USA, 2014. doi: 10.1109/SP.2014.44.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Xueqi Yang, Mariusz Jakubowski, Li Kang, Haojie Yu, and Tim Menzies. Sparsecoder: Advancing source code analysis with sparse attention and learned token pruning, 2024. URL https://arxiv.org/abs/2310.07109.

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

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. SPI: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–27, 2021.

A Appendix A: Empirical Study on Patches in Vulnerability Databases

To assess the current state of **patches** in vulnerability databases, such as Common Vulnerabilities and Exposures (CVE) and National Vulnerability Database (NVD), we conducted an empirical study on the following research questions:

- **RQ1: Patch Availability.** *What percentage of reported CVEs lack GitHub patch links?*
- **RQ2: Patch Functionality.** *What percentage of available patch links are non-functioning, and what are the primary causes?*
- **RQ3: Patch Recoverability.** *What percentage of CVEs with non-functioning patch links can be manually recovered?*

A.1 Patch Availability (RQ1)

We examined the availability of explicit patch commits in 193,448 CVE entries from 1999 to 2022 in National Vulnerability Database (NVD) (National Institute of Standards and Technology, 2025). To efficiently analyze this large volume of CVE entries, we first manually designed patch-matching regular expressions, and then automatically searched for vulnerability patches using these regular expressions within the CVE entries, focusing primarily on description section and references section. The results revealed that 30.99% of the CVEs contained explicit patch commits. The matching patches were primarily from open source repositories on GitHub, GitLab, BitBucket, SVN repositories, etc.

The aforementioned low percentage indicated limited public accessibility to vulnerable patches. This limited accessibility may lead to the following security consequences: (i) Systems that rely on software reuse may remain unpatched and susceptible to attacks. (ii) Security analysts and researchers might face difficulties in analyzing the vulnerability status. In this paper, we focus on searching for patches related to *open-source software vulnerabilities*.

A.2 Patch Functionality (RQ2)

Based on the $59,950$ CVE entries with explicit patch commits, we further automatically analyzed the functionality of these patch commits by cloning the repositories and checking out the commits. Patch commits with clonable repositories and check-out-able commits were considered functioning. The results revealed that only $35,164$ of these 59,950 patch commits were functioning, constituting 58.66%. This indicates that even among CVE entries with explicit patch commits, a significant portion (41.34%, 24,786) were non-functioning.

We categorized these non-functioning patch commits into three types: **Type-I: Repository relocation.** These patch commits lead to a 404 error due to the deletion or relocation of the repository; **Type-II: Branch relocation.** These patch commits display messages about the absence of the linked commit in any repository branch, often due to branch reorganization; **Type-III: Merged commits.** These patch commits are directed to merged pull requests containing multiple commits. Tracking patches for relocated repositories and branches remains necessary for legacy systems and those reusing code from these sources. While manual analysis of these cases is possible, it would be tedious and error-prone, highlighting the need for automatic patch search approaches.

A.3 Patch Recoverability (RQ3)

We further conducted patch recoverability analysis on non-existing patch commits and non-functioning patch commits.

- **Non-existing Patch Commits Recovery.** In this analysis, we randomly selected 800 CVE entries without associated patch commits, and examined whether the patch commits could be *manually* recovered. The methodology involved searching for patch commits and hosting repositories in vulnerability databases such as NVD (National Institute of Standards and Technology, 2025) and security advisories such as Debian (Debian Project, 2024) and Red Hat (Red Hat, Inc., 2024), searching for commits within the found repositories using commands tailored to commit messages, narrowing the search window to three months before and four months after each CVE report date,

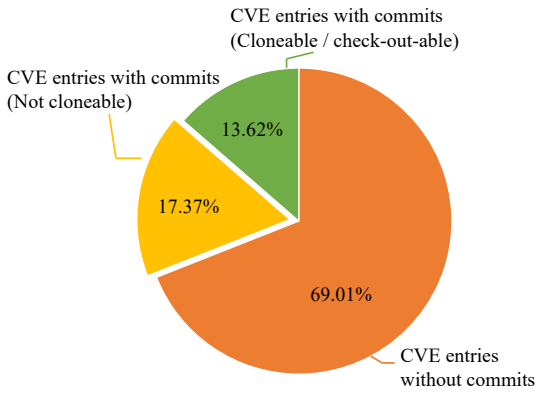

Figure 4: CVE database entries with explicit patch commits.

and rigorously reviewing potential patch commits. This meticulous validation process, conducted over 2.5 *months by* 3 *graduate students*, successfully associated approximately 53% of the CVEs that initially lacked patch commits.

- **Non-functioning Patch Commits Recovery.** In this analysis, we studies 765 non-functioning patch commits in TRACER (Xu et al., 2022) dataset, and examined whether the patch commits can be *manually* recovered. Our approches included: **Type-I**: Searching for the repository within online source code hosts such as GitHub to discover new repository URLs. **Type-II**: Searching within the project repository using git messages to locate new commit hash IDs corresponding to relocated patch commits. **Type-III**: Manually examining the merged pull requests, and cross-referencing information within the repository using author names of the pull requests and git messages to recover the correct patch commits. Ultimately, while we successfully restored 80% of the non-functioning patch commits, 152 CVEs with non-functioning patch commits could not be recovered. Of these, 103 were hosted on GitHub, and 49 were on the Apache SVN websites.

B APPENDIX B: MOTIVATING EXAMPLE AND ITS SAMPLE IN GITPATCHDB

```
// --- Backward Slice (Variables/Expressions Leading to `datalen - 13`) ---
// datalen: Length of the received SSH packet (unsigned integer)
if (datalen >= 5) {
  ...
  if (datalen >= 9) {
  --- message_len = _libssh2_ntohu32(data + 5); // Read from network data
  +++ _libssh2_get_u32(&buf, &reason);
  +++ _libssh2_get_string(&buf, &message, &message_len); // Validates buffer limits
  +++ _libssh2_get_string(&buf, &language, &language_len); // No manual arithmetic
    ...
    // Bounds check with potential integer overflow:
  --- if (message_len < datalen - 13) { // <-- PROBLEMATIC LINE
    // --- Forward Slice (Consequences of `datalen - 13`) ---
        message = (char *)data + 9; // Offset derived from `datalen`
        language_len = _libssh2_ntohu32(data + 9 + message_len);
        language = (char *)data + 9 + message_len + 4; // Arbitrary offset
        ...
```

Figure 5: Motivating example.

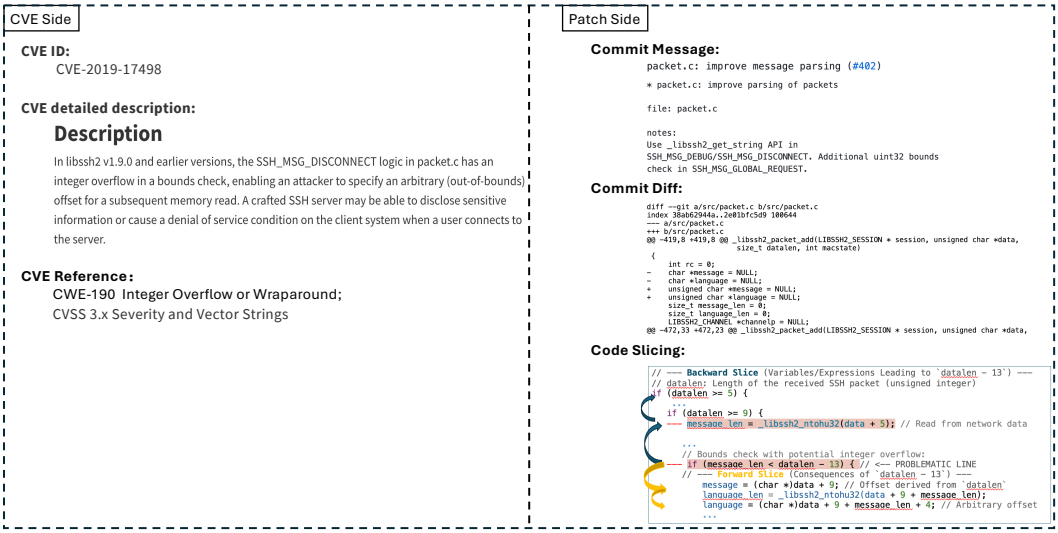

Figure 6: Motivating example sample in GITPATCHDB.

Motivating example. CVE-2019-17498 is an integer overflow vulnerability in `libssh2`, with details illustrated in Figure 5 and Figure 6. The patch for CVE-2019-17498, shown in Figure 5, includes both added (+++) and removed (---) lines from the code diff. To enrich the contextual understanding of this vulnerability, our program analysis in GITPATCHDB applies both forward slicing (orange arrows) and backward slicing (blue arrows), resulting in an interprocedural program slice that spans relevant control and data dependencies across function boundaries. Figure 5 presents the corresponding input sample in GITPATCHDB, which consists of two complementary components: the CVE side and the patch side.

C Appendix C: Program Analyses in GitPatchDB

C.1 Interprocedural Program Slicing

Program slicing primer. Program slicing extracts the subset of program statements that affect (backward slice) or are affected by (forward slice) a given slicing criterion such as a variable or line of code (Weiser, 1981; Tip, 1994). While early approaches focused on intraprocedural slicing, modern applications require interprocedural slicing to accurately track data and control dependencies across function boundaries (Reps et al., 1995a; Horwitz et al., 1990). These techniques extend classic program dependence graphs with interprocedural edges and call-return matching to preserve contextual semantics (Reps et al., 1995a). Additionally, robust slicing frameworks often integrate alias-aware def-use chains, which are particularly relevant when analyzing pointer-heavy languages like C/C++ (Binkley & Gallagher, 1996; Weiser, 1981). In our work, we adopt a static, context-sensitive interprocedural slicing approach, enhanced with pointer-aware analysis to resolve indirect dependencies during traversal.

Analysis motivation 1. Context richness. Slicing augments the diff with a contextual slice of surrounding code, which often spans functions and aliases, yielding an average 6.8× expansion in code relevant to the deleted lines. This captures how vulnerabilities propagate across program structure, enabling more precise matching with natural-language CVE descriptions. This motivates our *interprocedural program slicing*.

Analysis motivation 2. Analysis scalability. GitPatchDB performs convergence-based slicing at scale by alternating interprocedural slicing and flow-sensitive pointer refinement until a fixed point is reached. While effective, this co-analysis loop poses scalability challenges across thousands of patches. In order to analyze large-scale programs, prior works on scalable static analysis (Hackett et al., 2006; Hardekopf & Lin, 2009) often assume whole-program visibility or rely on coarse-grained modularity. Slicing engines such as PDG-based tools (Binkley & Gallagher, 1996; Weiser, 1981) or taint tracking systems (Livshits & Lam, 2005) typically operate over precomputed global graphs, with pointer analysis decoupled as a preprocessing phase, making them unsuitable for patch-level or real-time analysis. In contrast, this desire of analysis scalability motivates our *on-the-fly co-analysis of program slicing and pointer analysis*.

On-the-fly co-analysis of program slicing and pointer analysis. GitPatchDB introduces a fully *on-the-fly, patch-centric* pipeline that incrementally builds control and data flow graphs only for the code related to each patch. As shown in Algorithm 1, graph structures are updated dynamically (Line 14), and re-slicing is restricted to alias-affected nodes (Line 11), improving both precision and efficiency.

Co-analysis design details. Our convergence loop terminates when both program slicing and pointer analysis reach fix-points (Algorithm 1, Line 17), yielding a semantically complete, minimal patch slice. Unlike prior systems that decouple slicing and aliasing (Lhoták & Hendren, 2003), we design this co-analysis to eliminate both irrelevant statements and infeasible pointer aliases.

The process involves four stages:

Step 1. On-the-Fly Graph Construction. The co-analysis begins by parsing the program P into an abstract syntax tree (AST) and constructing the interprocedural control-flow graph (CFG) and def-use data-flow graph (DFG) (Horwitz et al., 1990). The CFG captures all control paths across function calls, while the DFG encodes definition-use chains, as well as aliasing pointers. By preserving call contexts in the co-analysis, we maintain precise analysis results across procedure boundaries (Reps et al., 1995a). We extract the changed lines Δ as the slicing criteria to initialize the slice worklist W.

Step 2. Pointer-Aware Def-Use Slicing. For each seed node $v \in W$, we compute a forward slice $fwd(v)$ and backward slice $bwd(v)$ over the DFG (Tip, 1994), traversing both def-use and control-dependence edges. Notably, this traversal is *pointer-aware*: when encountering indirect accesses (e.g., $\star\mathtt{p}$), we use the current points-to map Π to determine valid aliases (Lhoták & Hendren, 2003). For instance, if $p \mapsto \{x, y\}$, an assignment to $\star\mathtt{p}$ is treated as modifying both x and y. The union of all slices forms the updated slice set S', which is strictly more precise than traditional slicing due to alias disambiguation.

Algorithm 2: Flow-Sensitive Andersen's Pointer Analysis

Function FLOWSENSITIVEANDERSEN($\mathcal{G}_{CFG}$, $\mathcal{S}_0$, Δ)**:**
  Input : CFG $\mathcal{G}_{CFG}$, initial state $\mathcal{S}_0$, changed lines Δ
  Output : Updated pointer state $\mathcal{S}$
2 $\mathcal{Q} \leftarrow$ INITIALIZEWORKLIST(Δ) // BFS queue for CFG nodes
3 $\mathcal{S} \leftarrow \mathcal{S}_0$, VISITED $\leftarrow \emptyset$
4 **while** $\mathcal{Q} \neq \emptyset$ **do**
5 $n \leftarrow \mathcal{Q}$.DEQUEUE()
6 VISITED $\leftarrow$ VISITED $\cup \{n\}$
  /* Kill outdated relations before processing */
7 $\mathcal{S} \leftarrow \mathcal{S} -$ GETKILLSET(n)
8 **foreach** $s \in$ GETSTATEMENTS(n) **do**
9 $\mathcal{C}_s \leftarrow$ EXTRACTCONSTRAINTS(s)
10 **foreach** $c \in \mathcal{C}_s$ **do**
11 $(\mathcal{S}_{add}, \mathcal{S}_{kill}) \leftarrow$ PROCESSCONSTRAINT($c, \mathcal{S}$)
12 $\mathcal{S} \leftarrow (\mathcal{S} - \mathcal{S}_{kill}) \cup \mathcal{S}_{add}$
  /* Propagate to successors if state changed */
13 **if** $\mathcal{S} \neq$ GETPREVSTATE(n) **then**
14 **foreach** $succ \in$ GETSUCCESSORS(n) **do**
15 **if** $succ \notin$ VISITED **then**
16 $\mathcal{Q}$.ENQUEUE($succ$)
17 **return** $\mathcal{S}$
Function PROCESSCONSTRAINT(c, $\mathcal{S}$)**:**
  switch CONSTRAINTTYPE(c) **do**
  case *address-of* **do** // $p = \&q$
18 **return** $(\{pts(p) \subseteq \{q\}\}, \emptyset)$
  case *copy* **do** // $p = q$
20 **return** $(\{pts(p) \subseteq pts(q)\}, \emptyset)$
  case *store* **do** // $\star p = q$
22 **return** $(\{pts(\star p) \subseteq pts(q)\}, \emptyset)$

Step 3. Flow-Sensitive Pointer Analysis. We refine alias sets by applying a flow-sensitive Andersen-style points-to analysis (Sridharan & Bodík, 2006) restricted to nodes in the current slice S'. Unlike flow-insensitive pointer analyses (Hardekopf & Lin, 2011), this approach tracks pointer updates across control paths (Pearce et al., 2007), distinguishing aliasing pointers at different program points. The result is an updated points-to map Π', enabling us to prune infeasible aliasing relations and improve the accuracy of data dependencies in the next slicing iteration.

Step 4. Convergence and fixed-point check. We update the CFG and DFG based on the new pointer information Π_{new} (Algorithm 1, Lines 13-22). If control-flow or data dependencies change (e.g., resolving indirect calls), we rebuild edges and update the affected nodes. If neither the slice S' nor pointer state Π_{new} changed, the co-analysis converges and reaches a fix-point. Otherwise, we update the slicing result $R \leftarrow R \cup S'$ and repeat Phases 2-4. This loop continues until a fixed point is reached, yielding a minimal and precise slice R that captures both interprocedural control / data dependencies and pointer-based aliasing relations (Weiser, 1981; Reps et al., 1995b), critical for modeling vulnerability semantics.

This detailed description of our program analysis highlight core novelty of our co-analysis: an on-the-fly co-analysis that couples program slicing with pointer analysis to strengthen context richness and analysis scalability. This approach is especially well-suited to security patches, where flaws often span multiple files or functions and are mediated by aliasing or indirect control flow.

Co-analysis implementation details. In our co-analysis, GITPATCHDB incorporates *deleted lines* as slicing roots. These lines are recovered from Git parent commits and treated as entry points for backward slicing, providing richer semantic context. This integrates Git-based evolution tracking (Kim et al., 2025; Fluri et al., 2007) with static def-use semantics, a combination overlooked in earlier tools.

GITPATCHDB caches intermediate slices across patches with overlapping code, enabling parallelized ingestion and eliminating redundant recomputation. These design choices collectively allow us to scale slicing across 12,000 patches without sacrificing semantic precision or completeness.

GITPATCHDB stores backward and forward slices alongside patch metadata, enabling machine learning models to consume fine-grained semantic context for downstream tasks. Unlike raw diffs, these slices encode interpretable relationships between variables, control conditions, and pointer dereferences, supporting contrastive CVE-to-patch alignment and vulnerability pattern learning.

C.2 FLOW-SENSITIVE ANDERSEN'S POINTER ANALYSIS

In this section, we describe our flow-sensitive Andersen's pointer analysis in details. We propose a flow-sensitive variant of Andersen's inclusion-based pointer analysis (Andersen & Lee, 2005), as listed in Algorithm 2. Our flow-sensitive pointer analysis resolves constraints such as *address-of*, *copy*, and *store*, across the interprecedural control-flow graph (CFG) on-the-fly, starting from the original changed lines in the patch Δ (Line 1).

- Initialization (Lines 2-3): A worklist Q is initialized with affected CFG nodes based on Δ, and the pointer state S is initialized to S_0.
- Constraint Solving (Lines 4-12): For each node n dequeued from Q, outdated pointer relations are invalidated (Line 7). New constraints are extracted from statements in n (Lines 8-9) and processed (Line 10) using the PROCESSCONSTRAINT function (Lines 18-25).
- Constraint Types:
  - *Address-of* ($p := \&q$): Adds $\{q\}$ to PointsTo(p) (Line 21).
  - *Copy* ($p := q$): Propagates PointsTo(q) to PointsTo(p) (Line 23).
  - *Store* ($*p := q$): Updates all targets of PointsTo(p) with PointsTo(q) (Line 25).
- Constraint Propagation (Lines 13-16): If the pointer state changes, successors are enqueued for further processing.

Our flow- and context-sensitive pointer analysis balances precision and scalability, enabling fine-grained tracking of pointer relationships across control-flow and calling contexts. This makes it particularly suited for downstream tasks such as vulnerability detection, memory safety verification, and interprocedural slicing.

D APPENDIX D: STATISTICS OF OUR GITPATCHDB

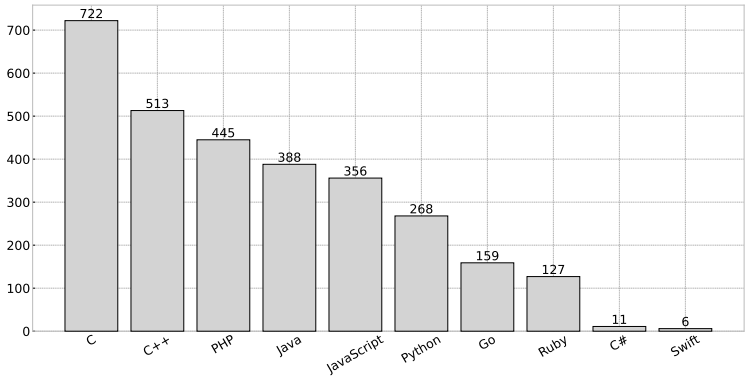

Figure 7: Programming languages distribution in our dataset.

Table 7: Top 10 repositories with most CVEs in our dataset.

Project	Category	Language	Description	URL	Count
Linux Kernel	Operating System	C	Operating System Kernel	http://git.kernel.org	1780
moodle	Education	PHP	E-learning platform	https://github.com/moodle/moodle	498
TensorFlow	AI/ML	Python	Machine learning framework	https://github.com/tensorflow/tensorflow	341
ChakraCore	Web	C++	JavaScript engine	https://github.com/chakra-core/ChakraCore.git	218
ImageMagick	Multimedia	C	Image processing software	https://github.com/ImageMagick/ImageMagick	171
WordPress	Web	PHP	Content management system	https://github.com/WordPress/WordPress	158
phpmyadmin	Management	PHP	Database management	https://github.com/phpmyadmin/phpmyadmin	138
tcpdump	Network	C	Network diagnostic tool	https://github.com/the-tcpdump-group/tcpdump	125
rails	Web	Ruby	Web application framework	https://github.com/rails/rails	119
vim	Editor	C	Text editor	https://github.com/vim/vim	117

The programming languages distribution and the top 10 most CVEs repositories in our dataset are illustrated in Figure 7 and Table 7, respectively. Note that C# and Swift appear only in the test set and were not included in training, reflecting the natural underlying distribution of the source CVE corpus during our collection window, where C#-based and Swift-based projects are comparatively rare. Nevertheless, our results show strong zero-shot cross-language generalization to these languages, achieving an Accuracy@10 of 95.99%.

E APPENDIX E: EMBEDDING BASELINES EVALUATION

Table 8: Additional baselines on GITPATCHDB with frozen encoders and learned projection heads.

Top N	Qwen3	SparseCoder	GITPATCHDB (CNPP)
1	73.68	58.33	**78.56**
2	**83.34**	69.44	83.17
3	**88.89**	73.15	88.11
4	**92.06**	78.74	91.37
5	92.95	81.23	**93.12**
6	93.48	83.33	**94.41**
7	94.33	86.55	**94.87**
8	95.13	87.96	**95.23**
9	**95.48**	87.97	95.43
10	95.95	89.09	**95.99**
15	96.39	90.12	**96.51**
20	97.04	90.33	**97.28**
25	97.25	90.31	**97.98**
30	98.61	92.48	**99.02**

Table 8 highlights the following key results:

- Qwen3 attains a Recall@10 of 95.95%, substantially outperforming SparseCoder (89.09%) and nearly matching our supervised CNPP model (95.99%).
- At Recall@30, CNPP achieves 99.02%, while Qwen3 and SparseCoder reach 98.61% and 92.48%, respectively.

E.1 SPARSECODER

Experimental Setup. SparseCoder (Yang et al., 2024) is designed and pre-trained for *sequence-to-sequence* generation tasks such as code summarization, as stated in the paper: "*We train all models for 10 epochs on the FunCom dataset*" (Section 4.2), and its architecture incorporates an encoder-decoder structure optimized for generating natural language summaries from code (Section 3.1).

In our contrastive retrieval framework, we therefore only utilized the encoder and extracted embeddings from the Top-K sparse activation layer (Section 3.3), which was originally intended for interpretability, not dense similarity matching.

Result Analysis. We attribute SparseCoder's relatively moderate performance to the following factors:

(i) SparseCoder is trained with a cross-entropy loss for decoder-based generation rather than a contrastive loss, making it misaligned with our retrieval objective, which relies on InfoNCE to align CVE descriptions with patch embeddings.

(ii) SparseCoder was trained exclusively on English-language corpora such as CodeSearchNet and FunCom (Section 4.1), and does not support multilingual inputs natively. Since GITPATCHDB includes CVE descriptions and commit messages in multiple natural languages, we had to translate non-English queries before encoding, introducing noise and additional limitations.

Despite these challenges, SparseCoder still achieved competitive results in our comprehensive evaluation. We believe future works with multilingual pretraining and contrastive fine-tuning can further improve its performance in retrieval tasks.

E.2 QWEN3

Experimental Setup. We applied Qwen3 as a frozen encoder for both CVE descriptions and code patches (via mean pooling and [CLS] token), followed by modality-specific projection heads f_θ and g_ϕ to embed inputs into a shared latent space. These heads were trained using a CLIP-style InfoNCE contrastive objective:

$$L_{CLIP} = -log\frac{exp(sim(f_\theta(z_{CVE}), g_\phi(z_patch))/\tau)}{\sum_j \exp(sim(f_\theta(z_{CVE}), g_\phi(z_j))/\tau)}$$

Result Analysis. Despite limited epochs and no encoder updates, Qwen3 achieved a Recall@10 of 95.95%, demonstrating its strong potential. With full contrastive fine-tuning and longer convergence,

we expect it to surpass baseline embeddings. In future work, we plan to fine-tune Qwen3 end-to-end using the following steps:

- Initialize from Qwen3-embedding as the encoder backbone.
- Freeze the lower transformer layers initially to stabilize gradients.
- Optimize the full system using our contrastive CLIP-style loss on CVE–patch pairs from GitPatchDB.