Feature Slice Matching for Precise Bug Detection

KE MA, Renmin University of China, China
JIANJUN HUANG*, Renmin University of China, China
WEI YOU, Renmin University of China, China
BIN LIANG, Renmin University of China, China
JINGZHENG WU, Institute of Software at Chinese Academy of Sciences, China
YANJUN WU, Institute of Software at Chinese Academy of Sciences, China
YUANJUN GONG, University of Trento, Italy

Measuring the function similarity to detect bugs is effective, but the statements unrelated to the bugs can impede the performance due to the noise interference. Suppressing the noise interference in existing works does not manage the tough job, i.e., eliminating the noise in the targets. In this paper, we propose Matus to mitigate the target noise for precise bug detection based on similarity measurement. Feature slices are extracted from both the buggy query and the targets to represent the semantic feature of (potential) bug logics. In particular, Matus guides the target slicing with the prior knowledge from the buggy code, in an end-to-end way to pinpoint the slicing criterion in the targets. All feature slices are embedded and compared based on the vector similarity. Buggy candidates are audited to confirm unknown bugs in the targets. Experiments show that Matus holds advantages in bug detection for real-world projects with acceptable efficiency. In total, Matus has spotted 31 unknown bugs in the Linux kernel. All of them have been confirmed by the kernel developers, and 11 have been assigned CVEs.

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

Additional Key Words and Phrases: Bug detection, Feature slice, Similarity measurement

ACM Reference Format:
Ke Ma, Jianjun Huang, Wei You, Bin Liang, Jingzheng Wu, Yanjun Wu, and Yuanjun Gong. 2026. Feature Slice Matching for Precise Bug Detection. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE052 (July 2026), 22 pages. https://doi.org/10.1145/3797080

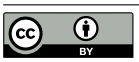
1 Introduction

Detecting bugs through code similarity measurement has proven to be effective [2, 17, 21]. Generally, code elements (i.e., *query*) related to a known bug are extracted and compared with the ones (i.e., *targets*) obtained from a code base of interest. High similarity between the query and a target will indicate a potential bug in the corresponding target function.

Directly measuring the similarity between functions is not always a good idea [55], as real-world functions usually contain not only the statements closely related to bugs but also many irrelevant

*Corresponding author

Authors' Contact Information: Ke Ma, Renmin University of China, Beijing, China, make@ruc.edu.cn; Jianjun Huang, Renmin University of China, Beijing, China, hjj@ruc.edu.cn; Wei You, Renmin University of China, Beijing, China, youwei@ruc.edu.cn; Bin Liang, Renmin University of China, Beijing, China, liangb@ruc.edu.cn; Jingzheng Wu, Institute of Software at Chinese Academy of Sciences, Beijing, China, jingzheng08@iscas.ac.cn; Yanjun Wu, Institute of Software at Chinese Academy of Sciences, Beijing, China, yanjun@iscas.ac.cn; Yuanjun Gong, University of Trento, Trento, Italy, yuanjun.gong@unitn.it.



ACM 2994-970X/2026/7-ARTFSE052
https://doi.org/10.1145/3797080

ones that may differ significantly across functions. Such noise statements can heavily impede the function-level similarity detection.

Recent studies tend to constrain the similarity measurement via various methods, but the core is to suppress the noise in the query via a slicing technique. For example, optimal vertex matching [55] and subgraph isomorphism decision [12] adopt heavyweight matching methods, shrinking the query to contain only the bug-related elements instead of matching the entire target functions. Slice-based graph matching [15] compares the similarity between slices related to input data, while FIRE [9] matches buggy/patched slices with all target slices that are not purposefully selected according to the known bug.

Though effective, the existing slice-based methods have explicit disadvantages. First, slicing only the query function is obviously affected by the noise in the target functions. In our experience, statement-by-statement matching can also emit lots of false matches [55], not to say a capability-restricted embedding model resolves a large target function to a small query slice [12]. Second, arbitrary slicing independent of the target bug can generate a large number of irrelevant slices. The influence of noise remains and probably affects the similarity measurement.

In this paper, we propose Matus to overcome the noise in both the query and targets for precisely detecting bugs based on similarity measurement. The query is sliced based on identified bug elements by Matus from the buggy/patched code. The targets are also sliced by Matus, to ensure each target function is represented accurately by a slice that maximally resembles the query slice. To achieve the goal, Matus leverages the knowledge of the known bug to direct the target slicing. More specifically, given a buggy function and its patched version, Matus extracts the statements and variables that are tightly closed to the bug. Using them as prior knowledge, Matus pinpoints the corresponding statements and variables in the target functions in an end-to-end way, with the help of similarity computation based on an encoder model. The pinpointed elements serve as the slicing criterion to guide target slicing. Because they are spotted via similarity measurement, the corresponding target slices are more likely to share similar semantic features with the query slice. Such slices are named *feature slices* in this paper. Feature slices are embedded into vectors to ease the measurement of similarity between the query and targets. The top-ranked candidates will be manually audited to confirm unknown bugs.

We have implemented a prototype of Matus. All embedding-related tasks utilize a fine-tuned pre-trained large code model, UniXcoder [13]. To improve efficiency, a coarse-grained function-level filtering is taken to screen out a small number of candidate functions from a large code base. Matus has been proved to be effective and scalable in bug detection. In total, 31 previously unknown bugs have been detected in the Linux kernel v6.4-rc2 and subsequently confirmed by the kernel developers. Eleven of them are assigned CVEs. Over a benchmark with 76 bug pairs, our method demonstrates an advantage over nine matching-based methods in discovering bugs in large real-world projects.

This paper makes the following contributions:

- We propose a method to mitigate the noise in similarity-measurement-based bug detection. The knowledge of a known bug is treated as a guide for pinpointing semantically similar code elements that are probably related to an unknown similar bug.
- We propose a novel approach to obtaining the critical target elements end-to-end, based on the masked embeddings from an encoder model. It has been proven to be effective.
- We design and implement Matus, and evaluate it on the Linux kernel. Thirty-one previously unknown bugs have been detected. Comparison with state-of-the-art (SOTA) tools demonstrates the effectiveness and efficiency of our method. The artifacts are publicly available at https://github.com/Kyew2r6qAe/MATUS.

2 Motivation

We use Figure 1 to motivate our technique, which contains (a) a known bug and (b) a similar new bug in the Linux kernel.

Fig. 1. Motivating example.

Figure 1(a) is responsible for registering a list of NTB client devices with the transport layer. Each *client_dev* is allocated and initialized before being registered at Line 20. If any registration fails, the current device is destroyed to avoid memory leaks (Line 22), and all registered devices are cleaned up at Line 30. The bug arises when the device is simply destroyed with `kfree`, which only deallocates the memory by Line 6 but does not release the device name allocated by `dev_set_name` at Line 15, i.e., leading to a memory leak. The fix replaces `kfree` with `put_device` (Line 23). The latter invokes a customized callback function `ntb_transport_client_release` (registered at Line 17) to release all memory chunks associated with the device.

In Figure 1(b), `locomo_init_one_child` initializes and registers a single child device on the LoCoMo bus. Similar to the process for registering a client device in Figure 1(a), this function contains the same memory leak bug if registration fails at Line 21, and the same fix applies. Identifying the bug alone would be challenging for auditors lacking sufficient domain expertise. Leveraging similarity-based matching techniques would help us recognize that the two functions implement similar semantics. However, the different code structures (loop *v.s.* non-loop structures, continuous *v.s.* intermittent operations on the devices) and device-specific operations (omitted lines) aggravate the function-level difference. The noise code makes the function difficult to stand out from hundreds of thousands of kernel functions when they are ranked based on their similarity to the left

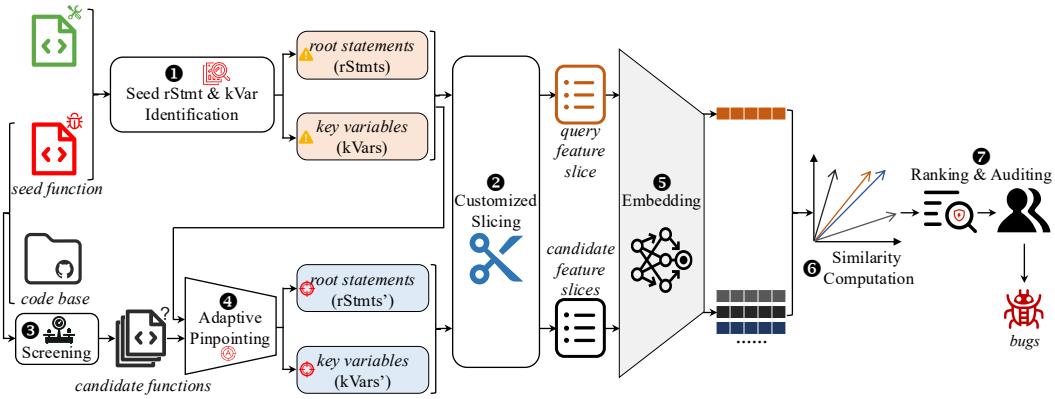


Fig. 2. The overall workflow of Matus.

one. In fact, when we embed the functions into vectors with a fine-tuned code embedding model UniXcoder [13] and compute the cosine similarity between vectors, `locomo_init_one_child` is only ranked the 885-*th* similar with `ntb_transport_register_client_dev`. Such a low ranking is far beyond the scope of a normal manual audit. To make the buggy function come into attention of the auditors, we need a method that can confidently enhance the similarity between the two functions and promote the ranking to a scope (e.g., within the top 10) acceptable to auditors.

We present Matus to highlight the bug based on similarity detection of bug-related slices that are embedded with pre-trained code models. According to the known bug and its fix, Matus first identifies the variables (labelled as kVar1 and kVar2) and the statements (rStmt1 and rStmt2), which are immediately closely related to the bug. Then a customized slicing extracts other related statements, forming a feature slice (*query*) of the bug, i.e., the highlighted lines in Figure 1(a). Facing the whole Linux kernel, Matus leverages an encoder model to discover the counterpart variables and statements (i.e., the labelled ones in Figure 1(b)) in each target function. Target feature slices potentially possessing semantically similar operations with the query are extracted, e.g., the highlighted lines in Figure 1(b). An encoder model embeds all the feature slices, and Matus ranks the target functions based on the cosine similarity between target slices and the query. By this means, Figure 1(b) is ranked first and can be quickly audited to identify the bug.

3 Methodology

3.1 Overview

Figure 2 shows the workflow of Matus. Given a seed function with a known bug and its fixed version, Matus heuristically identifies two kinds of important elements (❶), i.e., *key variables* and *root statements*, which are then used to extract a customized slice as the *query* (❷). We call the slice a *feature slice* as it is highly related to and illustrates the feature of the bug. The target functions in the code base are filtered based on their coarse-grained similarity to the buggy seed function (❸), leaving a group of candidate functions for fine-grained similarity detection. With the help of an embedding model, the counterpart key variables and root statements in each candidate function are acquired (❹). Using them as the slicing criterion, the candidate feature slices are extracted (❷). All feature slices are embedded into vectors through an encoder model (❺), and the cosine similarity between each candidate slice and the query is computed (❻). All candidates are ranked according to the similarity for manual auditing to discover potential bugs (❼).

3.2 Root Statements and Key Variables

In this section, we informally define the root statements and key variables that are critical in Figure 2 for extracting the query feature slice. The ones in the seed function also guide the discovery of the slicing criterion in candidate functions.

We define a *root statement* (shortened as *rStmt*) as a statement that can immediately trigger the bug in the buggy code. It involves two cases. First, the statement itself is buggy and hence modified or removed in the fixed version. Line 22 in Figure 1(a) is an example, which partially releases the allocated memory, resulting in a leak. Second, the statement immediately reaches the bug point through the control flow when a certain condition is satisfied. Line 20 in Figure 1(a) is an example. If the device registration fails, the immediate error handling runs into a bug.

A *key variable* (shortened as *kVar*) is an operand in the root statement, which typically holds an object or a value that is highly related to the causation of the bug. In this study, it can be either a simple variable or a compound expression, as an argument or the resultant value. In Figure 1(a), *client_dev* and *dev* are two key variables, but *rc* is not considered such because it only indicates a condition and does not involve an object that the buggy flow operates.

In a target (or candidate) function, we refer to the elements pinpointed using the seed rStmts and kVars as candidate rStmts and kVars, as they may exhibit semantic similarity to their seed counterparts and be involved in potentially similar bugs.

3.3 Identifying Seed Root Statements and Key Variables

According to Figure 2, identifying the seed rStmts and kVars is a critical step. Section 3.2 has presented a conceptual guide to the identification, but it is not suitable for implementing an automated tool. For example, a skilled human reviewer may understand why some variables are unnecessary to be kept as kVars, but the sense cannot be well automated. To enhance the efficiency of automated identification, we adopt an automated approach inspired by [5] to recognize the kVars and then heuristically pinpoint the rStmts.

Key variable identification. We adopt the term frequency [5] to calculate the importance of potential kVars, which measures the frequency of a variable's occurrence within the patch. A higher count indicates the variable is more critical to the bug. Any addition, deletion or modification will increase the frequency of the involved variables. The ones with the highest frequency and existing in the buggy code will be treated as kVars. When more than one variable reaches the same highest frequency, all of them are taken into account. Note that, if a variable appears in the *return* statement, either in the buggy or the fixed code, it is eliminated from the candidates. Such a rule is included because we find that in large code bases like the Linux kernel, a bug patch may contain multiple similar blocks for error handling and every block returns the same variable indicating a failure of the execution. Figure 1(a) is a perfect example. We can easily highlight *dev* and *client_dev* as two kVars as there are only two variables in the patch (Lines 22 ~ 23) and both have the same frequency (i.e., 1).

Root statement screening. Based on the identified kVars, we define two rules, corresponding to the two cases in Section 3.2, to collect the rStmts starting from the patch statements.

First, if a kVar var_k appears in a deleted/modified statement, it is directly collected, e.g., Line 22 in Figure 1(a). Second, if var_k appears in an inserted statement that is not in the buggy code, we find another correlated statement $stmt_r$ as the corresponding rStmt. Our heuristic picks $stmt_r$ that contains var_k and appears before and in the closest proximity to the inserted statement. Take Figure 1(a) as an example. With var_k = *dev*, we can determine that $stmt_r$ = Line 20 which uses *dev* and has only one step to Line 21.

3.4 Pinpointing Candidate Root Statements and Key Variables

Candidate rStmts and kVars in target functions are crucial to extract feature slices that are potentially semantically similar to the query. Provided a seed rStmt s_{stmt} and kVar s_{var}, we propose an end-to-end approach to identifying the candidate rStmt c_{stmt} and kVar c_{var} simultaneously in a target function F. The approach harnesses the power of a transformer-based encoder model, which learns the embedding representation of each token by taking all other tokens in the input into consideration. In other words, the embedding of a specific token has already comprised the context information. Inspired by it, our approach determines c_{stmt} and c_{var} in F with Equation 1, where the arg max is taken over all pairs $(stmt_i, var_j)$. Each pair contains a statement $stmt_i$ in F and a variable occurrence var_j in the statement. $\mathbf{v}_x$ is the embedding of x, and $sim()$ computes vector similarity.

$$c_{stmt},\ c_{var} = \underset{\forall (stmt_i, var_j)}{\arg\max}\ sim(\mathbf{v}_{s_{var}}, \mathbf{v}_{var_j}) \tag{1}$$

Note that, each var_j is constrained by $stmt_i$ where the variable appears. That means, if a variable appears in different statements, it will possess different embeddings associated with the occurrences. Moreover, as the position matters in transformer-based encoder models, a variable occurring twice in a statement will emit two different embeddings. We can leverage the characteristics to distinguish c_{var} from all the other occurrences of the same variable, and at the same time pinpoint the corresponding c_{stmt}.

There are two ways to obtain the embedding of a specific *var* in a *stmt*, as shown in Equations 2 and 3, where Tok: $C \rightarrow T$ is the tokenizer that splits a given text C into a sequence of tokens T and Enc: $T \rightarrow (R^d)^n$ is the encoder model which encodes each among the n tokens into a d-dimensional floating vector.

$$\mathbf{v}_{var} = \sum_{j=i}^{k} \mathbf{v}_{t_j},\ \text{with}\ \text{Enc}(\text{Tok}(stmt)) \rightarrow \cdots, \mathbf{v}_{t_j}, \cdots\ \textbf{and}\ \text{Tok}(var) \rightarrow t_i, \cdots, t_k \tag{2}$$

$$\mathbf{v}_{var} = \mathbf{v}_{\texttt{[MASK]}},\ \text{with}\ \text{Enc}(\cdots, t_{i-1}, \texttt{[MASK]}, t_{k+1}, \cdots) \rightarrow \cdots, \mathbf{v}_{t_{i-1}}, \mathbf{v}_{\texttt{[MASK]}}, \mathbf{v}_{t_{k+1}}, \cdots \tag{3}$$

Equation 2 aggregates the embeddings of the tokens corresponding to *var*, utilizing both the context information and the lexical components of the variable. Equation 3 replaces the variable with a special token `[MASK]`, leveraging the filling-the-blank ability of the model to infer the representation of *var*. Surprisingly, the lexical information does not provide much help and in Section 4.5.2, the advantage of the second method is illustrated compared to the first. In this study, we take Equation 3 to compute the variable embeddings for both the seed and the candidate kVars.

Considering that we aim to obtain a slice with multiple statements that can represent potential error-prone semantics of some objects of interest, Matus excludes the variables occurring only once (apart from the declarations) throughout the given function, i.e., they are not considered for candidate kVars and hence not masked in this step.

Example. Figure 3 illustrates the workflow. With the identified rStmt and kVar in (a), Matus masks the kVar and obtains its embedding $\mathbf{v}_{kv}$ (in (c)). For the variables in a candidate function as in (b), we mask their occurrences one by one. Take Line 7 in (b) as an example, which contains two variables. We build two masked statements as (d) shows, each with only one `[MASK]` corresponding to a specific variable, and compute the variables' embeddings, i.e., $\mathbf{v}_{err@7}$ and $\mathbf{v}_{bus->dev@7}$. Applying Equation 1 to all the variables' embeddings, Matus determines the candidate rStmt and kVar as in (e), i.e., Line 7 and *bus->dev* in (b), respectively. Following [27, 57], we consider *bus->dev* as a variable.

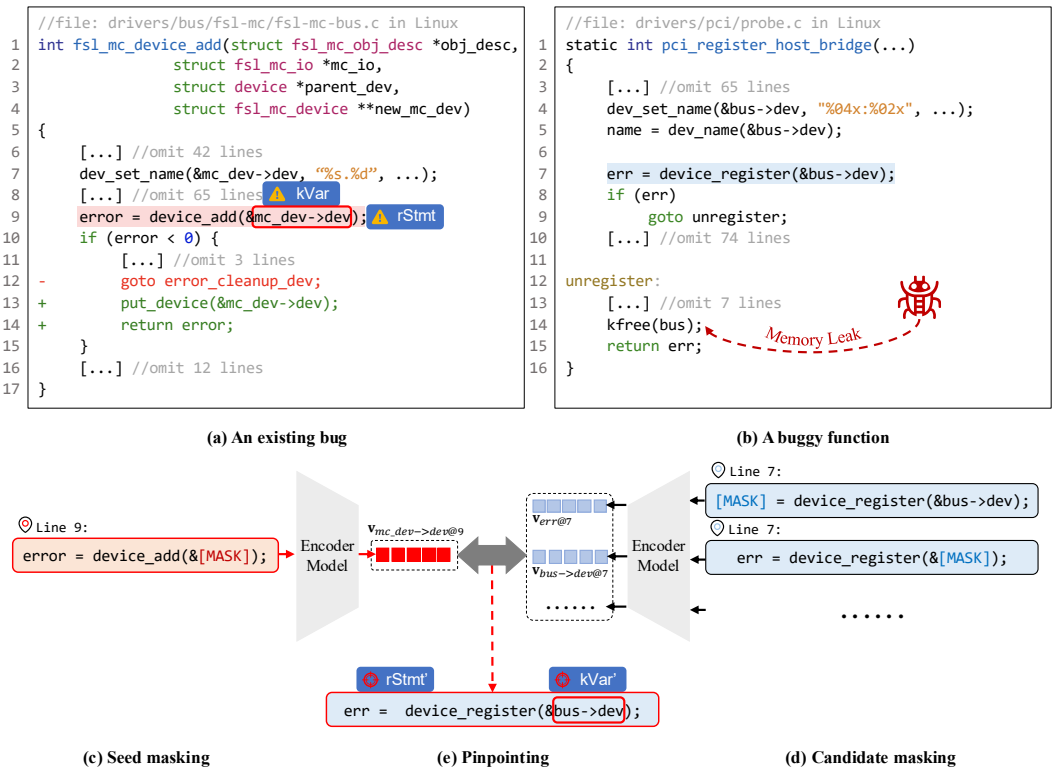


Fig. 3. An example illustrating how to pinpoint the candidate root statement and key variable in a candidate function based on their seed counterparts.

3.5 Customized Slicing

Instead of applying a traditional slicing algorithm to collect all the reachable statements in the dependency graph of function F with the slicing criterion of the identified rStmt and kVar, we customize the slicing to constrain the resultant slice to be tightly related to the (potential) bug. The customization is reflected in two aspects, as shown in Algorithm 1.

First, we limit the reachable depth in the dependency graph. Only when the kVar is the definition (solo use) of a unary expression, will one more step be forwarded to track the corresponding use (definition) variable (Lines 23 ~ 29). That is why *rc* and its correlated lines are not included in the slice for the kVar *dev* in Figure 1. Otherwise, only the statements directly depending on the kVar will be collected. The constraint is enforced by binding a *depth* to the variables, which are checked for each tracked variable (Line 7) and updated for a potentially trackable variable (Lines 25 and 28).

Second, statements referring to the trackable variables are not all kept in the slice. Instead, we only retain the statements in the control flow path that can cover the rStmt and most of the trackable variables. The strategy makes the slice compact and simultaneously keeps the semantics of the involved objects as far as possible in the slice. Line 16 filters the result collected through the while loop (Lines 6 ~ 15).

It is also notable that we treat a member access expression in the form of *mc_dev->dev* in Figure 3(a) as a whole, as is done in [27, 57], and ignore the access to any member from a tracked variable via the '->' operator. Neither do we track the receiver pointer (e.g., *mc_dev*) individually

Algorithm 1: Customized slicing

```
1  Func customizedSlice(F, rStmt, kVar):
2      kVar.depth ← 0
3      Σ ← [rStmt]                                          ▷ collected stmts
4      Γ ← [kVar]                                           ▷ tracked variables
5      cfg, ddg ← buildCfgAndDdg(F)
6      while (γ ← Γ.removeFirst()) is not nil do
7          if γ.depth > 1 then continue                     ▷ limit slicing depth
8          v, stmt ← ddg.getDefinition(γ)
9          collectAndPrepare(γ, stmt, Σ, Γ)                 ▷ collect stmt @20 and var @26
10         Ψ ← ddg.getAllUses(v)                            ▷ all use stmts
11         for ψ ∈ Ψ do
12             if cfg.hasFwdPath(ψ, rStmt) then
13                 collectAndPrepare(v, ψ, Σ, Γ)            ▷ collect stmt @20 and var @29
14             else if cfg.hasFwdPath(rStmt, ψ) then
15                 collectAndPrepare(v, ψ, Σ, Γ)            ▷ collect stmt @20 and var @29
16     Σ ← filterByMaxCoveragePath(cfg, rStmt, Σ)
17     return sortByLineNumber(Σ, deduplicate = True)
18 Func collectAndPrepare(var, stmt, Σ, Γ):
19     if isNormalStmt(stmt) == False then return
20     Σ.append(stmt)                                       ▷ collect stmt
21     l ← getLhs(stmt)                                     ▷ l == v@8
22     R ← getRhsOperands(stmt)
23     if l is not nil and len(R) == 1 then                 ▷ unary op
24         if l == var then                                 ▷ define var, i.e., v@8 == γ@8
25             R[0].depth ← var.depth + 1
26             Γ.append(R[0])                               ▷ prepare one more step for R[0]
27         else                                             ▷ use var and var == R[0]
28             l.depth ← var.depth + 1
29             Γ.append(l)                                  ▷ prepare one more step for l
```

in such cases. The rule applies to both variable gathering and dependency analysis (Lines 8, 10, 21 and 22). In other words, v is literally equal to γ at Line 8.

All the statements of interest are maintained in Σ, which is sorted by the line numbers before the algorithm returns it as the resulting slice (Line 17). The duplicate statements are also refined along with the sorting. Furthermore, if more than one rStmt and kVar are identified for a function, their slices are combined to compose the final feature slice.

3.6 Embedding, Ranking and Auditing

All feature slices are embedded into vectors with a pre-trained encoder model. In this study, we choose UniXcoder [13], a unified cross-modal model for code representation, as the embedding model. Since it learns from multiple languages but we focus on C code, we fine-tune the model

following its instruction on the C-based code clone dataset POJ-104 [28]. The fine-tuned model is termed UniXcoder-POJ in this paper.

Cosine similarity is computed between the query and each candidate feature slice. The candidate functions are then ranked based on the similarity values in descending order. The results are manually audited to confirm unknown bugs. Following previous studies [12, 55] that audit the top-ranked functions, in this study, we select the top 10 candidates for auditing.

3.7 Function-level Candidate Function Screening

In large code bases like the Linux kernel, directly pinpointing candidate rStmts and kVars and then computing the similarity between their corresponding slices and the query will inevitably encounter the efficiency issue. To address the issue, we propose a function-level screening step to screen out a set of candidate functions from the whole code base (❸ in Figure 2). Functions are embedded into vectors, and similarity between each target and the seed function is computed. The top similar functions are left for further analysis, i.e., pinpointing candidate rStmts and kVars. In this study, we choose to keep the top 1000 functions as the candidates.

4 Evaluation

4.1 Experiment Setup

We have implemented a prototype of Matus. Code preprocessing and slicing are built on top of Joern [19] and fuzzyc2cpg [37], while the other components (including model tuning, embedding and matching) are implemented in Python. All experiments are conducted on a server with Intel(R) Xeon(R) Gold 5218 CPU@2.30GHz, 256 GB memory, Ubuntu 20.04 and NVIDIA GeForce RTX 3090 GPU (for fine-tuning, embedding and similarity calculation).

Datasets. We evaluate Matus on two datasets. First, we select the Linux kernel (shortened as *Linux*) v6.4-rc2 as the target of evaluation for detecting new bugs and assessing the efficiency. To improve the matching efficiency, we precompute the function embeddings, slices, their corresponding embeddings, and the `[MASK]` embeddings (see Figure 3(d)) offline in a one-time preprocessing phase, leveraging their reusable nature. In total, 293,292 functions are successfully extracted and 10,909,165 slices are obtained. Second, we construct a benchmark with 76 pairs of seed/target bugs. The benchmark consists of (D1) the 31 bug pairs by Matus from Linux v6.4-rc2, (D2) 31 collected from the Linux commit history at the inception of this study, and (D3) 14 from OpenSSL v3.0.8 and Linux v6.2.5 by SICode [12]. The dataset is publicly available at the URL provided in Section 1.

Comparative methods. We compare Matus with nine publicly available, open-source methods that measure the code similarity, including ReDeBug [16], SourcererCC [34], Infercode [3], NIL [29], VUDDY [21], FIRE [9], SICode [12], UniXcoder [13] and UniXcoder-POJ. The first four methods target discovering cloned functions; VUDDY, FIRE and SICode detect bugs in similar implementations; and the last two can be used as clone detectors and also serve as the embedding model in Matus. Unless specified otherwise, we manually audit the top 10 reported instances for Infercode, SICode and UniXcoder models where the ranking is available, or the reported candidates for VUDDY, ReDeBug, SourcererCC, NIL and FIRE that emit found/not-found results. In addition, we inspect the rankings of confirmed bugs across all tools. Specifically, the comparative tools are summarized as follows.

- ReDeBug [16]: a fast, syntax-driven tool for detecting unpatched code clones in large, multi-language code bases, excelling in scale and speed.
- SourcererCC [34]: a token-based code clone detector using an optimized inverted-index and filtering heuristics.

- Infercode [3]: a self-supervised learning approach for generating AST-based code embeddings, effective in retrieving similar functions.
- NIL [29]: SOTA token-based clone detector, using N-grams and inverted indexes to identify large variations in large code bases.
- VUDDY [21]: a scalable and efficient tool for detecting vulnerable code clones, focusing on function-level granularity.
- FIRE [9]: SOTA similarity-based bug detector, combining multi-stage filtering and differential taint path analysis to achieve efficient and precise detection.
- SICode [12]: SOTA embedding-based bug detector, identifying subgraph isomorphism based on code graph embeddings to detect bugs.
- UniXcoder [13] and UniXcoder-POJ: cross-modal pre-trained model designed for programming languages that enhances code representation with AST and comments, excelling in a range of code-related tasks. UniXcoder-POJ is a fine-tuned model (see Section 3.6).

The tools are executed with their default parameter setting, but we increase the maximum token sequence length from 512 to 1024 in the use of UniXcoder models. Functions exceeding the length are truncated, but statistics show 92.61% of the functions are kept unchanged. The dimension of embedding vectors is 768 in Matus.

4.2 Detecting New Bugs

We first show the effectiveness of Matus in detecting new bugs. Through the CVE list and the issue list in the Linux repository, we collect 55 seed functions with known bugs and attempt to detect bugs in Linux v6.4-rc2 with Matus. The first author, as a skilled auditor, spends on average two minutes reviewing the top 10 candidates for each bug query.

Among the 55 seeds, 28 led to the discovery of 31 new bugs that have been confirmed by the kernel developers and 11 of the bugs have been assigned CVEs. No other new bugs were detected by the tools for the other 27 seeds. The results are detailed in Table 1.

The competitors explicitly expose worse performance compared to Matus in detecting new bugs. ReDeBug and VUDDY fail to identify any bugs from the seed functions. SourcererCC discovers five ($\frac{5}{31}$ = 16.1%). As a similarity-based matching method, Infercode can always compute a similarity score between a target and the seed, i.e., every function would have a ranking. We label those functions ranked beyond 100th with a '×' symbol in Table 1, indicating that they are typically ignored in a manual audit. Even so, Infercode finds only seven (22.6%), with six (19.4%) among the top 10. NIL performs the best among the first seven detectors listed in Table 1, with 11 (35.5%) bugs detected. However, NIL got stuck on the large code base. To evaluate its capabilities, we created a subset containing the detected buggy functions and tested whether NIL could match them to the seed functions. FIRE, the relevant SOTA tool for bug detection based on sliced taint paths, finds only eight bugs, accounting for 25.8% of the total. SICode requires manual seed slicing but fails to specify detailed rules. Consequently, we build the seed graph using the query slice obtained by Matus, which may substantially affect the embeddings and subgraph isomorphism decisions. It only detects three bugs (9.7%), two of which are also easily identifiable by other tools. The two UniXcoder models can find more bugs within the top 10 than the other seven comparative tools, 15 (48.4%) and 19 (61.3%), respectively. UniXcoder-POJ, fine-tuned on the C-based code clone dataset, performs better than the original model. We also observe that, though UniXcoder-POJ emits a lower ranking for some buggy functions (e.g., #5 and #8), UniXcoder fails to find three within the top 1000 in the new bugs (i.e., #23, #30 and #31).

From the Matus column, 27 buggy functions (87%) are ranked within the top 5, making them easy to audit and the bugs easy to spot. Three are ranked between eighth (#28 and #29) and tenth

Table 1. Newly confirmed bugs in Linux v6.4-rc2.

ID	Seed / Detected Buggy Functions	ReDeBug	SourcererCC	Infercode	NIL	VUDDY	FIRE	SICode	UniXcoder	UniXcoder-POJ	Matus
1	*mlx5e_ipsec_remove_trailer* / *esp_remove_trailer(ipv6)*	×	×	×	×	×	×	×	4	4	3
2	*vb2_dc_put_userptr* / *vb2_vmalloc_put_userptr*	×	×	×	×	×	×	×	11	2	1
3	*radeon_tv_get_modes* / *amdgpu_vkms_conn_get_modes*	×	×	×	×	×	×	×	18	168	2
4	*ath11k_update_per_peer_tx_stats* / *ath12k_update_per_peer_tx_stats*	×	✓	2	✓	×	✓	×	2	2	1
5	*pdsc_auxbus_dev_register* / *add_adev*	×	×	×	×	×	×	6	33	530	1
6	*amdgpu_vkms_conn_get_modes* / *nv17_tv_get_hd_modes*	×	×	×	×	×	×	×	19	23	3
7	*amdgpu_vkms_conn_get_modes* / *radeon_add_common_modes*	×	×	×	✓	×	×	×	3	3	1
8	*nouveau_connector_get_modes* / *nv17_tv_get_ld_modes*	×	×	×	×	×	×	×	83	654	1
9	*probe_uprobe_multi_link* / *serial_test_fexit_stress*	×	×	×	×	×	×	×	38	10	2
10	*ath11k_mhi_register* / *ath12k_mhi_register*	×	×	29	×	×	×	×	3	2	1
11	*tpg110_get_modes* / *psb_intel_lvds_get_modes*	×	×	×	×	×	×	×	75	126	1
12	*aldebaran_mode2_suspend_ip* / *smu_v13_0_10_mode2_suspend_ip*	×	✓	3	✓	×	✓	×	3	3	1
13	*dm9601_mdio_read* / *sr_mdio_read*	×	×	×	✓	×	×	×	6	2	3
14	*imx8mp_blk_ctrl_probe* / *imx8m_blk_ctrl_probe*	×	✓	×	✓	×	✓	×	3	2	1
15	*amdgpu_vkms_prepare_fb* / *amdgpu_dm_plane_helper_prepare_fb*	×	×	×	×	×	×	×	2	2	1
16	*emif_get_id* / *_emif_get_id*	×	×	2	✓	×	✓	×	2	2	1
17	*register_device* / *parport_attach*	×	×	×	×	×	×	×	113	23	2
18	*mt7915_thermal_init* / *mt7921_thermal_init*	×	×	×	✓	×	×	×	2	2	1
19	*mt7915_thermal_init* / *mt7615_thermal_init*	×	×	×	×	×	×	×	3	3	1
20	*dcn32_enable_phantom_plane* / *enable_phantom_plane*	×	✓	1	✓	×	✓	×	2	2	1
21	*rt5682s_register_dai_clks* / *rt5682_register_dai_clks*	×	×	×	✓	×	✓	3	2	2	1
22	*esp_remove_trailer(ipv6)* / *esp_remove_trailer(ipv4)*	×	✓	2	✓	×	✓	1	2	2	1
23	*i2c_register_adapter* / *usb_new_device*	×	×	×	×	×	×	×	×	722	4
24	*versatile_panel_get_modes* / *tpg110_get_modes*	×	×	1	✓	×	✓	×	31	2	2
25	*ntb_transport_register_client_dev* / *locomo_init_one_child*	×	×	×	×	×	×	×	585	885	1
26	*i3c_master_register_new_i3c_devs* / *pci_alloc_child_bus*	×	×	×	×	×	×	×	318	220	4
27	*ch9getstatus* / *ast_udc_getstatus*	×	×	×	×	×	×	×	8	5	4
28	*fsl_mc_device_add* / *pci_register_host_bridge*	×	×	×	×	×	×	×	12	3	8
29	*amdgpu_dm_connector_add_common_modes* / *versatile_panel_get_modes*	×	×	×	×	×	×	×	92	175	8
30	*ptp_ocp_device_init* / *tegra_xusb_port_init*	×	×	×	×	×	×	×	×	123	10
31	*ptp_ocp_device_init* / *srp_add_port*	×	×	×	×	×	×	×	×	325	11★

Numbers represent the rankings, and the symbol '✓' indicates that the bug can be detected by a corresponding tool. Eleven bugs have been assigned CVEs, i.e., #3, #5, #6, #7, #8, #11, #20, #24, #27, #28 and #29. The case marked with '★' is ranked just beyond the top 10 (at 11th), but was noted due to its immediate adjacency to #30 for the same seed function.

(#30), in an acceptable scope that usually does not irritate the auditors. The last one (#31) is a notable exception. Its ranking is lower than the predefined top 10 (Section 3.6), but it immediately follows #30 for the same seed function and we noticed it. Another thing worth noting is that, when using the same seed function as in #30, Matus actually detected another real bug ranked

Table 2. Bug detection results on the benchmark with a strict top-10 candidate audit for MATUS.

Datasets	# Pairs	ReDeBug	SourcererCC	Infercode	NIL	VUDDY	FIRE	SICode	UniXcoder	UniXcoder-POJ	MATUS
D1	31	0%	14.3%	19.4%	35.5%	0%	22.9%	9.7%	48.4%	61.3%	96.8%
D2	31	0%	0%	19.4%	19.4%	9.7%	12.9%	9.7%	38.7%	35.5%	80.7%
D3	14	14.3%	0%	0%	7.1%	0%	0%	0%	21.4%	35.7%	35.7%
D2+D3	45	4.4%	0%	6.7%	15.6%	6.7%	8.9%	6.7%	33.3%	35.6%	64.4%
D1+D2+D3	76	2.6%	6.6%	11.8%	23.7%	3.9%	15.8%	7.9%	39.4%	46.1%	77.6%

first in Linux v6.4-rc2. However, when we reported it to the developers, we got a reply saying the same bug had been fixed in v6.13-rc1 by someone else. Therefore, we do not include this bug in Table 1. Comparing the last two columns, we observe that, when the function-level similarity makes some buggy candidate functions neglected, MATUS significantly promotes the probability of the involved bugs being spotted, e.g., #5, #8, #23 and #25. In fact, compared to UniXCoder-POJ, MATUS improves the ranking for 28 buggy functions (90.3%), demonstrating the effectiveness of the feature slice-based similarity measurement.

4.3 Benchmark Performance

Since the new bugs are a subset of the benchmark, in this section, we focus on the performance on the benchmark. The results are shown in Table 2. Be aware of that, to ensure a fair comparison, we strictly audit the top 10 candidates for each query in ranking-based methods such as MATUS.

MATUS discovers 59 ($\frac{59}{76}$ = 77.6%) over the benchmark (D1+D2+D3) and exceeds all other tools on each subset. Even without considering the new bugs, MATUS successfully detects 64.4% of existing bugs (D2+D3). Among the five bugs detected in D3 by MATUS, four are from OpenSSL. The two UniXcoder models continue to outperform the other seven tools on each subset and the whole benchmark, and NIL achieves the third-best performance among the nine. SICode fails to report any bugs on D3, the dataset from its paper, for the same reason as in Section 4.2. In addition, four other tools (SourcererCC, Infercode, VUDDY and FIRE) emit no bugs at all on D3. However, the overall results (on D1+D2+D3) are consistent with those in Table 1.

We manually inspect the 17 missed bugs by MATUS, especially the limited detection effectiveness on D3. Six are excluded as they fall outside the top-1000 candidates during the screening phase (five from D3); eight are affected by an excessive number of similar operations (four from D2 and three from D3), which dilute their ranking scores; one is due to parsing inaccuracies (from D3); one is missed because of a single slice statement; and one resulted from incorrect identification of kVars and rStmts in the seed function. While the other causes are easy to understand, an excessive number of similar operations can yield lots of similar slices, and a minor deviation in the slices (e.g., variable or function names) may affect the embeddings. As a result, the buggy slices may get slightly lower similarity with the query and fall outside the audit scope.

On the benchmark, only three bugs are detected by the competitors but missed by MATUS. VUDDY detects the last case among the 17 (failing kVar/rStmt identification). UniXcoder-POJ detects the one with parsing issues and the other one with lower ranking by MATUS (18th) due to similar operations.

Table 3. Time consumption. Online cost is measured per query on average. ScrCC is short for SourcererCC and UniXcoder-POJ has the same time cost as UniXcoder.

	ReDeBug	ScrCC	Infercode	NIL	VUDDY	FIRE	SICode	UniXcoder
Offline	19m44s	5m33s	4h6m54s	N/A	2h50m45s	58m	5h42m	1h10m25s
Online			1m	(8s)	15s	34s	4m39s	1m

MATUS					
Offline	**Candidate Screening**	**Pinpointing rStmt & kVar**	**Obtaining Feature Slices**	**Matching & Ranking**	**Total (Online)**
42h1m32s	1m	2m15s	42s	15s	4m12s

Overall, the benchmark experiment has well demonstrated the effectiveness of MATUS in bug detection.

4.4 Efficiency

Table 3 presents a comparison of the average time cost per query across different tools on Linux v6.4-rc2. Generally, we count the offline preprocessing and online matching separately, if applicable. Though the offline step may cost a lot, online matching is often fast. For example, Infercode, SICode and UniXcoder(-POJ) take hours to embed the functions offline but spend only a few minutes to finish a query on ~300k targets. VUDDY exhibits faster online matching performance, averaging 15 seconds per query. NIL, however, fails to process the whole Linux code base. Hence, we only evaluate its speed on the small subset mentioned in Section 4.3, which is about 8 seconds per query.

As mentioned in Section 4.1, MATUS processes some steps offline, which takes about 42 hours, as shown in Table 3. As a one-time effort, it is acceptable and can be further accelerated by introducing parallel processing. For the online stages, though embedding vector-based similarity computation can be significantly sped up with GPU-enhanced batch processing, to ensure a fair comparison, we do the computation one by one. Given a seed function pair, MATUS takes negligible time to automatically identify the kVars and rStmts from the buggy function, so we omit the corresponding time cost in Table 3. The most time-consuming step is to pinpoint rStmts and kVars from the candidate functions, and the total online matching requires about four minutes. We deem it acceptable for a large code base such as Linux in practice, especially taking the detection performance in Table 1 into account.

4.5 Ablation Study

In this section, we conduct two groups of ablation study to evaluate the effectiveness of different model selection and different technical choices in MATUS.

4.5.1 Selecting Different Embedding Models. We first evaluate how the fine-tuned model could affect the performance of MATUS compared to the original one. MATUS involves the embedding model at three steps, i.e., funciton-level screening, kVar/rStmt pinpointing and slice embedding. Applying either model (UniXcoder or UniXcoder-POJ) to the steps emits eight combinations in total. We present the combinations and the results on the benchmark in Table 4.

Clearly, UniXcoder-POJ has positive impact on each step. For example, with UniXcoder-POJ on screening and pinpointing (C5), the recall@10 achieves 75.0%, higher than imposing it on screening only (C2: 67.1%) or pinpointing only (C3: 68.4%). The worst recall occurs when using UniXcoder for all the steps (C1), with only 64.5% ($\frac{49}{76}$) bugs identified and 10 fewer than using UniXcoder-POJ for all steps (C8). Among the three steps, UniXcoder-POJ has the least impact on screening (2.6% ~ 6.6%)

Table 4. Ablation study result of selecting different embedding models on the benchmark.

ID	SCR	PIN	EMB	Recall@10	ID	SCR	PIN	EMB	Recall@10
C1	○	○	○	64.5%	C5	●	●	○	75.0%
C2	●	○	○	67.1%	C6	●	○	●	69.7%
C3	○	●	○	68.4%	C7	○	●	●	75.0%
C4	○	○	●	68.4%	C8	●	●	●	77.6%

Symbol ● (○) indicates that UniXcoder-POJ (UniXcoder) is used for function-level screening (SCR), kVar/rStmt pinpointing (PIN) or feature-slice embedding (EMB). Recall@10 indicates how many bugs are discovered within the top 10 candidates.

Table 5. Ablation study result of taking different technical choices.

Technical Choice	Recall@10
Default MATUS	77.6%
(1) Preserving all changed as seed kVars/rStmts	60.5%
(2.1) Strict one step slicing	68.4%
(2.2) Unconstrained slice depth	51.3%
(3) Pinpointing candidate kVar/rStmt with Equation 2	55.3%
(4) Obtaining target slice with direct `[MASK]` mapping	52.6%

and the most impact on pinpointing (3.9% ∼ 7.9%). The result well demonstrates the effectiveness of the fine-tuned model as the embedding model.

4.5.2 Taking Different Technical Choices. Furthermore, we conduct an additional study to evaluate the impact of the key technical choices. The choices involve four aspects: (1) seed kVar/rStmt selection, (2) slicing depth, (3) target kVar/rStmt identification and (4) target slice acquisition.

An alternative to the first aspect is a strategy that we preserve all changed statements and involved variables in the seed function as the seed rStmts/kVars and take the same pinpointing/slicing steps as MATUS to obtain feature slices. The second aspect involves two alternative options when the customized slicing in MATUS goes forward one more step in certain cases (see Section 3.5): (2.1) Strict one-step slicing discards an additional step in Algorithm 1 and forces the slicing depth to be one; (2.2) Unconstrained slicing depth adopts classic slicing algorithms without limiting the depth. The options for the third one are Equation 2 or Equation 3 for kVar/rStmt pinpointing, as described in Section 3.4. MATUS takes the latter one by default. For the last aspect, an alternative approach is to pinpoint the target rStmts one by one by using each statement in the query slice as the seed rStmt (along with the pre-collected seed kVars) while prohibiting slicing on the target function. In other words, the pinpointed rStmts form the target slice. We list the alternatives in the first column in Table 5. Note that, the alternative techniques are independently evaluated, i.e., only the tested technique is adopted by MATUS with all the others kept unchanged.

From Table 5, we can see that taking a different technical choice will reduces the recall, especially for the last three rows. One interesting observation is that, slicing less is better than slicing more, as extraneous noise statements will significantly influence the embedding-based slice similarity. Figure 4(a) presents a bug that is caught by MATUS (even with the strict one step slicing) but missed by adopting the (2.2) slicing strategy. It is also notable that, acquiring the target slice via direct `[MASK]` mapping (4) without further slicing can also introduce noise, causing declined recall (77.6% → 52.6%). Figure 4(b) shows an example, which is the same buggy function as Figure 1(b). The direct

(a) An example of unconstrained slice depth

(b) An example of direct [MASK] mapping

Fig. 4. Examples of introducing noise statements into target slices with certain technique choices.

mapping strategy introduces line 19, a noise statement of the bug, into the target slice, resulting in a degraded ranking of the target function.

The study shows the necessity and effectiveness of the corresponding techniques Matus adopts by default, which lead to better performance in similarity-based bug detection.

4.6 Parameter Sensitivity

We further examine how the number of candidate functions may affect the result in detecting new bugs in Linux. Besides the default value of 1000, three other conditions are evaluated, i.e., 500, 2000 and all. The last means that we do not perform the screening and leave all functions to subsequent steps. The result is shown in Table 6. There is obviously a trade-off between efficiency and performance.

The time cost is nearly linear with the number of candidate functions. Therefore, it takes over 14 hours on average to execute a query on the whole code base (the *All* row), 207 times slower than the default that requires only 4 minutes.

Keeping only the top 500 could miss four bugs, as expected from the UniXcoder-POJ column in Table 1. Analyzing the top 2000 emits the same result as the top 1000. Neither a bug in Table 1 is missed, nor a new bug is spotted. For the last row, on the one hand, nine other bugs are uncovered in Linux v6.4-rc2, which have already been fixed in newer versions. They can also be detected by retaining the top 10000 as candidates, which, however, would take about 1 hour and 10 minutes per query. On the other hand, four bugs in Table 1 drop their rankings below the top 15. We find that many similar but non-buggy functions are ranked higher than the buggy ones. Such a phenomenon will be further discussed in Section 5. Therefore, Matus's default choice balances efficiency and performance.

Table 6. Experiment result in detecting new bugs for different numbers of screened candidate functions.

#Candidates	Time Cost	Results
Top 1000	4m12s	see Table 1
Top 500	2m1s	~~#5~~, ~~#8~~, ~~#23~~ and ~~#25~~
Top 2000	8m4s	Same as Top 1000
All	14h27m54s	~~#11~~, ~~#27~~, ~~#30~~ and ~~#31~~; nine new that have been fixed.

5 Discussion

While MATUS has demonstrated its effectiveness in bug detection, there are some points that need further discussion.

Automated or manual identification of seed kVar and rStmt. MATUS implements an automated identification of the seed kVars and rStmts, which enhances analysis efficiency and reduces the usage barrier of the tool. Human expertise can help MATUS handle the hard cases that may fail in an automated heuristic-based process, e.g., the one MATUS incorrectly recognized the seed kVar/rStmt and caused a missed bug (see Section 4.3). However, in our experiment, MATUS succeeds in identifying the seed kVars/rStmts in 98.7% ($\frac{75}{76}$) cases, achieving nearly identical effectiveness to that of manual selection.

Limitations of the kVar/rStmt-based method. Though the experiments have well demonstrated the effectiveness of identifying kVar/rStmt in the seed and target functions, it has certain limitations in corner cases due to the code complexity. First, if the patch introduces entirely new statements and new variables that exhibit no interaction with any existing code, the current approach will fail to spot any seed kVar/rStmt for further analysis. Second, term frequency-based kVar identification may exclude the real object of interest, or at least include useless variables, leading to a redundant query slice and failing the matching. Third, if a patch of one function involves multiple (types of) bugs, it is highly possible to emit a large feature slice that cannot match any other target slices, when a small target slice possesses only one similar bug with the query. We can leverage human expertise to manually and precisely obtain code slices, thereby addressing the issues mentioned above. In particular, we can split the query slice in the third case into multiple small slices to enhance the performance.

Task-specific fine-tuning. Pre-trained models are often fine-tuned specifically for downstream tasks. In this study, we only tune the model on a dataset targeting code clone detection. Neither is the dataset strongly related to the target of evaluation or bug detection, nor do we perform fine-grained tuning for masked embedding and feature slicing matching. Typically we should do that, but in practice, we have observed sufficiently high performance with the simply tuned model in combination with other techniques, making task-specific tuning unnecessary. However, due to the complexity of real-world code bases, there may be some scenarios where task-specific tuning must be taken to achieve acceptable performance. We will explore such cases in the future.

Side-effect of fixed/non-buggy targets. In large code bases such as Linux, there are many semantically similar functions, and many of them are non-buggy or have been fixed for the same bug in a given seed function. Their similarity with the seed function may significantly influence the result, when too many literally/semantically similar root statements can be found in the code base and the non-buggy code is considered more similar to the known buggy one. In such a case, the one with an unknown bug can have a low ranking and be ignored for auditing. For example, in Linux v6.4-rc2, there are 545 statements semantically similar to the rStmt calling `of_clk_add_hw_provider` in a known bug. A new bug ranked 31-*st* by MATUS has the correctly pinpointed rStmt invoking `devm_of_clk_add_hw_provider`. Among the 30 functions with higher

ranking than the new buggy one, 94% of them expose similar feature slices with the query but they are all non-buggy.

A possible solution to excluding the similar but non-buggy candidates is to leverage the fixed version of the known bug, as done in [49]. Higher similarity with the fixed one may probably indicate a non-buggy one. However, as Zhang et al. [55] has illustrated, the assumption that fixed code is more similar to non-buggy code is not applicable in many cases for embedding-based methods. Nonetheless, we think it represents a chance to reduce the influence of fixed/non-buggy targets, though purposeful fine-tuning of the embedding models may be required. We leave it as a future exploration.

LLM-assisted candidate screening and slice extraction. The powerful generative large language models (LLMs) can be an alternative to screen the candidate functions, pinpoint rStmts and kVars, and extract feature slices in an end-to-end manner. Ideally, we can feed LLM with the query slice of a known bug and a target function, asking LLM to extract a feature slice semantically similar to the query or to discard the function if no candidate slice can be found. We selected five hard samples (bug pairs with low similarity) and five easy (bug pairs with high similarity) in Table 1, and evaluated popular LLMs including ChatGPT [1], Deepseek-R1 [26] and LLaMA-7B [40]. For easy examples, the models may show satisfactory performance in pinpointing rStmts and kVars, and generate proper candidate feature slices within one test. We also find that the larger the LLM, the more likely it is to produce an answer as expected. However, inconsistency occurs with the same inputs, indicating stability issues. For hard examples, the models can barely pinpoint correct rStmts and kVars, or extract satisfied feature slices. With limited computational resources, fine-tuning these LLMs becomes impractical for us. We therefore leave it as an open problem.

6 Related work

6.1 Code Clone Detection

In general, a pair of functions {f1, f2} is identified as clones if they are similar according to some definitions. In the current researches, a variety of methods have been developed for code clone detection. Depending on the data structures employed, these methods can generally be classified into categories such as text-based [32, 38], token-based [20, 24, 25, 31, 34, 43], tree-based [11, 18, 22, 47, 52–54], graph-based [35, 42, 48, 56, 58], hybrid-method [8, 23, 33, 41, 44–46], etc. *NICAD* [32] employs text normalization techniques to identify potential changes as simple text differences, making it effective for detecting near-miss intentional clones. *Amain* [47] uses Markov chain models to transform ASTs into state transition matrices, followed by feature extraction and machine learning classification to identify similar code fragments scalably. *DSFM* [52] enhances functional code clone detection by incorporating deep subtree interactions, comparing subtrees from the abstract syntax trees of code snippets to introduce finer-grained semantic similarity. *TECCD* [11] utilizes a tree embedding technique with *word2vec* [4] to convert ASTs into vector representations, improving the efficiency of tree-based code clone detection and achieving high precision and recall for detecting type I, II, and III clones. *CCGraph* [58] is a PDG-based code clone detector that employs graph kernels and a two-stage filtering strategy with characteristic vector measurement, followed by an approximate graph matching algorithm using the Weisfeiler-Lehman graph kernel [36] to detect semantic clones. *Prism* [23] proposes a code clone detection method that leverages behavior semantics from multiple architecture assembly codes to enhance program understanding, utilizing an embedding technique and a multi-feature fusion strategy to create a more expressive representation of code.

6.2 Large Models Used in Code-Related Tasks

Currently, identifying semantic clone pairs with low textual similarity remains a significant challenge. With the advent of Artificial Intelligence (AI), researchers are seeking to address this issue through AI methods. In recent years, the rapid advancements in the field of large models [1, 6, 10, 14, 30, 39, 40], with their increasing complexity and capabilities, have also brought new developments to the task of code clone detection.

CodeBERT [10] utilizes a Transformer architecture and is pre-trained on a replaced token detection task, enabling it to learn from both NL-PL pairs and unimodal data. While not directly mentioned for code clone detection, *CodeBERT* [10]'s capability to understand code semantics could enhance the detection of semantic clones. It achieves great performance in NL-PL tasks, suggesting potential applications in improving code clone detection accuracy. *GraphCodeBERT* [14] achieves breakthroughs in code clone detection by leveraging semantic-level code structure and introducing structure-aware pre-training tasks. *CONCORD* [6] employs a self-supervised pre-training strategy that incorporates code clones and their deviants to enhance the learning of general-purpose code representations, improving performance on downstream software engineering tasks such as clone and bug detection with reduced pre-training resource requirements. Additionally, large models such as *LLaMA* [40], *GPT4* [1], *Alpaca* [39], and *Vicuna* [30], which have more parameters, are also being explored. Researchers have attempted to apply these models to the task of code clone detection.

6.3 Bug Detection Based on Code Clone

In the field of bug detection, bug detection [2, 7, 17, 21, 49–51] based on code clone is instrumental in preventing the propagation of software defects. *VGRAPH* [2] employs a graph-based approach to detect vulnerable code clones robustly by utilizing code property triplets, consisting of relationships from contextual, vulnerable, and patched code. *BugGraph* [17] addresses source-binary code similarity detection by first identifying the compilation provenance of the target binary and compiling the source code accordingly. It then employs a graph triplet-loss network on attributed control flow graphs to rank code similarity. This method bridges the gap between source and binary code analysis. *MVP* [49] proposes a program slicing technique, which extracts and matches bug and patch signatures for detecting recurring bugs with reduced false positives and negatives. *ISRD* [50] utilizes a multi-level birthmark model incorporating function, basic block, and instruction levels, along with Minimum Branch Path representation and intent search based on anchor recognition. *DiscovRE* [7] employs control flow graphs and numeric feature pre-filters for rapid bug detection across binary architectures.

7 Conclusion

In this paper, we present Matus, an effective and scalable bug detection approach based on similar code matching. Matus extracts the root statements and key variables from a snippet with a known bug, and then emits a query slice representing the semantic feature of the bug. The root statements and key variables are used to pinpoint their counterparts in target functions, which in turn guide the candidate feature slice extraction. The pinpointing leverages the power of an encoder model, based on which we propose a masked embedding-based approach to identification in an end-to-end way. Feature slices are embedded with a fine-tuned pre-trained code model, and the similarity between each candidate and the query is computed. The top-ranked code slices and their corresponding functions are audited to confirm unknown bugs. Matus successfully detected 31 previously unknown bugs from the Linux kernel and outperformed nine competitors. Experiments

also demonstrate the effectiveness of the technical choices in MATUS and the acceptable efficiency for detecting bugs in a large code base.

Data Availability

The code and data necessary to reproduce the findings and analyses in this paper are available in an anonymized repository at https://github.com/Kyew2r6qAe/MATUS. To protect the double-blind review process, the authors have omitted any identifying information. The complete, non-anonymized replication package will be made publicly available upon publication of this paper.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. The work is supported in part by Beijing Natural Science Foundation under grant 4262027, and National Natural Science Foundation of China (NSFC) under grants 62272465 and 62272464, and Public Computing Cloud, Renmin University of China.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Benjamin Bowman and H Howie Huang. 2020. VGRAPH: A robust vulnerable code clone detection system using code property triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 53–69.

[3] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1186–1197. doi:10.1109/ICSE43902.2021.00109

[4] Kenneth Ward Church. 2017. Word2Vec. *Natural Language Engineering* 23, 1 (2017), 155–162.

[5] Lei Cui, Zhiyu Hao, Yang Jiao, Haiqiang Fei, and Xiaochun Yun. 2021. VulDetector: Detecting Vulnerabilities Using Weighted Feature Graph Comparison. *Trans. Info. For. Sec.* 16 (Jan. 2021), 2004–2017. doi:10.1109/TIFS.2020.3047756

[6] Yangruibo Ding, Saikat Chakraborty, Luca Buratti, Saurabh Pujar, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2023. CONCORD: clone-aware contrastive learning for source code. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 26–38.

[7] Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. 2016. Discovre: Efficient cross-architecture identification of bugs in binary code.. In *Ndss*, Vol. 52. 58–79.

[8] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 516–527. doi:10.1145/3395363.3397362

[9] Siyue Feng, Yueming Wu, Wenjie Xue, Sikui Pan, Deqing Zou, Yang Liu, and Hai Jin. 2024. FIRE: combining multi-stage filtering with taint analysis for scalable recurring vulnerability detection. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 105, 18 pages.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[11] Yi Gao, Zan Wang, Shuang Liu, Lin Yang, Wei Sang, and Yuanfang Cai. 2019. TECCD: A tree embedding approach for code clone detection. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*. ieee, 145–156.

[12] Yuanjun Gong, Jianglei Nie, Wei You, Wenchang Shi, Jianjun Huang, Bin Liang, and Jian Zhang. 2024. SICode: Embedding-Based Subgraph Isomorphism Identification for Bug Detection. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 304–315.

[13] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. 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)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 7212–7225. doi:10.18653/v1/2022.acl-long.499

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*

(2020).
[15] Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu. 2021. Hunting Vulnerable Smart Contracts via Graph Embedding Based Bytecode Matching. *IEEE Trans. Inf. Forensics Secur.* 16 (2021), 2144–2156. doi:10.1109/TIFS.2021.3050051
[16] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 48–62.
[17] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 702–715. doi:10.1145/3433210.3437533
[18] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.
[19] joern.io. 2024. *Joern: The Bug Hunter's Workbench*. https://github.com/joernio/joern
[20] Toshihiro Kamiya. 2021. Ccfinderx: An interactive code clone analysis environment. *Code Clone Analysis: Research, Tools, and Practices* (2021), 31–44.
[21] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 595–614.
[22] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*. IEEE, 253–262.
[23] Haoran Li, Siqian Wang, Weihong Quan, Xiaoli Gong, Huayou Su, and Jin Zhang. 2024. Prism: Decomposing Program Semantics for Code Clone Detection through Compilation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[24] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 249–260. doi:10.1109/ICSME.2017.46
[25] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering* 32, 3 (2006), 176–192.
[26] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).
[27] Dinghao Liu, Zhipeng Lu, Shouling Ji, Kangjie Lu, Jianhai Chen, Zhenguang Liu, Dexin Liu, Renyi Cai, and Qinming He. 2024. Detecting kernel memory bugs through inconsistent memory management intention inferences. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 228, 18 pages.
[28] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).
[29] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. Nil: large-scale detection of large-variance clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–841.
[30] Michael Platzer and Peter Puschner. 2021. Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation. In *33rd euromicro conference on real-time systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[31] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering* 24, 4 (2019), 2236–2284.
[32] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*. IEEE, 172–181.
[33] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 354–365.
[34] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*. 1157–1168.
[35] Junjie Shan, Shihan Dou, Yueming Wu, Hairu Wu, and Yang Liu. 2023. Gitor: Scalable Code Clone Detection by Building Global Sample Graph. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 784–795.
[36] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research* 12, 9 (2011).

[37] ShiftLeftSecurity. 2024. *Code Property Graph: Specification and Tooling.* https://github.com/ShiftLeftSecurity/codepropertygraph

[38] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. 2016. Identifying functionally similar code in complex codebases. In *2016 ieee 24th international conference on program comprehension (icpc)*. IEEE, 1–10.

[39] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html* 3, 6 (2023), 7.

[40] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[41] Tijana Vislavski, Gordana Rakić, Nicolás Cardozo, and Zoran Budimac. 2018. LICCA: A tool for cross-language clone detection. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 512–516.

[42] Min Wang, Pengcheng Wang, and Yun Xu. 2017. CCSharp: An Efficient Three-Phase Code Clone Detector Using Modified PDGs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 100–109. doi:10.1109/APSEC.2017.16

[43] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: A Token Based Large-Gap Clone Detector. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1066–1077. doi:10.1145/3180155.3180179

[44] Wenjie Wang, Zihan Deng, Yinxing Xue, and Yun Xu. 2023. CCStokener: Fast yet accurate code clone detection with semantic token. *Journal of Systems and Software* 199 (2023), 111618.

[45] Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code.. In *IJCAI*. 3034–3040.

[46] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 87–98.

[47] Yueming Wu, Siyue Feng, Deqing Zou, and Hai Jin. 2022. Detecting semantic code clones by building AST-based Markov chains model. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[48] Yueming Wu, Deqing Zou, Shihan Dou, Siru Yang, Wei Yang, Feng Cheng, Hong Liang, and Hai Jin. 2020. SCDetector: Software functional clone detection based on semantic tokens analysis. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 821–833.

[49] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. {MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*. 1165–1182.

[50] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. 2021. Interpretation-enabled software reuse detection based on a multi-level birthmark model. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 873–884.

[51] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–387.

[52] Zhiwei Xu, Shaohua Qiang, Dinghong Song, Min Zhou, Hai Wan, Xibin Zhao, Ping Luo, and Hongyu Zhang. 2024. DSFM: Enhancing Functional Code Clone Detection with Deep Subtree Interactions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[53] Mohammad A Yahya and Dae-Kyoo Kim. 2023. CLCD-I: cross-language clone detection by using deep learning with infercode. *Computers* 12, 1 (2023), 12.

[54] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. doi:10.1109/ICSE.2019.00086

[55] Xiaohui Zhang, Yuanjun Gong, Bin Liang, Jianjun Huang, Wei You, Wenchang Shi, and Jian Zhang. 2022. Hunting bugs with accelerated optimal graph vertex matching. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 64–76.

[56] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 141–151.

[57] Qingyang Zhou, Qiushi Wu, Dinghao Liu, Shouling Ji, and Kangjie Lu. 2022. Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 3253–3267. doi:10.1145/3548606.3560661

[58] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 931–942.

Received 2025-09-11; accepted 2025-12-22