SICode: Embedding-Based Subgraph Isomorphism Identification for Bug Detection

Yuanjun Gong
Renmin University of China
Beijing, China
gongyuanjun@ruc.edu.cn

Jianglei Nie
Renmin University of China
Beijing, China
rucnjl@ruc.edu.cn

Wei You
Renmin University of China
Beijing, China
youwei@ruc.edu.cn

Wenchang Shi
Renmin University of China
Beijing, China
wenchang@ruc.edu.cn

Jianjun Huang*
Renmin University of China
Beijing, China
hjj@ruc.edu.cn

Bin Liang*
Renmin University of China
Beijing, China
liangb@ruc.edu.cn

Jian Zhang
SKLCS, Institute of Software, Chinese Academy of Sciences
Beijing, China
zj@ios.ac.cn

ABSTRACT

Given a known buggy code snippet, searching for similar patterns in a target project to detect unknown bugs is a reasonable approach. In practice, a search unit, such as a function, may appear quite different from the buggy snippet but actually contains a similar buggy substructure. Utilizing subgraph isomorphism identification can effectively hunt potential bugs by checking whether an approximate copy of the buggy subgraph exists within the target code graphs. Regrettably, subgraph isomorphism identification is an NP-complete problem.

In this paper, we propose an embedding-based method, SICode, to efficiently perform subgraph isomorphism identification for code graphs. We train a graph embedding model and the subgraph isomorphism relationship between two graphs can be measured by comparing their embedding vectors. In this manner, we can efficiently identify potential buggy code graphs via vector arithmetic without solving an NP-complete problem. A cascading loss scheme is presented to ensure the identification performance.

SICode exhibits greater scalability than classic subgraph isomorphism algorithms, such as VF2, and maintains high precision and recall. Experiments also demonstrate that SICode offers advantages in detecting sub-structurally similar bugs. Our approach spotted 20 previously-unknown bugs in real-world projects, among which, 18 bugs were confirmed by their developers and ranked within the top ten results of retrieval. This result is very encouraging for detecting subtle sub-structurally similar bugs.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; • **Security and privacy → Software and application security**.

KEYWORDS

Subgraph isomorphism, Bug detection, Cascading loss, Graph embedding

ACM Reference Format:
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 *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3643916.3646556

1 INTRODUCTION

It has been proven that detecting bugs based on code matching is feasible [14, 37, 38]. Since the code can be naturally represented as graphs (e.g., CFGs and PDGs), adopting graph matching for code detection can catch code structure characteristics and achieve more accurate results, as done in [7, 15].

The way of straightforward graph-to-graph matching can work well for the target code with simple logic and high homogeneity, such as smart contracts [15]. Unfortunately, in practice, bug-related elements within a function (a typical matching unit) occupy only a portion of the function, namely, only a part of nodes and edges in the corresponding code graph. The matchable bug logics in target functions are often sub-structurally similar to a known bug. Furthermore, the bug-related code elements may be the majority or minority of the functions. Besides, the functions can have either

*Corresponding authors.

ICPC '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0586-1/24/04…$15.00
https://doi.org/10.1145/3643916.3646556

```
//file: drivers/rtc/rtc-mxc.c in Linux
//     static int mxc_rtc_probe(struct platform_device *)
380 pdata = devm_kzalloc(dev, sizeof(*pdata), GFP_KERNEL);
386 res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
387 pdata->ioaddr = devm_ioremap_resource(&pdev->dev, res);
388 if (IS_ERR(pdata->ioaddr))
389    return PTR_ERR(pdata->ioaddr);
391 pdata->clk = devm_clk_get(dev, NULL);
392 if (IS_ERR(pdata->clk)) {
393   dev_err(...);
394   return PTR_ERR(pdata->clk);
395 }
397 clk_prepare_enable(pdata->clk);
398 rate = clk_get_rate(pdata->clk);
400 if (rate == 32678) ... // 10 lines if-block
413 writew(reg, (pdata->ioaddr + RTC_RTCCTL));
446 exit_put_clk:
447   clk_disable_unprepare(pdata->clk);
449   return ret;
```

(a) Known Bug: Commit 1b3d22

```
//file: drivers/spi/spi-bcm2835.c in Linux
//   static int bcm2835_spi_probe(struct platform_device *)
1331 ctlr = devm_spi_alloc_master(&pdev->dev, sizeof(*bs));
1335 platform_set_drvdata(pdev, ctlr);
1348 bs = spi_controller_get_devdata(ctlr);
1351 bs->regs = devm_platform_ioremap_resource(pdev, 0);
1355 bs->clk = devm_clk_get(&pdev->dev, NULL);
1356 if (IS_ERR(bs->clk))
1357    return dev_err_probe(&pdev->dev, PTR_ERR(bs->clk),
1358        "could not get clk\n");
1360 ctlr->max_speed_hz = clk_get_rate(bs->clk) / 2;
1362 bs->irq = platform_get_irq(pdev, 0);
1366 clk_prepare_enable(bs->clk);
1367 bs->clk_hz = clk_get_rate(bs->clk);
1369 err = bcm2835_dma_init(ctlr, &pdev->dev, bs);
1374 bcm2835_wr(bs, BCM2835_SPI_CS, ...);
1397 out_clk_disable:
1398   clk_disable_unprepare(bs->clk);
1399   return err;
```

(b) New Bug

Figure 1: The query bug and a previously unknown bug. This pair of functions show the subgraph isomorphism relationship between the query graph and the candidate graph. Two functions differ greatly globally, but they have sub-structural similarity in the highlighted statements.

numerous or only a few lines of code. The bug-unrelated elements can hazard the whole-graph matching for bug detection.

Naturally, graph-matching-based bug detection can be converted to a subgraph-matching problem. Namely, determining whether a graph formed by bug-related elements in a buggy function is *subgraph isomorphic* to the code graph of a target function. If the answer is yes, the target function can be considered to suffer a similar bug. In this way, the interference caused by bug-unrelated elements can be effectively reduced.

However, identifying whether a graph is subgraph isomorphic to the other one is an NP-complete problem. Existing classic algorithms such as VF2 [9] and QuickSI [26] are very time-consuming and not scalable in large-scale scenarios for isomorphism-based bug detection. To demonstrate it, we perform an empirical study on the Linux kernel, from which more than two million target code graphs are extracted. The result shows that at least three hours are required to decide the subgraph isomorphism between a given graph and all the targets. Therefore, how to scalably decide subgraph isomorphism for practical bug detection is an urgent issue.

In this paper, we address the issue by proposing an embedding-based approach to subgraph isomorphism identification, named **SICode** (**S**ubgraph **I**somorphism-based buggy **Code** detection)[1]. The basic idea is to embed the buggy graph (termed *query* in this paper) and the *target* graph into low-dimensional dense vectors, and then measure the relation between the two vectors to make an efficient subgraph isomorphism decision. Note that, there have been a lot of studies that employ graph neural networks (GNNs) to embed the code graphs and measure the vector similarities. However, the existing ones are oriented to whole-graph similarity measurement, without considering if a graph is subgraph isomorphic to the other one. SICode, in contrast, aims to develop a graph embedding model that can learn the subgraph isomorphism relations between code graphs and thus the relations can be utilized to detect bugs in a way of subgraph matching.

Ideally, we hope the embedding model can perfectly support qualitative subgraph isomorphism decision, i.e., directly deciding whether two graphs are subgraph isomorphic. However, in theory, the embedding-based method is essentially statistics-based and usually provides a probability solution. In other words, it provides a quantitative confidence of a solution.

To solve the subtle subgraph isomorphism problem, it is very difficult to train a relatively precise model in an all-at-once scheme. Especially in the scenario of making decisions for complicated heterogeneous code graphs, the all-at-once training scheme may lead to many missing bugs.

In this paper, we propose a *cascading loss* scheme to train the desired embedding model. The major difference between a cascading scheme and an all-at-once scheme lies in the positive training instance. In a cascading scheme, a positive training instance includes a series of code graphs in the form of $\langle P_0, \cdots, P_t \rangle$, in which P_i is subgraph isomorphic to P_{i+1}. The cascading loss is enforced to reflect the subgraph isomorphism relation between consecutive graphs in the positive instance. By this means, the model can learn how more likely a graph is subgraph isomorphic to another one than to the others. In a classic all-at-once scheme, a positive instance contains only a pair of graphs, i.e., $\langle P_0, P_1 \rangle$, and the straightforward loss only represents that P_0 is subgraph isomorphic to P_1. It will lead to overfitting. The ablation study in Section 4.5 demonstrates the cascading loss scheme has a significant improvement on the performance of subgraph isomorphism identification.

[1]The project is available at https://github.com/scdlcsce/SICodeOpenSource.

In addition, we also present a clustering-based normalization to reduce the heterogeneity of the graph nodes. Hundreds of thousands of functions in a target software project are clustered, and thus semantically similar nodes in two graphs can be regarded as the same, which can help detect potential semantically similar bugs.

SICode is proven to be scalable and effective. Taking the Linux kernel as an example, one query across the whole project takes less than four minutes, much faster than classic isomorphism algorithms such as VF2 (3h17min). SICode is also applied to several real-world large open-source projects for bug detection and the experiment demonstrates that SICode has advantages in finding sub-structurally similar bugs. Twenty previously unknown bugs have been confirmed by the developers and 18 among them are ranked within the top ten in subgraph matching. Without the cascading loss, most of the bugs are missed.

The contributions of this work are as follows:

- We design a novel bug detection method, SICode, with embedding-based subgraph isomorphism identification. Subgraph isomorphism relations are encoded in the vectors and hence vector-based comparison can support scalable subgraph isomorphism decisions.
- We propose a cascading loss function to train a desirable embedding model, which greatly improves the model's performance of subgraph isomorphism identification.
- SICode spotted 20 real-world bugs, among which, 18 bugs are confirmed by the developers. While the matching-based competitors may miss most of the bugs. This result is very encouraging for detecting subtle sub-structurally similar bugs.

This work advances the knowledge and practice of program comprehension by detecting similar bugs existing in the software. The matching-based method can help code auditors quickly comprehend the bug report and determine if the reports are true bugs. In addition, the subgraph isomorphism decision makes it easier to comprehend a piece of code that matches a known piece.

2 MOTIVATION & BACKGROUND

In this section, we first present a motivating example and then introduce the concept of subgraph isomorphism. In the end, we discuss the graph isomorphism network (GIN) that has been applied to social networks and bioinformatic graphs.

2.1 Motivating Example

Figure 1a shows a piece of code snippet with a known bug in Linux, in which *clk_prepare_enable()* at line 397 may fail and its return value should be before getting its rate. Fixing the bug can be done by assigning the return value of *clk_prepare_enable()* to a variable, e.g., *ret*, and check *ret*. A non-zero value of *ret* indicates an error occurs and hence the function could just return *ret* to its caller. The function *mxt_rtc_probe()* contains 63 lines of code (LoC) excluding space and comments, and operations most correlated to the bug have been highlighted. At line 380, a chunk to hold device data is allocated. After retrieving device-specific resources and checking corresponding error conditions, a clock is got at line 391. Typically, the clock would pass an error check (line 392) and then be prepared and enabled for later use.

Figure 1b presents a bug SICode spotted by subgraph matching in Linux v6.5-rc2. The buggy function *bcm2835_spi_probe()* has 57 LoC and contains the same bug, i.e., the return value of *clk_prepare_enable()* is not checked. Correlated lines are also highlighted. Device data is obtained (line 1348) and device-specific resources are retrieved (line 1351). Then a clock is got (line 1355) and checked (line 1356). The clock rate is visited and later the clock is prepared and enabled (line 1367). After that, its rate is queried again (line 1367).

Though both functions probe to install devices and have the same operations on device data preparation and clock acquisition/preparation/enabling, the other operations vary. For example, there are about 41 LoCs after calling *clk_prepare_enable()* in Figure 1a while only 26 lines after the operation in Figure 1b, given that both functions have 63 and 57 lines, respectively, in total. The involved operations are different, e.g., *writew()* v.s. *bcm2835_dma_init()* that are immediately invoked after enabling the clock and getting the clock rate. About 12 function calls exist in the 41 lines in Figure 1a and five have associated return value checks. A big *if* block starting at line 400 checks three types of clock rates. In contrast, ten calls happen after *clk_prepare_enable()* in Figure 1b and only three have return value checks. In addition, both functions call different functions to process device-specific resources (lines 386-387 *v.s.* line 1351).

Irrelevant lines (i.e., those not highlighted) can be treated as noise elements, which impose great differences on the code graphs, making the new bug not to be discovered via whole-graph matching.

SICode performs subgraph matching to reduce the influence of noise elements. We train a graph embedding model and subgraph isomorphism relations can be estimated by measuring the embedding vectors. A query code graph (CFG+PDG) corresponding to the associated lines in Figure 1a is extracted and then embedded into a vector. A target graph corresponding to the function in Figure 1b is extracted and embedded. Then SICode compares the vectors to decide whether the query graph is subgraph isomorphic to the target. By this means, Figure 1b is flagged as a candidate with bugs. Human auditing confirms the bug, which is also confirmed and fixed by Linux developers.

2.2 Subgraph Isomorphism

Graph isomorphism and subgraph isomorphism are formally defined as follows.

Graph isomorphism. Let $G_S = (V_S, E_S)$, $G_T = (V_T, E_T)$ be two graphs. G_S is graph isomorphic to G_T if and only if there exists a bijection $f : V_S \to V_T$ such that $\{v_1, v_2\} \in E_S \iff \{f(v_1), f(v_2)\} \in E_T$.

Subgraph isomorphism. G_S is subgraph isomorphic to G_T, if and only if there exists a subgraph $G_0 = (V_0, E_0)$ where $V_0 \subseteq V_T$ and $E_0 \subseteq E_T \cap (V_0 \times V_0)$, such that G_S is graph isomorphic to G_0. In other words, there exists a bijection $f : V_S \to V_0$ that makes $\{v_1, v_2\} \in E_S \iff \{f(v_1), f(v_2)\} \in E_0$ satisfied.

Subgraph isomorphism is a generalization of the graph isomorphism problem, i.e., the smaller graph is isomorphic to a subgraph of the larger graph.

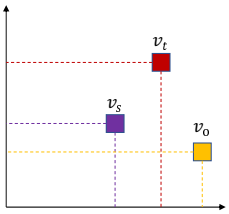

Figure 2: A schematic diagram showing subgraph isomorphism relations between graphs in the embedding space.

2.3 Graph Isomorphism Network

A typical GNN aggregates the features from the neighbor nodes of any node n and combines the aggregation with n's feature from the last hidden layer to make the feature of n in the current hidden layer. Summing or pooling the node features emits the graph feature. Such a GNN network cannot effectively learn the graph isomorphism relation from graphs. Xu et al. [32] proposed a graph isomorphism network (GIN). They suggest using a multi-layer perceptron function for aggregation and combination.

Based on GIN, Lou et al [21] proposed NeuroMatch to approximately solve the subgraph isomorphism decision problem. NeuroMatch imposes a geometric constraint to the embedding space, as shown in Figure 2 that exemplifies the concept in a two-dimensional embedding space. If an embedding vector v_s is at the lower-left of another vector v_t, the corresponding graphs will have a qualitative subgraph isomorphism relation, i.e., G_s is subgraph isomorphic to G_t. Eq. 1 formalizes the relation.

$$\forall_{i=1}^{D}, v_s[i] \leq v_t[i] \quad iff \quad G_s \text{ is subgraph isomorphic to } G_t \quad (1)$$

where D is the number of dimensions of the embedding vectors. In Figure 2, G_s is subgraph isomorphic to G_t but not to G_o. The loss function of NeuroMatch is defined as Eq. 2 where samples s, t in P are subgraph isomorphic while samples in N are non-subgraph isomorphic.

$$\mathcal{L}(v_s, v_t) = \sum_{(v_s,v_t)\in P} E(v_s, v_t) + \sum_{(v_s,v_t)\in N} max\{0, \alpha - E(v_s, v_t)\} \quad (2)$$

$$E(v_s, v_t) = \| \max\{0, v_s - v_t\}\|_2^2 \quad (3)$$

3 METHODOLOGY

3.1 Overview

In this paper, we present a bug detection-oriented subgraph isomorphism embedding, SICode. Figure 3 shows the workflow. In general, SICode consists of three components, i.e., (A) preparing graphs for the target system, (B) training an embedding model, and (C) making subgraph isomorphism decisions for bug detection. Some steps in three components are the same and the same step label is used.

Code graphs are extracted from the source code base or a piece of the buggy query (❶, Section 3.2.1 and Section 3.2.2). The graphs are converted to featured graphs with node encoding (❷, Section 3.2.3). Then the target graphs are embedded into vectors along the blue dashed arrows while the query graph follows the red solid arrows(❺, Section 3.4). SICode then decides whether the query graph is subgraph isomorphic to any target graphs via vector-based measurement and candidates are audited to flag new bugs (❻ and ❼, Section 3.5). Note that, training the model (❹, Section 3.3) requires a different process of preparing graphs (❶ and ❸), and the details are presented in Section 3.3.2.

3.2 Graph Preparation

We first transform a code snippet (e.g., a function) into a code graph via *fuzzyc2cpg* [1]. Specifically, we merge the CFG and the PDG into a hybrid-directed graph, with each node corresponding to a rich semantic element.

First, we define a rich semantic element as invoking a method, performing a certain operation (e.g., '+'), or returning from a method, as it can help match similar code snippets for bug detection. If a node does not involve the above three types of elements, it is pruned from the graph. Some operators are regarded as non-important elements. For example, indirect member access like "->" in C is pruned as we consider it the same as direct access to a variable. If the operator is kept in graphs, we can never match a graph accessing normal variables to the other graph accessing members even if both code snippets have completely the same logic and calls. We prune a non-important node in the hybrid code graph by connecting its predecessors to its successors according to the edge attributes and removing the node itself. Further, we highlight the core semantic of nodes, e.g., a node involving a method call will have only the method name to represent the node, and the redundant information such as variables and constants is omitted.

The processing procedure after code graph pruning in target graph generation and query graph generation is different. The detailed procedure in target graph generation will be described in Section 3.2.1 while the query graph will be described in Section 3.2.2. In training sample generation, the code graphs are saved as backbones, preserving only the type information of each node, seeing Section 3.3.2.

3.2.1 Target Graph Generation. The simplified code graphs will be further processed before node encoding. In case the graphs are too large to capture the information in a fixed-depth graph neural network adequately, we extract k-hop target graphs from the code graphs as done in [21] for large social network graphs.

In this study, a k-hop target graph starts from an invocation node, and all reachable neighbors and edges within k hops are included. As code is represented in directed graphs, we extract a k-hop graph following the edge direction. Specifically, given a node n and an edge e, if $e : n' \rightarrow n$ is an in-edge, we collect all nodes as 2-hop neighbors that can reach n' in the same direction of e. If $e : n \rightarrow n'$ is an out-edge, 2-hop neighbors include all nodes that can be reached from n' in the same direction. For example, in Figure 4a, starting from *clk_prepare_enable*, we can collect *devm_clk_get* and *devm_kzalloc* as reachable neighbors, but *IS_ERR* and *clk_get_rate* are unreachable. Therefore, Figure 4 is in fact a 1-hop subgraph starting from *devm_clk_get*.

3.2.2 Query Graph Generation. The query graph is a subgraph of the buggy function graph that every node and edge is closely related to the bug logic. To properly select a buggy subgraph from

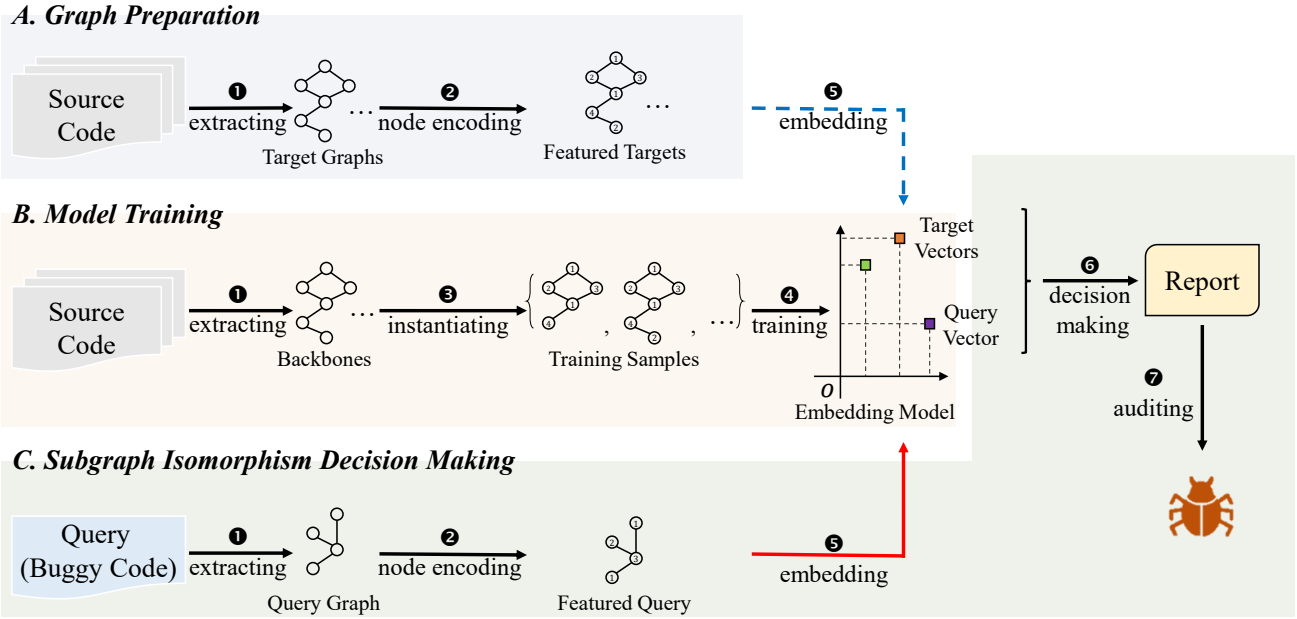

Figure 3: The overall workflow of SICode. The whole procedure is divided into three components and each component consists of several steps. The query graph and target graphs are embedded into vectors, based on which we decide if the query is subgraph isomorphic to a target. The result is audited for bug detection.

the function, first, we find out the method invocation node that contains the key statement of the bug as the center node. Then we implement a program slicing on the function in both forward and backward, regarding the key method invocation as the slicing criteria. The number of steps the slicing program takes is equal to the number of GNN layers, i.e., equals to k. Once we have generated the automated sliced query graph, we remove any unrelated nodes based on certain empirical rules. For example, if a variable in the center statement is related to log or time, and is not relevant to the bug, we exclude it from the analysis.

3.2.3 Node Encoding. To obtain a discrete label for each node while highlighting its core semantics, we do feature normalization on code graphs. Given a label set of size $\mathcal{F}$, each node will be assigned a label from 0 to $\mathcal{F}-1$, according to its feature. We call the normalized graph a *featured graph*. We propose the following three strategies to acquire the size $\mathcal{F}$.

Return encoding. Ignoring the returned value, all *return*-related nodes are the same. We use 0 as the label.

Operator encoding. For a programming language, the number of supported operators is fixed. Assuming there are L_o operators, each operator will be assigned a label within $[1, L_o]$.

Method invocation encoding. Encoding method names into distinct labels can result in a very large vocabulary, which also discards the semantic information among the methods. Therefore, we cluster the method names and assign each cluster one distinct label. The clustering is based on function vectorization, following the procedure of func2vec [12]. We first build a corpus for a given target system by collecting the invoked methods in every function in physical order. We treat a method sequence as a sentence and a method as a word and train a fastText [6] model. We use the K-means algorithm to cluster the vectorized methods. If there are L_m clusters, each cluster (and all the methods in this cluster) will acquire a label within $[L_o + 1, L_o + L_m]$.

In summary, we have $\mathcal{F} = L_o + L_m + 1$, $\mathcal{F}$ varies for different target projects that have different numbers of methods. Figure 4 shows an example of converting the simplified code graph (Figure 4a) associated with the highlighted lines in Figure 1a to a featured graph (Figure 4b).

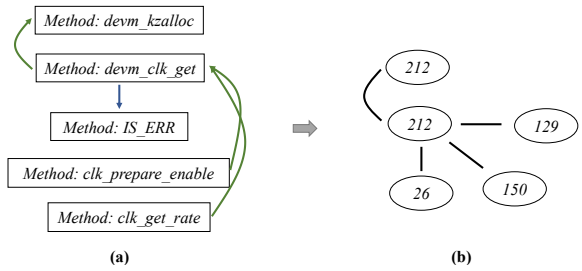

Figure 4: Normalizing a code graph (a) into a featured graph (b). Forward arrows control flow edges and back denotes data dependence.

3.3 Model Training

As estimation-based embedding vectors might not perfectly represent subgraph isomorphism relation between graphs as Eq. 1 and Figure 2 show, we want to train a graph embedding model that can represent quantitative relations in embedding vectors, answering

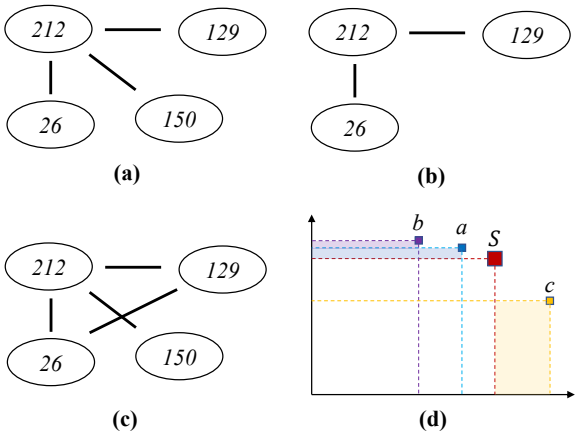

Figure 5: An illustration of subgraph isomorphism with error tolerance.

not only if G_Q is subgraph isomorphic to G_T, but also how more likely G_{Q_1} is subgraph isomorphic to G_T than G_{Q_2}.

Figure 5 shows an example. Let Figure 4b be an anchor graph, denoted as S in Figure 5d. Figure 5a and Figure 5b are two subgraphs of S, and Figure 5b is also a subgraph of Figure 5a. Figure 5c is not a subgraph of S, a or b. Except S, the other three graphs are denoted in Figure 5d using their figure labels. While estimation-based embedding may not place a subgraph to the lower-left of its supergraph as in Figure 2, we use the uncovered area to denote the error, i.e., the shadow in Figure 5d. If we tolerate the error to some extent, we could still achieve a quantitative inference that, b is a subgraph of a, a is a subgraph of S, a is closer to S than b, but c is not a subgraph of them.

Below we will show the details of training a desirable embedding model. We first discuss the loss function to train the model and then talk about how to effectively build a training data set.

3.3.1 Loss Function. To make a quantitative decision well, we propose a *cascading loss* on a series of graphs so that the embedding model can learn to place the embedding space in an order associated with the subgraph coverage degree. The likelihood between G_{Q_1} and G_T is higher than that between G_{Q_2} and G_T when G_{Q_1} covers more nodes/edges of G_T.

The cascading loss requires a series of graphs in each training sample. A positive sample P is in the form of $\langle P_0(S), P_1, \ldots, P_t\rangle$, where $S = P_0$, and a negative sample N is $\langle S, N_1, \ldots, N_t\rangle$. We leave the construction of training samples in Section 3.3.2.

Based on the training samples, we first define two partial loss functions that focus on only the positive samples (Eq. 4) and the negative samples (Eq. 5), respectively.

$$\mathcal{L}_{pos}(S,P) = \sum_{i=1,d=0}^{i\le t,d<|v|} (\max\{0, v(P_{i-1})_d - v(P_i)_d + margin_{pos}\}) \quad (4)$$

$$\mathcal{L}_{neg}(S,N) = \sum_{i=1}^{i\le t} (\max\{0, margin_{neg}\times(t-i) - W(S,Ni)\}) \quad (5)$$

where $v(g)_d$ means the floating value at the d-th dimension of the graph g's dense vector and $|v|$ indicates the dimensions of the dense vector. $W(G_Q, G_T)$ represents the violation score of G_Q being a subgraph of G_T, which is used to make subgraph isomorphism decision and can be analogous to the shadows in Figure 5d. The smaller the violation score, the higher the likelihood.

In practice, we use Eq. 6 to compute the violation score.

$$W(G_Q, G_T) = \sum_{d=0}^{d<|v|} (\max(0, v(G_Q)_d - v(G_T)_d))^2 \quad (6)$$

Eq. 4 says that, for each isomorphic subgraph pairs P_{i-1} and P_i, every dimension of the embedding vector of the larger graph should exceed the smaller one by $margin_{pos}$. For the negative samples, Eq. 5 imposes that the violation score between a negative sample N_i and the anchor graph S is larger than $margin_{neg}\times(t-i)$, since the larger N_i is, the more common nodes it shares with S.

With the above two partial loss functions, we finally define the overall loss function in Eq. 7.

$$\mathcal{L}(S,P,N) = \mathcal{L}_{pos}(S,P) + \mathcal{L}_{neg}(S,N) \quad (7)$$

3.3.2 Training Samples. To train a universal embedding model, we use instantiated backbone samples instead of directly extracting training samples from projects. The backbone samples are language-related but not project-related. By studying popular, large enough, and mature software, we can learn patterns and templates that exhibit the local pattern of the C language grammar and the statistical features of invoking methods and operators. Guided by these templates, we generate uniformly distributed training samples, which helps to reduce model bias.

A backbone graph is a 1~k-hop featured subgraph centered at an invocation node in a function-level code graph. For the nodes in backbone graphs, only the types (e.g., invocation, operator, or return) are kept and the content (e.g., specific method or operator like '*clk_prepare_enable*' and '+') is removed.

Based on the backbone graphs, we generate positive training samples in the form of $\langle P_0(S), P_1, \ldots, P_t\rangle$. We select a backbone graph to instantiate P_t first. The label of P_t is assigned based on the type of P_t's backbone graph, i.e., the label of a node is randomly assigned to a number within $[L_o+1, L_o+L_m]$ if the node type of the corresponding node in the backbone graph is *invocation*. Then we remove some nodes and/or edges in P_t to obtain P_{t-1}, from which we get P_{t-2}. Repeating the procedure and with the constraint that the node number of P_0 is smaller than ten (to handle small code graphs), we get $t+1$ graphs. P_0 is treated as the anchor graph, i.e., S.

Next, we build negative training samples. Each negative sample $\langle S, N_1, \ldots, N_t\rangle$ is built from a given anchor graph S. Here, N_t is instantiated from another backbone graph that is irrelevant to S. Note that, in this study, we enforce that N_t contains all the nodes (labels) in S after the instantiation but S is not a subgraph of N_t. Following the same procedure of generating a positive sample, we get N_t's subgraph sequence, i.e., $N_{t-1}, \ldots, N_1$.

3.4 Graph Embedding

A featured subgraph is fed into the embedding model to generate a low-dimensional dense vector. Each node label is converted into a distributed feature through an embedding layer, after which a k-layer network embeds the featured subgraph into a floating vector. At the j-th layer, feature update for node n is done as Eq. 8, where $\mathcal{N}(n)$ is the neighbor of n.

$$h_n^{(j)} = MLP^{(j)}\left((1+\epsilon^{(j)})\cdot h_n^{(j-1)} + \sum_{u\in\mathcal{N}(n)} h_u^{(j-1)}\right) \quad (8)$$

The graph embedding finally concatenates each layer's summation as Eq. 9.

$$v_G = MLP_G\left(\sum_{n\in G} \underset{0\le j\le k}{\text{CONCAT}}\left(h_n^{(j)}\right)\right) \quad (9)$$

3.5 Bug Detection Based on Subgraph Isomorphism Decision

Bug detection on top of the vector-based measurement takes the following four steps, given a featured query graph G_Q and all the target graphs in the database.

First, we exclude a target graph G_T from candidates of G_Q if any node in G_Q does not exist in G_T. The exclusion is done by generating an $\mathcal{F}$-dimensional bag-of-words vector for each graph, where each dimension indicates occurrences of the corresponding node. Notating the vector as $v^m(\cdot)$, G_T is kept if $v^m(G_T)_d \ge v^m(G_Q)_d$ is satisfiable for $0 \le d < |v^m|$. Second, we examine if $W(G_Q, G_T) < \tau$, where τ is the error tolerance threshold, for each G_T that has passed the above check. Only when the violation score is lower than the threshold, can G_Q be regarded as a subgraph of G_T. Third, to eliminate similar code snippets that have already fixed the bugs, we take the fixed version of G_Q, i.e., $G_{Q'}$, to match G_T, following the above two steps. If $W(G_{Q'}, G_T) < W(G_Q, G_T)$, we consider the target code has been fixed and G_T will not be left for human auditing. Last, all the remaining candidates are sorted by their violation scores in ascending order. We then manually audit the top-ranked graphs and report discovered bugs to the developers.

It is notable that, some code snippets may consist of slightly different bug-related elements but have the same bug. For example, different operations on a freed pointer after a free operation can lead to the same use-after-free bug. However, a query can contain only one type of non-critical element. Limiting the matching to query-involved operations only can lead to missing bugs. We address the issue by sampling a few other node types to replace a non-critical node (e.g., the operation on a freed pointer) in the query graph and make a subgraph isomorphism decision between each new query graph and the target set. The result for each new query is separately ranked and audited, but the detected bugs are combined into G_Q's result.

4 EVALUATION

We implemented a prototype of SICode. Parsing source code is implemented on top of *fuzzyc2cpg* v1.1.19 [1]. Other components are implemented in Python. In this section, we will evaluate SICode to answer the following four questions.

Table 1: Performance on the synthetic testing set for τs.

τ	TP	FP	TN	FN	Precision	Recall	F1
1.5	825	175	9897	103	0.889	0.825	0.856
...			...				
1.75	850	150	9880	120	0.876	0.850	**0.863**
...			...				
2	857	143	9863	137	0.862	0.857	0.860

- **RQ1.** How scalable is SICode to make code-oriented subgraph isomorphism decisions in large software projects? (Section 4.2)
- **RQ2.** How effective is SICode in detecting bugs? (Section 4.3)
- **RQ3.** Can SICode outperform comparative methods in bug detection? (Section 4.4)
- **RQ4.** Is the proposed cascading loss necessary? (Section 4.5)

In the following sections, we will first introduce the experimental setup and scalability. Then, we present the results of bug detection and comparative analyses.

4.1 Experimental Setup

All the experiments are executed on a server with 64GB memory and a GeForce RTX 2080 Ti GPU. The depth of the graph neural network is set to four, i.e., $k = 4$. The number of nodes of training samples is strictly bounded to make sure of the model's performance, i.e., the node number of P_0/S is less than ten and the node number of the target graphs P_is/N_is increases from 10 to 120 according to curriculum training scheme [4], and the same restriction is set to the testing samples. The number of positive and negative instances in a series sample is set to 5, i.e., $t = 5$.

Training set. We choose OpenSSL, a popular C language project, with around 0.7MLOC, to generate backbones. The embedding models of different sizes in this paper are all trained based on these backbones. We use 80% of the functions in OpenSSL to extract the backbone graphs for the training set.

Synthetic Testing Set. A synthetic test set can provide a more universal test result than that from a specific project to test our model's performance. For hyper-parameter tuning and Section 4.5, we use the rest 20% of OpenSSL functions to extract the backbone graphs for the testing set. To allow all tests to pass the first check in Section 3.5, all nodes in Q exist in T_P/T_N in $\langle Q, T_P\rangle/\langle Q, T_N\rangle$. Since there are far more non-isomorphic pairs in the real world, the proportion of positive to negative samples is set to 1:10. Furthermore, the size of tested graphs influences the embedding model's performance. Since 95% of the 4-hop featured graphs include fewer than 28 nodes, the size of target graphs is set to 28.

Target Projects. SICode can be applied to any C project for detecting intra-procedural bugs. We apply the method to perform matching-based bug detection in large open-source C projects, OpenSSL, VLC, and the Linux kernel (Linux for short, v5.17 and v6.2.5). These projects have detailed git logs and responsible maintainers, indicating that the submitted bug reports are likely to receive a quick response. Moreover, these projects have a rich history of submission, making it easier to choose seed bugs for retrieval.

Table 2: Time cost for each step of SICode on three code bases.

Targets	Code graph & corpus generation	Method encoding model training	Clustering	Featured graph embedding	Retrieval
Linux	2h3m48s	4m17s	1h2m24s	2h30m39s	3m39s
VLC	3m22s	19s	7s	4m1s	22s
OpenSSL	2m38s	13s	6s	4m43s	9s

We train two embedding models, namely $Model_S$ and $Model_L$, to embed Openssl/VLC and Linux v5.17/v6.2.5 respectively, since their size varies greatly. The values of L_o and L_m are set to 23 and 76 in $Model_S$, 23 and 476 in $Model_L$. For these projects, the number of 4-hop featured graphs ranges from 78k to 2.6 million.

Selection of k. The models are not project-specific, therefore, to encode a certain project, k can be assigned to a number less than or equal to L_m. In our preliminary research, we manually analyzed the clustering results, and we consider that the average number of methods in clusters ranges from tens to hundreds is suitable. The values of k are set to 76 for Openssl/VLC, and 476 for Linux.

Metrics. The headers *TP*, *FP*, *TN*, *FN* in Table 1 and Table 5 indicate the number of True Positives (isomorphism graphs reported as isomorphism), False Positives (non-isomorphism graphs reported as isomorphism), True Negatives (non-isomorphism graphs reported as non-isomorphism), and False Negatives (isomorphism graphs reported as non-isomorphism). *Precision* describes the quality of the positive reports, calculated as $\frac{TP}{TP+FP}$, while *Recall* describes the ability to capture all positive samples, calculated as $\frac{TP}{TP+FN}$. The metric *F1-Score* comprehensively evaluates the model, calculated as $\frac{2\times Precision\times Recall}{Precision+Recall}$.

Comparative isomorphism decision algorithms. Two isomorphism algorithms are compared, VF2 [9] for straightforward subgraph isomorphism decision and NeuroMatch for embedding-based subgraph isomorphism decision. We compare SICode to VF2 to show how much the isomorphism decision can be accelerated by the embedding-based method. Meanwhile, we compare SICode to NeuroMatch to demonstrate the effectiveness of the cascading loss.

Decision Threshold Tunning. The decision threshold τ varies for embedding models with different input/output sizes. For instance, we evaluate the embedding model with 22 types of operator labels, 76 types of method labels (100 types of labels in total), and 256-dimensional output embedding vectors. We test the model on the synthetic testing with different τ (from 1 to 2). As Table 1 shows, increasing the decision threshold drops the precision while raising the recall. The best F1 score is achieved when $\tau = 1.75$. Therefore, 1.75 is the decision threshold for this embedding model. The threshold will be used in Section 4.5.

4.2 Scalability

We evaluate the scalability of SICode about subgraph isomorphism decisions on Linux v5.17, VLC, and OpenSSL.

Fix-based warning suppression and sampling-based detection augmentation are not included, i.e., only the first two steps in Section 3.5 are taken in the scalability evaluation. Table 2 describes the time cost of each step. The one-time offline stage (the four columns as a whole) costs about 5.7 hours on Linux, which is acceptable for such a large-scale code base, and less than eight minutes for the other two. Given one query, SICode takes 3.65 minutes to retrieve the candidate graphs in Linux and a few seconds in VLC and OpenSSL. The query efficiency can be further improved by paralleling techniques.

Comparison to isomorphism decision algorithms. We compare the embedding-based methods to a classic subgraph isomorphism decision algorithm, i.e., VF2 [9], to further demonstrate the efficiency of our approach to the Linux data set. We use VF2's public Python implementation [23] for comparison. Without paralleling, VF2 takes **3h17m54s**, for a query graph on the whole Linux data set. In contrast, the embedding-based subgraph isomorphism decision methods (NeuroMatch and SICode) require **3mim24s** and **3min39s**, respectively. Hence, it makes more than 50 times acceleration.

NeuroMatch takes less time than SICode because SICode performs a filtering step (see Section 3.5), which means one more comparison of many pairs. However, from the time cost result, we can see that, even for a large code base like Linux, the filtering step does not bring much extra time (15 seconds, 6.8% of the total time).

In summary, the embedding-based subgraph isomorphism method has shown great scalability compared to classic algorithms.

4.3 Real-world Bug Detection

We apply SICode to open-source C projects OpenSSL, VLC, Linux v5.17, and Linux v6.2.5 to find real-world bugs.

We collect bugs from the issue lists and manually select bug-related elements to build query graphs, as described in Section 3.2.2. Then we go through the report list for each retrieval, checking the top twenty results that have the smallest violation scores. When reviewing the reports, we first examine the matched position to see if it is similar to the known buggy locality, and then we read the code to determine whether there is a bug. The suspected bugs will be reported to the developers. It takes a skilled auditor roughly half an hour to go through the report list for one retrieval.

Table 3 shows the bugs SICode detected. All bugs are reported to their developers and have been confirmed except Bug #19. Bugs #1 ~#19 rank in the top ten whereas Bug #20 ranks at the 14th. The last column in Table 3 shows the SICode ranking of each detected buggy function corresponding to its query. The result demonstrates that SICode effectively detects bugs via subgraph isomorphism decision-based code matching.

4.4 Comparison with Other Tools

We employ several code-matching-based bug-finding tools on the target projects with the same buggy queries that SICode used. The comparative tools implement code matching on different data structures such as tokens, text, trees, and graphs, and different matching levels such as text matching, set matching, graph matching, and embedding vector matching. Specifically, VUDDY [18] is based on whole-text hash matching, ReDeBug [16] is based on n-token matching, CCGraph [42] is based on graph matching using WL-kernel [27], and NIL [22] is based on longest common subsequence matching, which is a kind of sequential token matching. InferCode [8] is a code embedding method based on a language model that tries to predict subtrees, and it identifies similar codes by

Table 3: The details of bug detection.

ID	Projects	Buggy Queries	Detected Buggy Functions	NeuroMatch	V	R	C	N	I	SICode
1	OpenSSL	*asn1_print_info*	*www_body*	89	✗	✓	✓	✗	✗	7
2			*rev_body*	89	✗	✓	✗	✗	✗	9
3			*setup_trace_category*	89	✗	✗	✗	✗	✗	10
4		*www_body*	sv_body	44	✗	✓	✓	✗	✓	9
5			*s_client_main*	44	✗	✗	✓	✗	✗	2
6		*ndef_prefix*	*do_dump*	30	✗	✗	✓	✗	✗	5
7	VLC	*EsOutProgramAdd*	*EsOutAddLocked*	59	✗	✗	✗	✗	✗	6
8	Linux v5.17	*lpfc_els_rcv_lcb*	*__add_inode_ref*	N/A	✗	✗	✗	✗	✗	3
9		*lpass_platform_pcmops_open*	*pm8001_send_abort_all*	N/A	✗	✗	✗	✗	✗	2
10	Linux v6.2.5	*s3c2410wdt_probe*	*mmphw_probe*	294	✗	✗	✗	✗	✗	1
11			*aspeed_gfx_load*	48	✗	✗	✗	✗	✗	8
12		*dryice_rtc_probe*	*gsbi_probe*	N/A	✗	✗	✗	✗	✗	2
13			*bcm_sf2_sw_probe*	89	✗	✗	✗	✗	✗	1
14			*imx_ocotp_probe*	89	✗	✗	✗	✗	✗	3
15		*mxc_rtc_probe*	*korina_probe*	N/A	✗	✗	✗	✗	✗	3
16			*nokia_bluetooth_serdev_probe*	195	✗	✗	✗	✗	✗	4
17			*bcm2835_spi_probe*	195	✗	✗	✗	✗	✗	10
18		*ip6erspan_tunnel_xmit*	*erspan_fb_xmit*	3	✗	✗	✓	✓	✗	3
19*	Linux v6.2.5	*mmci_probe*	*spmmc_drv_probe*	N/A	✗	✗	✗	✗	✗	9
20**	OpenSSL	*asn1_print_info*	*setup_trace_category*	89	✗	✗	✗	✗	✗	14

Headers 'V','R','C','N', and 'I' refer to VUDDY[18], ReDeBug[16], CCGraph[42], NIL [22], and InferCode[8] for short which are compared with our tool in Section 4.4. Bug #1 in function *www_body* is different from the buggy query #4 in the same function. Bug #20 is a different function from Bug #3, though sharing the same function name.
*has not been confirmed by the developers.
**confirmed by the developers but ranked at the 14th.

Table 4: Detection results of the comparative tools based on the same buggy queries in Table 3.

Tool	Reports	Bugs	R@10(%)	R@20(%)	SICode Hits
VUDDY	0	0	-	-	-
ReDeBug	3	3	10.7	-	3
CCGraph	top 20	9	7.1	32.1	6
InferCode	top 20	6	17.9	21.4	1
SICode	top 20	**20**	**67.9**	**71.4**	-

comparing the similarity of embedding vectors. CCGraph and InferCode report the most matched functions sorted by similarity, and we audit the top twenty results for each query. NIL got stuck in the Linux project detection, therefore we only implemented it to a subdirectory that includes all the buggy queries and bugs that SICode found.

The performance of the five comparative tools is shown in the *V*~*I* columns in Table 3. VUDDY (*V*) fails to find any bugs even working at its highest abstraction level. ReDeBug (*R*) discovers bugs #1, #2 and #4. CCGraph (*C*), which performs graph isomorphism for bug detection, has the best performance among the five tools. Bugs #1, #4~#6, #18 are within the top twenty highest WL-kernel similarity to the queries. NIL (*N*) finds bug #18, which shows both similar logic and tokens with the query function. Infercode (*I*) only ranks bug #4 within the top twenty.

To test the tools' recall, we audit all the reports based on the buggy queries in Table 3. SICode and four comparative tools (VUDDY, ReDeBug, CCGraph, and InferCode) find 28 bugs in total. We analyze the bugs and the details are shown in Table 4. Among the five tools, SICode can find the most amount of bugs. Then we calculated the recall of bugs within the top 10 and top 20 results (columns *R@10* and *R@20*), which reflect the accuracy of the tools to some degree. According to the results, SICode achieves the highest R@10 and R@20, indicating that our method is effective and accurate.

We discovered that some bugs, which are only relevant to a local part of code, can be matched through our tool, and could not be captured by whole-graph (or text, tree) matching tools, such as CCGraph, VUDDY, and InferCode. In addition, our tool extracts the semantics of the invoked methods, and therefore other tools like ReDeBug/NIL cannot match most of the bugs that have textual differences from the query bug. Nevertheless, CCGraph and InferCode find bugs that SICode missed (the last column in Table 4). These bugs have high overall similarity to the queries but are not substructural similar to the query, while SICode focuses on substructure similarity. Therefore they are not detected by SICode. However, according to the experiment result, in real-world bug detection, concentrating on substructural similarity is more efficient.

4.5 Ablation Study

The cascading loss function can be considered as an updated version of NeuroMatch's loss (described in Section 2.3). The key difference is that the cascading loss considers the relationship between samples within a group, which is an important piece of information. When only considering one pair of graphs in a group of training samples, the cascading loss collapses to NeuroMatch's loss. In this section, we first compare the effectiveness of the full-featured cascading

Table 5: Ablation experiment on the synthetic testing set.

Model	TP	FN	TN	FP	Precision	Recall	F1
NeuroMatch	543	457	7844	2156	0.201	0.543	0.293
SICode	850	150	9880	120	0.876	0.850	**0.863**

loss and its collapsed version (i.e., the loss adopted by NeuroMatch) to demonstrate its necessity.

Table 5 shows the highest F1 score of each model and the precision and recall it achieves. Model *NeuroMatch* is the original embedding model from its open source repository [2]. Both models are trained to accept the same size of inputs and output the same size of embeddings. *NeuroMatch* achieves the highest F1 when τ is set to 1, the precision is 20.1%, and the recall is 54.3%. The overall F1 is only 29.3%. SICode, in contrast, achieves a higher F1 as 86.3% ($\tau = 1.75$), with 87.6% precision and 85.0% recall.

Second, we compare the model's performance in detecting bugs with two loss functions accordingly. For the bugs listed in Table 3, only bug #18 can be discovered in top-ten auditing by NeuroMatch, from the *NeuroMatch* column. Among the others, three bugs require inspecting more than one hundred candidates (#10, #16, and #17) and five are never matched (#8, #9, #12, #15, and #19). SICode produces a high ranking for all of them, so the corresponding bugs are detected.

It is proved that the proposed cascading loss, associated with the specially constructed training set, has a significant impact on facilitating the model's ability to decide subgraph isomorphism relations for code graphs.

5 DISCUSSION

Iterative Retrieval. While our approach works well, there can be some cases in which the bug logic ranges too wide to be summarized into a single query graph. Besides, we may leverage some additional information, such as the patches used by MVP [31], to suppress false positives in which a patched code snippet is matched.

Our approach can be extended to handle these cases with an iterative retrieval strategy. The buggy snippet is decomposed into a series of query graphs denoted as $\{Q_1, Q_2, ..., Q_n\}$. The patched graph centered at the same node as Q_i is denoted as P_i. The retrieved set of Q_i is denoted as $R_{Q_i} = \{r_{Q_i,j} | j \in \{1, 2, ..., |R_{Q_i}|\}\}$. A function $F(r)$ is defined to acquire the function name of the reported graph in r. The intersection operator on sets R_m and R_n is redefined as:

$$R_m \cap R_n = \{r | r \in R_m \wedge F(r) \in \{F(rr) | rr \in R_n\}\} \tag{10}$$

For a pair $\langle Q_1, P_1 \rangle$, if the buggy snippet is repaired by adding some statement, the buggy query graph Q_1 can be a subgraph of a potential buggy code while the patched code graph P_1 is not. Therefore, we can use $R_{Q_1} - R_{P_1}$ to eliminate false positive result.

The final result of the iterative retrieval can be gathered by Eq. 11.

$$R = (R_{Q_1} - R_{P_1}) \cap (R_{Q_2} - R_{P_2}) \cap ... \cap R_{Q_{(n-1)}} \cap R_{Q_n} \tag{11}$$

Complicated Similarity. Our method concentrates on a certain type of bug that is substructurally similar to the query. The substructural likeness reflects the coding habits of the developers. For example, though the functionalities of code vary greatly, developers write similar substructures to check the type of variables.

Due to the characteristic of subgraph isomorphism, the detected bugs are substructurally identical to the query bugs, and these similar substructures can be viewed as Type II or Type III clones. Our method fails to specifically identify those bugs with the same semantics but almost completely different substructures like the Type IV clones, because of the strict isomorphic relationship.

To capture more complicated substructural similarity, one idea is to strengthen the ability of normalization, for instance, rewriting the source code. For example, two synonymous value judgment statements *if (code)* and *if (code != NULL)* will be parsed into different graph structures, resulting in differences in embedding vectors. The rewriting procedure can identify the similarity and convert them to similar expressions.

6 RELATED WORK

6.1 Static Analysis for Bug Detection

Experienced analyzers use checkers(e.g. Cui et al. [10]) or templates (e.g. iComment [28], AutoISES [29]) based on security knowledge of the code system to implement bug detection. To acquire security knowledge automatically, PR-Miner [19] utilizes frequent item set mining to efficiently extract implicit programming rules. Rasthofer et al. [25] find the privacy concerned APIs with machine learning. VCCFinder [24] highlights the author of the source code as a feature. APISan [39] analyzes code with symbolic executing, calculating the frequency of occurrences of coding patterns, and recognizing violations as bugs. AntMiner [20] improves the precision of rule mining by taking critical operations as slicing criteria. APEx [17] gathers the executing paths to find the differences between the main-logic path and the fault-handling path. NAR-miner [5] extracts negative association programming rules from large-scale systems.

6.2 Code Matching Based Bug Detection

Apart from extracting rules automatically, matching-based methods alleviate the problem of lacking security knowledge.

Yamaguchi et al. [38][36] extract raw features from code property graphs for matching. ReDeBug[16] uses a syntax-based approach to find unpatched code clones and has scalability. David et al. [11] search similar executables using the short piece of executing path called tracelet as a matching unit. VUDDY [18] uses the hash of normalized source code statement as the feature set.

A code graph contains richer information than plain text and, therefore is widely used in code matching. DiscovRE [13] identifies similar functions across different compilers based on the structure of control flow graphs. Genius [14] encodes the graph structural information of CFGs into feature vectors to make the feature more robust through different compilers. Xu et al. [34] locate structural similarity on a multi-scale using a feature called Multi-level Birthmark to overcome the obfuscation caused by cross-compilation. Zhang et al. [40] use maximum vertex matching to measure the similarity of the buggy snippet and the candidate code. However, it cannot make sure the vertex in the buggy substructure is connected. On the contrary, using subgraph isomorphism can make sure the buggy substructure exists in the candidate code.

Patch information shows the essential statements that cause a bug. Taking advantage of patches in code matching can restrain false positive reports. Xu et al. [35] extract traces on binary to express the code graph structure and identify potential bugs when the trace set is more similar to the buggy function than the patched one. VGRAPH[7] generates the graph-based representations of the contextual code, the buggy code, and the patched code, and proposes a graph-matching algorithm to find cloned bugs. MVP [31] extracts signatures on source code, and similarly, finds functions with the signature that have higher similarity to the signature of buggy function than the patched function.

6.3 Neural Networks Based Code Matching

With the development of Graph Neural Networks, several methods are proposed to process the information in code based on various code graphs like CFGs and PDGs. Allamanis et al. [3] propose an approach to construct code graphs and scale Gated Graph Neural Networks training to large graphs, and has good performance on VarNaming and VarMisuse tasks. Lambdanet [30] predicts the type of variables using a graph embedding network.

Some neural network-based methods are proposed to find bugs. Gemini [33] is a Siamese-network-based approach that computes the embedding of binary functions and measures the distance between the embedding vectors to find similar bugs. Devign [41] learns the semantics of programs and uses graph classification to identify bugs. Huang et al.[15] propose bytecode-oriented normalization and slicing techniques to embed graphs to vectors to find similar buggy smart contracts.

7 CONCLUSION

In this paper, we propose an effective and scalable bug detector, SICode. SICode applies an embedding-based method to get an approximate solution to the subgraph isomorphism identification problem. The code containing a similar sub-structure to the buggy query can be detected as potential bugs. SICode focuses on substructural similarity, reducing the perturbation caused by bug-unrelated elements through the subgraph matching scheme. SICode trains a graph embedding model, which can generate proper embedding vectors, and subgraph isomorphism relations are maintained in vector-based comparison. A cascading loss function is proposed and an associated training data set is built, making the vector-based measurement to not only decide subgraph isomorphism relations but also quantitatively rank potential subgraphs.

Experiments have shown that SICode has a high scalability compared to traditional subgraph isomorphism algorithms. The cascading loss also makes a much higher identification accuracy than an all-at-once loss. Besides, 20 new bugs are detected by applying SICode to large open-source code bases and 18 confirmed bugs rank within the top ten results of the buggy queries, demonstrating the effectiveness of detecting bugs with subgraph isomorphism identification.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their constructive comments. The work is supported in part by the National Natural Science Foundation of China (NSFC) under grants 62272465, 62272464, and 62132020, and the CCF-Huawei Populus Grove Fund under grant CCF-HuaweiSE202310.

REFERENCES

[1] 2019. fuzzyc2cpg: A fuzzy parser for C/C++ that creates semantic code property graphs. https://github.com/ShiftLeftSecurity/fuzzyc2cpg.
[2] 2021. NeuroMatch. http://snap.stanford.edu/subgraph-matching.
[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).
[4] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, Vol. 382. ACM, 41–48.
[5] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. Narminer: Discovering negative association rules from code for bug detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 411–422.
[6] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.
[7] 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.
[8] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1186–1197. https://doi.org/10.1109/ICSE43902.2021.00109
[9] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2001. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*. Citeseer, 149–159.
[10] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. 2013. Verifying systems rules using rule-directed symbolic execution. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 329–342.
[11] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.
[12] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 423–433. https://doi.org/10.1145/3236024.3236059
[13] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *NDSS*.
[14] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 480–491.
[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 Transactions on Information Forensics and Security* 16 (2021), 2144–2156.
[16] Jiyong Jang, Maverick Woo, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. *login Usenix Mag.* 37, 6 (2012).
[17] Yuan Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: Automated inference of error specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 472–482.
[18] 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.
[19] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 306–315.
[20] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: mining more bugs by reducing noise interference. In *Proceedings of the 38th International Conference on Software Engineering*. 333–344.
[21] Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. 2020. Neural Subgraph Matching. *arXiv preprint arXiv:2007.03092* (2020).
[22] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. NIL: large-scale detection of large-variance clones. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 830–841.
[23] NetworkX. 2022. VF2 algorithm. https://networkx.org/documentation/stable/reference/algorithms/isomorphism.vf2.html.
[24] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential

vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 426–437.

[25] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks.. In *NDSS*.

[26] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* 1, 1 (2008), 364–375. https://doi.org/10.14778/1453856.1453899

[27] 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).

[28] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 145–158.

[29] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations.. In *USENIX Security Symposium*. 379–394.

[30] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161* (2020).

[31] 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.

[32] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[33] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 363–376.

[34] 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.

[35] 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.

[36] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[37] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*. 13–13.

[38] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 359–368.

[39] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security 16)*. 363–378.

[40] 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 *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 64–76. https://doi.org/10.1145/3533767.3534393

[41] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496* (2019).

[42] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 931–942.