Hunting Bugs with Accelerated Optimal Graph Vertex Matching

Xiaohui Zhang[1,#], Yuanjun Gong[1,#], Bin Liang[1,*], Jianjun Huang[1],
Wei You[1], Wenchang Shi[1], Jian Zhang[2]
[1]Key Laboratory of DEKE (MOE), School of Information, Renmin University of China, Beijing, China
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{xiaohuizhang,gongyuanjun,liangb,hjj,youwei,wenchang}@ruc.edu.cn,zj@ios.ac.cn

ABSTRACT

Various techniques based on code similarity measurement have been proposed to detect bugs. Essentially, the code fragment can be regarded as a kind of graph. Performing code graph similarity comparison to identify the potential bugs is a natural choice. However, the logic of a bug often involves only a few statements in the code fragment, while others are bug-irrelevant. They can be considered as a kind of noise, and can heavily interfere with the code similarity measurement. In theory, performing optimal vertex matching can address the problem well, but the task is NP-complete and cannot be applied to a large-scale code base. In this paper, we propose a two-phase strategy to accelerate code graph vertex matching for detecting bugs. In the first phase, a vertex matching embedding model is trained and used to rapidly filter a limited number of candidate code graphs from the target code base, which are likely to have a high vertex matching degree with the seed, i.e., the known buggy code. As a result, the number of code graphs needed to be further analyzed is dramatically reduced. In the second phase, a high-order similarity embedding model based on graph convolutional neural network is built to efficiently get the approximately optimal vertex matching between the seed and candidates. On this basis, the code graph similarity is calculated to identify the potential buggy code. The proposed method is applied to five open source projects. In total, 31 unknown bugs were successfully detected and confirmed by developers. Comparative experiments demonstrate that our method can effectively mitigate the noise problem, and the detection efficiency can be improved dozens of times with the two-phase strategy.

CCS CONCEPTS

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

∗ Corresponding Author.
# Both authors contributed equally to this research.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534393

KEYWORDS

bug detection, code similarity, optimal vertex matching, graph convolutional neural network

ACM Reference Format:
Xiaohui Zhang[1,#], Yuanjun Gong[1,#], Bin Liang[1,*], Jianjun Huang[1], and Wei You[1], Wenchang Shi[1], Jian Zhang[2]. 2022. Hunting Bugs with Accelerated Optimal Graph Vertex Matching. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534393

1 INTRODUCTION

In recent years, bug detection techniques based on code similarity measurement have proven to be very effective [16, 21, 22, 31, 47–52]. Such approaches match a code fragment, e.g., a function containing known bug (seed) with the target ones to detect the suspects that are similar to it. The detected fragments are likely to contain unknown bugs same as the seed.

The code fragments are essentially a kind of graph and can be naturally represented as various code graphs, such as CFGs (Control Flow Graphs) and PDGs (Program Dependency Graphs), etc. In fact, some recent studies have employed the graph embedding technique to analyze programs [13, 14, 19, 22, 31, 37, 44, 48, 51], and detect bugs based on code graph similarity comparison [22, 31, 48, 51].

However, in practice, the logic of a bug often involves only a few statements in the code fragment, while others are bug-irrelevant and can be considered *noise*. As shown in Fig. 1a, if we encode the whole code graph as one vector, the information of the noise code will also be encoded into the vector. As a result, the code similarity measurement task will be heavily interfered with by the irrelevant statements and lead to false positives and false negatives. A natural way to address the noise problem is to remove the noise vertices before embedding and matching. Actually, we can remove the noise from the seed graph because the bug-related statements are often definite and can be identified. For example, in some studies [47, 52], the noise in the seed function are excluded by applying the slicing technique. Unfortunately, we cannot assume there is a specific bug in the target code and mark the related statements to exclude noise in advance. In other words, it is unavoidable that there is still noise in the target code when comparing them with seed. As shown in Fig. 1b, the similarity measurement is still interfered with when only excluding the noise from the seed.

Essentially, the noise problem can be effectively addressed via optimal vertex matching. Namely, if the vertices are optimally paired one by one, the vertices corresponding to the sliced seed can be located and recognized from the target code. The code similarity can be precisely measured based on the paired vertices. As shown in Fig. 1c, the graph similarity can be measured on the vertices of

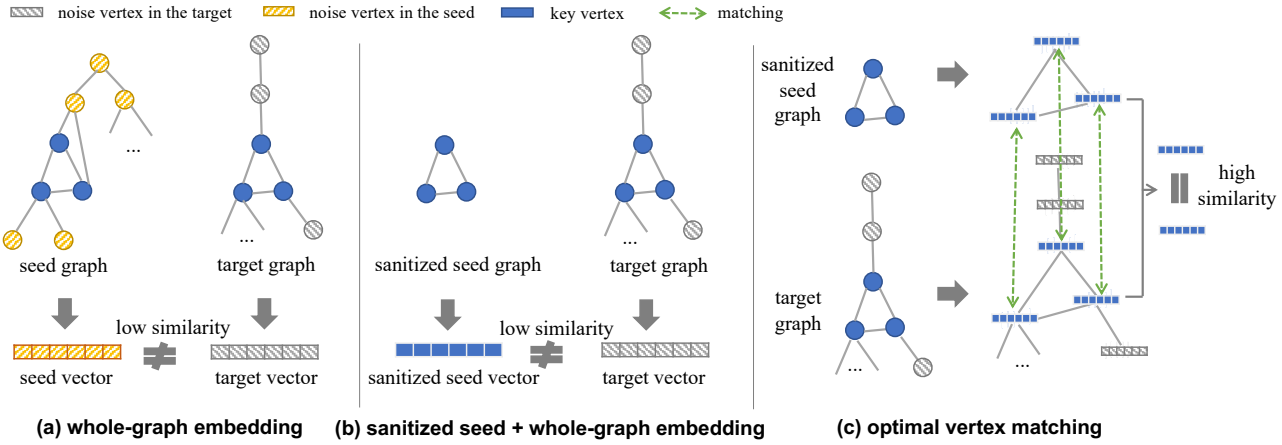

Figure 1: A toy example to motivate our approach. Removing noise from the seed is called sanitizing.

interest, i.e., the bug-related vertices in the seed and their counterparts in the target graph. As a result, the noise vertices are excluded and the obtained similarity is reasonable.

Unfortunately, optimal vertex matching will bring great time overhead. To catch the graph structure information, optimal vertex matching needs to take into account both the first-order and second-order (or even higher-order) similarity between vertices from separate graphs. The obtained matching should maximize the similarity between the matched vertices. Due to its high-order combinatorial nature, the task is in general NP-complete [25, 46, 53]. In practice, it is often necessary to match the seed with hundreds of thousands of targets, and the total time cost is unacceptable.

Based on the above discussion, an important question arises as *how to accelerate optimal vertex matching for measuring the code graph similarity scalably*. In this paper, we design a two-phase method to address the efficiency challenge while suppressing the influence of noise. As illustrated in Fig. 2, we first quickly identify a limited number of candidates with a vertex-matching-oriented embedding model. Subsequently, high-order similarity information between candidates and seed is extracted with a graph convolutional network model, and then be used to get an approximately but accurate enough optimal vertex matching. Consequently, we can efficiently measure the code graph similarity and avoid solving an NP-complete task directly.

Specifically, in the first phase, we train a vertex-matching-oriented embedding model (VME) to encode code graphs to vectors, which can be used to quickly estimate the vertex matching degree between two graphs. With VME, the seed graph and each target graph are embedded into low dimensional dense vectors. The matching scores among them are calculated as cosine similarity on vectors. The target graphs with high matching scores are selected as candidates for further analysis in the next phase. They are likely to have a high vertex matching degree with the seed. In this way, we can exclude most of unmatched target graphs before performing optimal vertex matching.

In the second phase, a graph convolutional neural network, called high-order similarity embedding model (HSE), is built to further speed up vertex matching. Instead of directly seeking an optimal vertex matching for the seed and a candidate, we first input them into HSE to get a high-order affinity matrix. The high-order similarity among vertices can be directly encoded into the matrix with graph convolution layers to avoid combination explosion. From it, Hungarian algorithm [29] is employed to get an approximately optimal vertex matching. The similarity between the seed and candidate is computed by averaging the vector similarity of their matched vertex pairs. Finally, the candidates ranked high similarity will be audited to determine whether there is a bug.

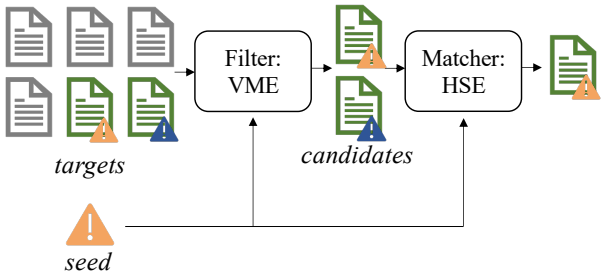

Figure 2: Two-phase bug detection method based on vertex matching.

We evaluate our method on five open source projects: OpenSC, SQLite, mruby, ImageMagick, and gpac. We successfully detect 31 unknown bugs, which have been confirmed by their developers. The comparison experiment shows that, our approach can get the best results in 25 (80.6%) detected bugs compared with the other methods. At the same time, the two-phase strategy can dramatically reduce the time by 94%, completing one query in the five projects only need 18 seconds on the average.

The contributions of this paper are as follows:

- We propose a bug detection method based on code graph similarity measurement. The influence of the noise code can be effectively suppressed by optimal vertex matching to reduce the false positives and false negatives.

```
//file: src/filters/reframe_adts.c in gpac
GF_Err adts_dmx_process(GF_Filter *filter)
{
  [...] // omit 187 lines
+ if (ctx->hdr.frame_size < ctx->hdr.hdr_size) {
+   GF_LOG(GF_LOG_WARNING, GF_LOG_PARSER, ("[ADTSDmx] Corrupted ADTS
      frame header, resyncing\n"));
+   ctx->nb_frames = 0;
+   goto drop_byte;
+ }
  [...] // omit 9 lines
  size = ctx->hdr.frame_size - ctx->hdr.hdr_size;
  [...] // omit 17 lines
  dst_pck = gf_filter_pck_new_alloc(ctx->opid, size, &output);
  [...] // omit 2 lines
  memcpy(output, sync + offset, size);
  [...] // omit 53 lines
}
```

(a) A known bug

```
//file: src/filters/dmx_mpegps.c in gpac
GF_Err m2psdmx_process(GF_Filter *filter)
{
  [...] // omit 71 lines
- if((buf[buf_len-4] == 0) && (buf[buf_len-3] == 0) &&
      (buf[buf_len-2] == 1)) buf_len -= 4;
+ if ((buf_len>4) && (buf[buf_len - 4] == 0) && (buf[buf_len - 3] ==
      0) && (buf[buf_len - 2] == 1)) buf_len -= 4;
  [...] // omit 22 lines
  dst_pck = gf_filter_pck_new_alloc(st->opid, buf_len, &pck_data);
  memcpy(pck_data, buf, buf_len);
  [...] // omit 21 lines
}
```

(b) A detected and confirmed bug

Figure 3: An motivating example in gpac.

- We design a two-phase strategy to accelerate optimal vertex matching to avoid solving an NP-complete problem. Two embedding models are designed to get an approximately but accurate enough optimal vertex matching. As a result, our method can be applied to large-scale code base.
- We detected 31 confirmed unknown bugs in five real-world projects with the proposed method. The comparison experiment shows that many of them are difficult to be detected with other similar methods.

2 MOTIVATING EXAMPLE

Taking a confirmed bug in gpac to motivate our technique of optimal vertex matching based code similarity measurement for bug detection.

Fig. 3a involves a known bug [2]. In the function `adts_dmx_process`, a crafted file may cause `ctx->hdr.frame_size` to be smaller than `ctx->hdr.hdr_size`, resulting in `size` to be a negative number and a heap overflow in `memcpy()`. Fig. 3b includes the same bug, in which, `buf_len` can be negative.

While both functions contain hundreds of lines of code, the core logic involves only a few statements. If we represent the code snippets as graphs in which each vertex relates to one statement, there are only a few bug-related vertices in each graph and all the others are bug-irrelevant. In other words, most of the vertices can be regarded as noise in a code similarity measurement based bug detection method.

Leveraging graph embedding techniques to encode each graph to a vector and measuring the similarity between vectors will be

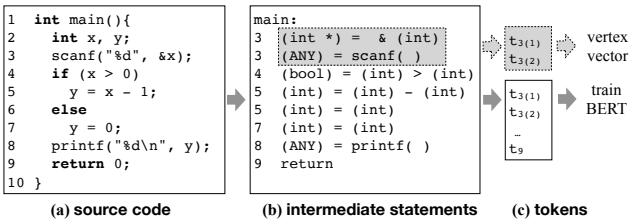

Figure 4: An example of code graph preprocessing.

inevitably interfered by the noise vertices. The potential bug can be difficult to identify. Our experiment shows that such a scheme ranks Fig. 3b the 212th using Fig. 3a as the seed. Note that removing the noise from the seed cannot make the situation much better. The buggy `m2psdmx_process` is ranked the 5031st, hardly to be noticed by an analyst in manual auditing.

A natural way is to match the bug-related statements (vertices) in the seed exactly to the ones in Fig. 3b, i.e., adopting an optimal vertex matching scheme. We propose to sanitize the seed graph, embed the vertices via a graph convolution network and adopt an attention mechanism to suppress the noise interference in the target graph. Our approach ranks Fig. 3b as the twelfth candidate to Fig. 3a. We report the bug to the developers and get it confirmed.

It is notable that, in a large-scale bug detection scenario, optimal vertex matching is extremely expensive. To address the challenge, we propose a two-phase retrieval method, which quickly filters the targets and leaves only a small number of candidates for optimal vertex matching. The efficiency can be significantly improved. More details will be presented in Section 3.

3 OUR APPROACH

We propose to detect bug by measuring the similarity between the buggy function (*seed*) and the other functions (*targets*) in a code base. The approach first generates the code graphs for the functions and adopts BERT [20] to initialize the vertex vectors. After that, a two-phase method, involving ***candidate filtering*** and ***vertex matching***, is performed on the vectors to detect bugs. Through a specially designed VME (Vertex-Matching-oriented Embedding model), the first phase can dramatically reduce the number of candidates that are left for vertex matching. In the second phase, we present HSE (High-order Similarity Embedding model) to encode the high-order similarity among vertices between a candidate and the seed, and then obtain an approximately optimal vertex matching. The similarity values of the matched vertices are averaged to denote the similarity between the candidate and the seed. Highly ranked candidates will be manually audited. Below we will discuss the code graph preprocessing and the two phases in detail.

3.1 Code Graph Preprocessing

We develop a robust parser to process the source code. The statements are parsed to three-address-code intermediate statements as done in GCC. And we extract one code feature graph per function by combining its CFG and PDG. Note that, different with traditional CFG [12], each generated basic block corresponds to only one statement in the source code. Our intuition is that a basic

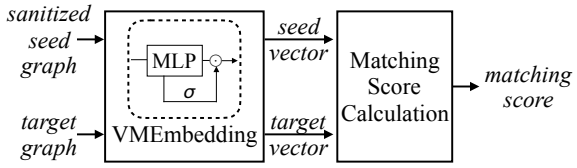

Figure 5: Workflow of VME.

block containing multiple statements will mix the features together and cause mutual interference. The PDG vertices are created by the same way. In other words, the intermediate statements of one source code statement form a vertex in the code feature graph. Identifier normalization, i.e., replacing variable names with their normalized types, is applied to the graph as well. Fig. 4(b) shows an example of normalized intermediate statements and the split of vertices (by source code lines). The code graph of the seed is further sanitized by removing the noise vertices with the knowledge of the bug and its corresponding patch, as done in [47, 52].

Treating each intermediate statement as a token and the token sequence of a function as a sentence, we employ the idea of the mask language model and leverage BERT [20] to train an embedding model. By this means, each token (i.e., an intermediate statement) can be embedded into a vector. If a vertex contains only one token, the vertex vector is exactly the token vector. Otherwise, we concatenate the tokens into a phrase and take the model to compute a vector for the vertex.

Take the function in Fig. 4(a) as example. Its intermediate representation is shown in Fig. 4(b). The corresponding sentence is composed of eight tokens, i.e., $t_{3(1)}, t_{3(2)}, t_4, \ldots, t_9$, where the subscripts indicate the original line numbers and the counters in parenthesis denote the indices of the intermediate statements corresponding to one source code statement. The vertex vectors for $t_4, \ldots, t_9$ are directly the token embeddings and the vertex vector for line 3 in Fig. 4(a) is the vector of the phrase "$t_{3(1)}\ t_{3(2)}$".

3.2 Phase 1: Candidate Filtering

We propose a vertex-matching-oriented embedding model (VME) in the first phase, which encodes the code graphs to vectors to support fast estimation of the vertex matching degree between two graphs. By this means, we can quickly filter the targets and leave only a limited number of candidates for further vertex matching.

The workflow of VME is shown in Fig. 5. One sanitized seed graph and a target graph are fed to the network. Each graph is encoded to a vector via an embedding module and a matching score between the two vectors is calculated. In the training stage, the matching score is used to guide the training. In the testing stage, it can indicate the estimated vertex matching degree.

3.2.1 Vertex-matching-oriented Embedding. The embedding module consists of a multi-layer perceptron network with a gate structure. The gate structure transforms vertex representations via Eq. 1.

$$\boldsymbol{h}'_i = \sigma(MLP(\boldsymbol{h}_i)) \odot MLP(\boldsymbol{h}_i) \tag{1}$$

where σ is the *sigmoid* function, $\boldsymbol{h}_i$ denotes the initial vector of the i^{th} vertex and $\boldsymbol{h}'_i$ represents the transformed output. The symbol $\odot$ denotes the Hadamard product, i.e., multiplying the elements in the same positions of the two matrices one by one. Theoretically, when Hadamard product is used to weight each dimension of the vector with learnable weights, the model can learn which dimensions are important or less important. The importance will be reflected in corresponding weights. Experimental results also show that $\odot$ outperforms simple summation.

We use max pooling to aggregate the vertex vectors and generate the graph vector $\boldsymbol{g}$ with d dimensions. Each dimension of $\boldsymbol{g}$ holds the maximum values of the same dimension of all $\boldsymbol{h}'_i$, as in Eq. 2.

$$\boldsymbol{g}[j] = \max(\boldsymbol{h}'_1[j], \boldsymbol{h}'_2[j], ..., \boldsymbol{h}'_n[j]) \quad j = 1, 2, ..., d. \tag{2}$$

To train a proper VME that can generate graph vectors to estimate optimal vertex matching between graphs, we build a series of quintets $\langle ACR, POS, NEG, s_1, s_2 \rangle$, each with three graphs and two labels. Given a graph POS, which is randomly chosen from the code graphs in the target project, we generate a sub-graph ACR from it. More specifically, ACR is generated from POS by taking a random vertex of POS as the centre and including its r-hop neighbor vertices. Then we randomly sample another target graph as NEG. We have $s_1 = 1$, indicating the satisfaction of optimal vertex matching between ACR and POS. The other label, s_2, is calculated via the Hungarian algorithm [29] to denote the optimal vertex matching degree between ACR and NEG.

We employ a triple loss, as shown in Eq. 3, as the loss function, to strengthen the model's ability of identifying that ACR and POS are a better matching.

$$L = \max\left(0, msc(v_{acr}, v_{pos}) - msc(v_{acr}, v_{neg}) - (s_1 - s_2)\right) \tag{3}$$

where msc measures the matching score of two graph vectors.

In the traditional triplet loss function, there is a constant margin ϵ, which can ensure that there is a gap between the positive sample and negative sample. In fact, $s_1 - s_2$ is the margin in our loss function, but it is dynamic and specific for each sample. For a positive sample and a negative one, we can measure the gap in advance ($s_1 = 1$, s_2 can be estimated through the Hungarian algorithm).

3.2.2 Matching Score Calculation. Instead of directly measuring the similarity between two graph vectors $\boldsymbol{s}$ and $\boldsymbol{t}$, we focus on only the dimensions which, in the seed vector, are greater than a certain threshold T. Those dimensions are extracted to form two new vectors that are used to calculate the matching score. Eq. 4 and Eq. 5 show the construction of the new seed and target vectors. Eq. 6 calculates their cosine similarity as the corresponding matching score. The experiment shows that $T = 0.02$ is a proper threshold.

$$\boldsymbol{s}' = \bigcup_{\boldsymbol{s}[i]>T} \boldsymbol{s}[i] \tag{4}$$

$$\boldsymbol{t}' = \bigcup_{\boldsymbol{s}[i]>T} \boldsymbol{t}[i] \tag{5}$$

$$msc(\boldsymbol{s}, \boldsymbol{t}) = \frac{\boldsymbol{s}' \cdot \boldsymbol{t}'}{\|\boldsymbol{s}'\|\|\boldsymbol{t}'\|} \tag{6}$$

where $\bigcup$ connects the satisfied dimensions and "$\cdot$" computes the dot product of two vectors.

In practice, we sort the candidates in a descending order and retain only the top $K1$ for a fine-grained optimal vertex matching.

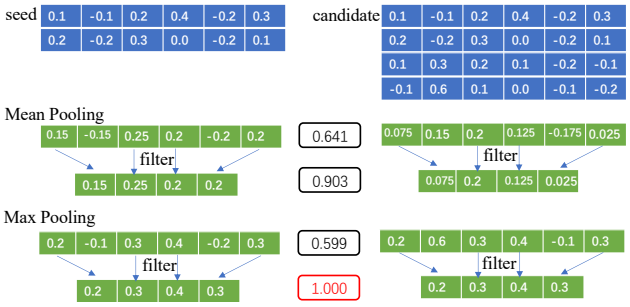

Figure 6: A demo example to show the effect of max pooling plus dimension filtering.

As a consequence, the filtering can dramatically reduce the number of candidates and thus significantly speed up the bug detection.

3.2.3 Example. We take Fig. 6 as an example to show that *max pooling + dimension filtering* behaves better than other techniques. Suppose the seed graph contains two vertices and the candidate contains four, with two the same as the seed and two noise nodes. Here, we assume that T is set to 0, i.e., to filter negative values.

Note that, for a seed vector s and a target vector t, we find out which dimensions in s are smaller than the threshold T. These dimensions will be deleted from s, and the remaining dimensions form a new vector s'. For t, we also delete the counterparts to get t', no matter whether the dimensions are smaller than the threshold T or not. For example, in Fig. 6, although 0.15 and 0.6 in t are greater than 0, they should be filtered out as the corresponding dimension in s is deleted.

Mean pooling along with direct similarity measurement produces a similarity of 0.641. Applying the same dimension filtering policy, the similarity is 0.903. Max pooling, in contrast, emits a similarity of 0.599, but the combination of max pooling and dimension filtering brings the similarity to 1.0.

Although mean pooling is commonly used in graph embedding models, we find it will completely mix the vertex information and every dimension in the candidate vector can be affected by the noise vertices. Max pooling keeps the dominant feature of each dimension, but it also leaves the noise features in the resulting vector and makes even lower similarity. Filtering the dimensions based on the seed actually highlights the important dimensions and reduces the impact of the seed-independent noise as far as possible. In the above example, the filtering can significantly lift the similarity and thus rank the candidate sufficiently high for next-phase analysis. Among the four methods, we can see that filtering plus max pooling achieves the best result, which has also been demonstrated in our preliminary experiment.

3.3 Phase 2: Vertex Matching

The second phase performs approximately optimal vertex matching to identify potential buggy candidates. To avoid combinational explosion resulting from directly solving the optimal vertex matching problem, we build a graph convolution neural network, named high-order similarity embedding model (HSE), to encode the high-order similarity among vertices into an affinity matrix. Hungarian algorithm is then employed to pair the vertices approximately optimally from the matrix. Fig. 7 shows the above process. Based on the matching, the similarity between the seed and the candidate is computed. Candidates are ranked by their similarity scores and the highly ranked ones will be manually audited.

To ease further discussion, we use V_s, E_s, and $\mathbf{A}_s$ to represent the vertex set, edge set, and adjacency matrix of graph s, respectively.

3.3.1 High-order Similarity Embedding. As shown in Fig. 7, HSE consists of a Siamese network, taking a seed graph and a candidate graph as inputs and employing attention mechanism to suppress the influence of noise during embedding. The network contains K graph convolution layers, which can encode the structure information of the graphs via a message passing mechanism, i.e., propagating vertex features from/to neighbors. It is notable that, because a candidate graph often contains noise vertices, we propose a different message passing scheme for it to suppress the noise propagation, compared to the mechanism in the seed graph. In fact, message passing in the candidate depends on the information from the seed. Therefore, below we first discuss the message passing scheme in the seed.

The message passing scheme in the seed graph is done as in [46]. In every iteration, the vertex feature, i.e., its vector, is aggregated from its adjacent vertices and the vertex itself. Eq. 7 averages the passed messages from the neighbors. Each neighborhood message is a transformation (f_{msg}) of the neighbor's feature $\boldsymbol{h}_{sj}^{(k-1)}$ in last layer. Eq. 8 passes the information of the vertex i in last layer to itself via a transformation f_{self}. Both f_{msg} and f_{self} are implemented as neural networks. Accumulating the messages, Eq. 9 updates the state of vertex i, where f_{aggr} is an aggregation function. Note that, we use $\boldsymbol{h}_{si}^{(k)}$ to indicate the feature vector of vertex i, layer k in s, and $\boldsymbol{h}_{si}^{(0)}$ is the initial feature vector generated by the BERT model, as mentioned in Section 3.1.

$$\boldsymbol{m}_{si}^{(k)} = \frac{1}{|(i,j) \in E_s|} \sum_{j:(i,j)\in E_s} f_{msg}(\boldsymbol{h}_{sj}^{(k-1)}) \tag{7}$$

$$\boldsymbol{n}_{si}^{(k)} = f_{self}(\boldsymbol{h}_{si}^{(k-1)}) \tag{8}$$

$$\boldsymbol{h}_{si}^{(k)} = f_{aggr}(\boldsymbol{m}_{si}^{(k)}, \boldsymbol{n}_{si}^{(k)}) \tag{9}$$

Neighbor features in the seed graph are equally fused, as shown in Eq. 7. However, such a propagation scheme is not suitable for the candidate, in which the noise propagation should be suppressed. Take Fig. 8 as an example. Averaging fusion of i's neighbor features will inevitably induce the noise feature from k, probably making i in the candidate and j in the seed dissimilar. Therefore, the candidate requires a message passing scheme that can suppress the noise.

To this end, we design the following message passing mechanism and leverage attention mechanism to pass messages from neighbor vertices in the candidate to every current vertex with a certain weight. First, for vertex i in a candidate t, its most similar vertex j in the seed graph is sought via Eq. 10.

$$j = arg\max_{j \in V_s} sim_{ti,sj}^{(k)} \tag{10}$$

where $sim_{ti,sj}^{(k)}$ indicates the cosine similarity of the feature vectors on layer k between vertex i in t and vertex j in s.

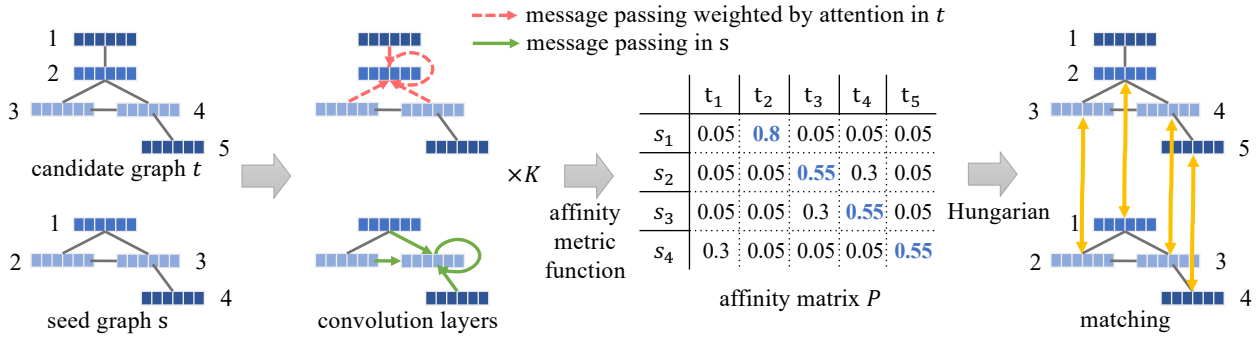

Figure 7: The overview of the optimal vertex matching model. We use vertex i in the candidate graph and vertex j in the seed graph as examples to show the message passing scheme.

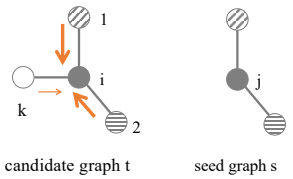

Figure 8: A toy example. The same grains indicate similar vertices and the arrows indicate the message passing. The thicker the arrow, the higher the attention weight.

Second, for i's every neighbor vertex n, we search its most similar vertex among j's neighbors in the seed and record the similarity. The similarity is then normalized as Eq. 11 to represent the attention weights of message passing from the vertex to i.

$$\beta_{ti\leftarrow n}^{(k)} = \frac{exp\left(\max_{j':(j,j')\in E_s}(sim_{tn,sj'}^{(k)})\right)}{\sum_{i':(i,i')\in E_t} exp\left(\max_{j':(j,j')\in E_s}(sim_{ti',sj'}^{(k)})\right)} \tag{11}$$

Finally, the neighbor vertices' features are fused based on their attention weights, as shown in Eq. 12.

$$\boldsymbol{m}_{ti}^{(k)} = \sum_{n:(i,n)\in E_t} \beta_{ti\leftarrow n}^{(k)} \times f_{msg}(\boldsymbol{h}_{tn}^{(k)}) \tag{12}$$

The above scheme guarantees that the noise propagation can be restrained as far as possible while the vertices possessing high similarity with seed vertices will play an important role in message passing. For example, in Fig.8, the message from k has negligible impact on i.

Aggregating the messages from other vertices and i itself, we get an updated feature for i using Eq. 13.

$$\boldsymbol{h}_{ti}^{(k)} = f_{aggr}(\boldsymbol{m}_{ti}^{(k)}, \boldsymbol{n}_{ti}^{(k)}) \tag{13}$$

where $\boldsymbol{n}_{ti}^{(k)}$ takes the same transformation as in Eq 8.

Based on the above techniques, the convolution layers encode each vertex in the seed graph and candidate graphs into vectors. Then, we model the high-order similarity among vertices using a vertex-vertex affinity matrix. The affinity between $j \in V_s$ and $i \in V_t$ is computed by Eq. 14.

$$\mathbf{S}_{ji} = f_{aff}(\boldsymbol{h}_{sj}^{(K)}, \boldsymbol{h}_{ti}^{(K)}), \quad j \in V_s, i \in V_t \tag{14}$$

where K denotes the number of graph convolution layers, and $\mathbf{S}$ denotes the affinity matrix and we implement f_{aff} as Eq. 15 [46].

$$\mathbf{S}_{ji} = exp(\frac{\boldsymbol{h}_{sj}^{(K)\top}\mathbf{W}\boldsymbol{h}_{ti}^{(K)}}{\tau}), \tau > 0 \tag{15}$$

in which, $\mathbf{W}$ is a matrix of learnable parameters and τ is a hyper parameter. The smaller τ is, the more discriminative the affinity matrix is.

We further employ Sinkhorn algorithm [11] to normalize the matrix (Eq. 16) as done in [46] and eventually acquire a normalized affinity matrix $\mathbf{P}$.

$$\mathbf{P} = Sinkhorn(\mathbf{S}) \tag{16}$$

Using the ground truth vertex-vertex correspondence $\mathbf{P}^{gt}$ as the supervision, we train the model by minimizing the cross entropy loss between $\mathbf{P}$ and $\mathbf{P}^{gt}$, as done in Eq. 17 [46].

$$L = -\sum_{j\in V_s, i\in V_t} (\mathbf{P}_{j,i}^{gt} \log \mathbf{P}_{j,i} + (1-\mathbf{P}_{j,i}^{gt}) \log(1-\mathbf{P}_{j,i})) \tag{17}$$

We build the training samples as follows. Picking a variable in a statement, we take the corresponding vertex as the slicing criteria and then traverse all its r-hop neighbors from the PDG to obtain a sub-graph. The sub-graph g_s and the whole graph g_w is paired and the ground truth $\mathbf{P}_{gt}$ can be recorded during the slicing, making a training sample in a form of a triplet $\langle g_s, g_w, \mathbf{P}_{gt}\rangle$. In order to improve the robustness of our model, some random noise is added to the initial vertex feature vectors of g_s.

3.3.2 Similarity Measurement for Bug Detection. After the embedding model is trained, a normalized affinity matrix $\mathbf{P}$ is generated for a seed graph s and a given candidate t. Hungarian algorithm [29] is then applied to get an approximately optimal vertex matching based on the matrix and a similarity is computed between s and t. The similarity computation is done as Eq. 18, by averaging the similarity of the initial feature vectors between matched vertices.

$$sim(s,t) = \frac{\sum_{j\in V_s} sim_{ti_j,sj}^{(0)}}{N} \tag{18}$$

Table 1: The basic features of five comparative methods.

Method Name	Noise in Seed	Matching Approach
GCN	preserved	Graph-level Matching
ReGCN	removed	Graph-level Matching
VGraph	preserved	Set Matching
PM	removed	Set Matching
HSE–	removed	Optimal Vertex Matching

where N is the number of vertices in s. Note that a candidate graph with fewer vertices than the seed will be excluded in our model.

For a given seed graph, its similarity with all candidates will be measured and the candidates are ranked according to their similarity scores. Due to the extensive workload of code auditing, we choose only the top $K2$ candidates (top 25 in our experiment) for manual inspection.

4 EVALUATION

4.1 Experimental Environment and Parameters

All experiments are carried out on a server with Ubuntu 18.04, 64G memory and two GeForce RTX 2080 Ti GPUs.

To demonstrate the effectiveness of our approach, we implement a prototype and evaluate it on five open-source projects, ImageMagick, gpac, OpenSC, mruby and SQLite. We train one VME and one HSE model by treating the five projects (34,129 functions) as a whole, to assess the efficiency of the two-phase approach.

We also perform a comparative analysis with five other similarity measurement based methods. Table 1 shows their basic features, including the seed noise suppression policies and the matching methods. The details of the five methods are as follows.

- GCN: measuring code similarity based on the whole-graph embedding extracted by a traditional graph convolution network implemented by us as in [22, 31, 48, 51]. We extend the training samples of HSE by adding a negative sample to each triplet and use it to train GCN with the triplet loss [41] adopted.
- ReGCN: same as GCN but refining the seed graph, i.e., removing the noise vertices.
- PM: leveraging patch information to match vulnerable candidates. We refer to the work [52] and [47], and reproduce the method therein. The work, based on bag of words model and TF-IDF algorithm [23], measures the similarity with both the vulnerable code slice and the patched code slice. We rank the targets by their similarity to the vulnerable code slice.
- VGraph [16]: converting the code graphs into a set of code property triplets and calculating the overlap ratio between a target triplet set and the positive triplets, context triplets and negative triplets of a seed graph, respectively. We rank the targets by the sum of their similarity to the positive triplets and context triplets.
- HSE–: same as HSE but excluding the attention mechanism. Namely, the seed graph is sanitized and optimal vertex matching is employed.

All the above methods, plus our approach, are applied to the five projects. If not specified, we manually audit the top 25 reported by each method, i.e., $K2 = 25$.

Below we discuss the hyper-parameter settings.

BERT. Most of the hyper-parameters in our BERT model take the default values of the bert-base model [20]. However, limited by the GPU capacity, we use a smaller hidden size (384) and max sequence length (256).

VME. The dimension filtering threshold T is 0.02, as our empirical experiment achieves the best performance with this value.

HSE. Most of the hyper-parameters of the HSE model are the same as [46], except that, the `FEATURE_CHANNEL` is 384 to fit the output size of BERT.

$K1$ and $K2$. The number of candidates retained for further vertex matching, $K1$, varies depending on the requirement of the detection rate. We set $K1 = 1,000$ in our experiment and will further discuss its impact on the effectiveness and efficiency in Section 4.2.2. $K2$, the number of matched candidates for manual auditing, is fixed to 25 due to the burdensome task of code auditing.

4.2 Experiment Results

In this section, we discuss in detail the performance of our approach over two aspects, i.e., effectiveness and efficiency. Section 4.2.1 will show how effective our approach can detect unknown bugs and Section 4.2.2 will present the advantage of the filtering stage in accelerating the detection.

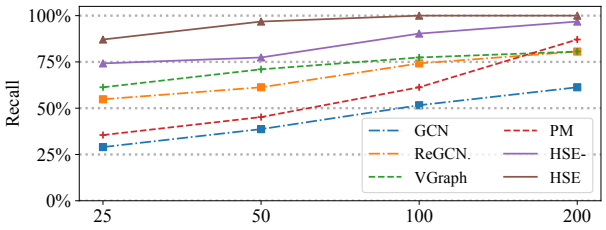

Figure 9: Recall rates with different $K2$.

4.2.1 Effectiveness. Totally, we have detected 31 bugs that have been confirmed by the developers of the open-source projects. The seeds and the corresponding buggy functions are listed in Table 2. More details about the bugs can be found at [1, 3–9].

If a bug is within the top 25 reported by a method, we also check its ranking in the results by other methods. For example, the bug #17 is ranked 14th by HSE and 15th by HSE–, but exceeds the scope of top 25 for the other four methods. Note that, the bug #30 ranks beyond 25 in all six methods, but in the VME+HSE scheme (i.e., filtering before optimal vertex matching, with $K1 = 1000$), it ranks 17th and gets noticed by us. Besides, to demonstrate the effectiveness of filtering, we list the rankings of the bugs in the candidate filtering phase, in the last column of Table 2 (VME).

There are some noticeable findings in the results.

First, the optimal vertex matching methods, i.e., HSE and HSE–, discover 27 out of the 31 bugs (87.1%) with $K2 = 25$. In contrast, under the same setting, the graph-level matching based methods (GCN and ReGCN) can report 22 bugs and the set matching based methods (PM and VGraph) reports 19, 16.1% and 25.8% fewer than our method.

Second, 25 (80.6%) bugs get the highest rank by HSE and HSE–. In particular, 13 (41.9%) of them are ranked top one. The higher the

Table 2: Experiment result. The symbol "-" means that the corresponding function cannot be identified due to a parsing failure.

ID	Project	Seeds	Detected Buggy Functions	GCN	ReGCN	VGraph	PM	HSE-	HSE	VME
1	ImageMagick-7.0.10-29	WritePDFImage	WriteVIFFImage	63	72	10	20	1	1	10
2			WriteXWDImage	173	244	19	66	1	1	33
3			WriteBMPImage	20	59	7	4	1	1	14
4			WriteDIBImage	47	11	16	40	1	1	11
5			WriteMAPImage	59	6	49	103	1	1	4
6			WritePCLImage	4	212	9	13	1	1	6
7			WritePCXImage	12	82	8	9	8	1	8
8			WritePS2Image	2	94	1	1	8	1	24
9			SerializeImageIndexes	79	10	139	152	1	1	1
10			WritePALMImage	8	155	24	38	11	11	42
11			WritePICTImage	247	439	18	23	12	12	67
12		LevelPixel	GenerateDifferentialNoise	1000	10	31	91	5	12	88
13			ScaleResampleFilter	6385	5	18	91	19	4	71
14		RemoveDuplicateLayers	OptimizeLayerFrames	540	5	-	61	1	1	2
15	gpac-1.0.1	HintFile	gf_sm_encode_od	73	2	23	208	1	3	1
16			gf_media_export_saf	647	8	9	130	57	33	8
17			gf_media_make_3gpp	201	36	93	154	15	14	42
18			gf_import_isomedia_track	144	37	60	33	75	17	14
19			gf_media_export_webvtt_metadata	1430	157	1	3	62	12	17
20			gf_export_isom_copy_track	280	10	4	144	16	25	4
21			UpdateODCommand	390	4	1169	291	55	61	112
22			gf_media_export_six	647	24	3	8	33	26	16
23		AV1_RewriteESDescriptorEx	VP9_RewriteESDescriptorEx	1	3	1	9	1	1	1
24		adts_dmx_process	m2psdmx_process	212	5031	48	133	12	12	622
25	OpenSC-0.19.0	main @ src/tools/eidenv.c	main @ src/tools/netkey-tool.c	14	22	11	14	154	6	11
26			main @ src/tests/p15dump.c	46	974	670	-	106	21	166
27		sc_oberthur_read_file	sc_pkcs15_read_file	3	17	8	3	12	2	1
28	mruby-3.0.0-preview	mrb_irep_free	codedump_recur	146	16	3811	166	9	10	1
29			lv_defined_p	47	19	4532	166	5	7	2
30			ipa_draw_polypolygon@**ImageMagick**	4247	2359	4029	75	588	30	168
31	sqlite-3.33.0	fts5TriTokenize	fts5UnicodeTokenize	1	1	1	-	1	1	1

ranking, the easier to hit the bug during auditing. In other words, our method makes the bugs stand out to be discovered.

Third, the largest ranking is 61 for HSE and the second largest is only 33. That means, all the bugs (or at least 30 of them) can be relatively easily discovered by an analyst with an acceptable larger $K2$, e.g., 50. On the contrary, more than ten bugs are ranked after 50 for GCN, ReGCN and PM, and the number is eight for VGraph. Even worse, some bugs are ranked after one thousand, which will never attract the eyes of the analyst. We conduct a further study by taking the 31 bugs as the benchmark and calculating the recall rates of the six methods under different $K2$. Fig. 9 shows the result. When $K2 = 200$, the HSE method reports all the 31 bugs, HSE- reports 96.8% but all the other methods (graph or set matching) can only reach a maximum recall of 87.1%.

We owe the better detection performance of our method to the consideration of fine-grained semantic and structural information in the code graphs. Graph-level matching based methods mix the semantics and structures and generally lower down the distinguishability of dissimilar code graphs. The set matching based methods, on the other hand, do not embed statements into vectors containing their semantics, making it difficult to identify the statements with similar semantics. Based on the above findings, we can claim that our approach can uncover more bugs when incorporating human efforts.

In addition, compared to HSE– that does not employ the attention mechanism to the message passing in candidate graphs, HSE has 12 (38.7%) bugs ranking higher than HSE– and 13 (41.9%) equally ranked (10/13 are top one). The remaining six bugs get lower rankings in HSE than in HSE–. We inspect those cases and find that some functions that look quite similar to the seed but without a bug are ranked higher when the noise is suppressed by HSE. However, such cases do not pose important impact on the result and the overall results illustrate the effectiveness of the attention mechanism on noise suppression.

Besides, most of the bugs get high rankings in the filtering phase. 24 (77.4%) are ranked top 50, 27 (87.1%) are within top 100 and all are within top 1000. It demonstrates that, the VME-based candidate filtering will not pose important influence to our optimal vertex matching method. With a proper $K1$, nearly all bugs can be covered.

At last, our approach successfully detect a cross-project bug (bug #30). Taking a bug in mruby as the seed, it matches a similar bug

Table 3: Experiment results with different $K1$.

$K1$		1000	200	100	HSE (w/o VME)
Recall		100%	96.8%	87.1%	-
Time (s)	Phase 1	7	7	7	0
	Phase 2	11	4	4	290
	Total	18	11	11	290
Time Reduction		94%	96%	96%	-

in ImageMagick, which proves that our approach has the ability of cross-project detection.

We also apply Fortify [10], one of the most prominent static analysis tools, on the target projects. However, none of the bugs detected with our proposed method can be found by Fortify. The main reason is that the performance of traditional static analyzers does depend on the priori knowledge about the bugs (detection rules). When there is no rule for a specific bug, it cannot be found by the analyzer. For the detected bugs in this paper, there are no corresponding rules in Fortify. It is demonstrated again that developing the match-based methodology is necessary.

4.2.2 Efficiency. Optimal vertex matching inevitably brings great time overhead. In fact, it takes 290 seconds on average to measure the similarity between one seed and all candidates without the filtering phase. The other methods, e.g., GCN, consume less than 20 seconds. Fortunately, candidate filtering by VME can dramatically reduce the number of candidates with a proper $K1$ and thus significantly accelerate the similarity measurement.

It is notable that, different $K1$ can affect the detection effectiveness and the efficiency. We have studied the influence of $K1$ and put the result in Table 3. When $K1 = 1000$, all the 31 bugs will be kept for the second-phase analysis that takes about ten seconds for optimal vertex matching. The filtering phase costs only 7 seconds and the total time cost (18 seconds) is reduced by 94%. Smaller $K1$, e.g., 200 or 100, drops the recall but further speeds up the detection. Only 11 seconds are required, with a reduction of 96%, illustrating a comparable speed with the other methods but in the meantime higher detection rate than others.

4.3 Case Studies

We examine two cases to demonstrate the advantages of considering fine-grained semantic and structural information and the attention mechanism respectively. Besides, we analyze the limitation of our approach with a negative example.

4.3.1 Fine-grained Structure and Semantics. Fig. 10 shows two code snippets. One has a known bug (Fig. 10a) and the other (Fig. 10b) is reported by our approach. We refer them to `main_1` and `main_2`, respectively, to ease our discussion. In `main_1`, line 5 allocates the memory that is used at line 7. If a failure occurs at line 7, an error message is dumped and the function returns. A missing release causes memory leak. We can find similar logic in `main_2`, which calls different functions and requires a different release function.

Fig. 11 shows the seed graph (left, with noise removed) and part of the candidate graph (right). The whole-graph matching methods and set matching methods are affected by the noise and rank it far beyond top 25, e.g., 974th by ReGCN and 670th by VGraph. On the contrary, since our BERT model generates similar vectors for APIs with similar semantics, optimal vertex matching can succeed when the influence of the noise in the candidate can be suppressed as far as possible. Our HSE model ranks Fig. 10b the 21st.

Besides, although the memory leakage is a well-known bug type, the related memory allocation/release operations (i.e., `sc_test_init` and `sc_test_cleanup`) in this bug are application-specific. There are often not corresponding detection rules in the traditional static analysis tools, preventing the bug to be discovered.

4.3.2 Attention. Fig. 12 shows an example in gpac. A NULL pointer dereference may occur at line 8 in Fig. 12a. Using the code snippet as a seed, we find an unknown bug in Fig. 12b (#18 in Table 2). The sanitized seed graph is very small, consisting of only a few lines. Many other statements are indeed noise in optimal vertex matching. For example, line 5 in Fig. 12a and line 5 in Fig. 12b should be exactly matched, while the corresponding vertex in the candidate graph has many noise neighbors. Among its 22 neighbors, 19 are noise. Without the attention mechanism, the vertex in the seed graph can hardly match the one in the candidate. In fact, HSE– ranks Fig. 12b the 75th, which is potentially neglected by the analyst. HSE, on the contrary, ranks it the 17th, illustrating that the attention mechanism can help suppress the noise propagation in candidate graphs.

In fact, null pointer dereference is also a well-known bug type. However, the involved function `gf_isom_get_esd` and the type of `origin_esd` are also application-specific. Therefore, it is difficult for the rule-based methods to detect the bugs.

4.3.3 A Negative Example. As shown in Table 2, the HSE model has not achieved good results in all examples. For example, bug #21 ranks only 61st with HSE. The function involved in bug #21 is `UpdateODCommand` in gpac project, whose bug-related statements are shown in Fig. 13. In fact, line 5 in Fig. 13 should have matched line 5 in the seed function in Fig. 12a. However, because of the different key functions `gf_list_enum` and `gf_isom_get_esd`, they can not match each other successfully. The essential cause is that the BERT model cannot capture the similarity of the two key functions. But if we specifically generate data and train the model for the gpac project, `UpdateODCommand` in Fig. 13 will be ranked the 20th, because the BERT model trained alone is of higher quality for the specific project.

5 DISCUSSION

5.1 Multiple or Single Project?

In this paper, we built the models by treating multiple projects as a whole. In fact, we can also do it for each single project, which can even achieve better detection results. Take gpac as an example. The bugs in Table 2 can all be ranked within top 25 by HSE, indicating less manual audit cost. At the same time, the experiment also shows that the efficiency can be improved 10 times by applying the two-phase strategy to a single project.

The difference in modeling multiple projects lies in that the normalization is not sufficient due to the diverse naming habits across the projects, resulting in different distributions for corresponding corpus. Therefore, the quality of the BERT model and the vectors are affected. In the future work, we intend to investigate a better normalization technique that fits multi-project embedding.

```
//file: src/tools/eidenv.c in OpenSC
int main(int argc, char **argv)
{
    [...] // omit 13 lines
    r = sc_context_create(&ctx, &ctx_param);
    [...] // omit 4 lines
    r = util_connect_card(ctx, &card, opt_reader, opt_wait, 0);
    if (r) {
        fprintf(stderr, "Failed to connect to card: %s\n",
        sc_strerror(r));
+       sc_release_context(ctx);
        return 1;
    }
    [...] // omit 27 lines
}
```

(a) The seed function with a known bug

```
//file: src/tests/p15dump.c in OpenSC
int main(int argc, char *argv[])
{
    int i;

    i = sc_test_init(&argc, argv);
    [...] // omit 6 lines
    i = sc_pkcs15_bind(card, NULL, &p15card);
    /* Keep card locked to prevent useless calls to sc_logout */
    if (i) {
        fprintf(stderr, "failed: %s\n", sc_strerror(i));
+       sc_test_cleanup();
        return 1;
    }
    [...] // omit 15 lines
}
```

(b) A candidate function with a detected bug

Figure 10: An example in OpenSC.

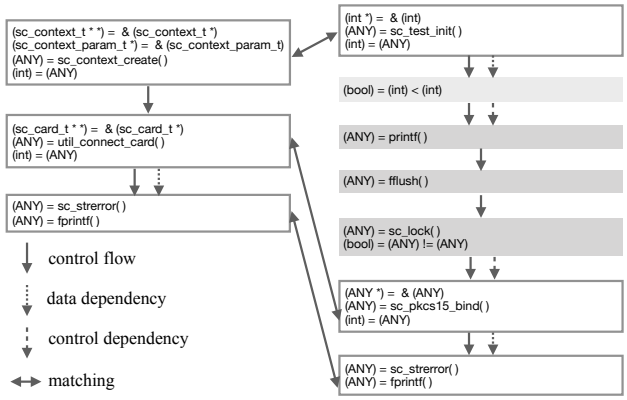

Figure 11: Optimal vertex matching between `main_1` (left) and `main_2` (right). Some vertices in the candidate graph (right) are omitted.

5.2 Why Not Use Patches?

Patch involves very important information of a bug. Existing set matching methods [16, 47, 52] have leveraged patches to filter out the false positives. The idea does not work well in our approach. In practice, integrating the idea can lead to serious false negatives. We have carefully examined the causes and found that, existing approaches treat each element orthogonal while in our method, each vertex may contribute differently to the similarity metric. Therefore, the patch information is not suitable to be directly employed and integrating it is left as a future work.

5.3 Limitation on Bug Type

Theoretically, the bug detection method based on code similarity is suitable for any type of bugs. But in fact, our approach does better in detecting intra-procedural bugs than inter-procedural bugs. This limitation seems to be the essential disadvantage of bug detection methods based on code similarity measurement, because most match-based bug detection methods only consider the feature of the implementation of a function. Li et al. [30, 33] propose a novel method to extract the context-based code representation for

```
//file: applications/mp4box/main.c in gpac
GF_Err HintFile(GF_ISOFile *file, u32 MTUSize, u32 max_ptime, u32
      rtp_rate, u32 base_flags, Bool copy_data, Bool interleave, Bool
      regular_iod, Bool single_group, Bool hint_no_offset)
{
    [...] // omit 93 lines
    esd = gf_isom_get_esd(file, i+1, 1);
-   if (esd) {
+   if (esd && esd->decoderConfig) {
      streamType = esd->decoderConfig->streamType;
    [...] // omit 79 lines
}
```

(a) Buggy `HintFile`

```
//file: src/media_tools/media_import.c in gpac
static GF_Err gf_import_isomedia_track(GF_MediaImporter *import)
{
    [...] // omit 63 lines
    origin_esd = gf_isom_get_esd(import->orig, track_in, 1);
    [...] // omit 16 lines
    if (mtype==GF_ISOM_MEDIA_VISUAL) {
        [...] // omit 7 lines
-       if (origin_esd &&
-       (origin_esd->decoderConfig->objectTypeIndication
-       ==GF_CODECID_MPEG4_PART2)) {
+       if (origin_esd && origin_esd->decoderConfig &&
+       (origin_esd->decoderConfig->objectTypeIndication
+       ==GF_CODECID_MPEG4_PART2)) {
    [...] // omit 334 lines
}
```

(b) Detected `gf_import_isomedia_track`

Figure 12: An example in gpac.

```
//file: src/scene_manager/loader_isom.c in gpac
static void UpdateODCommand(GF_ISOFile *mp4, GF_ODCom *com)
{
    [...] // omit 12 lines
    while ((esd = (GF_ESD *)gf_list_enum(od->ESDescriptors, &j))) {
      [...] // omit 6 lines
        switch (esd->decoderConfig->streamType) {
        case GF_STREAM_OD:
            continue;
    [...] // omit 80 lines
}
```

Figure 13: The source code of `UpdateODCommand`.

the bug detection. The function is embedded not only based on its implementation (AST), but also its context in the PDG and DFG (Data Flow Graph). The introduction of the function context can

greatly reduce the false positives when matching a given code with a known buggy one. The method presented by Li et al. extends the scope of the embedded information and improves the model performance. In theory, the method is compatible with our method, and can be directly integrated in our model. We plan to introduce the function context in our embedding model as done in [30, 33] to further improve the detection performance in the future.

5.4 Other Available Code Features

We measure code similarity based on code graphs which combine CFG and PDG. In fact, there are more features that can be utilized. The study [38] discusses the effectiveness of various graph-based representations in detecting off-by-one bugs. Experiments show that the heterogeneous graph is a good choice because it allows us to take advantage of the rich grammatical and semantic relationships between nodes and edges in AST. Our graph representation can also be converted to be heterogeneous by labelling the edges in CFGs and PDGs with their properties explicitly, such as the data dependence or the control dependence. In addition to using code as a query to retrieve bugs, there are some studies devoted to studying how to search code by a natural language query. For example, Gu et al. [26] embeds the code description (comment) and the source code into the same vector space for code searching, and Wang et al. [45] directly generate code description from user's query with reinforcement learning. We believe the code description is also a kind of beneficial knowledge for bug detection. How to properly bring the code description to our model is one of our future works.

6 RELATED WORK

6.1 Bug Detection Based on Code Similarity

Static code analysis techniques have proven to be very effective for bug detection. The mainstream approaches are rule-based [15, 18, 24, 27, 34, 35, 42, 43]. Different from the rule-based scheme, the bug detection methods based on code similarity do not need rules for specific bug types. One of the most representative approaches is the vulnerability extrapolation method proposed by Yamaguchi et al. [49, 50]. In this approach, the API symbols [49] and AST [50] are chosen as features, and mapped into vectors to measure the similarity between functions.

It is also a natural way to represent code fragments as graphs and employ graph embedding techniques [17, 22, 28, 31, 32, 39, 40, 48, 51]. For example, Feng et al. [22] encode the ACFGs of functions into vectors by clustering algorithm. Ji, Li, Xu, and Yu et al. [28, 31, 48, 51] embed binary code graphs through a Siamese graph neural network. Among them, BugGraph [28] presents a method to measure the source-binary code similarity. It first identifies which compiler is used to build an application. The target source code is compiled to binary with the same compiler. As a result, the source-binary matching can be done in the binary level graph matching.

These methods transform a whole code graph into a vector which contains the structure features of code, thus the code similarity can be measured effectively. However, sometimes a large number of bug-irrelevant statements will be encoded into the whole-graph level vector, and interfere the bug detection results consequently.

There are also some researchers trying to address the noise problem. For example, Li et al. [52] and Xiao et al. [47] use slicing method to remove the noise statements in the seed. Besides, Li, Xiao and Benjamin et al. [16, 47, 52] measure code similarity based on set matching rate. Through slicing techniques and set matching, the noise problem can be alleviated. However, when the code are represented as a set of orthogonal elements, all or part of their structure information will be lost, and the different statements with similar semantics will be omitted while detecting bugs.

6.2 Vertex Matching

In this paper, we base our bug detection on optimal vertex matching. Vertex matching is actually an NP-complete problem [25, 46, 53]. Sometimes, people choose to ignore the edges in the graph directly and simplify it into a set matching problem, but the structure information of graphs will be completely lost in this way. Therefore, there are many studies devoted to finding approximate solutions. The matching problem is essentially a general quadratic assignment programming (QAP) problem [36], where we need to define an affinity matrix to record the first-order, second-order or even higher-order similarity between graphs. The quality of affinity matrix determines how much graph information the model can capture. In this paper, we base our vertex matching on the model proposed in [46]. Through the model we can get a learnable affinity matrix, which can better model the graph structure. The original model is designed to match two graphs of the same size. However, we need to match a small graph (seed graph) with a large graph (target graph) in our task, so we modify the model structure and design an asymmetric network with attention mechanisms to adapt to the asymmetric matching task.

7 CONCLUSION

In this paper, we propose a two-phase bug detection approach based on optimal code graph vertex matching. The noise propagation is suppressed through a seed-based attention mechanism and we accelerate large-scale optimal vertex matching by presenting a filtering phase, which leverages the vertex-matching-oriented embeddings to quickly filter the targets and leaves a limited number of candidates for fine-grained analysis. We have evaluated our approach on five real-world open-source projects and detected 31 previously unknown bugs. The experiment has demonstrated that our approach outperforms the other matching-based methods in the effectiveness and achieves high efficiency due to the two-phase design.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their constructive comments and suggestions. The RUC authors are supported in part by National Natural Science Foundation of China (NSFC) under grants U1836209, 61802413, and 62002361, and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under grant 22XNKJ29. The ISCAS author is supported by the National Natural Science Foundation of China (Grant No. 62132020), and the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036).

REFERENCES

[1] 2020. *SQLite Forum.* https://www.sqlite.org/forum

[2] 2021. *CVE-2021-30019.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30019

[3] 2021. *Issue list of gpac.* https://github.com/gpac/gpac/issues

[4] 2021. *Issue list of ImageMagick.* https://github.com/ImageMagick/ImageMagick/issues

[5] 2021. *Issue list of OpenSC.* https://github.com/OpenSC/OpenSC/issues

[6] 2021. *Pull Request list of gpac.* https://github.com/gpac/gpac/pulls

[7] 2021. *Pull Request list of ImageMagick.* https://github.com/ImageMagick/ImageMagick/pulls

[8] 2021. *Pull Request list of mruby.* https://github.com/mruby/mruby/pulls

[9] 2021. *Pull Request list of OpenSC.* https://github.com/OpenSC/OpenSC/pulls

[10] 2022. *Fortify.* http://www.fortify.net/

[11] Ryan Prescott Adams and Richard S. Zemel. 2011. Ranking via Sinkhorn Propagation. *CoRR* abs/1106.1925 (2011). arXiv:1106.1925 http://arxiv.org/abs/1106.1925

[12] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. https://www.worldcat.org/oclc/12285707

[13] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.* OpenReview.net. https://openreview.net/forum?id=BJOFETxR-

[14] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada.* 3589–3601. https://proceedings.neurips.cc/paper/2018/hash/17c3433fecc21b57000debdf7ad5c930-Abstract.html

[15] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-miner: discovering negative association rules from code for bug detection. 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.* ACM, 411–422. https://doi.org/10.1145/3236024.3236032

[16] Benjamin Bowman and H. Huang. 2020. VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets. 53–69. https://doi.org/10.1109/EuroSP48549.2020.00012

[17] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *2nd International Conference on Learning Representations, ICLR 2014.*

[18] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. 2013. Verifying systems rules using rule-directed symbolic execution. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013.* ACM, 329–342. https://doi.org/10.1145/2451116.2451152

[19] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open Vocabulary Learning on Source Code with a Graph-Structured Cache. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 1475–1485. http://proceedings.mlr.press/v97/cvitkovic19b.html

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers).* Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[21] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016.* The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/discovre-efficient-cross-architecture-identification-bugs-binary-code.pdf

[22] 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, Vienna, Austria, October 24-28, 2016.* ACM, 480–491. https://doi.org/10.1145/2976749.2978370

[23] Stuart James Fitz-Gerald and Bob Wiggins. 2010. Introduction to modern information retrieval, 3rd ed., C.G. Chowdhury. Facit Publishing, London (2010). *Int. J. Inf. Manag.* 30, 6 (2010), 573–574. https://doi.org/10.1016/j.ijinfomgt.2010.08.004

[24] Chushu Gao and Jun Wei. 2013. Generating Open API Usage Rule from Error Descriptions. In *Seventh IEEE International Symposium on Service-Oriented System Engineering, SOSE 2013, San Francisco, CA, USA, March 25-28, 2013.* IEEE Computer Society, 245–253. https://doi.org/10.1109/SOSE.2013.32

[25] M. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman.

[26] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018.* ACM, 933–944. https://doi.org/10.1145/3180155.3180167

[27] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, V. N. Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. 2015. Vetting SSL Usage in Applications with SSLINT. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* IEEE Computer Society, 519–534. https://doi.org/10.1109/SP.2015.38

[28] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network. In *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021.* ACM, 702–715. https://doi.org/10.1145/3433210.3437533

[29] Harold W. Kuhn. 2010. The Hungarian Method for the Assignment Problem. In *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art.* Springer, 29–47. https://doi.org/10.1007/978-3-540-68279-0_2

[30] Yi Li. 2020. Improving bug detection and fixing via code representation learning. In *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume, Seoul, South Korea, 27 June - 19 July, 2020,* Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 137–139. https://doi.org/10.1145/3377812.3382172

[31] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97).* PMLR, 3835–3845. http://proceedings.mlr.press/v97/li19d.html

[32] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings.* http://arxiv.org/abs/1511.05493

[33] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 162:1–162:30. https://doi.org/10.1145/3360588

[34] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005.* ACM, 306–315. https://doi.org/10.1145/1081706.1081755

[35] 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, ICSE 2016, Austin, TX, USA, May 14-22, 2016.* ACM, 333–344. https://doi.org/10.1145/2884781.2884870

[36] Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura Netto, Peter Hahn, and Tania Maia Querido. 2007. A survey for the quadratic assignment problem. *Eur. J. Oper. Res.* 176, 2 (2007), 657–690. https://doi.org/10.1016/j.ejor.2005.09.032

[37] Mingming Lu, Yan Liu, Haifeng Li, Dingwu Tan, Xiaoxian He, Wenjie Bi, and Wendbo Li. 2019. Hyperbolic Function Embedding: Learning Hierarchical Representation for Functions of Source Code in Hyperbolic Space. *Symmetry* 11, 2 (2019), 254. https://doi.org/10.3390/sym11020254

[38] Amir Makhshari and Mifta Sintaha. [n.d.]. On The Effect of Graph Representation of Source Code in Bug Detection. ([n. d.]).

[39] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning Convolutional Neural Networks for Graphs. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48).* JMLR.org, 2014–2023. http://proceedings.mlr.press/v48/niepert16.html

[40] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Trans. Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[41] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015.* IEEE Computer Society, 815–823. https://doi.org/10.1109/CVPR.2015.7298682

[42] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007.* ACM, 145–158. https://doi.org/10.1145/1294261.1294276

[43] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA.* USENIX Association, 379–394. http://www.usenix.org/events/sec08/tech/full_papers/tan_l/tan_l.pdf

[44] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal Attention Network Learning for Semantic Source Code Retrieval. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 13–25. https://doi.org/10.1109/ASE.2019.00012

[45] Chaozheng Wang, Zhenhao Nong, Cuiyun Gao, Zongjie Li, Jichuan Zeng, Zhenchang Xing, and Yang Liu. 2022. Enriching query semantics for code search with reinforcement learning. *Neural Networks* 145 (2022), 22–32. https://doi.org/10.1016/j.neunet.2021.09.025

[46] Runzhong Wang, Junchi Yan, and Xiaokang Yang. 2019. Learning Combinatorial Embedding Networks for Deep Graph Matching. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 3056–3065. https://doi.org/10.1109/ICCV.2019.00315

[47] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 1165–1182. https://www.usenix.org/conference/usenixsecurity20/presentation/xiao

[48] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 363–376. https://doi.org/10.1145/3133956.3134018

[49] Fabian Yamaguchi, Felix "FX" Lindner, and Konrad Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *5th USENIX Workshop on Offensive Technologies, WOOT'11, August 8, 2011, San Francisco, CA, USA, Proceedings*, David Brumley and Michal Zalewski (Eds.). USENIX Association, 118–127. http://static.usenix.org/event/woot11/tech/final_files/Yamaguchi.pdf

[50] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*. ACM, 359–368. https://doi.org/10.1145/2420950.2421003

[51] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 1145–1152. https://ojs.aaai.org/index.php/AAAI/article/view/5466

[52] LI Zan, BIAN Pan, SHI Wen-Chang, and LIANG Bin. 2018. Approach of Leveraging Patches to Discover Unknown Vulnerabilities. *Journal of Software* 005 (2018), 1199–1212. (in Chinese).

[53] Andrei Zanfir and Cristian Sminchisescu. 2018. Deep Learning of Graph Matching. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 2684–2693. https://doi.org/10.1109/CVPR.2018.00284