SECRET: Towards Scalable and Efficient Code Retrieval via Segmented Deep Hashing

Wenchao Gu*
The Chinese University of Hong Kong
wcgu@cse.cuhk.edu.hk

Ensheng Shi*
Xi'an Jiaotong University
s1530129650@stu.xjtu.edu.cn

Yanlin Wang†‡
Sun Yat-sen University
yanlin-wang@outlook.com

Lun Du‡
Ant Research
dulun.dl@antgroup.com

Shi Han
Microsoft
shihan@microsoft.com

Hongyu Zhang
Chongqing University
hyzhang@cqu.edu.cn

Dongmei Zhang
Microsoft
dongmeiz@microsoft.com

Michael R. Lyu
The Chinese University of Hong Kong
lyu@cse.cuhk.edu.hk

***Abstract*—Code retrieval, which retrieves code snippets based on users' natural language descriptions, is widely used by developers and plays a pivotal role in real-world software development. The advent of deep learning has shifted the retrieval paradigm from lexical-based matching towards leveraging deep learning models to encode source code and queries into vector representations, facilitating code retrieval according to vector similarity. Despite the effectiveness of these models, managing large-scale code database presents significant challenges. Previous research proposes deep hashing-based methods, which generate hash codes for queries and code snippets and use Hamming distance for rapid recall of code candidates. However, this approach's reliance on linear scanning of the entire code base limits its scalability. To further improve the efficiency of large-scale code retrieval, we propose a novel approach SECRET (Scalable and Efficient Code Retrieval via SegmEnTed deep hashing). SECRET converts long hash codes calculated by existing deep hashing approaches into several short hash code segments through an iterative training strategy. After training, SECRET recalls code candidates by looking up the hash tables for each segment, the time complexity of recall can thus be greatly reduced. Extensive experimental results demonstrate that SECRET can drastically reduce the retrieval time by at least 95% while achieving comparable or even higher performance of existing deep hashing approaches. Besides, SECRET also exhibits superior performance and efficiency compared to the classical hash table-based approach known as LSH under the same number of hash tables.**

I. Introduction

Code retrieval, a technology enabling the search for relevant code within a codebase using natural language, has garnered significant attention. Due to its pivotal role in real-world software development, many code search approaches [1]–[8] have been proposed in recent years. The open-source communities such as GitHub and Stack Overflow also provide a huge amount of open-source code with natural language descriptions, making it possible to adopt deep learning-based models for code retrieval [9]–[11]. Deep learning approaches employing the dual encoder architecture have become predominent in code retrieval tasks [6], [9], [11]. These approaches use two encoders to encode source code and queries separately into representation vectors. After the encoding, dense retrieval is adopted to retrieve the representation vectors of the source code, which have a strong similarity (e.g., inner product) with the given representation vector of the query [6], [12].

However, efficiency in code retrieval for large-scale code databases remains a significant challenge. Since dense retrieval requires a linear scan of the whole code database, the calculation cost of the retrieval for a single query will be extremely high. The engineering team at GitHub also has underscored the unique challenges posed by GitHub's massive scale. Their search engine encompasses more than 45 million repositories and a vast 115 TB of code. Previous retrieval approaches have struggled with efficiency, resulting in subpar user experiences[1]. Thus, effectively searching code within an extremely large code database using natural language descriptions has become a crucial issue in large-scale code retrieval.

To enhance code retrieval efficiency, Gu et al. [13] introduced a deep hashing-based method called CoSHC. Deep hashing is a technique which leverages deep learning to convert high-dimensional data into low-dimensional hash codes, facilitating efficient data retrieval and storage. This mapping function preserves the high-dimensional features of the data, ensuring that similar data yield similar hash codes in the hash space [14]–[16]. A typical deep hashing approach uses a deep neural network, such as a CNN or Transformer, similar to those employed for representation learning, to extract features from the original data. After feature extraction, the network maps these features to a low-dimensional hash code space, usually through three fully connected layers and a sign function. The sign function is often not used during the training stage but is adopted during inference to convert the continuous output from the deep learning model into final binary hash code. Unlike methods that rely on cosine similarity between vectors, deep hashing approaches evaluate the similarity between data points by measuring the Hamming distance between their hash codes. CoSHC utilizes a "recall-rerank" pipeline, which initially retrieves code candidates based on the Hamming distance of hash codes and then

*Work done while the author was an intern at Microsoft Research.
†Yanlin Wang is the corresponding author.
‡Work done while the author was a fulltime researcher at Microsoft.

[1]https://github.blog/2023-02-06-the-technology-behind-githubs-new-code-search/

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00154

reorders them according to semantic similarity with the query. By adopting deep hashing, their approach significantly reduces computational costs. While calculating the Hamming distance between binary hash codes can be efficiently performed using the XOR instruction on modern computer architectures [17], CoSHC still requires scanning the entire large-scale code database for Hamming-distance calculations. Consequently, its retrieval efficiency remains a notable consideration, especially with extremely large code databases.

To further improve the efficiency of previous deep hashing-based approaches, we propose a code retrieval acceleration framework SECRET. By converting the long hash codes from the previous deep hashing approaches into several segmented hash codes and constructing the hash tables for these hash codes, SECRET can retrieve the code candidates with lookup hash tables and the retrieval time can be greatly reduced.

Specifically, SECRET splits the long hash code from previous deep hashing-based approaches into several segmented hash codes. Hash tables will be constructed for each segmented hash code. During the code retrieval, the segmented hash codes of the given query will be looked up in the corresponding hash table, and the code candidates that are hit in any hash table will be retrieved for the "re-rank" step. To reduce the possibility of hash collision between the unmatched codes and queries, SECRET proposes the strategy named dynamic matching objective adjustment, which tries to assign a unique hash value for each segmented hash code in the matched pair of code and query. To reduce the difficulty of the hash code alignment between the code and query modality, SECRET proposes another strategy named adaptive bit relaxing, which allows SECRET to give up the prediction of the hash bits that are found to be hard to align during the training.

Extensive experiments have been conducted to validate the performance of the proposed approach. Experimental results indicate that SECRET can reduce at least 95% of the retrieval time of current deep hashing approaches meanwhile retaining comparable performance or even outperforming previous deep hashing approaches in the recall step. Additionally, we conducted a comparison between SECRET and a conventional hash table-based method known as Locality-sensitive hashing (LSH). Based on our experimental findings, SECRET demonstrates superior performance and efficiency compared to LSH when the number of hash tables is kept constant.

This work makes the following key contributions:

- We propose a novel approach, SECRET, to improve the retrieval efficiency of previous deep hashing-based approaches in the task of code retrieval. SECRET is the first approach that can convert the long hash code from the deep hashing-based approach into segmented hash codes for the hash table construction.
- SECRET proposes the dynamic matching objective adjustment strategy, which allows the SECRET to dynamically adjust the hash value for each pair of code and query to reduce the false positive hash collision condition.
- SECRET proposes the adaptive bit relaxing strategy, which allows the SECRET to give up the prediction of the hash bits that are hard to align during the training.
- The comprehensive experiments on benchmarks demonstrate that SECRET greatly reduces recall computational complexity while keeping advanced performances of previous deep hashing approaches.

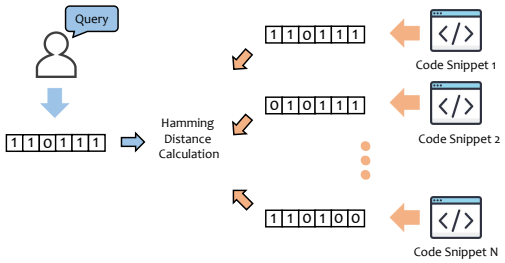

Fig. 1. Illustration of recall with previous deep hashing approaches.

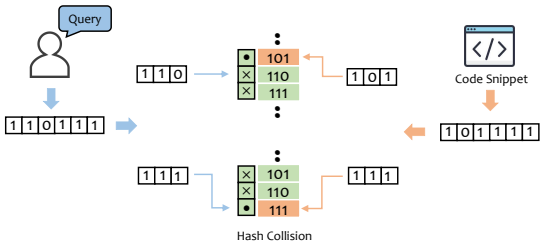

Fig. 2. Illustration of recall with the combination of deep hashing approaches and SECRET.

II. Method

A. Overview

Figure 1 illustrates the recall achieved with deep hashing techniques in code search [13]. In traditional deep hashing approaches, the query and code snippets are encoded into binary hash codes, and code candidates are recalled based on the Hamming distance between the query and code snippet hash codes. In contrast, Figure 2 illustrates the recall using a combination of deep hashing techniques and SECRET. Unlike previous methods [13], SECRET converts the long hash codes from traditional deep hashing approaches into shorter hash segments. By constructing lookup hash tables for these segments, we can utilize hash collisions instead of Hamming distance calculations during the recall stage, reducing the time complexity from $O(n)$ to $O(1)$.

Figure 3 depicts the training process for SECRET. This process comprises two main stages: the initial hashing generation training stage and the iterative training stage. During the initial hashing generation stage, traditional deep hashing techniques are employed to train two hashing models, which generate initial hash codes for both the code and the query. In the iterative training stage, one hashing model is kept fixed, and its output serves as the training objective for the other hashing model. After training for certain number of epoch, the roles are reversed: the newly trained hashing model becomes fixed, providing the training objective, while the previously fixed hashing model is unlocked for further training. These two steps are repeated iteratively until the model training converges. The goal of this iterative training is to ensure that the output from matched queries and code snippets aligns accurately.

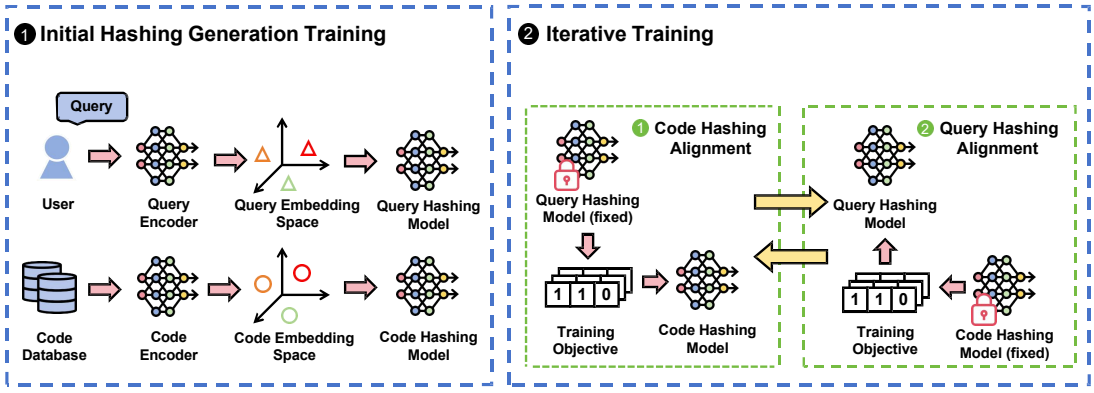

Fig. 3. Overall framework of SECRET. *Initial Hashing Generation Training*: train the code hashing model and query hashing model with representation vectors from the code encoder and query encoder; *Iterative Training*: consists of two sub-training steps which are code hashing alignment and query hashing alignment. In the sub-training step of hashing alignment, one of the hashing model will be fixed and provide the training objective for training of the other hashing model. These two steps will be performed alternately until the model training converges.

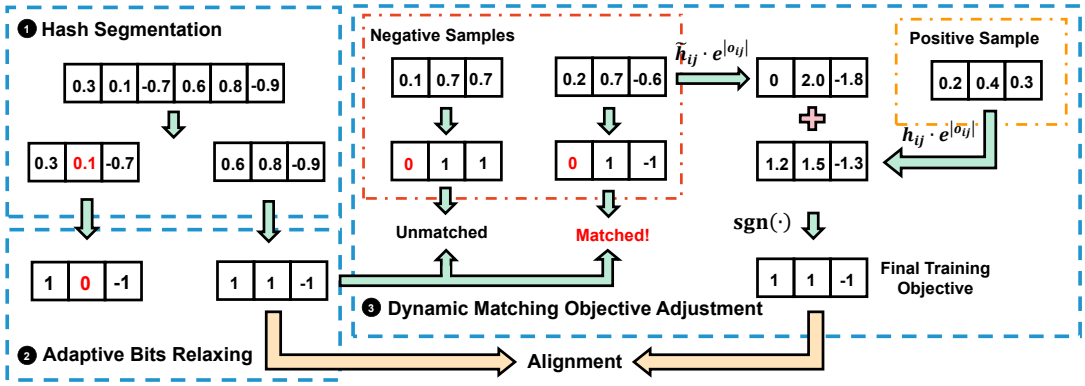

Fig. 4. Steps in the iterative training strategy. *Hash Segmentation*: split the long hash code into several segmented hash codes and convert continuous output value into the discrete value; *Adaptive Bits Relaxing*: select the hash bits from each segmented hash code according to the absolute output value from the model and overwrite the hash value as "unknown" on these hash bits, which represented as 0; *Dynamic Matching Objective Adjustment*: Assign the suitable matching objective for each pair of query and code snippet. The hash code from the positive sample will be utilized as the ground-truth label and adjusted according to the hash collision condition with the negative samples from the same batch.

B. Initial Hashing Generation Training

We leverage the existing deep hashing approaches to design our initial hashing method for code-query search. In our proposed framework, it contains two individual hashing models for the code modality and query modality, respectively. Each hash model is composed as three fully connection layers with a soft binary transformation module (e.g. Tanh activation). The loss function for the hashing models can be summarized as the following mathematical form:

$$L = \sum_{i}^{n} f\left(sim(\mathbf{c}^{(i)}, \mathbf{q}^{(i)})\right) + \kappa \cdot \mathbb{E}_{(j,k)\sim P_n}\left[g(sim(\mathbf{c}^{(j)}, \mathbf{q}^{(k)}))\right], \quad (1)$$

where n is the number of positive training pairs, $\mathbf{c}^{(i)}$ is the i-th code hashing representation, $\mathbf{q}^{(i)}$ is the i-th query hashing representation, $sim(\cdot,\cdot)$ is a similarity function (e.g., cosine similarity or dot production), $f(\cdot)$ is a monotonically decreasing function to rescale the similarity, $g(\cdot)$ a monotonically increasing function correlated to $f(\cdot)$, P_n is a negative sampling distribution from which we can sample a negative pair (i, j), and κ is the number of negative samples. After optimization, we will discretize the learned vectors, i.e., if a value is greater than 0, it will be set as 1 otherwise it will be set as 0.

C. Iterative Training

To construct an effective hash table, it is crucial that the code hashing model and query hashing model produce identical hash codes for matched code and queries. Since deep hashing models produce discrete values as outputs, simply aligning the output of two hashing models may result in the issue of opposite training objectives. To illustrate this problem, consider a scenario where the hash value of the code is 1 and the hash value of the query is -1 for a given pair of code and query. The training objective (target hash value) for modality alignment will differ for the code hashing model and query hashing model, posing challenges for convergence during model training.

To mitigate this issue, we introduce a novel training strategy called iterative training for hashing model training. During the iterative training stage, one hashing model is trained while the other remains fixed. The output of the fixed hashing model serves as the training objective for the updated hashing model. With the pre-defined number of training epochs, the fixed model and the updated model will be alternated. The iterative training will be stopped when the hashing models are converged.

There are still two problems remained to solve. Firstly, it is hard to ensure the hash code from the code hashing models

and query hashing models to be identical for the matched pair of code and query, which greatly harms the accuracy in the "recall" step. Secondly, an ideal hash table should avoid the false positive hash collision, which means the irrelevant code and query should have different hash value.

To tackle the first problem, we propose two strategies: hash segmentation and adaptive bits relaxation. In the hash segmentation strategy, long hash codes are split into short hash segments for constructing the hash table to reduce the difficulty of hash alignment. The adaptive bits relaxation strategy allows the hashing model to relax hard-to-align hash bits, further easing model alignment challenges.

Addressing the second problem, we introduce the dynamic matching objective adjustment strategy. Here, the output from the fixed model serves as the temporary training objective and adjusts based on hash collision conditions with hash codes from negative samples in the same training batch. Figure 4 illustrates the steps in the iterative training strategy, with detailed explanations provided in subsequent sections.

1) Hash Segmentation: To reduce the difficulty of the hash code alignment and increase the hash hit ratio, we split the long hash code from the initial hashing generation into several segments and construct a hash table for each segment. The segmented hash code is

$$H_i = \{h_{i1}, ..., h_{ik}\}, \tag{2}$$

where H_i is the i-th segmented hash code of the code or query, which is composed of k hash bits from the initial hash code. The j-th hash bit in the i-th segmented hash is determined by:

$$h_{ij} = \mathrm{sgn}(o_{(i-1)*k+j}), \tag{3}$$

where $o_{(i-1)*k+j}$ is the output value of the $((i-1)*k+j)$-th hash bit from the neural network and sgn is the sign function. For a more concise representation, $o_{(i-1)*k+j}$ will be replaced by o_{ij} in the following section.

Let's consider the example illustrated in Figure 4 (1) for clarification. Assume that the hash code generated by previous deep hashing methods is [0.3, 0.1, -0.7, 0.6, 0.8, -0.9], and the desired length for each hash segment is 3. In this step, the original hash code will be divided into two hash segments: [0.3, 0.1, -0.7] and [0.6, 0.8, -0.9].

2) Adaptive Bits Relaxing: To further reduce the difficulty of hash alignment, we propose a strategy called *adaptive bits relaxing*. In training, the target output on the hash bit is discrete (+1 and -1). The closer the model's output is to the target value, the better the model's convergence. Consequently, outputs with low absolute values indicate poor convergence and that those hash bits are hard to align. To address these mismatching problems, we omit predictions for these uncertain hash bits and set their outputs to both +1 and -1.

To achieve the adaptive bits relaxing, we first select the hash bits with top k smallest absolute value as the uncertain bits in each hash segment, which is shown below:

$$S_i = \{j \,|\, |o_{ij}| \,\text{is top k smallest in O}_\text{i}\}, \tag{4}$$

where S_i is the set that contains the hash bits with the top k smallest absolute value. k is the maximum number of relaxing bits allowed in a single segmented hash code. For these relaxing bits in each hash segment, we replace the initial hash value with 0 as the intermediate value, which is:

$$\tilde{h}_{ij} = \begin{cases} 0, & j \in S_i \text{ and } |o_{ij}| \le t \\ h_{ij}, & \text{otherwise} \end{cases}, \tag{5}$$

where $\tilde{h}_{ij}$ is the hash code on the i-th hash bit after the relaxing. Since the convergence condition of the model may be good on all the hash bits, we pre-define a threshold value t for the relaxing. Only the hash bits whose absolute value is lower than t are allowed to be relaxed.

Returning to the example illustrated in Figure 4 (2), we proceed from the point where the hash segments have been obtained following the hashing segmentation step. The next step involves relaxing the hash bits. For this example, we assume that the maximum allowed relaxed hash bit is 1 for each hash segment, and the threshold value for relaxation is set at 0.5. The smallest absolute values within each segment are found to be 0.1 and 0.6, respectively. Since only 0.1 is smaller than the predefined threshold value, the second bit in the first hash segment will be relaxed. Consequently, the final hash segments will be [1, 0, -1] and [1, 1, -1].

3) Dynamic Matching Objective Adjustment: To decrease the false positive ratio during the recall step, we introduce a new strategy called *Dynamic Matching Objective Adjustment*. This method assigns appropriate hash codes to each code-query pair. Initially, we obtain the hash codes from the fixed model as the training objective. Then, we evaluate the hash collision condition of this hash code with negative samples from the training batch. Subsequently, we adjust this training objective based on the hash collision condition with the negative samples and incorporate it into the training process.

In the first step, we need to check whether the hash code of the negative samples in the batch is the same as the hash code we retrieved from the fixed model. Equation 6 is the matching results for every bit in the hash code:

$$c_{ij} = \tilde{h}^-_{ij} \cdot \tilde{h}^+_{ij} \tag{6}$$

where $\tilde{h}^-_{ij}$ is the j-th hash bit in the i-th segmented negative hash codes from the modality which needs to be updated and $\tilde{h}^+_{ij}$ is the j-th hash bit in the i-th segmented positive hash codes from the fixed model. c_{ij} indicates whether the j-th hash bit between the i-th segmented positive hash code and the i-th segmented negative hash code is matched. Since we know that $\tilde{h}^-_{ij}, \tilde{h}^+_{ij} \in \{+1, 0, -1\}$, then we can get that $c_{ij} \in \{+1, 0, -1\}$. $c_{ij} = +1$ indicates that the two hash bits are identical, $c_{ij} = 0$ indicates that there is at least one hash bit is relaxed, and $c_{ij} = -1$ indicates that the two hash bits are unmatched. Then we define the C_i as follows:

$$C_i = \min\{c_{i1}, ..., c_{ik}\} \tag{7}$$

where $C_i = -1$ indicates that there exists at least one hash bit unmatched between two segmented hash codes. otherwise, these two segmented hash codes can be regarded as identical. For the purpose of convenient calculation, we define $\tilde{C}_i$ to

indicate whether the negative segmented hash code has the hash collision with the positive segmented hash code as

$$\tilde{C}_i = \begin{cases} 0, C_i = -1 \\ 1, \text{otherwise} \end{cases} \tag{8}$$

where $\tilde{C}_i = 1$ indicates that the i-th segmented negative sample has the hash collision with the segmented positive sample, otherwise does not.

Since we have checked the hash collision condition with negative samples, then we can adjust the training objective we get from the fixed model for the hash alignment with such information. The adjusted training objective is determined by:

$$l_{ij} = h_{ij}^{+} \cdot e^{\gamma \cdot \left| o_{ij}^{+} \right|} - \sum_{n=1}^{m} \tilde{C}_{in} \cdot \tilde{h}_{ij}^{-} \cdot e^{\gamma \cdot \left| o_{ijn}^{-} \right|} \tag{9}$$

where l_{ij} is the j-th hash bit in the ground-truth label for the i-th segmented hash code. m is the negative sample number in the batch. γ is the constant parameter. Since the value range of $o_i j$ is $[1, +1]$, γ can be utilized to amplify the value range so that there is less probability for the hash bits with good convergence conditions to change their sign. For the positive sample, hash bits before the adaptive bits relaxing are selected in the above equation since we still need to offer a clear optimization target for the neural network and the output of these hash bits may get out of the ill convergence condition in the following training epochs. For the negative samples, the hash bits after the adaptive bits relaxing are selected since the hash collision with the negative samples cannot be avoided by changing the output on these relaxed hash bits. Finally, we need to discrete ground-truth label for the hash code bit as:

$$\tilde{l}_{ij} = \text{sgn}(l_{ij}) \tag{10}$$

where $\tilde{l}_{ij}$ is the j-th hash bit in the i-th final segmented hash code template. The final training objective of the i-th segmented hash code for the hash alignment is

$$\tilde{L}_i = \{\tilde{l}_{i1}, ..., \tilde{l}_{ik}\} \tag{11}$$

Let's revisit the example in Figure 4 (3). Here, we assume that the hash segments obtained from previous steps are from the query modality (though it doesn't matter whether they are from the query modality or the code modality). We have one positive sample [0.2, 0.4, 0.3] (indicating the corresponding hash segment from the matched code snippets) and two negative samples [0.1, 0.7, 0.7] and [0.2, 0.7, -0.6] (indicating the corresponding hash segments from two randomly selected unmatched code snippets).

First, let's focus on the positive sample. By applying the first term in Equation 9, we get the value [1.2, 1.5, -1.3]. Next, we examine the negative samples. The discrete hash value for the hash segment [0.1, 0.7, 0.7] is [0, 1, 1], which does not match the given query hash segment [1, 1, -1], so this negative sample is ignored. The other negative sample matches the given query hash segment. Applying the second term in Equation 9, we get the value [0, 2.0, -1.8]. Finally, we sum these two terms via Equation 9 and apply a sign function to the output. The final training objective is [1, 1, -1], which will be used for training the given query hash segment in subsequent steps.

D. Hash Alignment

We align hash code with the following cross-entropy loss:

$$L = -(1 - \tilde{l}_{ij}) * \log(1 - o_{ij}) - (1 + \tilde{l}_{ij}) * \log(1 + o_{ij}) \tag{12}$$

where $\tilde{l}_{ij}$ is the j-th hash bit in the final training objective for the i-th segmented hash code. o_{ij} is the output of j-th hash bit in the i-th segmented hash code from the neural network.

E. Inference of Binary Hash Codes

In the inference stage of binary hash codes, binary hash codes of source code and queries will be generated by the corresponding hashing model. Firstly, the hashing model will output the continuous hashing value. Then hash code will be split into several segmented hash codes with adaptive bits relaxing strategy as we introduced in the Subsection II-C1 and Subsection II-C2. The unknown state for the hash bit will only be treated as an intermediate state in the inference and finally, it will be converted into both 1 and -1. The hash value of the rest hash bits where be converted into 1 or -1 according to the output hash value as a positive number or a negative number.

During searching with lookup hash tables, all the hit code snippets will be added to the recall candidate set. If the users want to set the maximum size of the recall candidate set, we will use a hash table to count the matched times and then apply a Bucket sort to re-rank these candidates.

III. Experimental Settings

A. Research Questions

In our evaluation, we focus on the following questions:

- RQ1: What is the Efficiency of SECRET?
- RQ2: What is the Effectiveness of SECRET?
- RQ3: What is the Effectiveness of Adaptive Bits Relaxing and Iterative Training?
- RQ4: How Many Error Bits Have Been Fixed?

To investigate RQ1, we conducted efficiency experiments to evaluate the time required in the recall step using SECRET compared to previous deep hashing approaches and conventional hashing approach. Due to computational resource limitations, we set the database sizes at 50,000, 100,000, 200,000, and 400,000, respectively. We then analyzed the increasing trend in time for different methods as the data size increased.

To address RQ2, we conducted performance experiments to compare the performance of SECRET with previous deep hashing approaches.

To investigate RQ3, we tested the performance of model training without the adaptive bit relaxing strategy, as well as directly adopting the adaptive bit relaxation strategy without training, in order to verify their respective contributions to the performance of SECRET.

In RQ4, we evaluated the ratio of mismatched hash bits repaired by the adaptive relaxing strategy and the number of hash bits relaxed by both code and query hashing models. This analysis sheds light on potential resource wastage caused by unnecessary relaxations.

TABLE I
DATASET STATISTICS.

Dataset	Training	Validation	Test
Python	412,178	23,107	22,176
Java	454,451	15,328	26,909

B. Datasets

The dataset utilized to evaluate the performance of our proposed approach is provided by CodeBERT [18]. CodeBERT selects code snippets and queries from CodeSearchNet [10] to construct positive and negative pairs. We remain the positive pairs from the dataset and finally get 412,178 $\langle \text{code}, \text{query} \rangle$ pairs as the training set, 23,107 $\langle \text{code}, \text{query} \rangle$ pairs as the validation set, and 22,176 $\langle \text{code}, \text{query} \rangle$ pairs as the test set in the Python dataset. We get 454,451 $\langle \text{code}, \text{query} \rangle$ pairs as the training set, 15,328 $\langle \text{code}, \text{query} \rangle$ pairs as the validation set, and 26,909 $\langle \text{code}, \text{query} \rangle$ pairs as the test set in the Java dataset, which is shown in Table I. We tested 10,000 queries across databases of varying sizes (50,000, 100,000, 200,000, and 400,000) for research question 1. For the remaining research questions, we tested 22,176 queries using a database of 22,176 Python code snippets and 26,909 queries using a database of 26,909 Java code snippets.

C. Baselines

We select two state-of-the-art deep learning-based code retrieval models with four deep hashing approaches and a non-learning-based approach as our baselines.

1) Code Retrieval Baselines: We select CodeBERT [18] and GraphCodeBERT [19] as our base code retrieval models. Both are state-of-the-art models in the code retrieval task.

- **CodeBERT** is a bi-modal pre-trained model based on a Transformer with 12 layers for programming languages and natural languages.
- **GraphCodeBERT** is another pre-trained Transformer-based model. Unlike previous pre-trained models which only utilize the sequence of code tokens as the features, GraphCodeBERT additionally considers the data flow of code snippets in the pre-training stage.

2) Deep Hashing Baselines: We select four state-of-the-art baseline models of deep hashing, which are CoSHC [13], DJSRH [20], DSAH [21], and JDSH [22]. We also select a hash table based approach LSH [23] and two non hash table based approaches which are BM25 [24] and TF-IDF [25].

- **CoSHC** is the first approach that combines the deep hashing techniques with code classification to accelerate the code search. For the sake of fairness of the experiment, we only adopt the deep hashing parts from this approach.
- **DJRSH** constructs a novel joint-semantic affinity matrix that contains specific similarity values instead of similarity order as in previous approaches.
- **DSAH** designs a semantic-alignment loss to align similarity between input features and generated binary hash.
- **JDSH** is a deep hashing approach that jointly trains different modalities with a joint-modal similarity matrix, which can fully preserve cross-modal semantic correlations.
- **LSH** is one of the most popular approaches that map high dimensional data to hash value by using random hash functions and constructing lookup hash tables for data searching. It is widely applied in data recall for single modality.
- **BM25** is a widely popular approach for ranking documents, renowned for its effectiveness in search engines and information retrieval systems. It calculates a relevance score by considering the frequency of query terms in a document, the length of the document, and the term's importance across the entire corpus.
- **TF-IDF** is a popular method in text analysis and information retrieval. It emphasizes terms that are important within a specific document but rare across the entire dataset, making it an effective tool for retrieving relevant data.

D. Metrics

We use R@k (recall at k), MRR (mean reciprocal rank), and N@k (Normalized discounted cumulative gain at k) as the evaluation metrics in this paper. R@k is the metric which widely used to evaluate the performance of the code retrieval models by many previous studies [26]–[29]. It is defined as:

$$R@k = \frac{1}{|Q|} \sum_{q=1}^{Q} \delta(FRank_q \leq k), \tag{13}$$

where Q denotes the query set and $FRank_q$ is the rank of the correct answer for query q. $\delta(Frank_q \leq k)$ returns 1 if the correct result is within the top k returning results, otherwise it returns 0. A higher R@k indicates better performance.

MRR is another metric widely used in the code retrieval task to evaluate the performance [18], [19]:

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{Q} \frac{1}{FRank_q} \tag{14}$$

A higher MRR indicates better performance.

N@k is a metric used to assess the effectiveness of recommendation systems by evaluating both the relevance and the ranking of the results they provide. It is defined as follows:

$$DCG@k = \sum_{i=1}^{k} \frac{rel_i}{\log(i+1)} \tag{15}$$

$$IDCG@k = \sum_{i=1}^{k} \frac{rel_i^{ideal}}{\log(i+1)} \tag{16}$$

$$NDCG@k = \frac{DCG@k}{IDCG@k} \tag{17}$$

where rel_i is the relevance score of the retrieved i-th code snippet, and rel_i^{ideal} is the ideal relevance score of the i-th code snippet. We assign a relevance score of 1 to the matched code snippet and 0 to the unmatched code snippets. A higher N@k indicates better performance.

E. Implementation Details

We use the dual encoder paradigm to use two CodeBERT or GraphCodeBERT to encode the source codes and queries into representation vectors. The dimension of the representation vectors is 768. We implement CoSHC, DJSRH, DSAH, and JDSH by ourselves and follow the hyperparameter settings

TABLE II
RESULTS OF TIME EFFICIENCY COMPARISON ON THE RECALL STEP OF DIFFERENT DEEP HASHING APPROACHES WITH DIFFERENT CODE RETRIEVAL MODELS ON THE PYTHON DATASET WITH THE SIZE 50,000, 100,000, 200,000 AND 400,000.

		50,000		100,000		200,000		400,000	
		128bit	**256bit**	**128bit**	**256bit**	**128bit**	**256bit**	**128bit**	**256bit**
CodeBERT	LSH	3.8s	7.5s	8.1s	16.4s	16.7s	37.6s	38.8s	82.8s
	BM25	308.6s	308.6s	623.2s	623.2s	1258.9s	1258.9s	2555.5s	2555.5s
	TF-IDF	309.2s	309.2s	670.9s	670.9s	1455.9s	1455.9s	3173.8s	3173.8s
	CoSHC	31.9s	43.7s	66.4s	90.1s	137.7s	184.8s	280.1s	375.7s
	$\text{CoSHC}_{\text{SECRET}}$	1.2s (↓96.2%)	2.2s (↓95.0%)	2.1s (↓96.8%)	3.8s (↓95.8%)	4.0s (↓97.1%)	7.1s (↓96.2%)	7.8s (↓97.2%)	14.2s (↓96.2%)
	DJSRH	31.2s	43.1s	65.2s	88.7s	135.1s	185.8s	274.5s	367.8s
	$\text{DJSRH}_{\text{SECRET}}$	1.2s (↓96.2%)	1.4s (↓96.8%)	2.1s (↓96.8%)	2.5s (↓97.1%)	3.9s (↓97.1%)	4.4s (↓97.6%)	7.9s (↓97.1%)	8.4s (↓97.6%)
	DSAH	31.2s	44.0s	65.3s	90.5s	135.0s	186.0s	275.5s	376.8s
	$\text{DSAH}_{\text{SECRET}}$	1.0s (↓96.8%)	1.4s (↓96.8%)	1.9s (↓97.1%)	2.5s (↓97.2%)	3.5s (↓97.4%)	4.5s (↓97.6%)	6.8s (↓97.5%)	8.4s (↓97.8%)
	JDSH	31.1s	42.2s	65.2s	90.6s	135.0s	185.7s	274.4s	368.6s
	$\text{JDSH}_{\text{SECRET}}$	1.2s (↓96.1%)	1.6s (↓96.2%)	2.0s (↓96.9%)	2.6s (↓97.1%)	3.8s (↓97.2%)	4.6s (↓97.5%)	7.5s (↓97.3%)	8.6s (↓97.7%)
GraphCodeBERT	LSH	3.7s	7.3s	7.7s	15.5s	16.9s	34.9s	38.2s	82.2s
	BM25	308.1s	308.1s	622.8s	622.8s	1256.1s	1256.1s	2552.9s	2552.9s
	TF-IDF	307.5s	307.5s	668.1s	668.1s	1450.2s	1450.2s	3165.2s	3165.2s
	CoSHC	31.9s	43.7s	66.5s	90.1s	137.6s	184.9s	280.0s	375.9s
	$\text{CoSHC}_{\text{SECRET}}$	1.1s (↓96.6%)	2.2s (↓95.0%)	2.0s (↓97.0%)	3.8s (↓95.8%)	3.8s (↓97.2%)	7.0s (↓96.2%)	7.4s (↓97.4%)	13.8s (↓96.3%)
	DJSRH	31.2s	43.0s	65.2s	88.5s	134.9s	181.7s	274.5s	367.8s
	$\text{DJSRH}_{\text{SECRET}}$	1.1s (↓96.5%)	1.5s (↓96.5%)	2.0s (↓96.9%)	2.6s (↓97.1%)	3.8s (↓97.2%)	4.6s (↓97.5%)	7.6s (↓97.2%)	8.8s (↓97.6%)
	DSAH	31.1s	43.9s	65.2s	90.4s	134.9s	185.7s	274.7s	377.4s
	$\text{DSAH}_{\text{SECRET}}$	1.0s (↓96.7%)	1.5s (↓96.6%)	1.8s (↓97.2%)	2.6s (↓97.1%)	3.4s (↓97.5%)	4.7s (↓97.5%)	6.6s (↓97.6%)	8.8s (↓97.7%)
	JDSH	31.1s	42.3s	65.1s	90.3s	134.9s	185.6s	275.2s	355.0s
	$\text{JDSH}_{\text{SECRET}}$	1.1s (↓96.5%)	1.7s (↓96.0%)	2.0s(↓96.9%)	2.6s (↓97.1%)	3.7s (↓97.3%)	4.9s (↓97.3%)	7.2s (↓97.4%)	9.2s (↓97.4%)

of deep hashing baselines described their original papers. We experiment on 128-bit and 256-bit for the generated binary hash codes. The hidden size of all deep hashing models is 1,536. We set the size of the binary hash code segment to 16 bits and we allow the deep hash model to predict no more than three unknown bits in each segment. Due to the low recall ratio of LSH, we reduce the length of hash segment into 8 bits. In addition, we set threshold value t as 0.5. In the overall performance comparison experiment in Section IV-B, deep hashing models retrieve the top 300 candidates of code snippets at first. Then the code retrieval models determine the final ranking order of these candidates. In the time efficiency experiment in Section IV-A, we only compare the time cost for the top 300 candidates retrieved by the deep hashing models since our focus is the retrieval efficiency of the recall step. The learning rate of the code retrieval models including CodeBERT and GraphCodeBERT is $1e^{-5}$ and the learning rate for all the deep hashing baselines is $1e^{-4}$. All models are optimized via the AdamW algorithm [30].

We train our models on a server with Tesla V100. We train CodeBERT or GraphCodeBERT for 10 epochs. The training epoch both either initial hashing projection and iterative training is 100. Early stopping is adopted to avoid over-fitting. We evaluate the retrieval efficiency of SECRET on a server with Intel Xeon E5-2698v4 2.2Ghz 20-core. The code for efficiency evaluation is written in C++ and the program is only allowed to use a single thread of CPU for fair comparison.

IV. EVALUATION

A. RQ1: What is the Efficiency of SECRET?

Table III-D shows the experiment results of time efficiency comparison in the recall step of different approaches with different sizes. Since the experiment results on different datasets are very similar, we only show the experiment results of recall efficiency on the Python dataset from the consideration of the paper length. To compare the recall efficiency of the previous deep hashing approaches with and without SECRET, only the time cost in the recall step is recorded.

First of all, we can find that SECRET can reduce more than 95% of the searching time for all sizes of the dataset and hash bits compared to previous deep Hamming distance-based hashing approaches. What's more, the efficiency of SECRET is also higher than the conventional approaches, which demonstrates the effectiveness of SECRET on the improvement of recall efficiency.

From Table III-D, we can also find that the time cost of the deep hashing approaches with SECRET has sublinear growth while the time cost of the deep hashing approaches has superlinear growth as the size of the dataset grows, which demonstrates that the deep hashing approach with SECRET is more efficient than the deep hashing approach without SECRET with the larger dataset. However, we can notice that although the increasing tendency of time cost of SECRET with the increase of dataset size is sublinear, it still does not meet the $O(1)$ complexity. The reason for it is that SECRET contains both searching and sorting processes in the recall step. Although the time complexity of the searching process with the lookup hash tables is $O(1)$, we still need to count the appearance times of the hit candidates in each lookup hash table and sort these candidates to determine the list of recall candidates according to preset the recall number. It is unnecessary to worry whether the sorting process will harm the effectiveness of SECRET since the previous deep hashing approaches also contain the sorting process with the time complexity of $O(n\log n)$ for the entire dataset in the recall

TABLE III
OVERALL PERFORMANCE COMPARISON OF DIFFERENT DEEP HASHING APPROACHES WITH DIFFERENT CODE RETRIEVAL MODELS.

	Model	Python						Java					
		128bit			256bit			128bit			256bit		
		R@1	MRR	N@10	R@1	MRR	N@10	R@1	MRR	N@10	R@1	MRR	N@10
CodeBERT	Original	0.455	0.562	0.606	0.455	0.563	0.606	0.322	0.420	0.459	0.322	0.420	0.459
	LSH	0.388	0.461	0.491	0.441	0.533	0.572	0.265	0.331	0.358	0.303	0.387	0.422
	BM25	0.448	0.541	0.580	0.448	0.541	0.580	0.245	0.305	0.329	0.245	0.305	0.329
	TF-IDF	0.451	0.548	0.606	0.451	0.548	0.606	0.263	0.330	0.372	0.263	0.330	0.372
	CoSHC	0.455	0.562	0.605	0.455	0.563	0.606	0.321	0.419	0.457	0.322	0.420	0.459
	$\text{CoSHC}_{\text{SECRET}}$	0.447 (↓1.8%)	0.547 (↓2.7%)	0.589 (↓2.6%)	0.452 (↓0.7%)	0.554 (↓1.6%)	0.596 (↓1.7%)	0.316 (↓1.6%)	0.408 (↓2.6%)	0.445 (↓2.6%)	0.319 (↓0.9%)	0.415 (↓1.2%)	0.454 (↓1.1%)
	DJSRH	0.454	0.561	0.604	0.455	0.563	0.606	0.321	0.418	0.457	0.322	0.420	0.459
	$\text{DJSRH}_{\text{SECRET}}$	0.446 (↓1.8%)	0.546 (↓2.7%)	0.587 (↓2.8%)	0.451 (↓0.9%)	0.553 (↓1.8%)	0.596 (↓1.7%)	0.316 (↓1.6%)	0.409 (↓2.2%)	0.446 (↓2.4%)	0.319 (↓0.9%)	0.414 (↓1.4%)	0.452 (↓1.5%)
	DSAH	0.450	0.552	0.594	0.451	0.554	0.596	0.317	0.411	0.448	0.319	0.414	0.452
	$\text{DSAH}_{\text{SECRET}}$	0.447 (↓0.7%)	0.547 (↓0.9%)	0.587 (↓1.2%)	0.452 (↑0.2%)	0.554 (0.0%)	0.596 (0.0%)	0.316 (↓0.3%)	0.409 (↓0.5%)	0.446 (↓0.4%)	0.319 (0.0%)	0.415 (↑0.2%)	0.454 (↑0.4%)
	JDSH	0.448	0.549	0.590	0.450	0.552	0.594	0.317	0.410	0.448	0.318	0.412	0.450
	$\text{JDSH}_{\text{SECRET}}$	0.447 (↓0.2%)	0.547 (↓0.4%)	0.587 (↓0.5%)	0.452 (↑0.4%)	0.554 (↑0.4%)	0.596 (↑0.3%)	0.316 (↓0.3%)	0.409 (↓0.2%)	0.447 (↓0.2%)	0.319 (↑0.3%)	0.415 (↑0.7%)	0.452 (↑0.4%)
GraphCodeBERT	Original	0.489	0.598	0.641	0.489	0.598	0.641	0.355	0.457	0.498	0.355	0.457	0.498
	LSH	0.409	0.478	0.506	0.469	0.562	0.600	0.281	0.343	0.368	0.330	0.415	0.450
	BM25	0.472	0.562	0.599	0.472	0.562	0.599	0.262	0.319	0.343	0.262	0.319	0.343
	TF-IDF	0.477	0.570	0.624	0.477	0.570	0.624	0.284	0.348	0.386	0.284	0.348	0.386
	CoSHC	0.489	0.597	0.640	0.489	0.598	0.641	0.355	0.455	0.496	0.355	0.457	0.498
	$\text{CoSHC}_{\text{SECRET}}$	0.479 (↓2.0%)	0.580 (↓2.8%)	0.620 (↓3.1%)	0.484 (↓1.0%)	0.587 (↓1.8%)	0.628 (↓2.0%)	0.348 (↓2.0%)	0.443 (↓2.6%)	0.482 (↓2.8%)	0.353 (↓0.6%)	0.451 (↓1.3%)	0.491 (↓1.4%)
	DJSRH	0.489	0.597	0.640	0.489	0.598	0.641	0.354	0.454	0.495	0.355	0.457	0.498
	$\text{DJSRH}_{\text{SECRET}}$	0.479 (↓2.0%)	0.579 (↓3.0%)	0.619 (↓3.3%)	0.482 (↓1.4%)	0.586 (↓2.0%)	0.629 (↓1.9%)	0.348 (↓1.7%)	0.444 (↓2.2%)	0.482 (↓2.6%)	0.353 (↓0.6%)	0.450 (↓1.5%)	0.490 (↓1.6%)
	DSAH	0.482	0.586	0.628	0.484	0.589	0.631	0.351	0.447	0.486	0.352	0.449	0.488
	$\text{DSAH}_{\text{SECRET}}$	0.480 (↓0.4%)	0.580 (↓1.0%)	0.620 (↓1.3%)	0.484 (0.0%)	0.587 (↓0.3%)	0.629 (↓0.3%)	0.349 (↓0.6%)	0.444 (↓0.7%)	0.482 (↓0.8%)	0.353 (↑0.3%)	0.450 (↑0.2%)	0.490 (↑0.4%)
	JDSH	0.482	0.585	0.629	0.483	0.587	0.629	0.350	0.446	0.485	0.351	0.448	0.487
	$\text{JDSH}_{\text{SECRET}}$	0.478 (↓0.8%)	0.579 (↓1.0%)	0.619 (↓1.6%)	0.483 (0.0%)	0.586 (↓0.2%)	0.628 (↓0.2%)	0.349 (↓0.3%)	0.443 (↓0.7%)	0.483 (↓0.4%)	0.353 (↑0.6%)	0.450 (↑0.4%)	0.490 (↑0.6%)

step. Since SECRET cannot recall the code candidates more than the dataset has, the upper bound of the time complexity of the sorting process in SECRET is $O(n\log n)$ with the dataset containing n code snippets, which is no large than the time complexity of the sorting process in previous deep hashing approaches. The efficiency of deep hashing approaches with SECRET will keep increasing with the increase of the dataset compared to the previous deep hashing approaches.

In summary, SECRET significantly reduces recall time compared to previous deep hashing approaches and even surpasses classical approaches like LSH. Moreover, the efficiency advantages of SECRET will further increase with the expansion of the data size.

B. RQ2: What is the Effectiveness of SECRET?

Table IV-A illustrates the results of the overall performance comparison of different approaches with different code retrieval models. First of all, we can find that SECRET can preserve at least 97.0%,of the performance with all the deep hashing based code retrieval baselines in all the datasets, respectively. In addition, SECRET also outperforms the conventional baselines in most metrics. These results demonstrate that SECRET can retain most of the retrieval performance.

Moreover, we can find that the performance gap between deep hashing approaches with and without SECRET shrinks when the hash codes have more bits. DASH and JDSH with SECRET even outperform baselines with 256 hash bits. The reason for this performance improvement is the mechanism of SECRET. The increase of the hash code length will directly increase the number of lookup hash tables under the setting of SECRET, which can effectively increase the possibility of recall of the corresponding code. Since the hash codes are very space-efficient and the extra space cost for the increase of the hash code's length can be almost neglected. This phenomenon indicates that the problem of performance drop with SECRET can be addressed by increasing hash code's length, which further demonstrates the potential of SECRET.

Lastly, we can find that the performance of SECRET is relatively stable under different deep hashing approaches, which demonstrates the generalizability of SECRET. However, we can still find that there is a small performance difference under different deep hashing approaches. The reason for this phenomenon is the hashing projection distribution, which will be discussed in Section IV-C.

In summary, SECRET retains more than 98% of performance compared to previous deep hashing approaches and even outperforms some of them. Additionally, SECRET significantly outperforms the classical hash table-based approach LSH when using the same number of hash tables.

C. RQ3: What is the Effectiveness of Adaptive Bits Relaxing and Iterative Training?

Table IV-C illustrates the performance comparison of the six variants of SECRET with the baseline of CodeBERT. There are five types of subscripts in Table IV-C, which are NA, A, NR, SR, and BR. NA represents the model only splits the long hash code into segmented hash codes without iterative training. A represents the model splits the long hash code into several segmented hash codes with the iterative training. NR represents the model doesn't adopt the adaptive bits relaxing strategy. SR represents the model only adopts the adaptive bits relaxing strategy on the code hash model. BR represents the model adopts the adaptive bits relaxing strategy on both hash

TABLE IV
THE COMPARISONS AMONG THE SIX SECRET VARIANTS WITH THE BASELINE OF CODEBERT.

Model	Python						Java					
	128bit			256bit			128bit			256bit		
	R@1	MRR	N@10	R@1	MRR	N@10	R@1	MRR	N@10	R@1	MRR	N@10
$\text{CoSHC}_{\text{NA\_NR}}$	0.271	0.312	0.328	0.334	0.391	0.413	0.214	0.263	0.283	0.251	0.313	0.339
$\text{CoSHC}_{\text{A\_NR}}$	0.385	0.460	0.491	0.411	0.494	0.529	0.266	0.337	0.366	0.291	0.371	0.403
$\text{CoSHC}_{\text{NA\_SR}}$	0.417	0.499	0.533	0.440	0.533	0.572	0.301	0.384	0.418	0.311	0.400	0.437
$\text{CoSHC}_{\text{A\_SR}}$	0.433	0.526	0.564	0.444	0.542	0.582	0.306	0.393	0.429	0.314	0.406	0.444
$\text{CoSHC}_{\text{NA\_BR}}$	0.445	0.543	0.584	0.451	**0.554**	**0.596**	0.315	0.405	0.442	**0.319**	0.414	0.453
$\text{CoSHC}_{\text{A\_BR}}$	**0.447**	**0.547**	**0.589**	**0.452**	**0.554**	**0.596**	**0.316**	**0.408**	**0.445**	**0.319**	**0.415**	**0.454**
$\text{DJSRH}_{\text{NA\_NR}}$	0.078	0.086	0.089	0.124	0.140	0.146	0.061	0.072	0.077	0.110	0.132	0.141
$\text{DJSRH}_{\text{A\_NR}}$	0.384	0.460	0.491	0.414	0.500	0.535	0.268	0.338	0.367	0.289	0.368	0.401
$\text{DJSRH}_{\text{NA\_SR}}$	0.250	0.289	0.304	0.319	0.375	0.397	0.183	0.225	0.243	0.249	0.312	0.339
$\text{DJSRH}_{\text{A\_SR}}$	0.434	0.526	0.564	0.445	0.542	0.582	0.305	0.392	0.428	0.313	0.404	0.441
$\text{DJSRH}_{\text{NA\_BR}}$	0.396	0.472	0.504	0.414	0.497	0.531	0.272	0.344	0.374	0.297	0.379	0.413
$\text{DJSRH}_{\text{A\_BR}}$	**0.446**	**0.546**	**0.587**	**0.451**	**0.553**	**0.596**	**0.316**	**0.409**	**0.446**	**0.319**	**0.414**	**0.452**
$\text{DSAH}_{\text{NA\_NR}}$	0.313	0.365	0.386	0.375	0.445	0.474	0.232	0.288	0.311	0.270	0.341	0.370
$\text{DSAH}_{\text{A\_NR}}$	0.388	0.466	0.498	0.417	0.502	0.537	0.268	0.339	0.369	0.291	0.371	0.403
$\text{DSAH}_{\text{NA\_SR}}$	0.421	0.509	0.544	0.438	0.533	0.572	0.299	0.383	0.417	0.310	0.398	0.434
$\text{DSAH}_{\text{A\_SR}}$	0.436	0.529	0.567	0.443	0.540	0.580	0.305	0.393	0.428	0.314	0.405	0.442
$\text{DSAH}_{\text{NA\_BR}}$	0.441	0.537	0.577	0.449	0.550	0.590	0.312	0.402	0.438	0.317	0.410	0.448
$\text{DSAH}_{\text{A\_BR}}$	**0.447**	**0.547**	**0.587**	**0.452**	**0.554**	**0.596**	**0.316**	**0.409**	**0.446**	**0.319**	**0.415**	**0.454**
$\text{JDSH}_{\text{NA\_NR}}$	0.326	0.384	0.408	0.384	0.459	0.489	0.246	0.307	0.332	0.281	0.355	0.385
$\text{JDSH}_{\text{A\_NR}}$	0.388	0.465	0.497	0.416	0.502	0.537	0.269	0.340	0.369	0.290	0.370	0.402
$\text{JDSH}_{\text{NA\_SR}}$	0.425	0.513	0.550	0.438	0.533	0.572	0.304	0.388	0.423	0.311	0.400	0.436
$\text{JDSH}_{\text{A\_SR}}$	0.436	0.529	0.567	0.445	0.543	0.584	0.308	0.395	0.431	0.314	0.405	0.443
$\text{JDSH}_{\text{NA\_BR}}$	0.441	0.537	0.577	0.448	0.549	0.590	0.314	0.404	0.441	0.318	0.411	0.449
$\text{JDSH}_{\text{A\_BR}}$	**0.447**	**0.547**	**0.587**	**0.452**	**0.554**	**0.596**	**0.316**	**0.409**	**0.447**	**0.319**	**0.415**	**0.452**

TABLE V
THE REPAIR RATIO OF ADAPTIVE BITS RELAXING IN BOTH CODE HASHING MODEL AND QUERY HASHING MODEL.

	Model	Python		Java	
		128bit	256bit	128bit	256bit
CodeBERT	$\text{CoSHC}_{\text{Code}}$	0.356	0.358	0.359	0.370
	$\text{CoSHC}_{\text{Query}}$	0.356	0.358	0.357	0.369
	$\text{DJSRH}_{\text{Code}}$	0.356	0.377	0.365	0.387
	$\text{DJSRH}_{\text{Query}}$	0.356	0.378	0.362	0.384
	$\text{DSAH}_{\text{Code}}$	0.353	0.374	0.360	0.382
	$\text{DSAH}_{\text{Query}}$	0.355	0.374	0.358	0.382
	$\text{JDSH}_{\text{Code}}$	0.353	0.374	0.360	0.385
	$\text{JDSH}_{\text{Query}}$	0.350	0.374	0.357	0.386
GraphCodeBERT	$\text{CoSHC}_{\text{Code}}$	0.353	0.358	0.359	0.375
	$\text{CoSHC}_{\text{Query}}$	0.355	0.359	0.356	0.374
	$\text{DJSRH}_{\text{Code}}$	0.356	0.377	0.365	0.387
	$\text{DJSRH}_{\text{Query}}$	0.356	0.378	0.362	0.384
	$\text{DSAH}_{\text{Code}}$	0.350	0.376	0.357	0.385
	$\text{DSAH}_{\text{Query}}$	0.348	0.376	0.355	0.384
	$\text{JDSH}_{\text{Code}}$	0.349	0.374	0.354	0.382
	$\text{JDSH}_{\text{Query}}$	0.350	0.374	0.351	0.381

TABLE VI
AVERAGE HASH BITS THAT BOTH CODE AND QUERY HASHING MODELS PREDICTED AS UNKNOWN IN SINGLE HASH CODE SEGMENT.

	Model	Python		Java	
		128bit	256bit	128bit	256bit
CodeBERT	$\text{CoSHC}_{\text{SECRET}}$	1.10	1.10	1.07	1.04
	$\text{DJSRH}_{\text{SECRET}}$	1.10	1.02	1.06	1.00
	$\text{DSAH}_{\text{SECRET}}$	1.11	1.04	1.07	1.01
	$\text{JDSH}_{\text{SECRET}}$	1.11	1.03	1.07	0.99
GraphCodeBERT	$\text{CoSHC}_{\text{SECRET}}$	1.11	1.09	1.08	1.01
	$\text{DJSRH}_{\text{SECRET}}$	1.12	1.01	1.08	1.01
	$\text{DSAH}_{\text{SECRET}}$	1.13	1.03	1.09	1.01
	$\text{JDSH}_{\text{SECRET}}$	1.13	1.03	1.09	1.01

models. The strategy of NA or A can be combined with the strategy of NR, SR, or BR arbitrary.

As shown in Table IV-C, $\text{Model}_{\text{A\_BR}}$ achieves the best performance, which demonstrates the effectiveness of the combination of iterative strategy and adaptive bits relaxing strategy. Besides, we can find that either iterative strategy or adaptive bits relaxing strategy can greatly improve the model performance. By comparing the performance between Model_{NA} and Model_{A}, we can find the performance for all the setting are improved. The performance of Model_{A} can be greatly improved when the performance of Model_{NA} is low. For example, we can find the performance of $\text{Model}_{\text{NA\_NR}}$ is far from the performance of the baselines and the performance of $\text{Model}_{\text{A\_NR}}$ improved a lot. By comparing the performance among Model_{NR}, Model_{SR}, and Model_{BR}, we can also get similar results as above. What's more, we can find the performance improvement brought by the adaptive bits relaxing strategy is higher than the improvement brought by the iterative training strategy. $\text{Model}_{\text{NA\_BR}}$ can preserve more than 98% performance of $\text{Model}_{\text{A\_BR}}$ for, CoSHC, DSAH, and JDSH baselines.

Interestingly, we observed that the performance of DJSRH_{NA} is notably inferior to the other models. This outcome underscores the significant variations in hash codes initialized by different depth hashing methods. Fortunately, the combination of iterative training strategy and adaptive bit relaxing strategy compensates for this initial projection deficiency. As a result, the final performance of DJSRH is only slightly worse than other deep hashing approaches. This further highlights the effectiveness and versatility of SECRET.

In summary, both the adaptive bit relaxing strategy and the iterative training strategy contribute to the performance of SECRET. However, the former has a larger impact on performance compared to the latter.

D. RQ4: How Many Error Bits Have Been Fixed?

Table V illustrates the repair ratio of the adaptive bits relaxing in both hashing models. $\mathrm{Model}_{\mathrm{Code}}$ and $\mathrm{Model}_{\mathrm{Query}}$ are the repair ratio of the relaxed bits in the code hashing model and query hashing model, respectively. The definition of the repair ratio is the ratio of whether the bits predicted as unknown are the misaligned bits of the binary hash codes generated from the two hashing models in the initial hashing projection training process. Noted that hash bits that are relaxed by both sides of hashing model are not counted.

As shown in Table V, the repair ratio of all the baselines with SECRET is very close. Another finding is that the repair ratio of DJSRH is slightly higher than the other two baselines. As shown in § IV-C, the initial hash projection of DJSRH is much worse than others, which provides more space for SECRET to play its advantages. In addition, we can find the repair ratio with 256 bits higher than that with 128 bits. The reason is that the relatively minimum resolution of the hash codes increases when the hash codes get longer, making it easier for SECRET to distinguish which hash bits have a higher probability to make mistakes.

Table VI illustrates the average hash bits which both hashing models predicted as unknown bits in the single hash code segment. It is unnecessary to predict as unknown in the same bit from both the hashing models since the two hash code segments can be matched as long as the misaligned hash bit is predicted as unknown in either the code hashing model or query hashing model. On the contrary, the prediction of uncertainty in one hash bit will also reduce the hamming distance of unrelated hash codes, which may bring false positives. Therefore, the average number of unknown predictions in both code and query hashing models is smaller, and the performance of our proposed approach will be better. As shown in Table VI, all the baselines with SECRET have similar performance. Similar to the condition in repair ratio, DJSRH has fewer average hash bits which both hashing models predicted as unknown. Besides, we can find the average hash bits predicted as unknown from both hashing models with 256 bits is less than the average hash bits with 128 bits. The reason for this phenomenon is similar to the condition in repair ratio.

With the increasing hash code length, the repair ratio increases and the number of hash bits relaxed by both hashing models decreases. This indicates less resource waste for SECRET as the hash code length increases.

V. Discussion

In this section, we will first examine how faithful SECRET is to the original deep hashing methods. Next, we will explore the impact of varying segment lengths on overall performance. Finally, we will analyze SECRET's performance on one-to-many datasets, where a single query may match multiple code snippets within the database.

A. Faithfulness of SECRET

The proportion of code snippets that both the original deep hashing model and SECRET can recall is shown in Table II in Appendix. The experimental results indicate that SECRET can recall approximately 90% of the code snippets that the original deep hashing approaches can recall. However, SECRET shows an overall performance drop of only around 2% compared to the original deep hashing approaches. This suggests that while SECRET may miss some correct candidates during the recall stage compared to the original deep hashing approaches, the disparity between these code snippets and queries is often too significant for even the re-rank models (original code retrieval models) to correctly retrieve them.

B. Impact of Segment Lengths

We chose 16 bits for the length of hash segments to ensure the segment length is a power of 2, optimizing calculation efficiency for modern 32-bit and 64-bit processors. An 8-bit segment is too short, as it only allows for 256 different hash values, leading to excessive hash collisions even if the data is evenly distributed. We also experimented with a 32-bit hash segment but found it challenging to achieve hash collisions for positive pairs with such a long segment.

Although there is a trade-off between hash segment length and the number of relaxed hash bits, which means we can improve the hash collision probability by increasing the number of relaxed hash bits. However, it will significantly raise storage costs. For example, the storage cost for hash segments with 3 relaxed hash bits is 2^3 times higher than that for segments without relaxed hash bits. Although storing hash codes is much more efficient than storing high-dimensional vectors, we recommend not relaxing too many hash bits within the segment.

We have not fully explored the influence of hash segment lengths that are not powers of 2. Therefore, selecting 16 bits as the segment length might not be the optimal solution for overall performance. However, theoretically, the calculation efficiency for lengths that are not powers of 2 could be compromised due to the principles of bit operations in modern processors. Despite this, the impact may be minimal, as the bit operations for our proposed hash code are already fast enough.

C. Applicability to One-to-Many Dataset

To make our evaluation more representative of real-world code retrieval scenarios, we also evaluate SECRET on a one-to-many dataset CoSQA+, where multiple correct answers exist for a given query. Due to the similar efficiency of hashing calculations and space limitations, we present only the overall performance comparison for this dataset. Detailed experiments can be found in Appendix Section III and Table III. The trend of SECRET's relative performance drop on this dataset is similar to that on the one-to-one dataset, with the relative performance drop being even smaller. This further demonstrates the effectiveness of SECRET.

VI. Related Work

A. Code Search

We briefly introduce recent deep learning-based code search approaches. NCS [5] firstly adopts FastText [31] to embed both queries and source code into representation vectors. CRaDLe [11] further considers the dependency feature within

the code snippets and adopts long short-term memory (LSTM) networks to model program dependency graph from source code. CodeBERT [18] is a bimodal pre-trained model for programming language and natural language. GraphCodeBERT [19] considers the data flow of code during pre-training. CodeT5 [32] is a unified pre-trained encoder-decoder model trained with the identifier-aware pre-training task. Similarly, SPT-Code [33] is a sequence-to-sequence pre-trained model that learns knowledge of source code, the corresponding code structure, and natural language descriptions of code. SyncoBERT [34] is another pre-trained model with the objectives of Identifier Prediction and AST Edge Prediction. UniXcoder [35] is a unified cross-modal pre-trained model which use mask attention matrices with prefix adapters to control the behavior of the model. Bui et al. [36] propose a self-supervised contrastive learning framework named Corder that distinguishes similar and dissimilar code snippets during the training process. Shi et al. [37] propose to dynamically mask tokens to generate positive code examples for contrastive learning. Liu et al. [38] propose to construct graphs and jointly learn the high-level semantics between code and queries.

B. Hashing Techniques

In this subsection, we briefly introduce some representative hash table-based hashing approaches and Hamming distance-based cross-modal hashing approaches, which can be classified into supervised and unsupervised cross-modal hashing approaches. Besides, we also introduce the related works about the ternary hashing and segmented hashing.

1) Hash table based approaches: Locality Sensitive Hashing (LSH) [23] is one of the most popular approaches for recalling data. LSH maps high dimensional data to hash value by using random hash functions. There are several variants of LSH [39]–[41]. Most of them need to build many lookup hash tables to guarantee the recall rate of data. Compared to these approaches, our proposed approaches can achieve similar performance with higher time efficiency and lower storage cost. Semantic Hashing [42] is a learning-based approach and can also construct the lookup hash table for the data recall. However, most of the above approaches are single-modal approaches and do not consider the cross-modal problem.

2) Supervised cross-modal hashing approaches: Bronstein et al. [43] consider the embedding of the input data from two arbitrary spaces into the Hamming space as a binary classification problem with positive and negative examples. SCM [44] is proposed to reduce the training time complexity of most existing SMH methods. SePH [45] converts semantic affinities of training data into a probability distribution and approximates it with to-be-learned hash codes. MCSCH [46] sequentially generates the hash code guided by different scale features through an RNN model with the scale information.

3) Unsupervised cross-modal hashing approaches: CMFH [47] learns unified hash codes by collective matrix factorization with latent factor model. UDCMH [48] incorporates Laplacian constraints into the objective function to preserve neighbors information. DJSRH [20] proposes to train the hashing model with a joint-semantics affinity matrix that integrates the original neighborhood information from different modalities. Yang et al. [21] propose DSAH with a semantic-alignment loss function to align the similarities between features. JDSH [22] utilizes Distribution-based Similarity Decision and Weighting (DSDW) for unsupervised cross-modal hashing to generate hash codes.

4) Ternary hashing and segmented hashing: Liu et al. [49] propose a novel ternary hash encoding for learning to hash methods. They adopt Kleene logic and Łukasiewicz logic to calculate the Ternary Hamming Distance (THD) for both the training and inference stage. Liong et al. [50] propose to split the hash layer into several segments to expand more information during the training and finally concatenate the segmented hash codes into one binary hash code in the inference stage, which is not the same as the purpose of hashing segmentation in our proposed approach.

VII. Threats to Validity

We have identified the following threats to validity:

Dataset Size. From the consideration of the experiment cost, we only select one Python dataset and one Java dataset in our evaluation. Such an amount of data size may not be sufficient to demonstrate the performance and efficiency of SECRET under huge databases.

Baseline Model Selection. From the consideration of experiment cost, we only select two code retrieval models with three deep hashing baselines. However, it is possible that when applying SECRET to other models, there is no significant time boost or the accuracy may be well preserved.

Evaluation Metrics. We only evaluate the proposed approach with the metric of R@1 and MRR in the overall performance experiment. However, these two metrics may not sufficiently reveal the performance gap between SECRET and deep hashing baselines.

VIII. Conclusion

In this paper, we have explored the efficiency aspect of code retrieval, which has received little attention in the existing literature but is very important to the industry. Our contribution lies in a novel hashing approach built upon existing deep hashing methods. This approach involves converting long hash codes from these methods into several segmented hash codes, which are then used to construct hash tables for code candidate recall. By adopting our approach, we significantly reduce the time complexity of previous deep hashing-based approaches during the code candidates recall stage. Our experimental results demonstrate that SECRET not only greatly reduces retrieval time but also achieves comparable or even higher performance than previous deep hashing approaches.

IX. Acknowledgement

Wenchao Gu's and Michael R. Lyu's work described in this paper was in part supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund).

REFERENCES

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, D. R. O. Jr., R. B. Arthur, K. Hinckley, M. R. Morris, S. E. Hudson, and S. Greenberg, Eds. ACM, 2009, pp. 1589–1598. [Online]. Available: https://doi.org/10.1145/1518701.1518944

[2] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 111–120. [Online]. Available: https://doi.org/10.1145/1985793.1985809

[3] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on API understanding and extended boolean model (E)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 260–270. [Online]. Available: https://doi.org/10.1109/ASE.2015.42

[4] E. Shi, Y. Wang, W. Gu, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cocosoda: Effective contrastive learning for code search," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2198–2210.

[5] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. Gottschlich and A. Cheung, Eds. ACM, 2018, pp. 31–41. [Online]. Available: https://doi.org/10.1145/3211346.3211353

[6] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 964–974. [Online]. Available: https://doi.org/10.1145/3338906.3340458

[7] Z. Yao, J. R. Peddamail, and H. Sun, "Coacor: Code annotation for code retrieval with reinforcement learning," in *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, and L. Zia, Eds. ACM, 2019, pp. 2203–2214. [Online]. Available: https://doi.org/10.1145/3308558.3313632

[8] C. Zeng, Y. Yu, S. Li, X. Xia, Z. Wang, M. Geng, L. Bai, W. Dong, and X. Liao, "degraphcs: Embedding variable-based flow graph for neural code search," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, pp. 34:1–34:27, 2023. [Online]. Available: https://doi.org/10.1145/3546066

[9] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 933–944. [Online]. Available: https://doi.org/10.1145/3180155.3180167

[10] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436

[11] W. Gu, Z. Li, C. Gao, C. Wang, H. Zhang, Z. Xu, and M. R. Lyu, "Cradle: Deep code retrieval based on semantic dependency learning," *Neural Networks*, vol. 141, pp. 385–394, 2021. [Online]. Available: https://doi.org/10.1016/j.neunet.2021.04.019

[12] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Retrieval augmented code generation and summarization," in *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 2719–2734. [Online]. Available: https://aclanthology.org/2021.findings-emnlp.232

[13] W. Gu, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and M. R. Lyu, "Accelerating code search with deep hashing and code classification," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 2534–2544. [Online]. Available: https://doi.org/10.18653/v1/2022.acl-long.181

[14] X. Luo, H. Wang, D. Wu, C. Chen, M. Deng, J. Huang, and X. Hua, "A survey on deep hashing methods," *ACM Trans. Knowl. Discov. Data*, vol. 17, no. 1, pp. 15:1–15:50, 2023. [Online]. Available: https://doi.org/10.1145/3532624

[15] A. Singh and S. Gupta, "Learning to hash: a comprehensive survey of deep learning-based hashing methods," *Knowl. Inf. Syst.*, vol. 64, no. 10, pp. 2565–2597, 2022. [Online]. Available: https://doi.org/10.1007/s10115-022-01734-0

[16] J. Rodrigues, M. Cristo, and J. G. Colonna, "Deep hashing for multi-label image retrieval: a survey," *Artif. Intell. Rev.*, vol. 53, no. 7, pp. 5261–5307, 2020. [Online]. Available: https://doi.org/10.1007/s10462-020-09820-x

[17] J. Wang, W. Liu, S. Kumar, and S. Chang, "Learning to hash for indexing big data - A survey," *Proc. IEEE*, vol. 104, no. 1, pp. 34–57, 2016. [Online]. Available: https://doi.org/10.1109/JPROC.2015.2487976

[18] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547. [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.139

[19] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[20] S. Su, Z. Zhong, and C. Zhang, "Deep joint-semantics reconstructing hashing for large-scale unsupervised cross-modal retrieval," in *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 2019, pp. 3027–3035. [Online]. Available: https://doi.org/10.1109/ICCV.2019.00312

[21] D. Yang, D. Wu, W. Zhang, H. Zhang, B. Li, and W. Wang, "Deep semantic-alignment hashing for unsupervised cross-modal retrieval," in *Proceedings of the 2020 on International Conference on Multimedia Retrieval, ICMR 2020, Dublin, Ireland, June 8-11, 2020*, C. Gurrin, B. Þ. Jónsson, N. Kando, K. Schöffmann, Y. P. Chen, and N. E. O'Connor, Eds. ACM, 2020, pp. 44–52. [Online]. Available: https://doi.org/10.1145/3372278.3390673

[22] S. Liu, S. Qian, Y. Guan, J. Zhan, and L. Ying, "Joint-modal distribution-based similarity hashing for large-scale unsupervised deep cross-modal retrieval," in *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, J. Huang, Y. Chang, X. Cheng, J. Kamps, V. Murdock, J. Wen, and Y. Liu, Eds. ACM, 2020, pp. 1379–1388. [Online]. Available: https://doi.org/10.1145/3397271.3401086

[23] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, J. Snoeyink and J. Boissonnat, Eds. ACM, 2004, pp. 253–262. [Online]. Available: https://doi.org/10.1145/997817.997857

[24] S. E. Robertson and S. Walker, "Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval," in *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*, W. B. Croft and C. J. van Rijsbergen, Eds. ACM/Springer, 1994, pp. 232–241. [Online]. Available: https://doi.org/10.1007/978-1-4471-2099-5_24

[25] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *J. Documentation*, vol. 60, no. 5, pp. 493–502, 2004. [Online]. Available: https://doi.org/10.1108/00220410410560573

[26] R. Haldar, L. Wu, J. Xiong, and J. Hockenmaier, "A multi-perspective architecture for semantic code search," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, D. Jurafsky, J. Chai, N. Schluter, and J. R.

Tetreault, Eds. Association for Computational Linguistics, 2020, pp. 8563–8568. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.758

[27] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," in *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 196–207. [Online]. Available: https://doi.org/10.1145/3387904.3389269

[28] S. Fang, Y. Tan, T. Zhang, and Y. Liu, "Self-attention networks for code search," *Inf. Softw. Technol.*, vol. 134, p. 106542, 2021. [Online]. Available: https://doi.org/10.1016/j.infsof.2021.106542

[29] G. Heyman and T. V. Cutsem, "Neural code search revisited: Enhancing code snippet retrieval through natural language intent," *CoRR*, vol. abs/2008.12193, 2020. [Online]. Available: https://arxiv.org/abs/2008.12193

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[31] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 135–146, 2017. [Online]. Available: https://transacl.org/ojs/index.php/tacl/article/view/999

[32] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.685

[33] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: Sequence-to-sequence pre-training for learning source code representations," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3510003.3510096

[34] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," *arXiv preprint arXiv:2108.04556*, 2021.

[35] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 7212–7225. [Online]. Available: https://doi.org/10.18653/v1/2022.acl-long.499

[36] N. D. Q. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, F. Diaz, C. Shah, T. Suel, P. Castells, R. Jones, and T. Sakai, Eds. ACM, 2021, pp. 511–521. [Online]. Available: https://doi.org/10.1145/3404835.3462840

[37] E. Shi, Y. Wang, W. Gu, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cocosoda: Effective contrastive learning for code search," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2198–2210. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00185

[38] S. Liu, X. Xie, J. K. Siow, L. Ma, G. Meng, and Y. Liu, "Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2839–2855, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3233901

[39] M. Bawa, T. Condie, and P. Ganesan, "LSH forest: self-tuning indexes for similarity search," in *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, A. Ellis and T. Hagino, Eds. ACM, 2005, pp. 651–660. [Online]. Available: https://doi.org/10.1145/1060745.1060840

[40] J. Gan, J. Feng, Q. Fang, and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds. ACM, 2012, pp. 541–552. [Online]. Available: https://doi.org/10.1145/2213836.2213898

[41] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *Proc. VLDB Endow.*, vol. 9, no. 1, pp. 1–12, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p1-huang.pdf

[42] R. Salakhutdinov and G. E. Hinton, "Semantic hashing," *Int. J. Approx. Reason.*, vol. 50, no. 7, pp. 969–978, 2009. [Online]. Available: https://doi.org/10.1016/j.ijar.2008.11.006

[43] M. M. Bronstein, A. M. Bronstein, F. Michel, and N. Paragios, "Data fusion through cross-modality metric learning using similarity-sensitive hashing," in *The Twenty-Third IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2010, San Francisco, CA, USA, 13-18 June 2010*. IEEE Computer Society, 2010, pp. 3594–3601. [Online]. Available: https://doi.org/10.1109/CVPR.2010.5539928

[44] D. Zhang and W. Li, "Large-scale supervised multimodal hashing with semantic correlation maximization," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, C. E. Brodley and P. Stone, Eds. AAAI Press, 2014, pp. 2177–2183. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8382

[45] Z. Lin, G. Ding, M. Hu, and J. Wang, "Semantics-preserving hashing for cross-view retrieval," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 2015, pp. 3864–3872. [Online]. Available: https://doi.org/10.1109/CVPR.2015.7299011

[46] Z. Ye and Y. Peng, "Multi-scale correlation for sequential cross-modal hashing learning," in *2018 ACM Multimedia Conference on Multimedia Conference, MM 2018, Seoul, Republic of Korea, October 22-26, 2018*, S. Boll, K. M. Lee, J. Luo, W. Zhu, H. Byun, C. W. Chen, R. Lienhart, and T. Mei, Eds. ACM, 2018, pp. 852–860. [Online]. Available: https://doi.org/10.1145/3240508.3240560

[47] G. Ding, Y. Guo, and J. Zhou, "Collective matrix factorization hashing for multimodal data," in *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 2014, pp. 2083–2090. [Online]. Available: https://doi.org/10.1109/CVPR.2014.267

[48] G. Wu, Z. Lin, J. Han, L. Liu, G. Ding, B. Zhang, and J. Shen, "Unsupervised deep hashing via binary latent factor models for large-scale cross-modal retrieval," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, J. Lang, Ed. ijcai.org, 2018, pp. 2854–2860. [Online]. Available: https://doi.org/10.24963/ijcai.2018/396

[49] C. Liu, L. Fan, K. W. Ng, Y. Jin, C. Ju, T. Zhang, C. S. Chan, and Q. Yang, "Ternary hashing," *CoRR*, vol. abs/2103.09173, 2021. [Online]. Available: https://arxiv.org/abs/2103.09173

[50] V. E. Liong, J. Lu, L. Duan, and Y. Tan, "Deep variational and structural hashing," pp. 580–595, 2020. [Online]. Available: https://doi.org/10.1109/TPAMI.2018.2882816