Towards Intelligent Program Development based on Code Semantics Learning

Ph.D. Oral Defense of Wenchao Gu

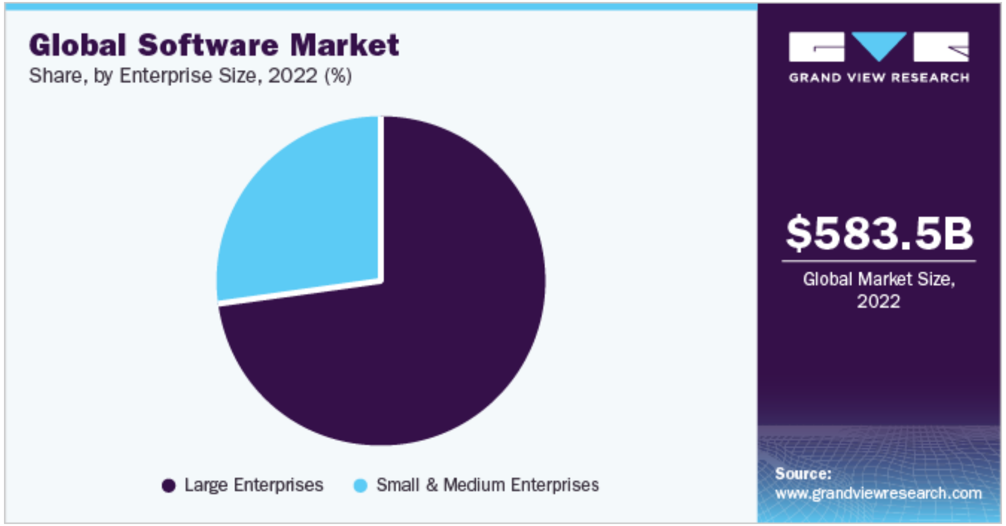
Supervised by Prof. Michael Lyu

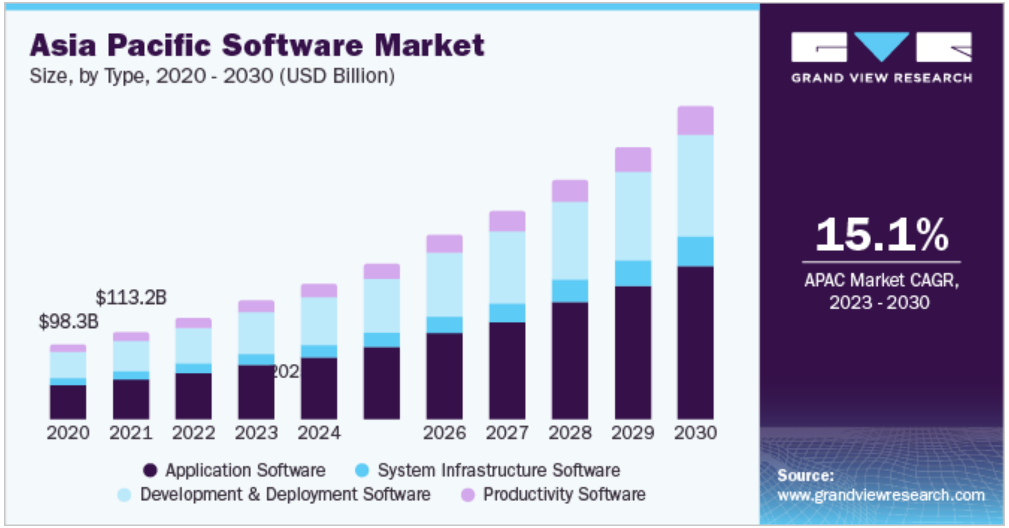
Thursday 14 December 2023

ARISE
Automated Reliable Intelligent Software Engineering

香港中文大學
The Chinese University of Hong Kong

Software is Everywhere





Demand of Software Developer is Increasing

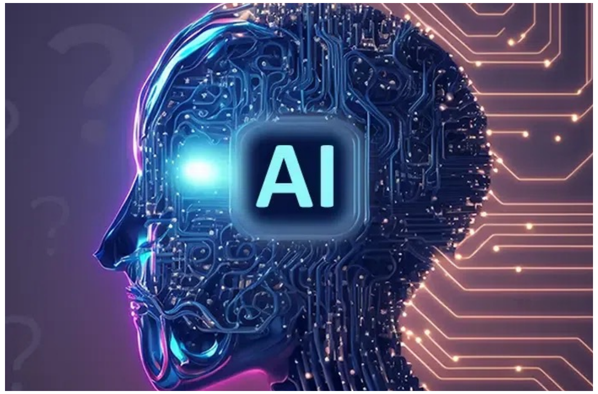
Quick Facts: Software Developers, Quality Assurance Analysts, and Testers	
2022 Median Pay	$124,200 per year $59.71 per hour
Typical Entry-Level Education	Bachelor's degree
Work Experience in a Related Occupation	None
On-the-job Training	None
Number of Jobs, 2022	1,795,300
Job Outlook, 2022-32	25% (Much faster than average)
Employment Change, 2022-32	451,200

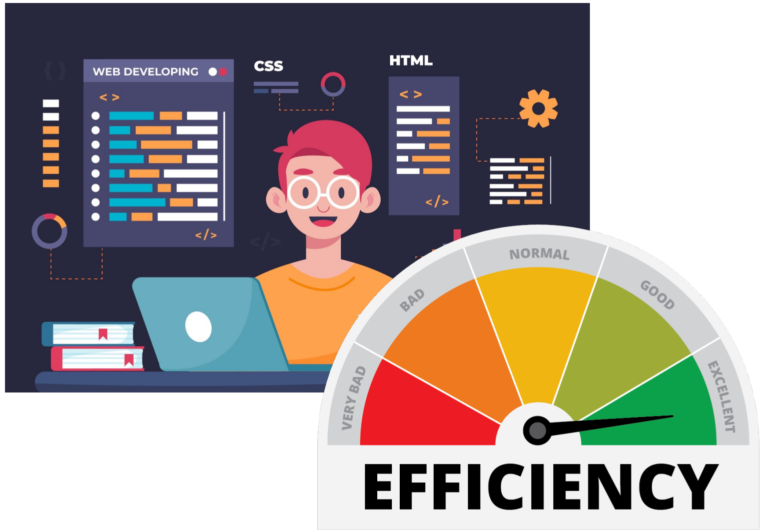
The market's demand for software developers is also expected to increase rapidly!

Occupation code	Occupation title (click on the occupation title to view its profile)	Level	Employment	Employment RSE	Employment per 1,000 jobs	Median hourly wage	Mean hourly wage	Annual mean wage	Mean wage RSE
00-0000	**All Occupations**	total	147,886,000	0.0%	1000.000	$22.26	$29.76	$61,900	0.2%
15-1250	Software and Web Developers, Programmers, and Testers	broad	2,049,920	0.4%	13.861	$54.90	$60.07	$124,940	0.6%
15-1251	Computer Programmers	detail	132,740	2.8%	0.898	$47.02	$49.42	$102,790	1.4%
15-1252	Software Developers	detail	1,534,790	0.4%	10.378	$61.18	$63.91	$132,930	0.6%
15-1253	Software Quality Assurance Analysts and Testers	detail	196,420	2.0%	1.328	$47.89	$50.84	$105,750	0.6%

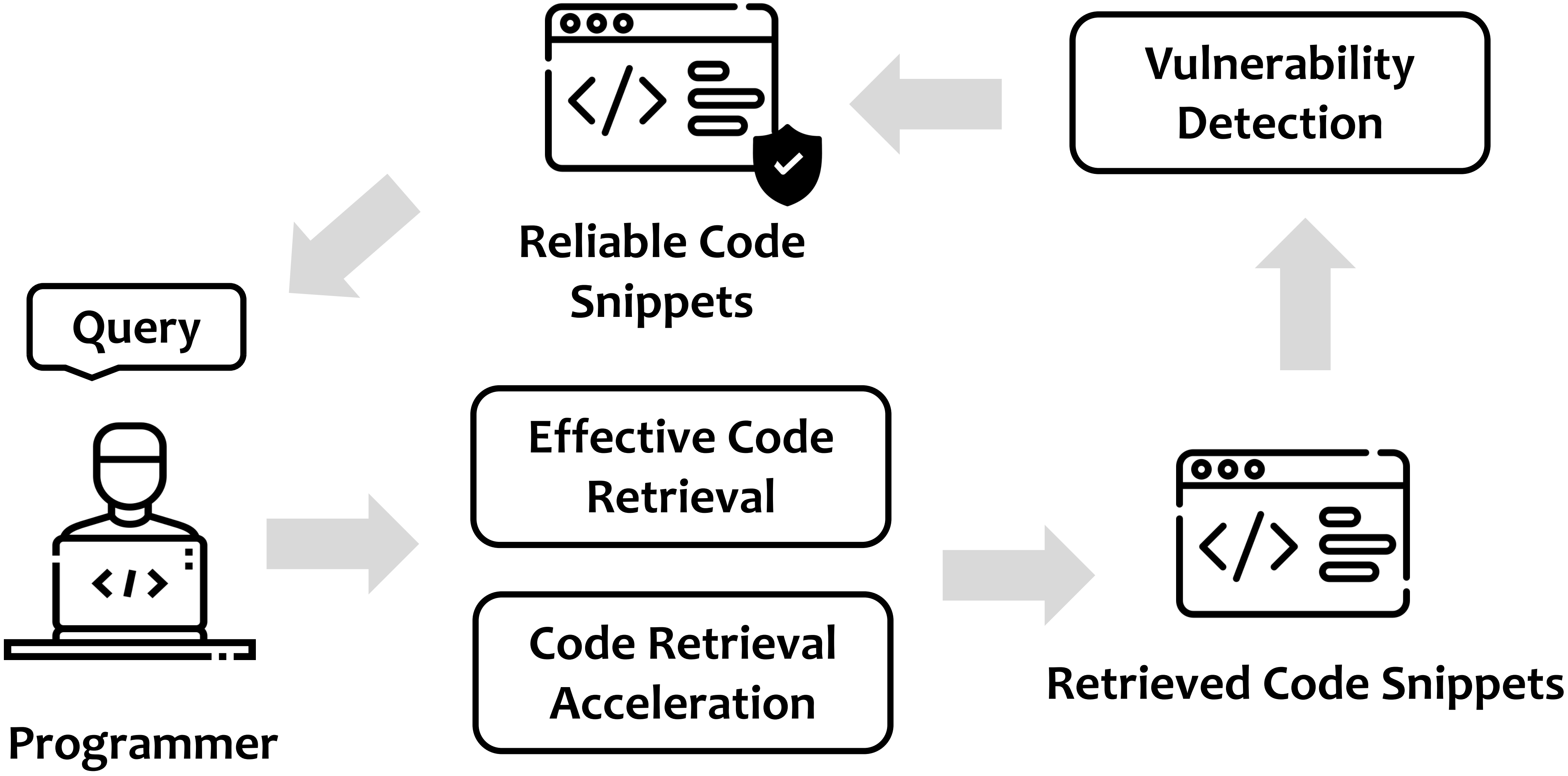
Intelligent Program Development

- Intelligent program development aims to use automation technology to solve some tasks that need to be solved manually by software engineers during the software development period

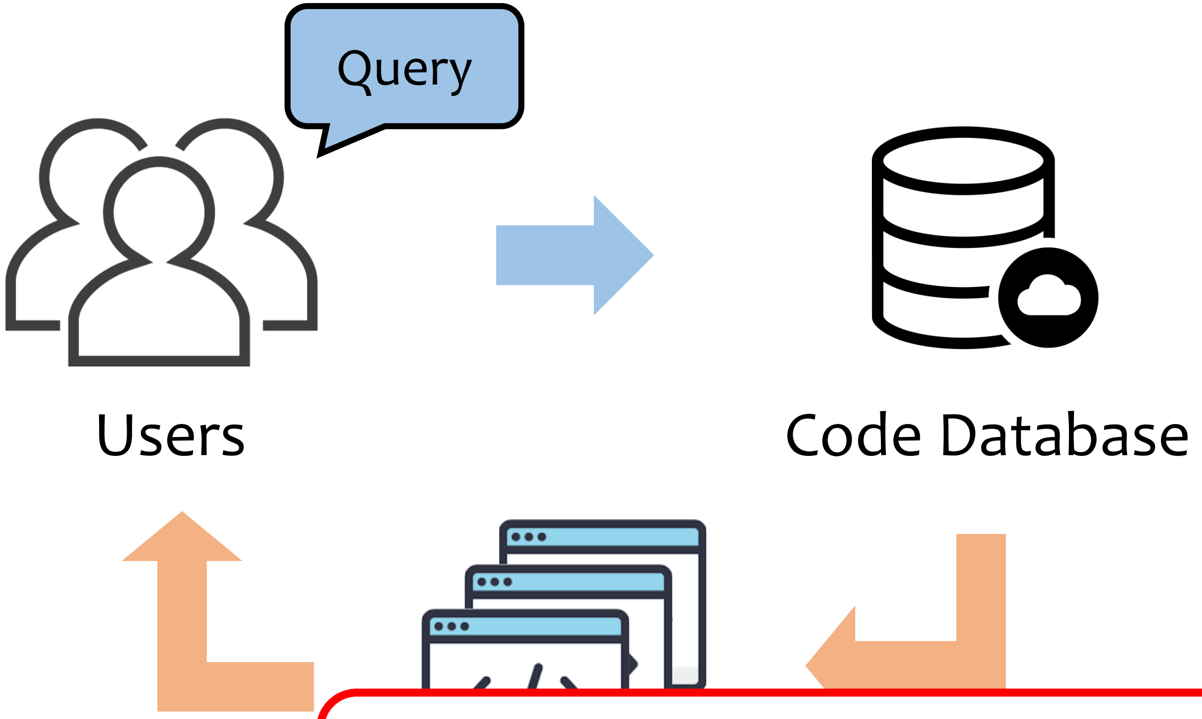




Roadmap


Vulnerability Detection
Reliable Code Snippets
Query
Programmer
Effective Code Retrieval
Code Retrieval Acceleration
Retrieved Code Snippets

Code Retrieval



Advantage of code retrieval:

- Shorten the software development cycle
- Reduce the difficulty of software development
- Help software developers improve the code reuse rate in their projects

Semantic gap between the programming language and natural language hinders the performance!

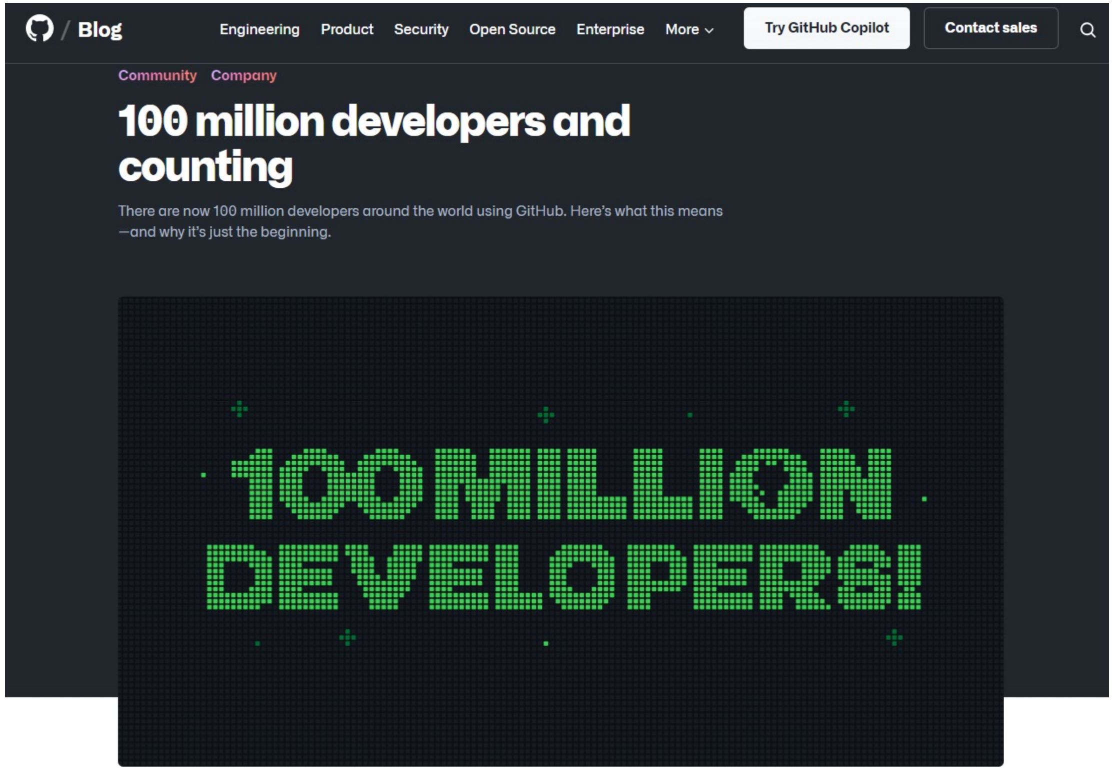
Code Retrieval Acceleration

Blog | Engineering | Product | Security | Open Source | Enterprise | More | Try GitHu

from queries, and, we want to search with regular expressions.

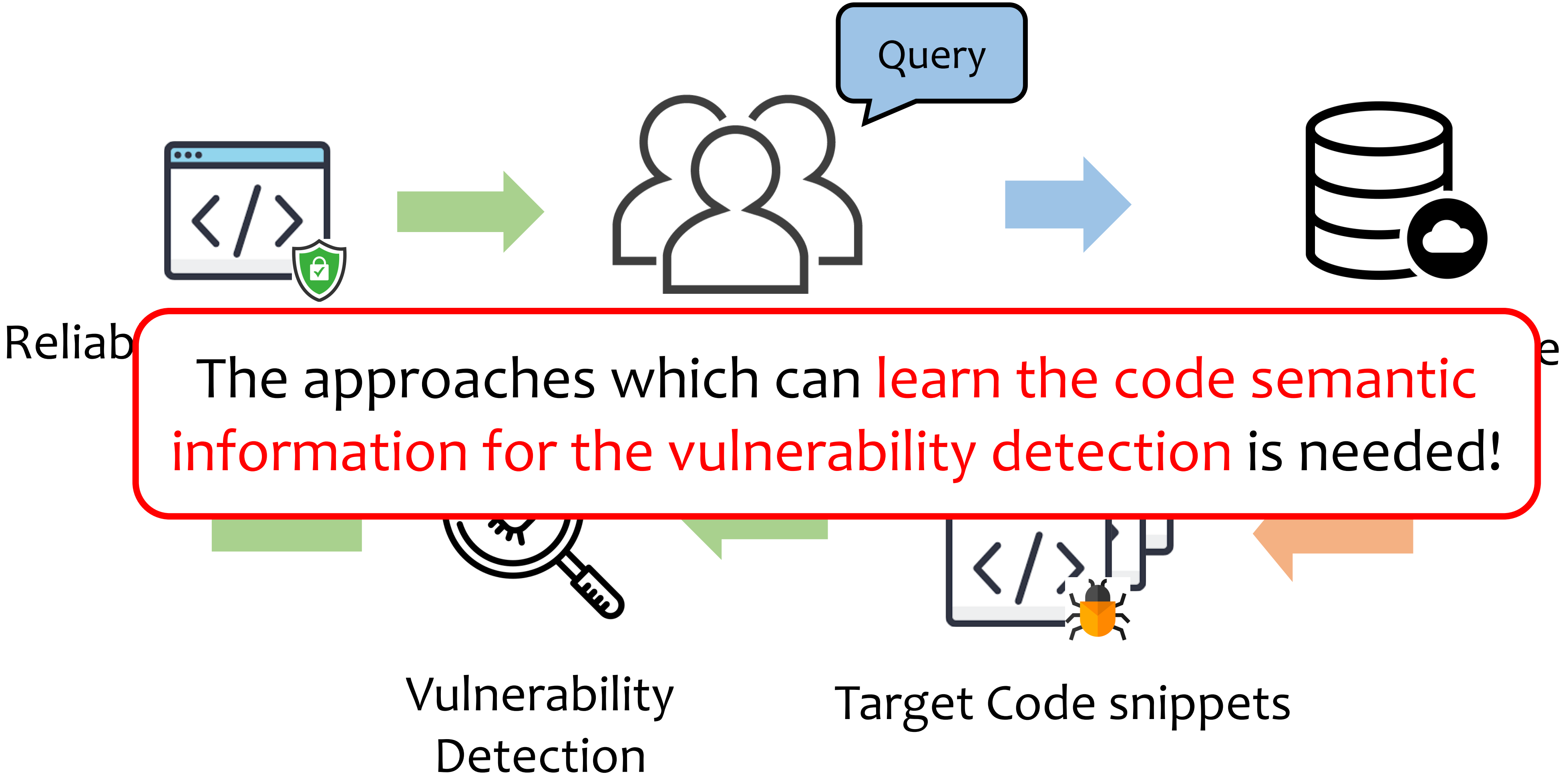
3. GitHub's scale is truly a unique challenge. When we first deployed Elasticsearch, it took months to index all of the code on GitHub (about 8 million repositories at the time). Today, that number is north of 200 million, and that code isn't static: it's constantly changing and that's quite challenging for search engines to handle. For the beta, you can currently search almost 45 million repositories, representing 115 TB of code and 15.5 billion documents.

At the end of the day, nothing off the shelf met our needs, so we built something from scratch.



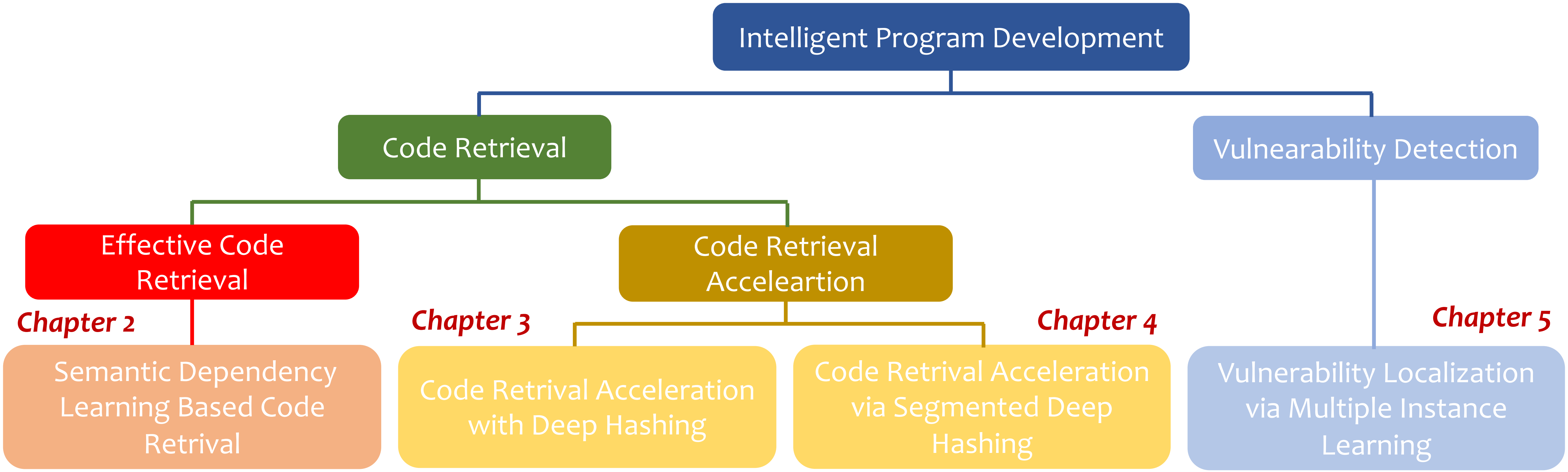
The efficiency is more and more important in code retrieval!

Vulnerability Detection



The approaches which can learn the code semantic information for the vulnerability detection is needed!

Thesis Contribution



- First method to integrate the dependency and semantics information at the statement level for code learning

  [Nerual Networks]

- First method to integrate code classification and deep hashing to improve the retrieval efficiency

  [ACL'22]

- First method to convert the long hash code from the deep hashing approach into segmented hash codes for the hash table construction

  *[ICDE'24]**

- Method can localize vulnerabilities at the statement level without requiring additional labeling at the statement level

  *[TOSEM]**

** Under review*

Thesis Contribution

Intelligent Program Development

- Code Retrieval
  - Effective Code Retrieval
    - *Chapter 2*: Semantic Dependency Learning Based Code Retrival
  - Code Retrieval Acceleartion
    - *Chapter 3*: Code Retrival Acceleration with Deep Hashing
    - *Chapter 4*: Code Retrival Acceleration via Segmented Deep Hashing
- Vulnearability Detection
  - *Chapter 5*: Vulnerability Localization via Multiple Instance Learning

Chapter 2 – Semantic Dependency Learning Based Code Retrival

➢ First method to integrate the dependency and semantics information at the statement level for code learning

[Nerual Networks]

Chapter 3 – Code Retrival Acceleration with Deep Hashing

➢ First method to integrate code classification and deep hashing to improve the retrieval efficiency

Chapter 4 – Code Retrival Acceleration via Segmented Deep Hashing

➢ First method to convert the long hash code from the deep hashing approach into segmented hash codes for the hash table construction

Chapter 5 – Vulnerability Localization via Multiple Instance Learning

➢ Method can localize vulnerabilities at the statement level without requiring additional labeling at the statement level

** Under review*

Thesis Contribution

Intelligent Program Development

Code Retrieval

Vulnearability Detection

Effective Code Retrieval

Code Retrieval Acceleartion

Chapter 2

Semantic Dependency Learning Based Code Retrival

- First method to integrate the dependency and semantics information at the statement level for code learning

Chapter 3

Code Retrival Acceleration with Deep Hashing

- First method to integrate code classification and deep hashing to improve the retrieval efficiency

[ACL'22]

Chapter 4

Code Retrival Acceleration via Segmented Deep Hashing

- First method to convert the long hash code from the deep hashing approach into segmented hash codes for the hash table construction

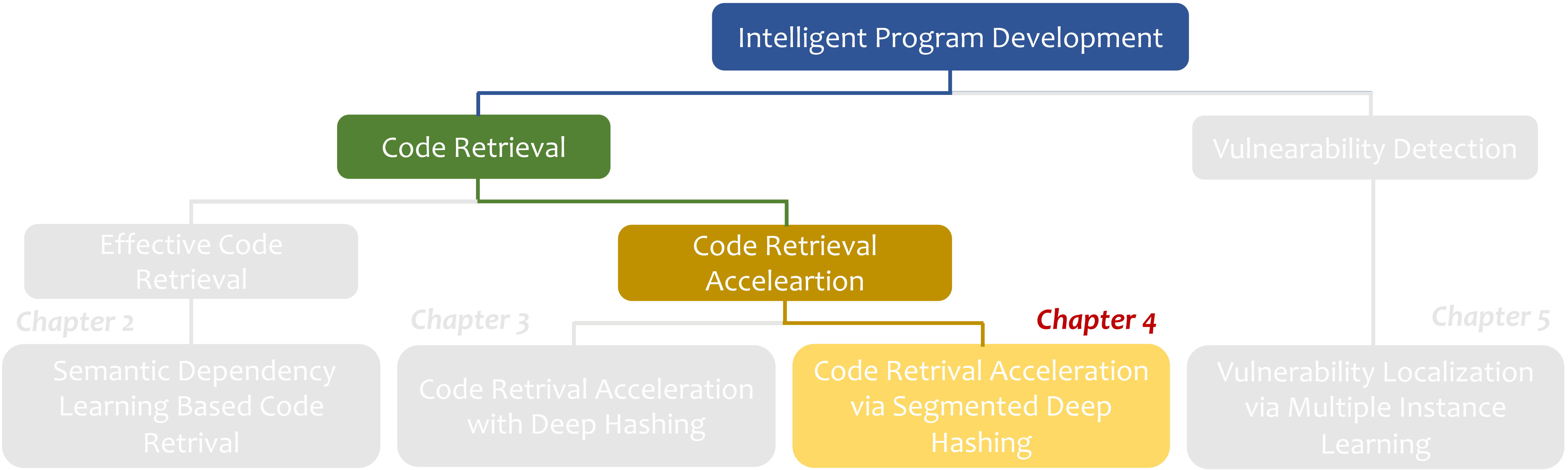
Chapter 5

Vulnerability Localization via Multiple Instance Learning

- Method can localize vulnerabilities at the statement level without requiring additional labeling at the statement level

** Under review*

Thesis Contribution



- First method to integrate the dependency and semantics information at the statement level for code learning
- First method to integrate code classification and deep hashing to improve the retrieval efficiency
- First method to convert the long hash code from the deep hashing approaches into segmented hash codes for the hash table construction

  *[ICDE’24]**
- Method can localize vulnerabilities at the statement level without requiring additional labeling at the statement level

*** Under review**

Thesis Contribution

Intelligent Program Development

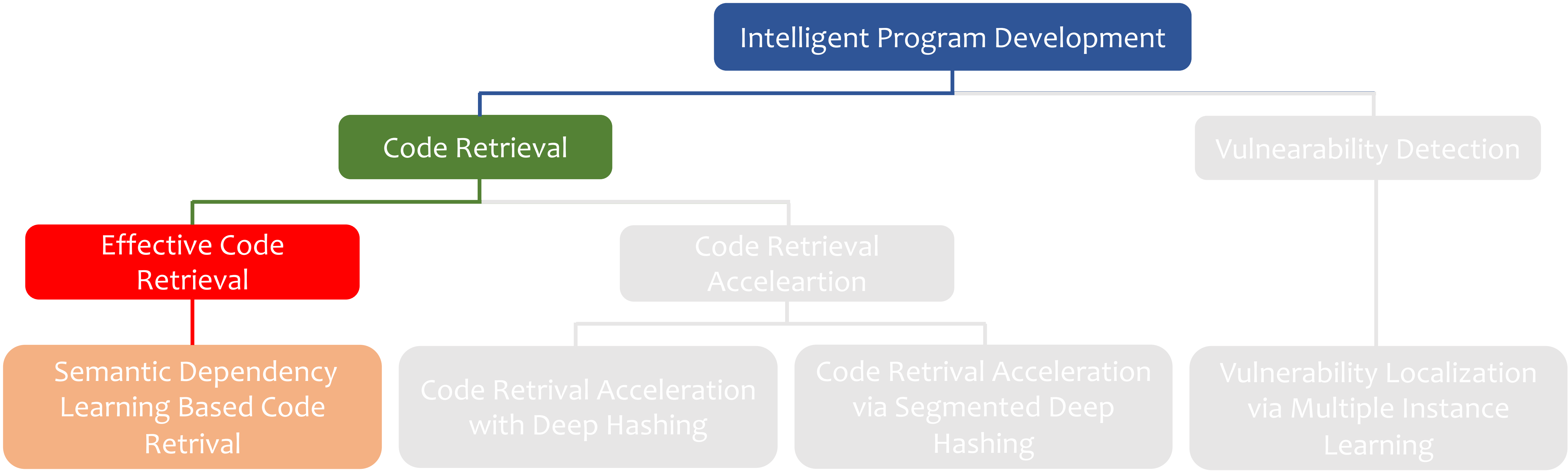
- Code Retrieval
  - Effective Code Retrieval
    - Chapter 2: Semantic Dependency Learning Based Code Retrival
      - ➢ First method to integrate the dependency and semantics information at the statement level for code learning
  - Code Retrieval Acceleartion
    - Chapter 3: Code Retrival Acceleration with Deep Hashing
      - ➢ First method to integrate code classification and deep hashing to improve the retrieval efficiency
    - Chapter 4: Code Retrival Acceleration via Segmented Deep Hashing
      - ➢ First method to convert the long hash code from the deep hashing approach into segmented hash codes for the hash table construction
- Vulnearability Detection
  - Chapter 5: Vulnerability Localization via Multiple Instance Learning
    - ➢ Method can localize vulnerabilities at the statement level without requiring additional labeling at the statement level

*[TOSEM]**

** Under review*

CONTENTS

Thesis Structure



1

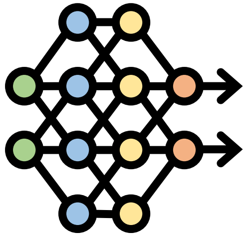
Semantic Dependency Learning Based Code Retrival

Motivation

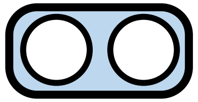
Where is the captical of China?

where | is | the | ...

Token sequence



Nerual Network



Representation Vector

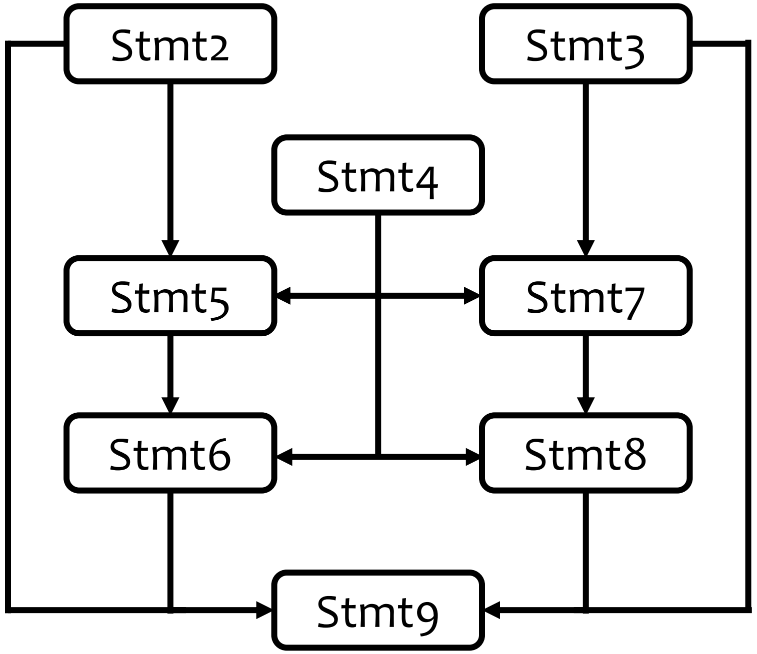
```
def ReturnMax(a, b):
  if a > b:
     return a
  else:
     return b
```

Syntax information is important to code!

Motivation

Program dependency graph is a good choice!

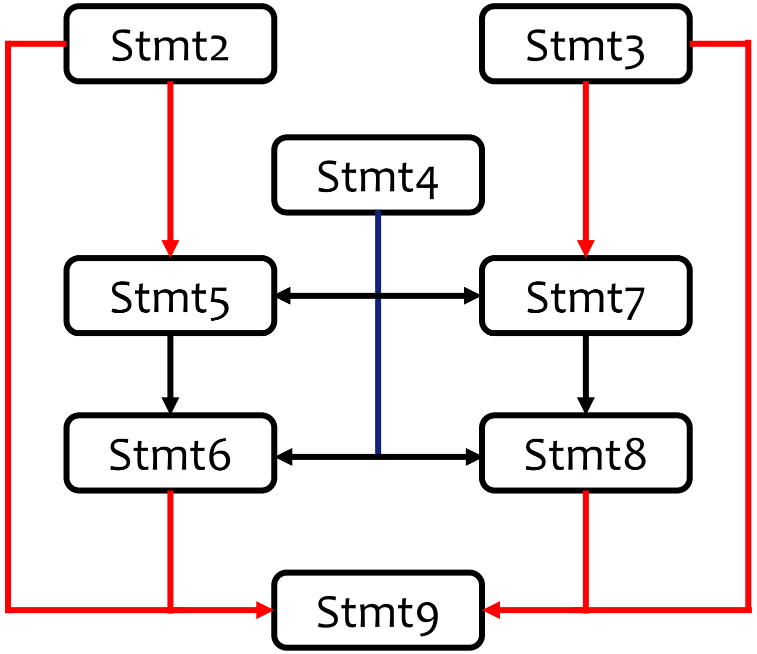
```
1. def ReturnMaxMin(nums):
2.   max = 0
3.   min = 100
4.   for num in nums:
5.     if max < num:
6.       max = num
7.     if min > num:
8.       min = num
9.   return max, min
```



Motivation

Program dependency graph is a good choice!

```
def ReturnMaxMin(nums):
  max = 0
  min = 100
  for num in nums:
     if max < num:
         max = num
     if min > num:
         min = num
  return max, min
```



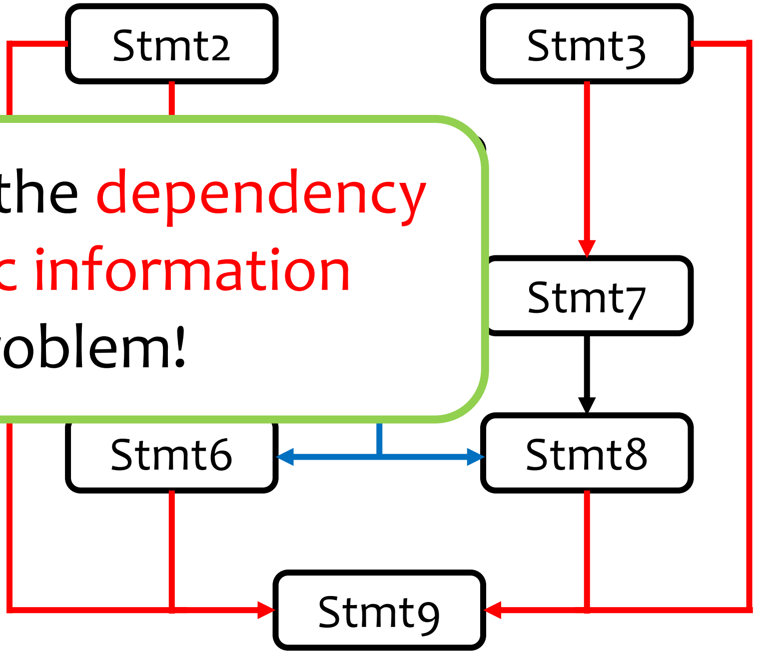
Program dependency graph is composed of data dependency graph

Motivation

Program dependency graph is a good choice!

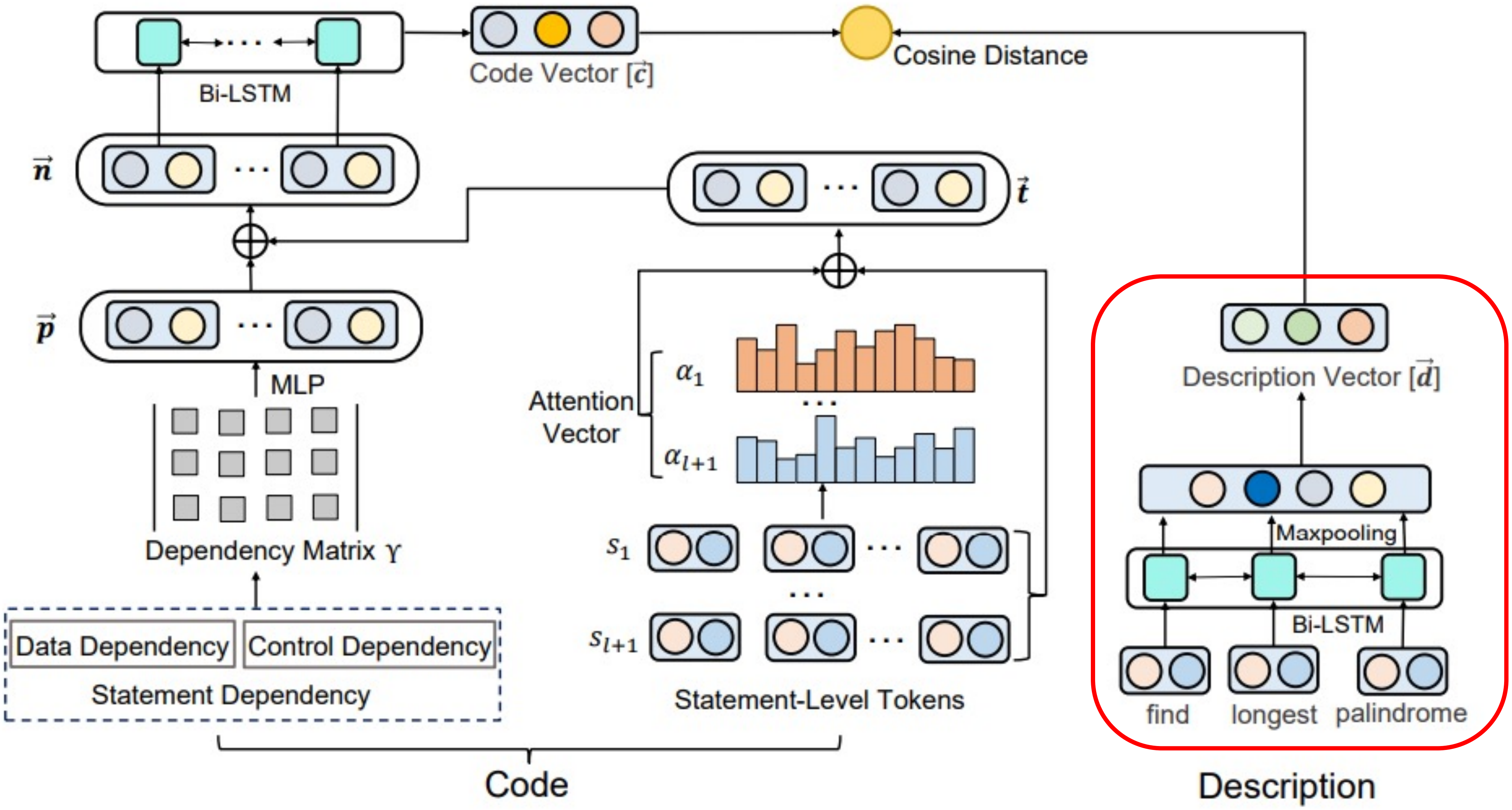
```
1. def ReturnMaxMin(nums):
2.   max = 0
3.   min = 10
4.   for num
5.     if max
6.       max
7.     if min > num:
8.       min = num
9.   return max, min
```

How to effectively combine the dependency information with semantic information becomes the key problem!



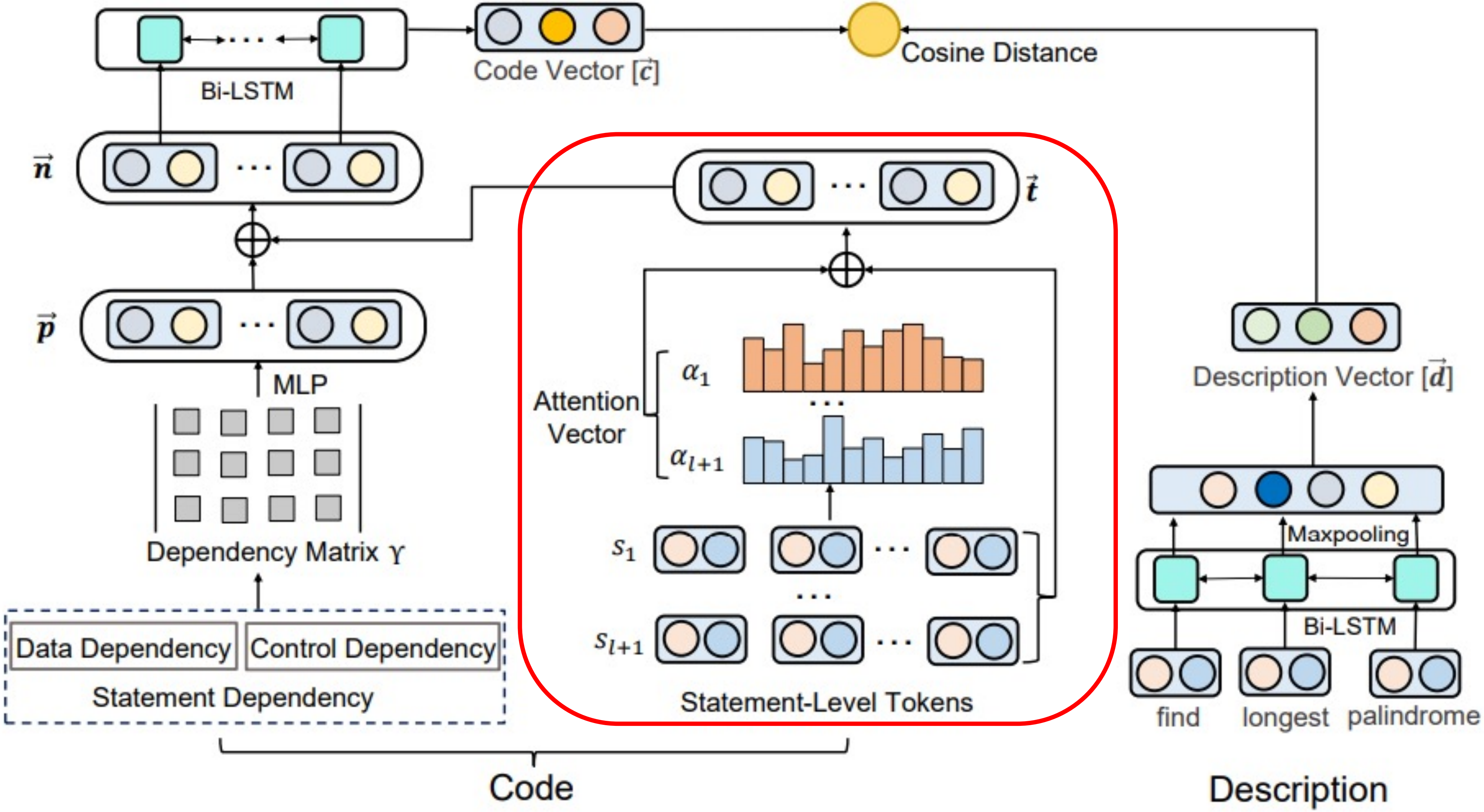
Program dependency graph is composed of data dependency graph and control dependency graph

Overview



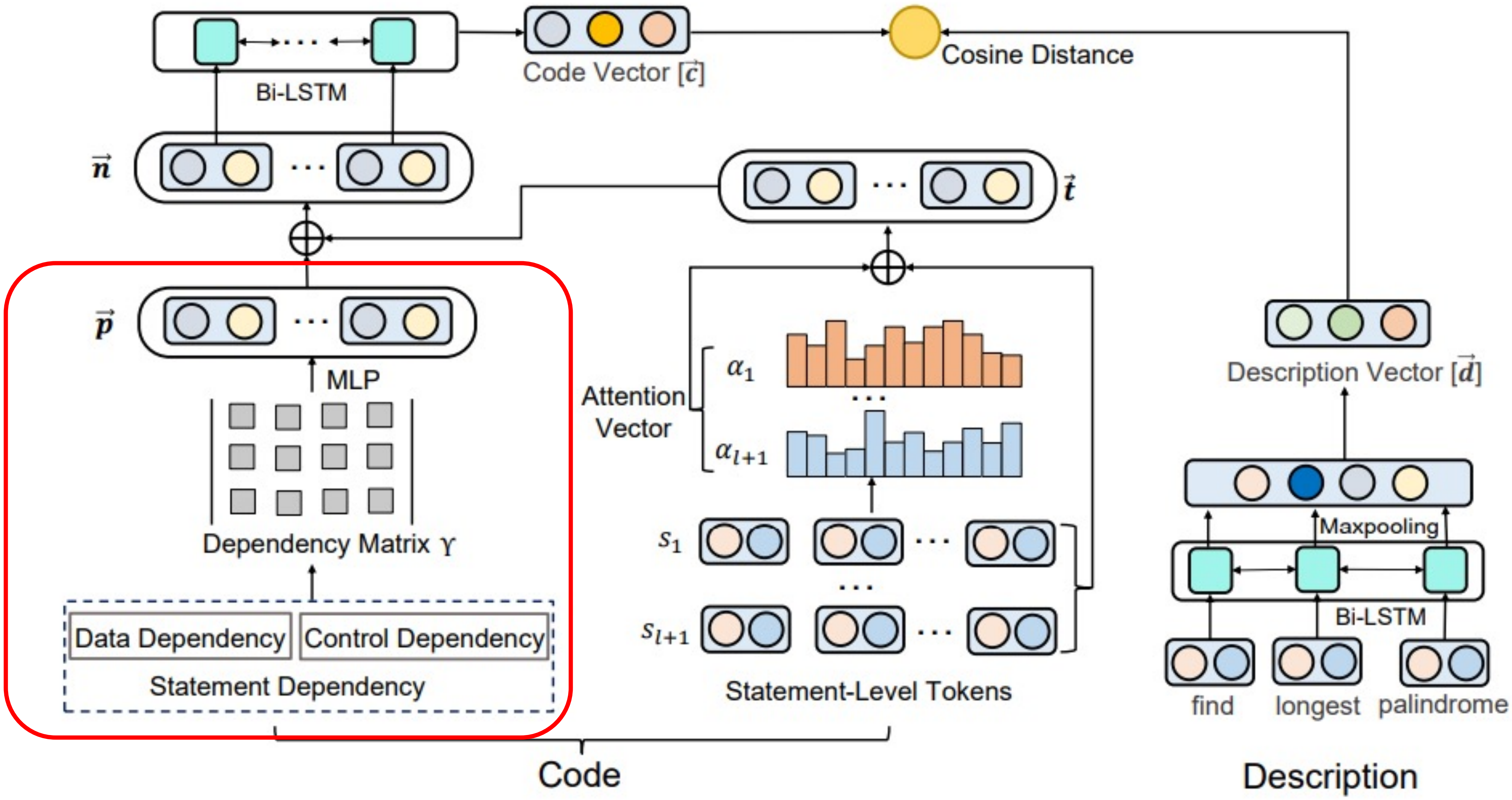
Bi-LSTM
Code Vector [$\vec{c}$]
Cosine Distance
$\vec{n}$
$\vec{t}$
$\vec{p}$
MLP
Attention Vector
α_1
α_{l+1}
Dependency Matrix γ
s_1
s_{l+1}
Data Dependency
Control Dependency
Statement Dependency
Statement-Level Tokens
Code
Description Vector [$\vec{d}$]
Maxpooling
Bi-LSTM
find
longest
palindrome
Description

Overview



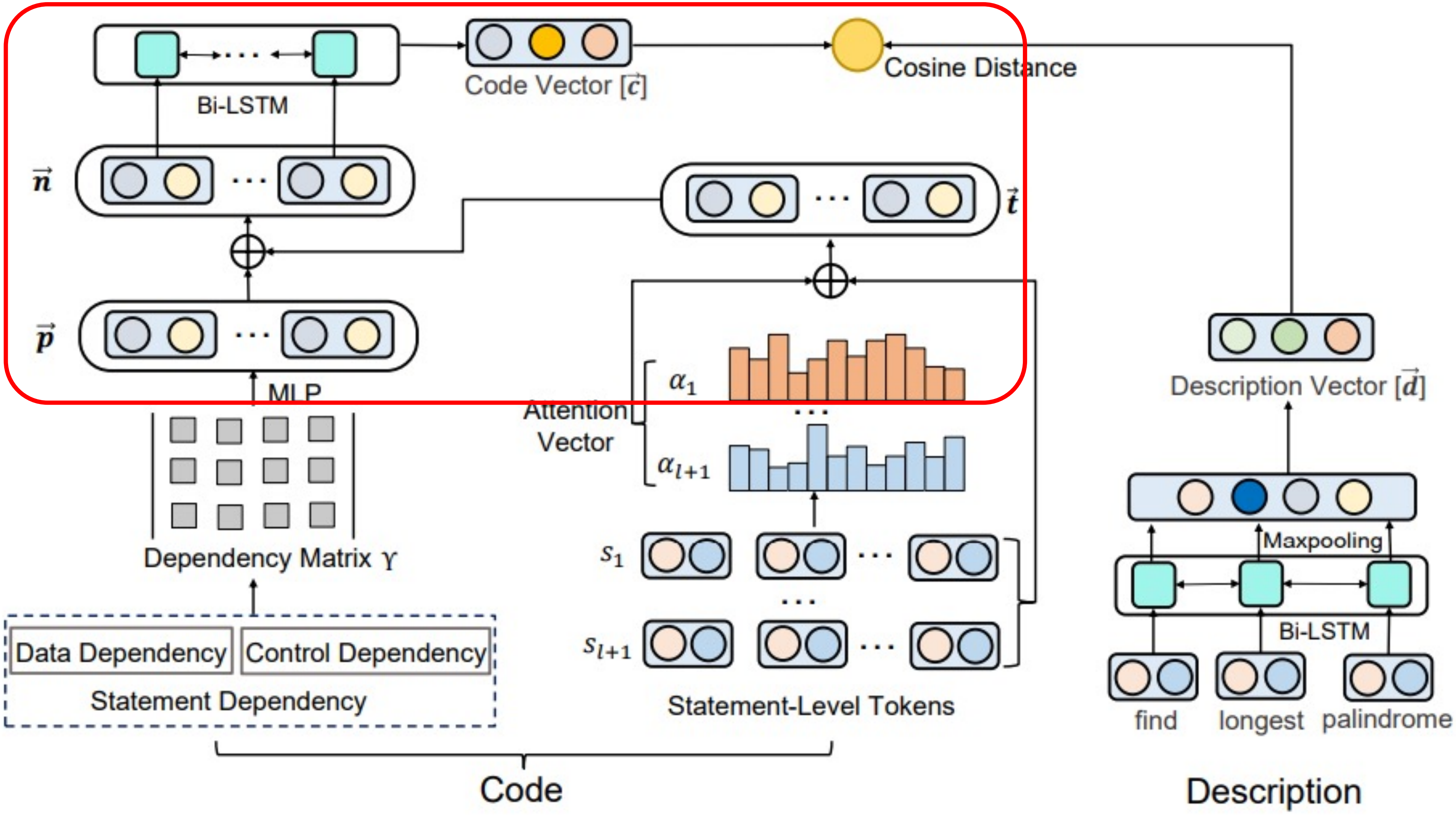
Bi-LSTM
Code Vector [$\vec{c}$]
Cosine Distance
$\vec{n}$
$\vec{t}$
$\vec{p}$
MLP
Attention Vector
α_1
α_{l+1}
s_1
s_{l+1}
Dependency Matrix γ
Data Dependency
Control Dependency
Statement Dependency
Statement-Level Tokens
Description Vector [$\vec{d}$]
Maxpooling
Bi-LSTM
find
longest
palindrome
Code
Description

Overview



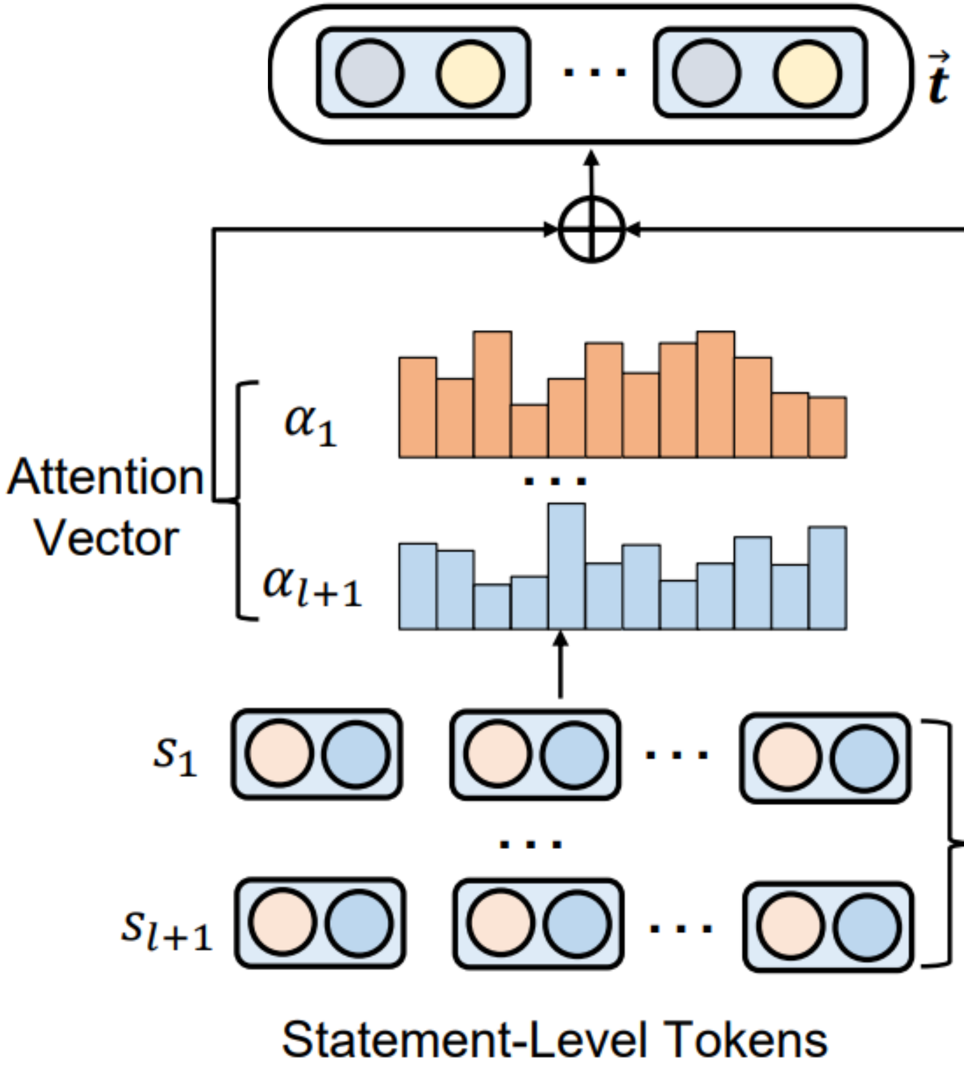
Bi-LSTM
Code Vector [$\vec{c}$]
Cosine Distance
$\vec{n}$
$\vec{t}$
$\vec{p}$
MLP
Dependency Matrix Υ
Data Dependency
Control Dependency
Statement Dependency
Attention Vector
α_1
α_{l+1}
s_1
s_{l+1}
Statement-Level Tokens
Code
Description Vector [$\vec{d}$]
Maxpooling
Bi-LSTM
find
longest
palindrome
Description

Overview



Bi-LSTM
Code Vector [$\vec{c}$]
Cosine Distance
$\vec{n}$
$\vec{t}$
$\vec{p}$
MLP
Attention Vector
α_1
α_{l+1}
Dependency Matrix γ
s_1
s_{l+1}
Data Dependency
Control Dependency
Statement Dependency
Statement-Level Tokens
Description Vector [$\vec{d}$]
Maxpooling
Bi-LSTM
find
longest
palindrome
Code
Description

Statement-Level Semantic Embedding



Tokenize statement and embed them into vectors:

$$s_i = \{e_{i1}, \ldots, e_{ij}, \ldots\}$$

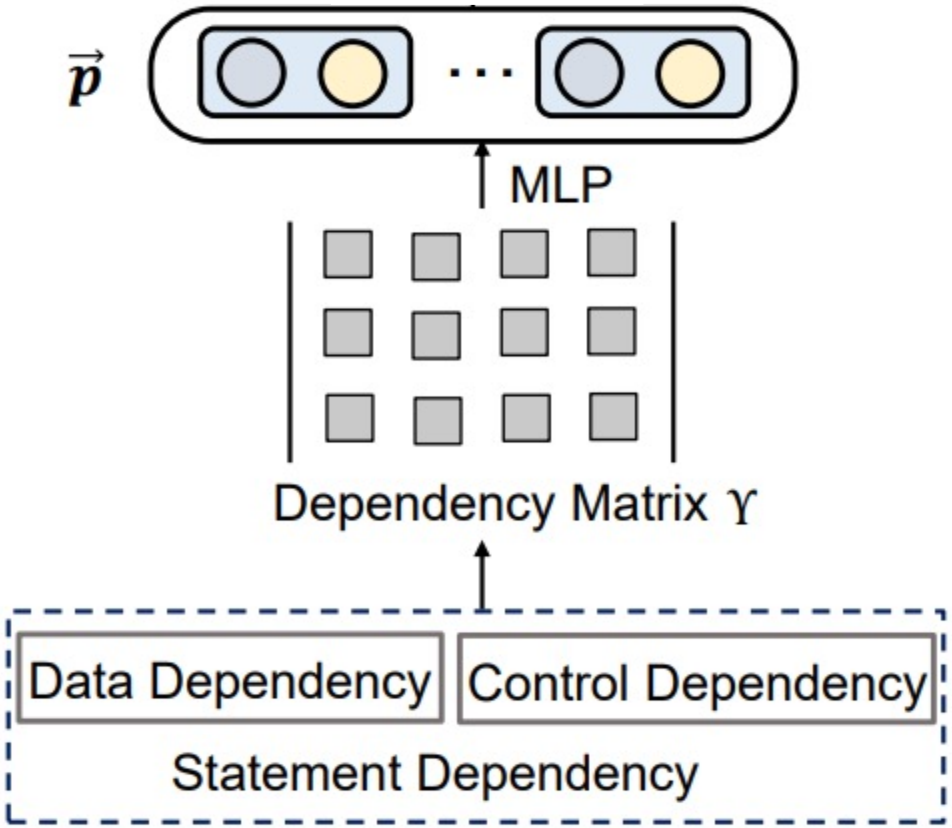
Calculate the attention score for each vector:

$$\alpha_{ij} = \frac{\exp\left(e_{ij}^T\right)}{\sum_j \exp\left(e_{ij}^T\right)}$$

Get the statement-level vector based on attention weight:

$$s_i = \sum_j \alpha_{ij} e_{ij}^T$$

Statement-Level Dependency Embedding



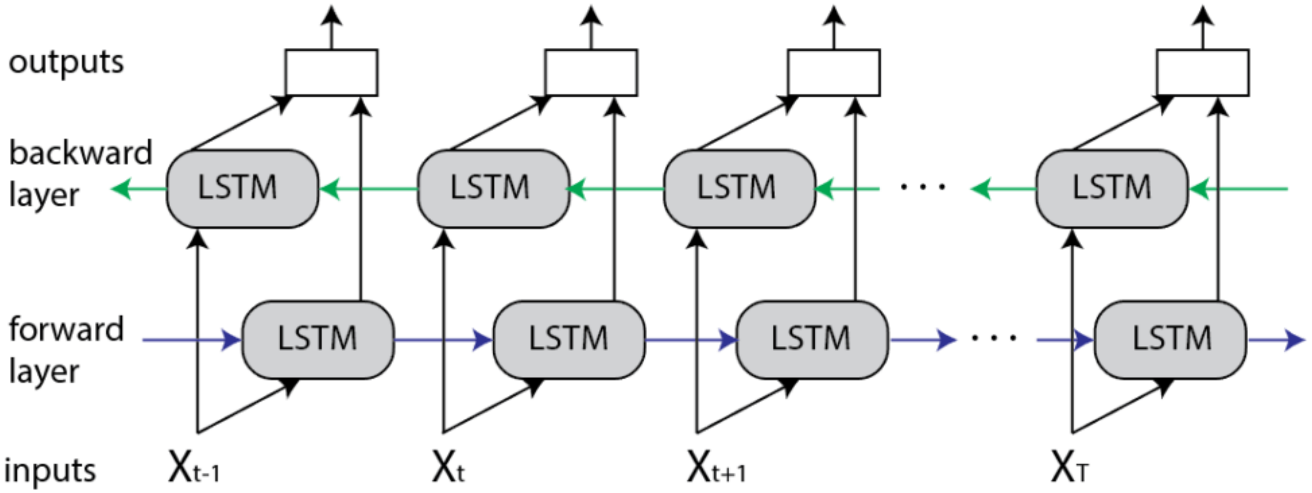
Construct the dependency matrix:

$$\Gamma_i = \{v_{i1}, \ldots, v_{ij}, \ldots\}, v_{ij} \in \{0,1\}$$

Embed the dependecy matrix via multilayer perceptron (MLP):

$$p_i = \tanh(W^{\Gamma} \Gamma_i)$$

Function-Level Vector Generation



Vector concatenation:

$$x_t = [s_i, p_i]$$

Feed into bi-LSTM:

$$h_t = \mathrm{bi} - \mathrm{LSTM}(h_{t-1}, x_t)$$
$$c = \mathrm{atten}(h_1, \ldots, h_t, \ldots)$$

Experiments

- Dataset:
  - CodeSearchNet (Python): released by H. Husain et al.
  - Code2Seq (Python): released by U. Alon et al. in ICLR 2019
- Baselines
  - CODEnn, UNIF, NeuralBoW, RNN, CONV, CONVSelf, SelfAttn
- Metrics:
  - Recall@1 (R@1), R@5, R@10, Mean Reciprocal Rank (MRR)

Experiments

We achieve state-of-the-art performance on both dataset

TABLE I
COMPARISON RESULTS WITH BASELINE MODELS ON THE CODESEARCHNET DATASET. THE BEST RESULTS ARE HIGHLIGHTED IN BOLD FONTS.

Approach	R@1	R@5	R@10	MRR
CODEnn	0.367	0.573	0.652	0.465
UNIF	0.379	0.615	0.706	0.490
NeuralBoW	0.521	0.747	0.807	0.622
RNN	0.556	0.772	0.832	0.654
CONV	0.475	0.703	0.776	0.579
CONVSelf	0.571	0.788	0.845	0.668
SelfAttn	0.580	0.786	0.840	0.673
$\text{CRaDLe}_{\text{maxpooling}}$	0.777	0.914	0.946	0.838
CRaDLe	**0.791**	**0.923**	**0.951**	**0.843**

TABLE II
COMPARISON RESULTS WITH BASELINE MODELS ON THE CODE2SEQ DATASET. THE BEST RESULTS ARE HIGHLIGHTED IN BOLD FONTS.

Approach	R@1	R@5	R@10	MRR
CODEnn	0.330	0.532	0.617	0.427
UNIF	0.380	0.588	0.668	0.478
NeuralBoW	0.546	0.693	0.738	0.615
RNN	0.438	0.623	0.688	0.526
CONV	0.425	0.584	0.645	0.502
CONVSelf	0.470	0.642	0.700	0.552
SelfAttn	0.525	0.683	0.731	0.599
$\text{CRaDLe}_{\text{maxpooling}}$	0.664	0.843	0.892	0.745
CRaDLe	**0.668**	**0.849**	**0.897**	**0.749**

Ablation Study

The combination of data dependency and control dependency can improve the performance

TABLE III
ABLATION STUDY ON THE CODESEARCHNET DATASET.

Approach	R@1	R@5	R@10	MRR
$\text{CRaDLe}_{\text{Full}}$	0.791	0.923	0.951	0.843
$\text{CRaDLe}_{\text{DataDependency}}$	0.779	0.910	0.946	0.840
$\text{CRaDLe}_{\text{ControlDependency}}$	0.785	0.918	0.950	0.845

TABLE IV
ABLATION STUDY ON THE CODE2SEQ DATASET.

Approach	R@1	R@5	R@10	MRR
$\text{CRaDLe}_{\text{Full}}$	0.668	0.849	0.897	0.749
$\text{CRaDLe}_{\text{DataDependency}}$	0.645	0.827	0.880	0.724
$\text{CRaDLe}_{\text{ControlDependency}}$	0.645	0.828	0.882	0.730

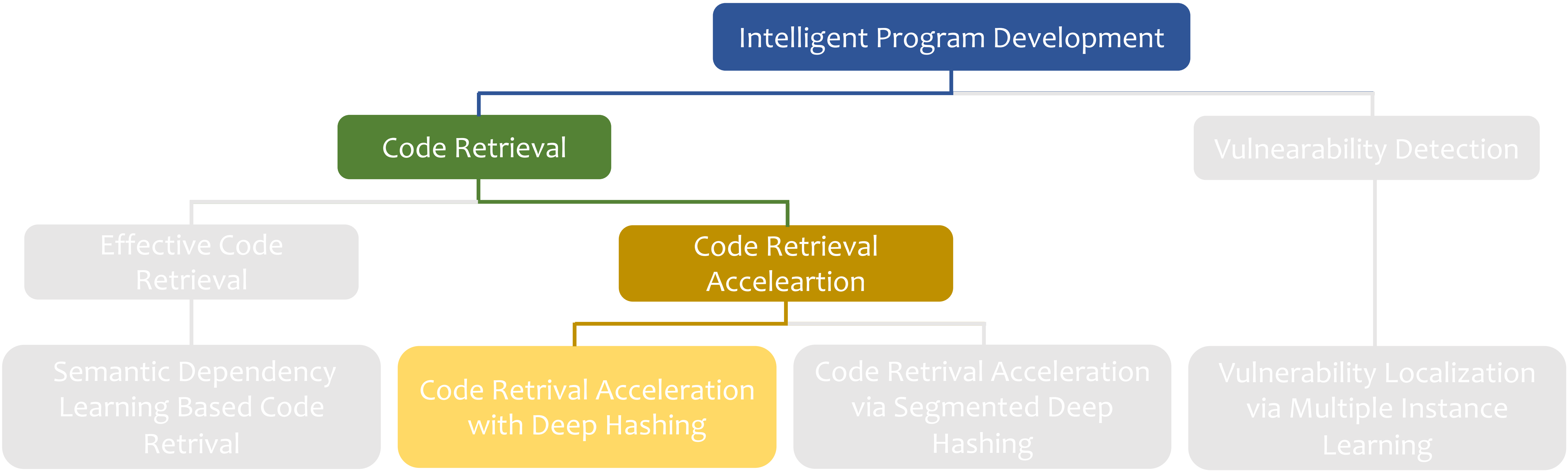
Summary

- Propose a novel code retrieval model which firstly integrates the dependency and semantics information at the statement level

- The experiment results demonstrate its superior performance over the baseline models

* Under review

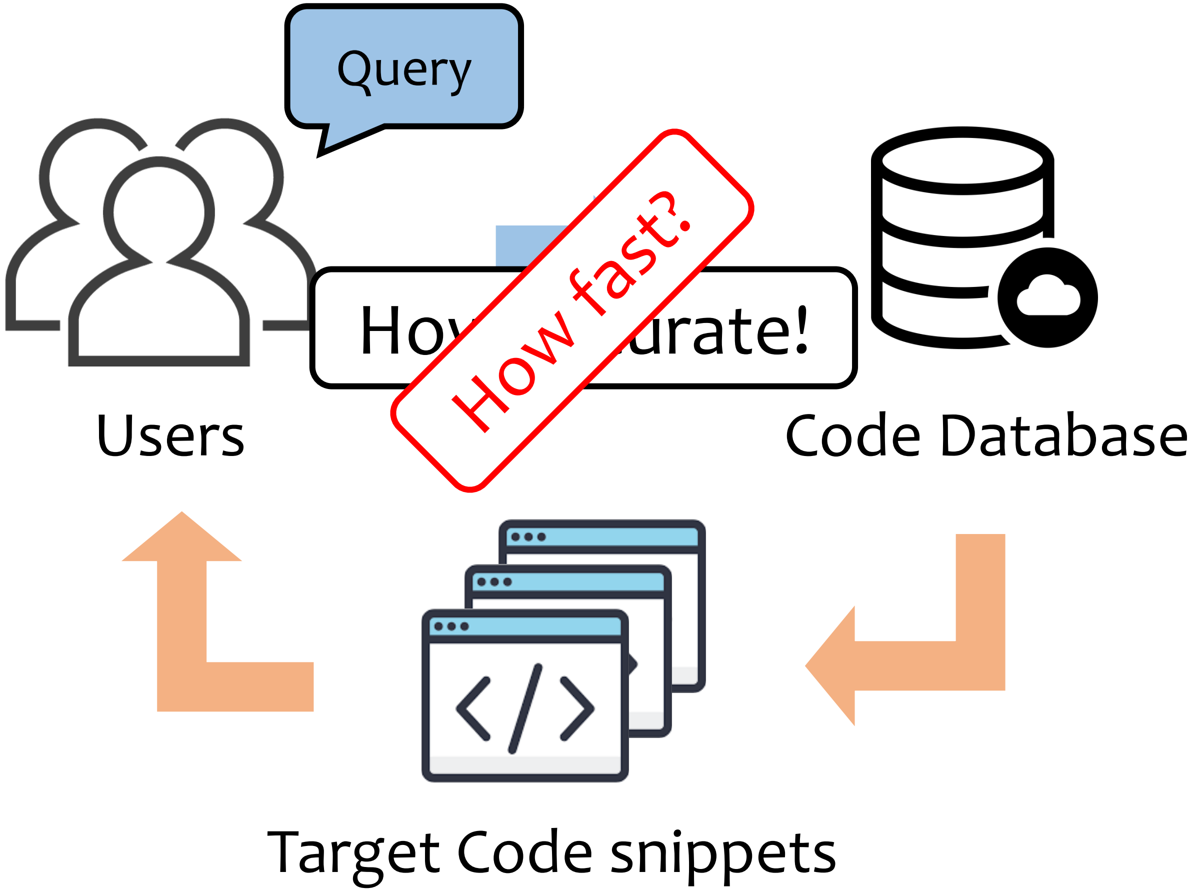
Thesis Structure



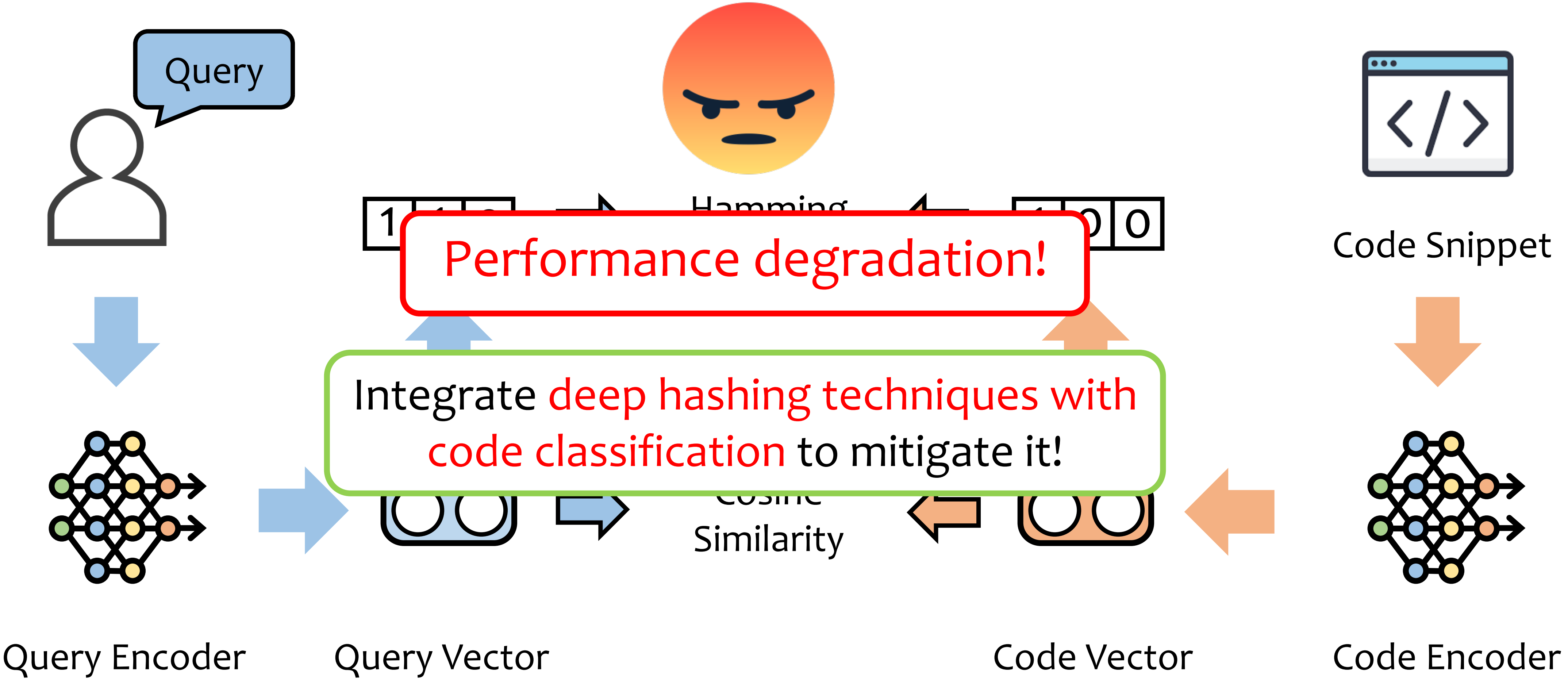
2

Code Retrival Acceleration with Deep Hashing

Motivation



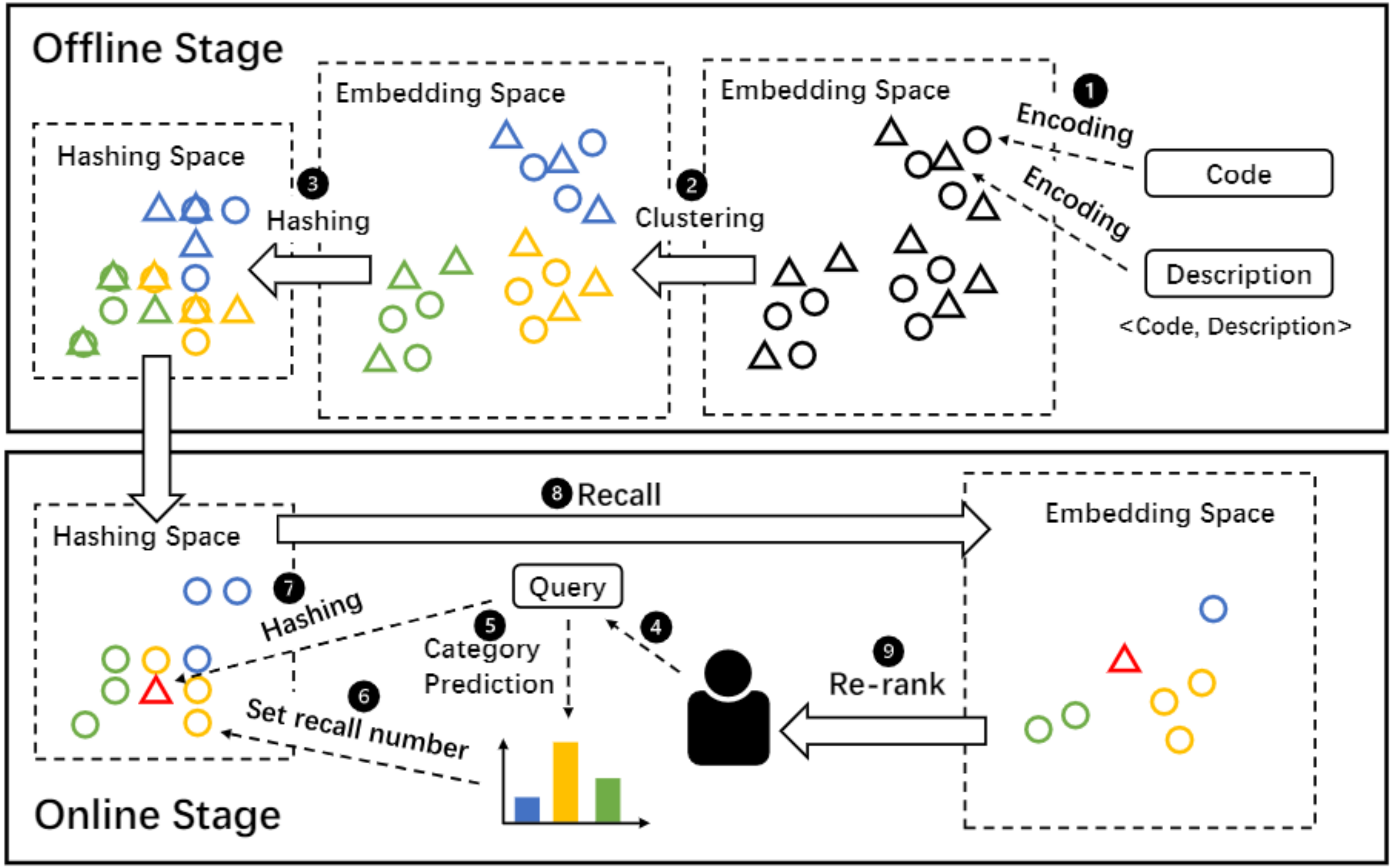
Motivation



Performance degradation!

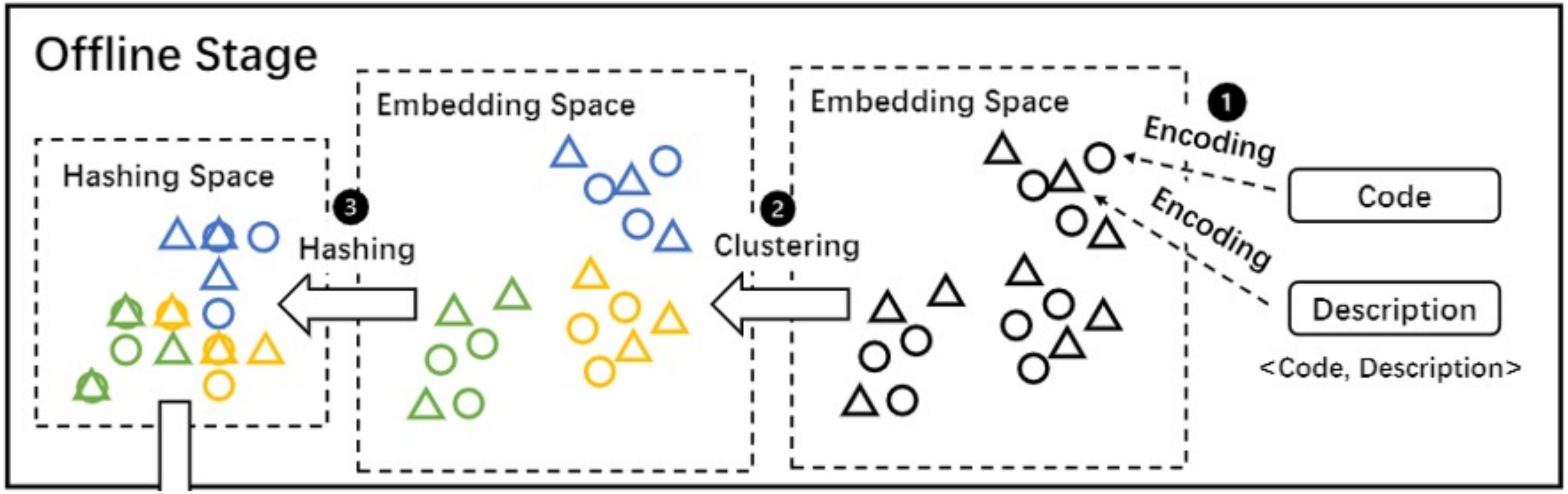
Integrate deep hashing techniques with code classification to mitigate it!

Overview



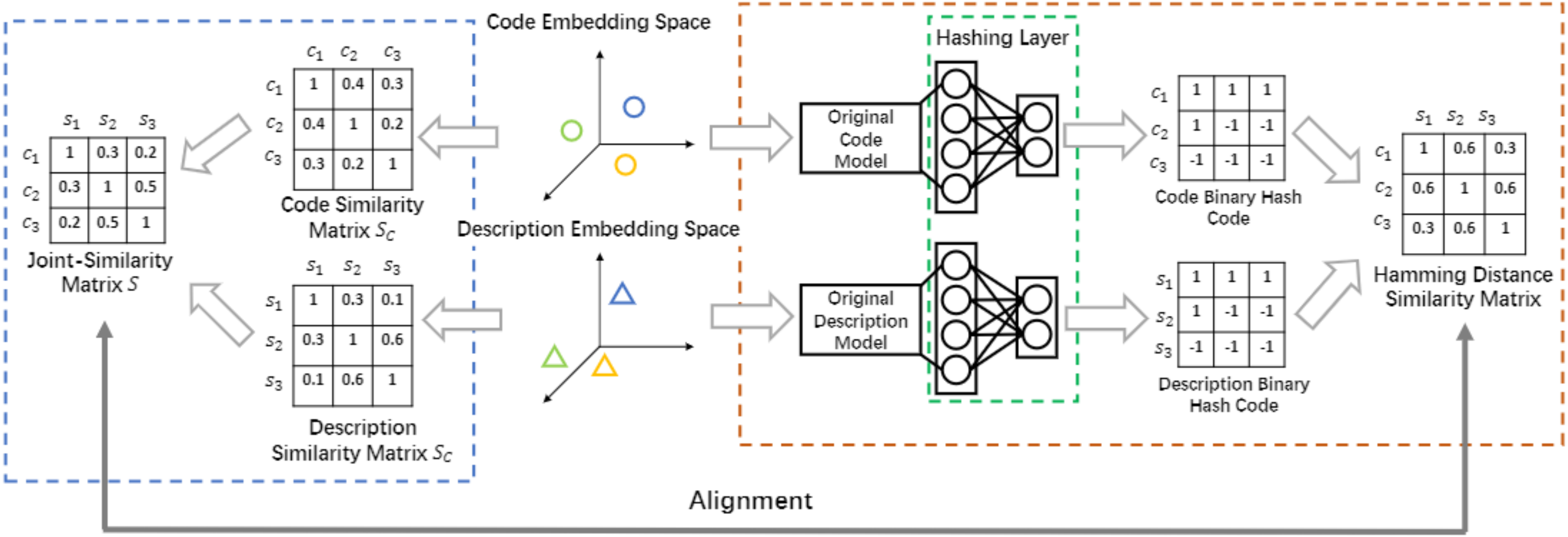
Offline Stage
Embedding Space
Embedding Space
1
Encoding
Code
Encoding
Description
<Code, Description>
Hashing Space
3
Hashing
2
Clustering
Online Stage
8 Recall
Hashing Space
Embedding Space
7
Hashing
Query
5
Category
Prediction
4
9
Re-rank
6
Set recall number

Offline Stage

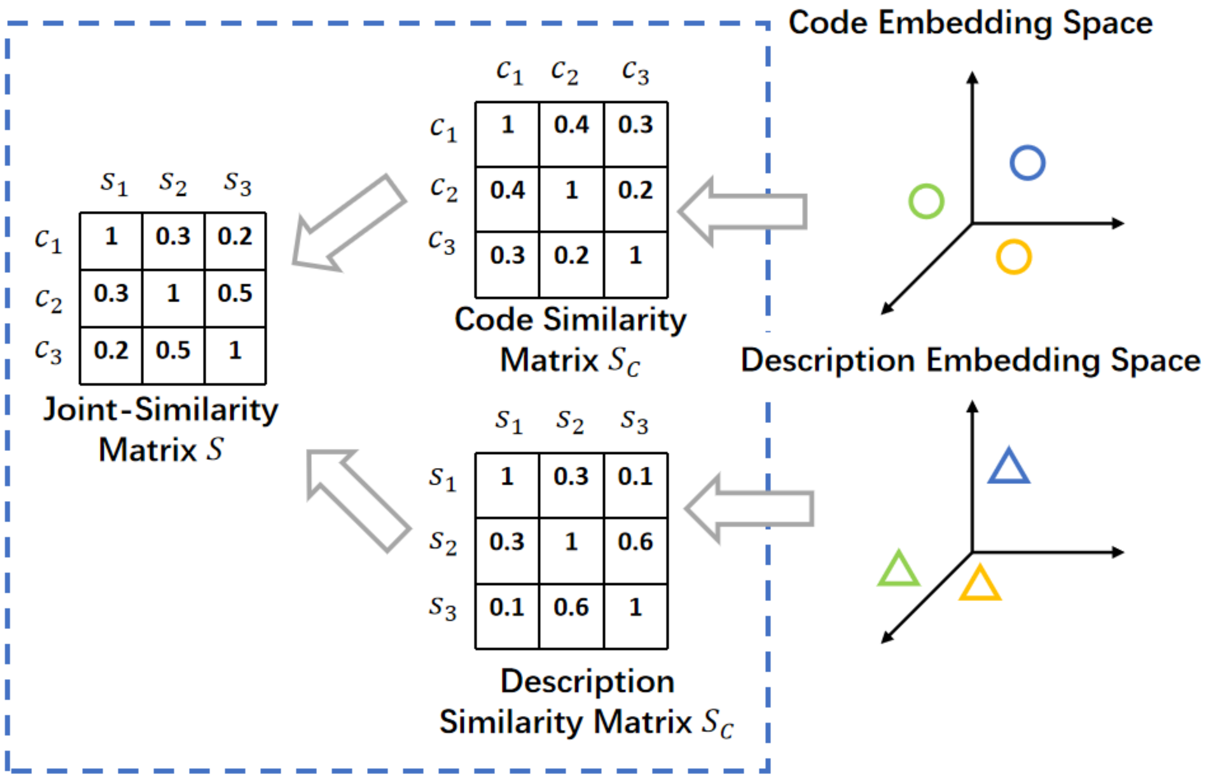


- Encode code and description (query) with existing deep learning approaches
- Cluster the vectors into several categories with k-mean algorithm and train a category predictor
- Hashing the code and description into binary hash codes

Hashing Training



Joint-Similarity Matrix



- Construct the similarity matrix for code modality and description modality:

$$S_c = V_C V_C^T, S_D = V_D V_D^T$$

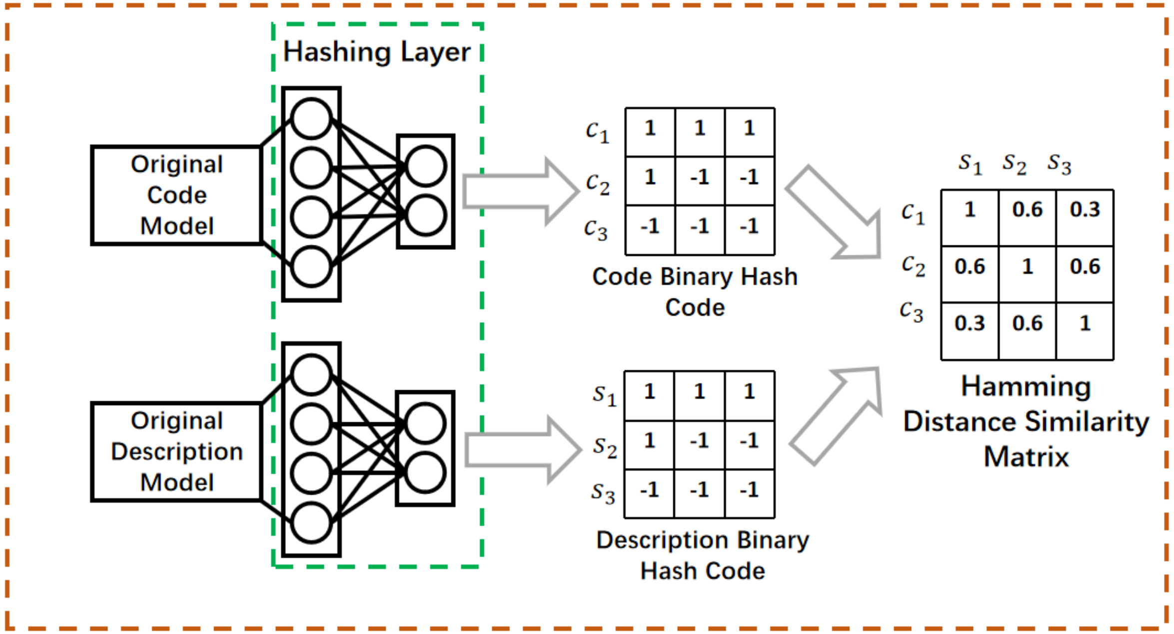
- Construct the joint-similarity matrix:

$$\tilde{S} = \beta S_c + (1-\beta) S_D$$

$$S = \eta \tilde{S} + (1-\eta) \frac{\tilde{S}\tilde{S}^T}{m}$$

$$S_{F_{ij}} = \begin{cases} 1, & \mathrm{i} = \mathrm{j} \\ S_{ij}, & \mathrm{otherwise} \end{cases}$$

Hamming Distance Similarity Matrix



- Generate the hash code from the hashing model:

$$B = \mathrm{sgn}(H) \in \{-1,1\}^{m \times d}$$

- Construct the Hamming Distance similarity matrix:

$$S_{CC} = \frac{B_C B_C^T}{d}$$

$$S_{CD} = \frac{B_C B_D^T}{d}$$

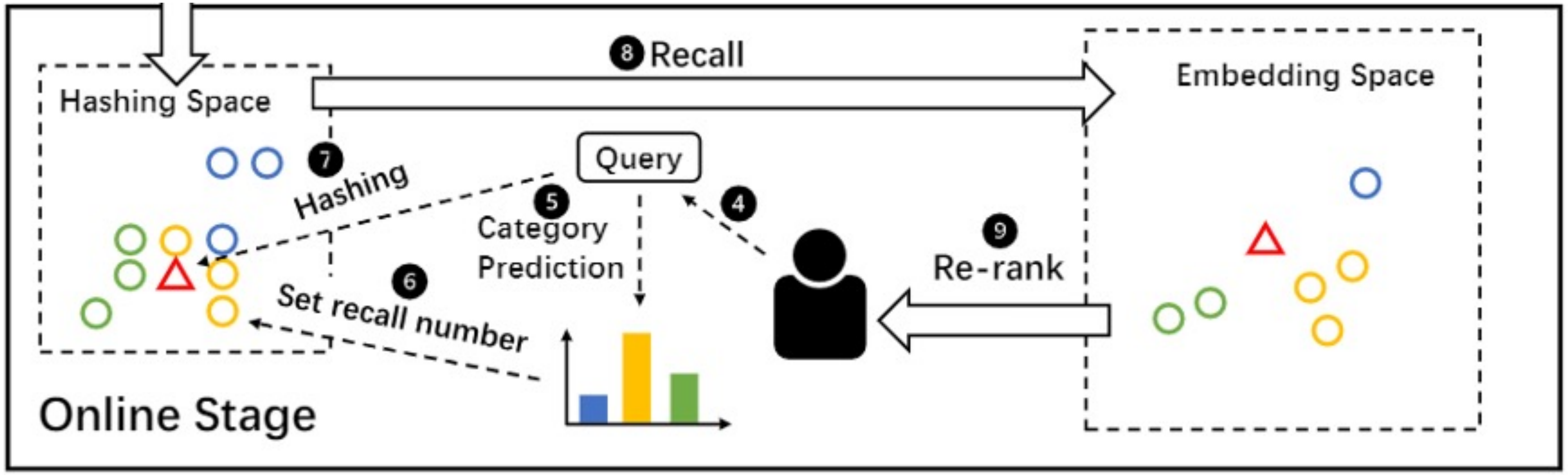
$$S_{DD} = \frac{B_D B_D^T}{d}$$

Training Loss Design

- Alignment between the joint-similarity matrix and Hamming Distance similarity matrix

$$\mathcal{L}(\theta) = \min_{B_C,B_D} \|\min(\mu S_F, 1 - S_{CC})\|_F^2 + \lambda_1 \|\min(\mu S_F, 1 - S_{CD})\| + \lambda_2 \|\min(\mu S_F, 1 - S_{DD})\|$$

Online Stage



- Predict the probability of given query category and set the recall number for each category

$$R_i = min(\lfloor p_i \cdot (N - k) \rfloor, 1), \qquad \forall i = 1, \ldots, k$$

- Generate the hash code for query and recall the candidates in each category via Hamming distance

- Re-rank the code candidates according to the cosine simiarity of the vectors

Experiments

- Dataset:
  - CodeSearchNet (Python, Java): released by H. Husain et al.

- Baselines
  - Non pre-trained model: RNN, UNIF
  - Pre-trained model: CodeBERTa, CodeBERT, GraphCodeBERT

- Metrics:
  - R@1, R@5, R@10

Experiments

- Our method can reduce more than 90% of retrieval time

TABLE V
TIME EFFICIENCY OF COSHC.

	Python	Java
	Total Time	
CodeBERT	572.97s	247.78s
CSSDH	33.87s (↓94.09%)	15.78s (↓93.51%)
	(1) Vector Similarity Calculation	
CodeBERT	531.95s	234.08s
CSSDH	14.43s (↓97.29%)	7.25s (↓96.90%)
	(2) Array Sorting	
CodeBERT	41.02s	13.70s
CSSDH	19.44s (↓53.61%)	8.53s (↓37.74%)

Experiments

Our method can retain most of performance and even outperforms the original code retrieval models when its performance is bad

TABLE VI
RESULTS OF CODE SEARCH PERFORMANCE COMPARISON.

Model	Python			Java		
	R@1	**R@5**	**R@10**	**R@1**	**R@5**	**R@10**
UNIF	0.071	0.173	0.236	0.084	0.193	0.254
$\text{CSSDH}_{\text{UNIF}}$	0.072 (↑1.4%)	0.177 (↑2.3%)	0.241 (↑2.1%)	0.086 (↑2.4%)	0.198 (↑2.6%)	0.264 (↑3.9%)
RNN	0.111	0.253	0.333	0.073	0.184	0.250
$\text{CSSDH}_{\text{RNN}}$	0.112 (↑0.9%)	0.259 (↑2.4%)	0.343 (↑5.0%)	0.076 (↑4.1%)	0.194 (↑5.4%)	0.265 (↑6.0%)
CodeBERTa	0.124	0.250	0.314	0.089	0.203	0.264
$\text{CSSDH}_{\text{CodeBERTa}}$	0.123 (↓0.8%)	0.247 (↓1.2%)	0.309 (↓1.6%)	0.090 (↑1.1%)	0.210 (↑3.4%)	0.272 ((↑3.0%)
CodeBERT	0.451	0.683	0.759	0.319	0.537	0.608
$\text{CSSDH}_{\text{CodeBERT}}$	0.451 (0.0%)	0.679 (↓0.6%)	0.750 (↓1.2%)	0.318 (↓0.3%)	0.533 (↓0.7%)	0.602 (↓1.0%)
GraphCodeBERT	0.485	0.726	0.792	0.353	0.571	0.640
$\text{CSSDH}_{\text{GraphCodeBERT}}$	0.483 (↓0.4%)	0.719 (↓1.0%)	0.782 (↓1.3%)	0.350 (↓0.8%)	0.561 (↓1.8%)	0.625 (↓2.3%)

Ablation Study

TABLE VII

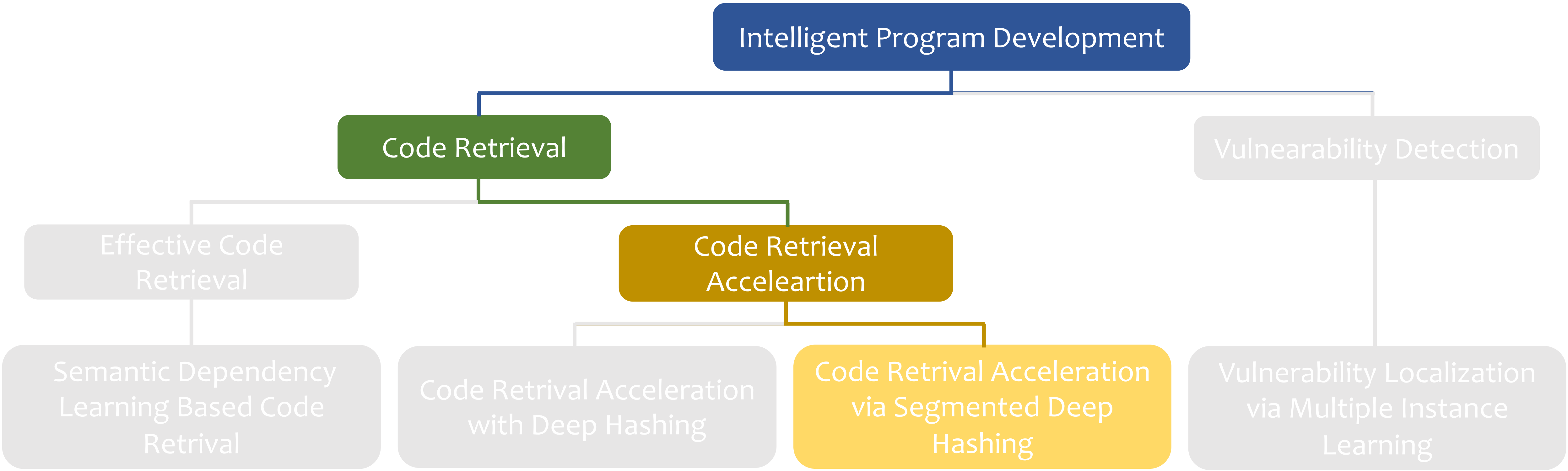
RESULTS OF CODE SEARCH PERFORMANCE COMPARISON. THE BEST RESULTS AMONG THE THREE COSHC VARIANTS ARE HIGHLIGHTED IN BOLD FONT.

Model	Python			Java		
	R@1	R@5	R@10	R@1	R@5	R@10
$\text{CSSDH}_{\text{UNIF}}$	**0.072 (↑1.4%)**	**0.177 (↑2.3%)**	**0.241 (↑2.1%)**	**0.086 (↑2.4%)**	**0.198 (↑2.6%)**	**0.264 (↑3.9%)**
−w/o classification	0.071 (0.0%)	0.174 (↑0.6%)	0.236 (0.0%)	0.085 (↑1.2%)	0.193 (0.0%)	0.254 (0.0%)
−one classification	0.069 (↓2.8%)	0.163 (↓5.8%)	0.216 (↓8.5%)	0.083 (↓1.2%)	0.183 (↓5.2%)	0.236 (↓7.1%)
$\text{CSSDH}_{\text{RNN}}$	**0.112 (↑0.9%)**	**0.259 (↑2.4%)**	**0.343 (↑5.0%)**	**0.076 (↑4.1%)**	**0.194 (↑5.4%)**	**0.265 (↑6.0%)**
−w/o classification	**0.112 (↑0.9%)**	0.254 (↑0.4%)	0.335 (↑0.6%)	0.073 (0.0%)	0.186 (↑1.1%)	0.253 (↑1.2%)
−one classification	**0.112 (↑0.9%)**	0.243 (↓4.0%)	0.311 (↓6.6%)	0.075 (↑2.7%)	0.182 (↓1.1%)	0.240 (↓4.0%)
$\text{CSSDH}_{\text{CodeBERTa}}$	**0.123 (↓0.8%)**	**0.247 (↓1.2%)**	**0.309 (↓1.6%)**	**0.090 (↑1.1%)**	**0.210 (↑3.4%)**	**0.272 ((↑3.0%)**
−w/o classification	0.122 (↓1.6%)	0.242 (↓3.2%)	0.302 (↓3.8%)	0.089 (0.0%)	0.201 (↓1.0%)	0.258 (↓2.3%)
−one classification	0.116 (↓6.5%)	0.221 (↓11.6%)	0.271 (↓13.7%)	0.085 (↓4.5%)	0.189 (↓6.9%)	0.238 (↓9.8%)
$\text{CSSDH}_{\text{CodeBERT}}$	**0.451 (0.0%)**	**0.679 (↓0.6%)**	**0.750 (↓1.2%)**	**0.318 (↓0.3%)**	**0.533 (↓0.7%)**	**0.602 (↓1.0%)**
−w/o classification	0.449 (↓0.4%)	0.673 (↓1.5%)	0.742 (↓2.2%)	0.316 (↓0.9%)	0.527 (↓1.9%)	0.593 (↓2.5%)
−one classification	0.425 (↓5.8%)	0.613 (↓10.2%)	0.665 (↓12.4%)	0.304 (↓4.7%)	0.483 (↓10.1%)	0.532 (↓12.5%)
$\text{CSSDH}_{\text{GraphCodeBERT}}$	**0.483 (↓0.4%)**	**0.719 (↓1.0%)**	**0.782 (↓1.3%)**	**0.350 (↓0.8%)**	**0.561 (↓1.8%)**	**0.625 (↓2.3%)**
−w/o classification	0.481 (↓0.8%)	0.713 (↓1.8%)	0.774 (↓2.3%)	0.347 (↓1.7%)	0.553 (↓3.2%)	0.616 (↓3.7%)
−one classification	0.459 (↓5.4%)	0.653 (↓10.1%)	0.698 (↓11.9%)	0.329 (↓7.8%)	0.505 (↓11.6%)	0.551 (↓13.9%)

Summary

- Propose a novel approach which firstly integrates code calssificiation and deep hashing
- The experiment results demonstrate its ability to greatly improve the retrieval efficiency meanwhile preserve almost the same performance

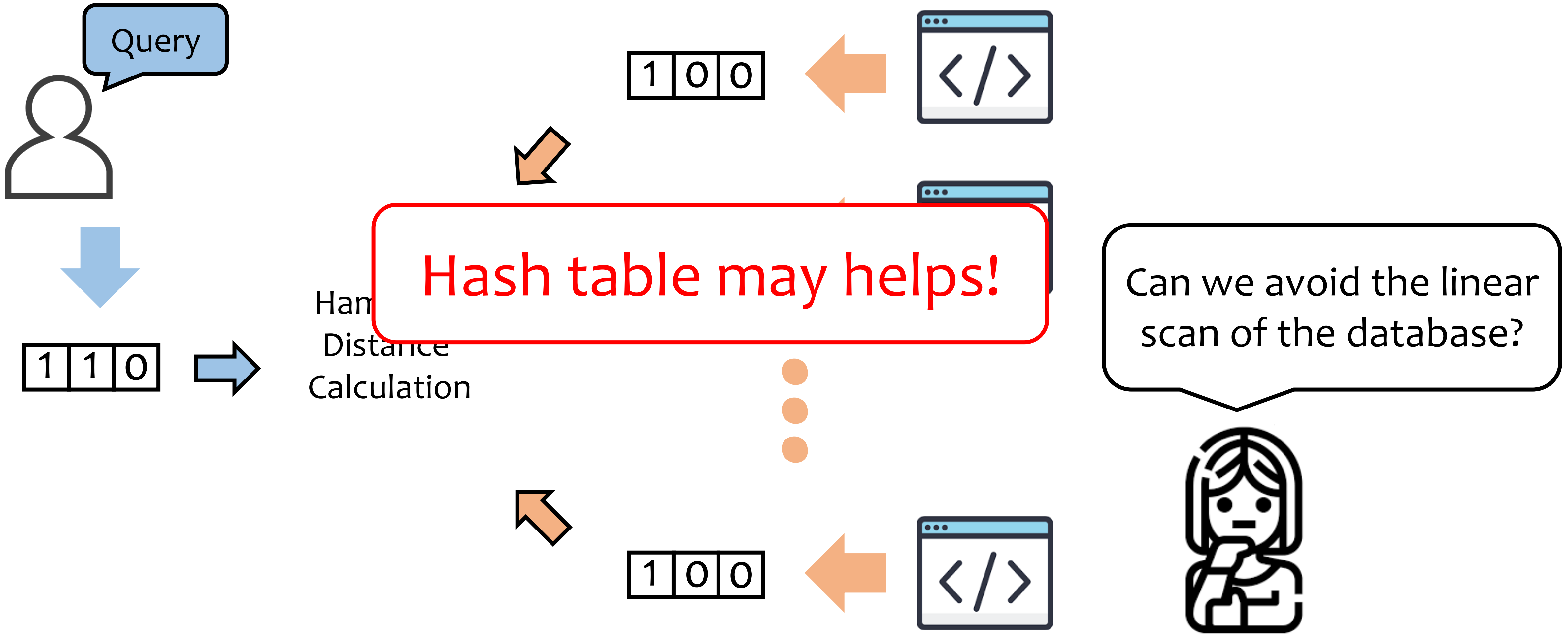
Thesis Structure



3

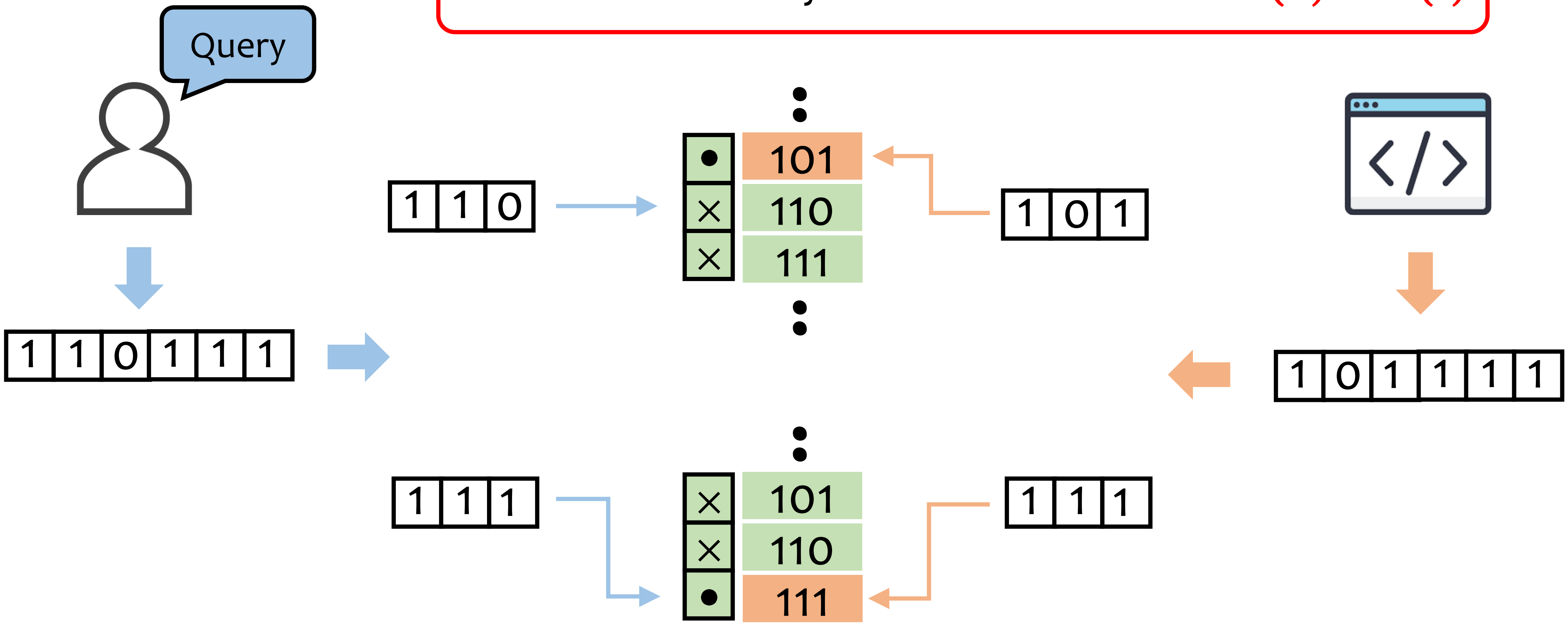
Code Retrival Acceleration via Segmented Deep Hashing

Motivation

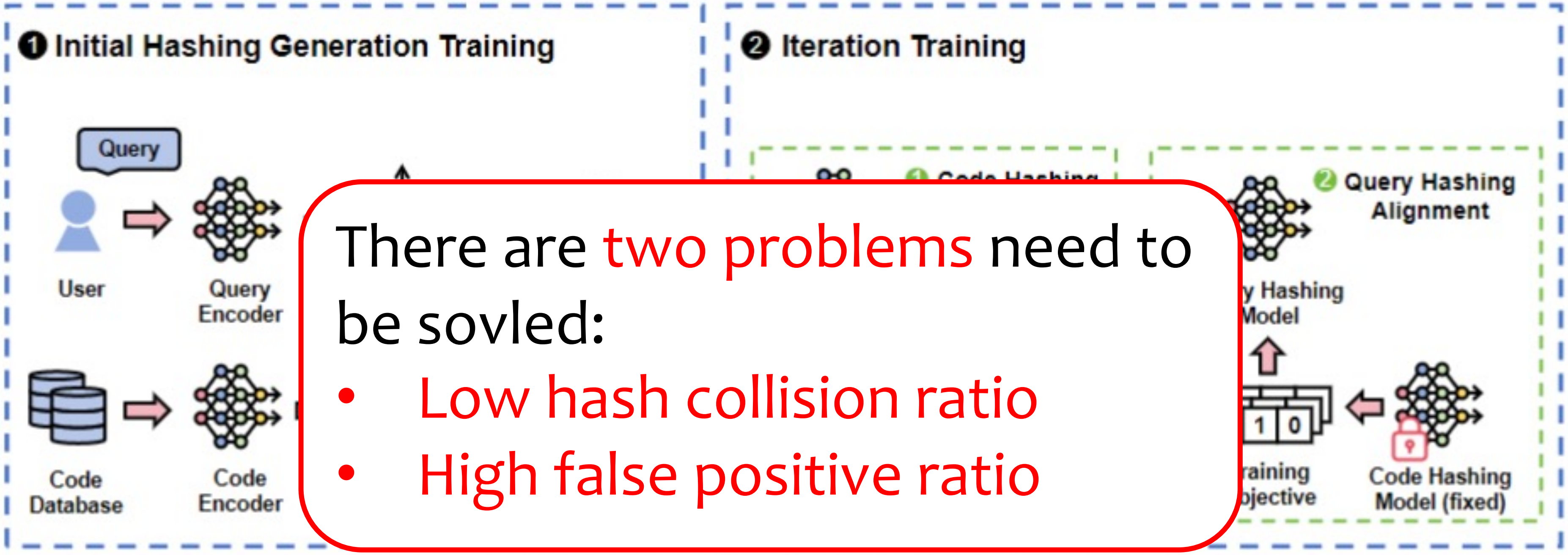

Query
1 1 0
Ham
Distance
Calculation
1 0 0
Hash table may helps!
1 0 0
Can we avoid the linear scan of the database?

Motivation

The time comlexity can be reduced from O(n) to O(1)



Training Strategy

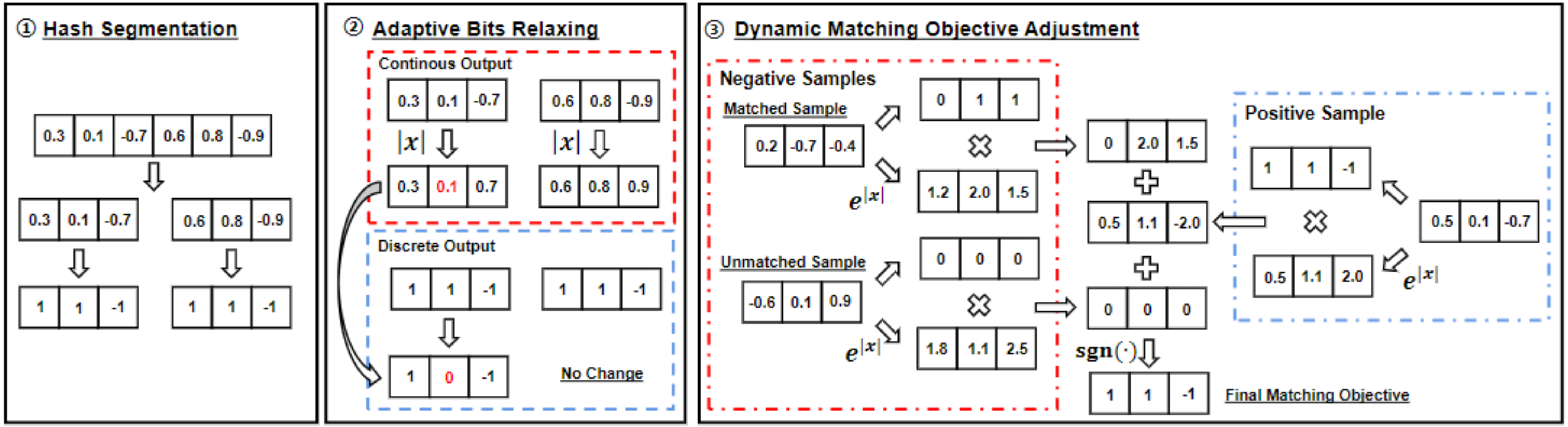


There are two problems need to be sovled:

- Low hash collision ratio
- High false positive ratio

- Train the hash code with previous deep hashing approaches
- Alternately lock one hashing model and train the other hashing model

Methodology



To address low hash collision ratio:

- Hash segementation
- Adaptive bits relaxing

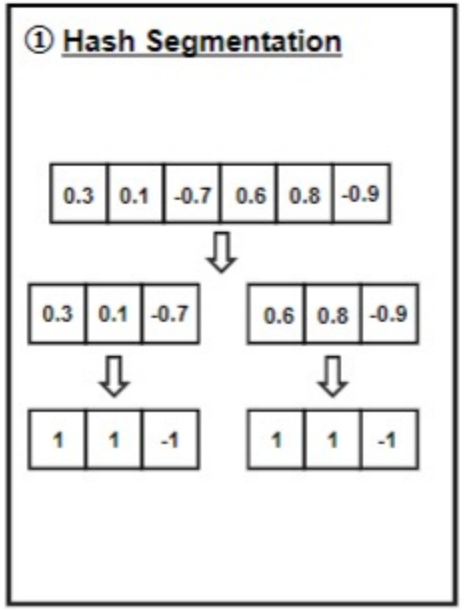
To address high false positive ratio:

- Dynamic matching objective adjustment

Hash Segmentation

Hash Segmentation: Split the long hash code into segmented hash codes

- Reduce the difficulty of alignment
- Increase the number of hash table



- Hash Segmentation:

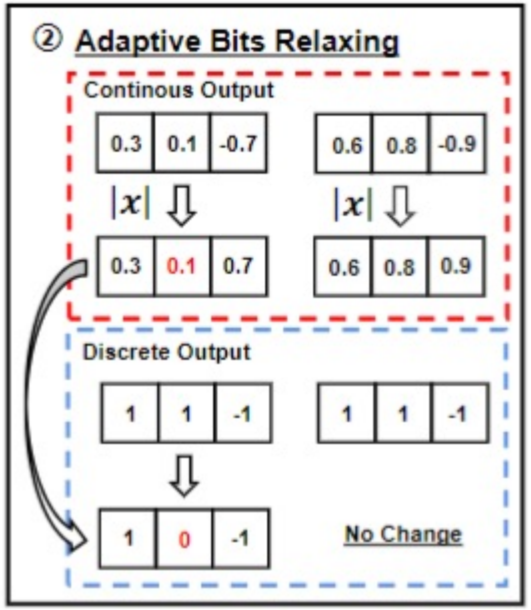
$$H_i = \{h_{i1}, \ldots, h_{ik}\}$$

- Value discretization:

$$h_{ij} = \mathrm{sgn}\left(o_{(i-1)*k+j}\right)$$

Adaptive bits relaxing

Adaptive bits relaxing: give up the prediction on the hash bits which are hard to align



- Select the hash bits with top k smallest absolute value:

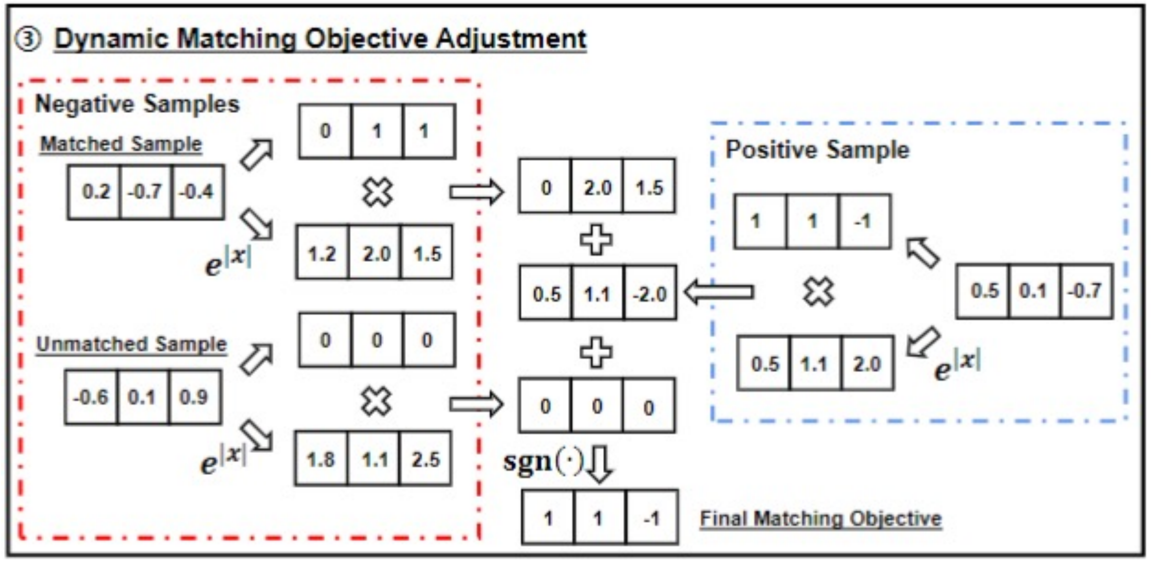
$$S_i = \{j \mid |o_{ij}| \text{ is top k smallest in } O_i\}$$

- Replace the initial hash value with 0 as the intermediate value:

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

Dynamic Matching Objective Adjustment

Dynamic matching objective adjustment: Assign a suitable hash code for each pair of code and query to address high false positive ratio



- Check whether the given hash code has hash collision with negative samples:

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

$$C_i = \min\{c_{i1}, \ldots, c_{ik}\}$$

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

- Determine the matching objective:

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

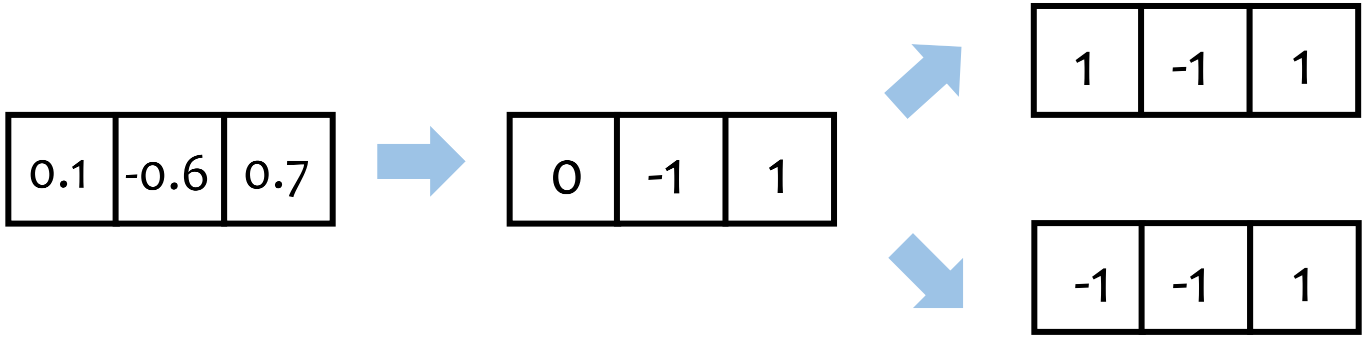
$$L_i = \{l_{i1}, \ldots, l_{ik}\}$$

Training Loss and Hash Code Inference

- Training Loss:

$$\mathcal{L}(\theta) = -\left(1 - \tilde{l}_{ij}\right) \cdot \log\left(1 - o_{ij}\right) - \left(1 + \tilde{l}_{ij}\right) \cdot \log\left(1 + o_{ij}\right)$$

- Inference of hash code:



Experiments

- Dataset:
  - CodeSearchNet (Python, Java): released by H. Husain et al.

- Baselines
  - Code retrieval model: CodeBERT, GraphCodeBERT
  - Conventional hashing approach: Locality-sensitive hashing
  - Deep hashing model: CoSHC, DJSRH, DSAH, JDSH

- Metrics:
  - R@1, MRR

Experiments

TABLE VIII
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
	CoSHC	31.9s	43.7s	66.4s	90.1s	137.7s	184.8s	280.1s	375.7s
	CoSHC$_{CSSDH}$	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
	DJSRH$_{CSSDH}$	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
	DSAH$_{CSSDH}$	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	44.0s	65.2s	90.6s	135.0s	185.7s	274.4s	368.6s
	JDSH$_{CSSDH}$	1.2s (↓96.1%)	1.5s (↓96.6%)	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
	CoSHC	31.9s	43.7s	66.5s	90.1s	137.6s	184.9s	280.0s	375.9s
	CoSHC$_{CSSDH}$	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
	DJSRH$_{CSSDH}$	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
	DSAH$_{CSSDH}$	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	43.8s	65.1s	90.3s	134.9s	185.6s	275.2s	376.3s
	JDSH$_{CSSDH}$	1.1s (↓96.5%)	1.5s (↓96.6%)	2.0s(↓96.9%)	2.6s (↓97.1%)	3.7s (↓97.3%)	4.9s (↓97.3%)	7.2s (↓97.4%)	9.1s (↓97.6%)

Experiments

TABLE IX

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

	Model	Python				Java			
		128bit		256bit		128bit		256bit	
		R@1	MRR	R@1	MRR	R@1	MRR	R@1	MRR
CodeBERT	Original	0.455	0.562	0.455	0.563	0.321	0.419	0.322	0.420
	LSH	0.390	0.460	0.438	0.531	0.264	0.330	0.302	0.386
	CoSHC	0.455	0.562	0.455	0.563	0.321	0.419	0.322	0.420
	$\text{CoSHC}_{\text{CSSDH}}$	0.447 (↓1.8%)	0.547 (↓2.7%)	0.452 (↓0.7%)	0.554 (↓1.6%)	0.316 (↓1.6%)	0.408 (↓2.6%)	0.319 (↓0.9%)	0.415 (↓1.2%)
	DJSRH	0.454	0.561	0.455	0.563	0.321	0.418	0.322	0.420
	$\text{DJSRH}_{\text{CSSDH}}$	0.446 (↓1.8%)	0.546 (↓2.7%)	0.451 (↓0.9%)	0.553 (↓1.8%)	0.316 (↓1.6%)	0.409 (↓2.2%)	0.319 (↓0.9%)	0.414 (↓1.4%)
	DSAH	0.450	0.552	0.451	0.554	0.317	0.411	0.319	0.414
	$\text{DSAH}_{\text{CSSDH}}$	0.447 (↓0.7%)	0.547 (↓0.9%)	0.452 (↑0.2%)	0.554 (0.0%)	0.316 (↓0.3%)	0.409 (↓0.5%)	0.319 (0.0%)	0.415 (↑0.2%)
	JDSH	0.448	0.549	0.450	0.552	0.317	0.410	0.318	0.412
	$\text{JDSH}_{\text{CSSDH}}$	0.447 (↓0.2%)	0.547 (↓0.4%)	0.452 (↑0.4%)	0.554 (↑0.4%)	0.316 (↓0.3%)	0.409 (↓0.2%)	0.319 (↑0.3%)	0.415 (↑0.7%)
GraphCodeBERT	Original	0.489	0.598	0.489	0.598	0.355	0.457	0.355	0.457
	LSH	0.412	0.480	0.471	0.565	0.279	0.340	0.334	0.420
	CoSHC	0.489	0.597	0.489	0.598	0.355	0.455	0.355	0.457
	$\text{CoSHC}_{\text{CSSDH}}$	0.479 (↓2.0%)	0.580 (↓2.8%)	0.484 (↓1.0%)	0.587 (↓1.8%)	0.348 (↓2.0%)	0.443 (↓2.6%)	0.353 (↓0.6%)	0.451 (↓1.3%)
	DJSRH	0.489	0.597	0.489	0.598	0.354	0.454	0.355	0.457
	$\text{DJSRH}_{\text{CSSDH}}$	0.479 (↓2.0%)	0.579 (↓3.0%)	0.482 (↓1.4%)	0.586 (↓2.0%)	0.348 (↓1.7%)	0.444 (↓2.2%)	0.353 (↓0.6%)	0.450 (↓1.5%)
	DSAH	0.482	0.586	0.484	0.589	0.351	0.447	0.352	0.449
	$\text{DSAH}_{\text{CSSDH}}$	0.480 (↓0.4%)	0.580 (↓1.0%)	0.484 (0.0%)	0.587 (↓0.3%)	0.349 (↓0.6%)	0.444 (↓0.7%)	0.353 (↑0.3%)	0.450 (↑0.2%)
	JDSH	0.482	0.585	0.483	0.587	0.350	0.446	0.351	0.448
	$\text{JDSH}_{\text{CSSDH}}$	0.478 (↓0.8%)	0.579 (↓1.0%)	0.483 (0.0%)	0.586 (↓0.2%)	0.349 (↓0.3%)	0.443 (↓0.7%)	0.353 (↑0.6%)	0.450 (↑0.4%)

Ablation Study

TABLE X
THE COMPARISONS AMONG THE SIX CSSDH VARIANTS WITH THE BASELINE OF CODEBERT.

Model	Python				Java			
	128bit		256bit		128bit		256bit	
	R@1	**MRR**	**R@1**	**MRR**	**R@1**	**MRR**	**R@1**	**MRR**
$\mathrm{CoSHC}_{\mathrm{NA\_NR}}$	0.270	0.312	0.334	0.390	0.214	0.263	0.251	0.313
$\mathrm{CoSHC}_{\mathrm{A\_NR}}$	0.383	0.459	0.410	0.493	0.267	0.338	0.290	0.370
$\mathrm{CoSHC}_{\mathrm{NA\_SR}}$	0.417	0.499	0.440	0.533	0.301	0.384	0.311	0.400
$\mathrm{CoSHC}_{\mathrm{A\_SR}}$	0.435	0.527	0.445	0.542	0.304	0.391	0.313	0.406
$\mathrm{CoSHC}_{\mathrm{NA\_BR}}$	0.445	0.543	0.451	**0.554**	0.315	0.405	**0.319**	0.414
$\mathbf{CoSHC}_{\mathbf{A\_BR}}$	**0.447**	**0.547**	**0.452**	**0.554**	**0.316**	**0.408**	**0.319**	**0.415**
$\mathrm{DJSRH}_{\mathrm{NA\_NR}}$	0.078	0.086	0.125	0.140	0.061	0.072	0.110	0.132
$\mathrm{DJSRH}_{\mathrm{A\_NR}}$	0.384	0.460	0.414	0.500	0.268	0.338	0.289	0.368
$\mathrm{DJSRH}_{\mathrm{NA\_SR}}$	0.250	0.289	0.319	0.375	0.183	0.226	0.249	0.312
$\mathrm{DJSRH}_{\mathrm{A\_SR}}$	0.432	0.524	0.445	0.541	0.305	0.392	0.313	0.404
$\mathrm{DJSRH}_{\mathrm{NA\_BR}}$	0.396	0.472	0.414	0.497	0.273	0.345	0.297	0.379
$\mathbf{DJSRH}_{\mathbf{A\_BR}}$	**0.446**	**0.546**	**0.451**	**0.553**	**0.316**	**0.409**	**0.319**	**0.414**
$\mathrm{DSAH}_{\mathrm{NA\_NR}}$	0.313	0.365	0.374	0.444	0.232	0.288	0.271	0.341
$\mathrm{DSAH}_{\mathrm{A\_NR}}$	0.388	0.466	0.417	0.502	0.268	0.339	0.292	0.371
$\mathrm{DSAH}_{\mathrm{NA\_SR}}$	0.421	0.509	0.438	0.533	0.299	0.383	0.310	0.398
$\mathrm{DSAH}_{\mathrm{A\_SR}}$	0.436	0.529	0.443	0.540	0.305	0.392	0.314	0.405
$\mathrm{DSAH}_{\mathrm{NA\_BR}}$	0.441	0.537	0.449	0.550	0.312	0.402	0.317	0.410
$\mathbf{DSAH}_{\mathbf{A\_BR}}$	**0.447**	**0.547**	**0.452**	**0.554**	**0.316**	**0.409**	**0.319**	**0.415**
$\mathrm{JDSH}_{\mathrm{NA\_NR}}$	0.326	0.384	0.384	0.459	0.246	0.307	0.280	0.354
$\mathrm{JDSH}_{\mathrm{A\_NR}}$	0.388	0.465	0.416	0.502	0.269	0.340	0.290	0.370
$\mathrm{JDSH}_{\mathrm{NA\_SR}}$	0.425	0.513	0.438	0.533	0.304	0.388	0.311	0.400
$\mathrm{JDSH}_{\mathrm{A\_SR}}$	0.436	0.529	0.445	0.543	0.308	0.396	0.314	0.405
$\mathrm{JDSH}_{\mathrm{NA\_BR}}$	0.441	0.537	0.448	0.549	0.314	0.404	0.318	0.411
$\mathbf{JDSH}_{\mathbf{A\_BR}}$	**0.447**	**0.547**	**0.452**	**0.554**	**0.316**	**0.409**	**0.319**	**0.415**

Summary

- Propose a novel approach which firstly convert the long hash code into segmented hash codes
- Propose the dynamic matching objective adjustment strategy to reduce the false positive hash collision ratio
- Propose the adaptive bit relaxing strategy to increase the hash collision ratio
- The experiments demonstrate its great ability to reduce recall calculation cost while perserve most performance

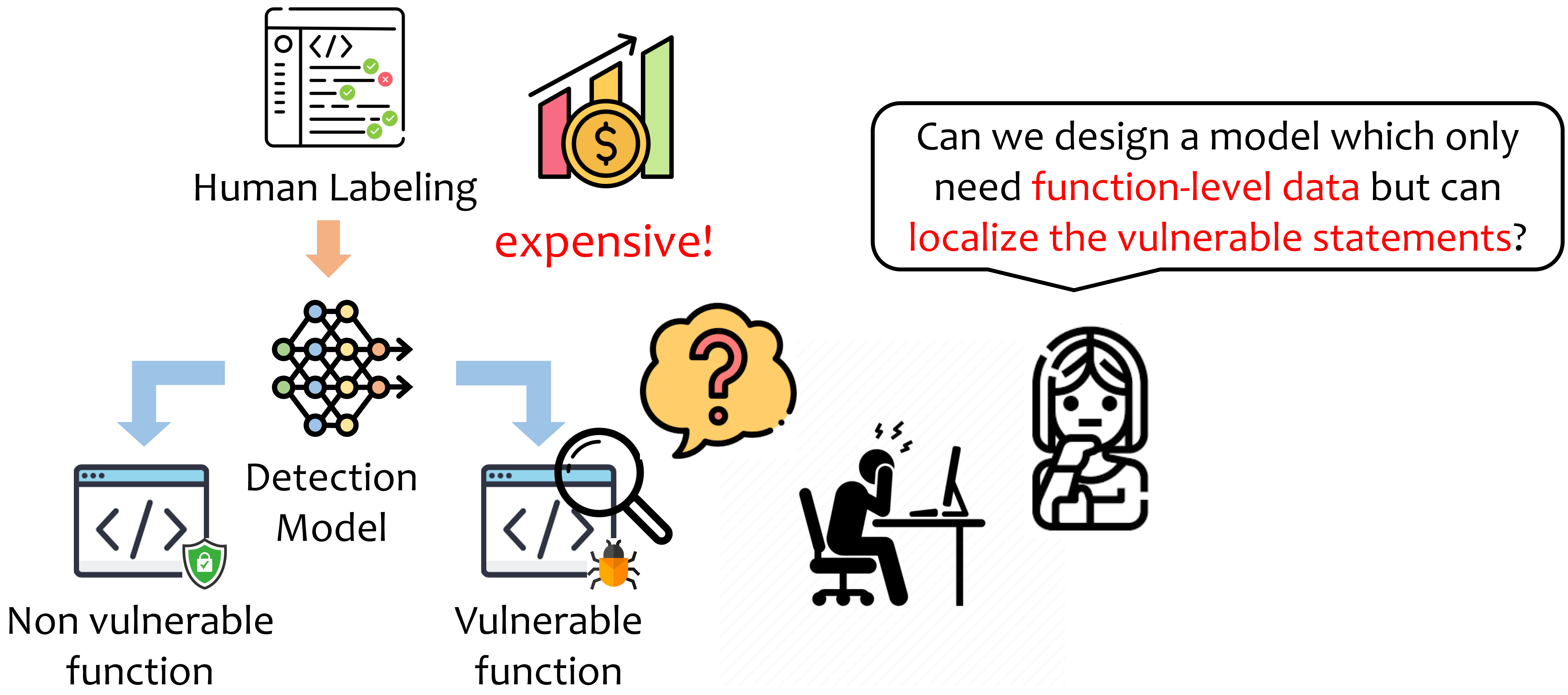
Thesis Structure

- Intelligent Program Development
  - Code Retrieval
    - Effective Code Retrieval
      - Semantic Dependency Learning Based Code Retrival
    - Code Retrieval Acceleartion
      - Code Retrival Acceleration with Deep Hashing
      - Code Retrival Acceleration via Segmented Deep Hashing
  - Vulnearability Detection
    - Vulnerability Localization via Multiple Instance Learning

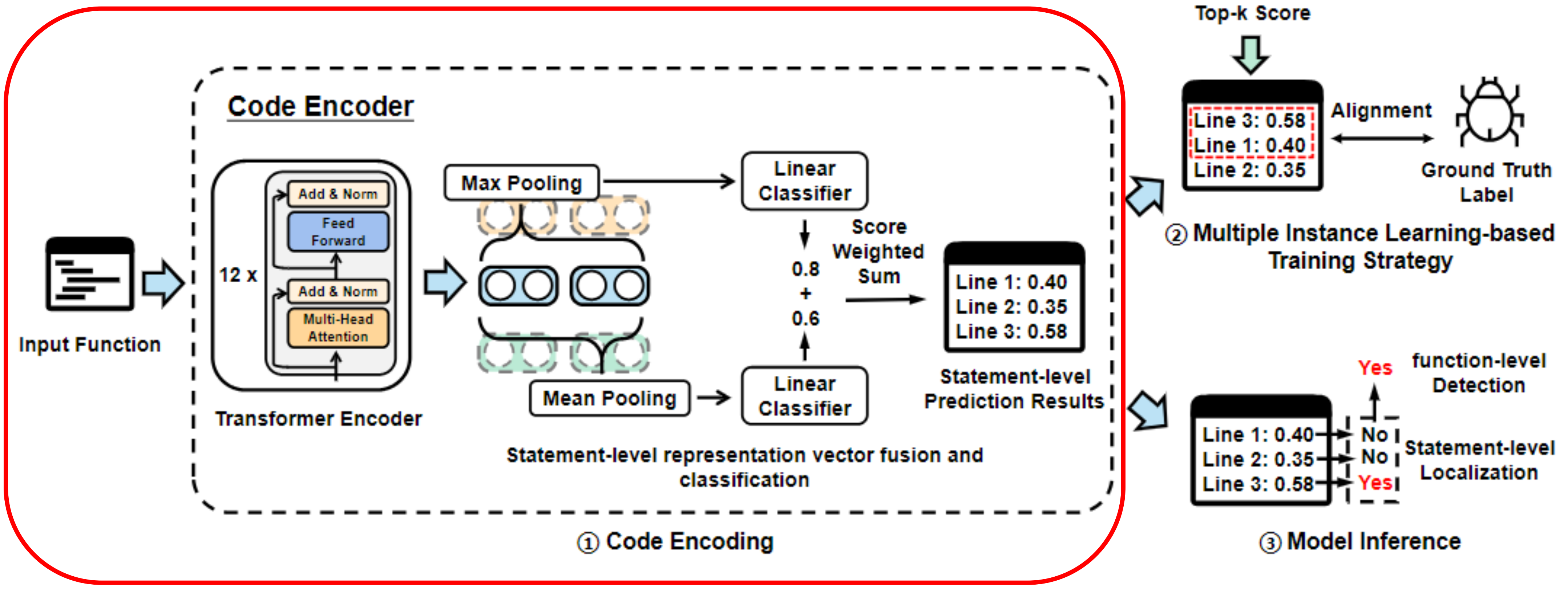
4

Vulnerability Localization via Multiple Instance Learning

Motivation

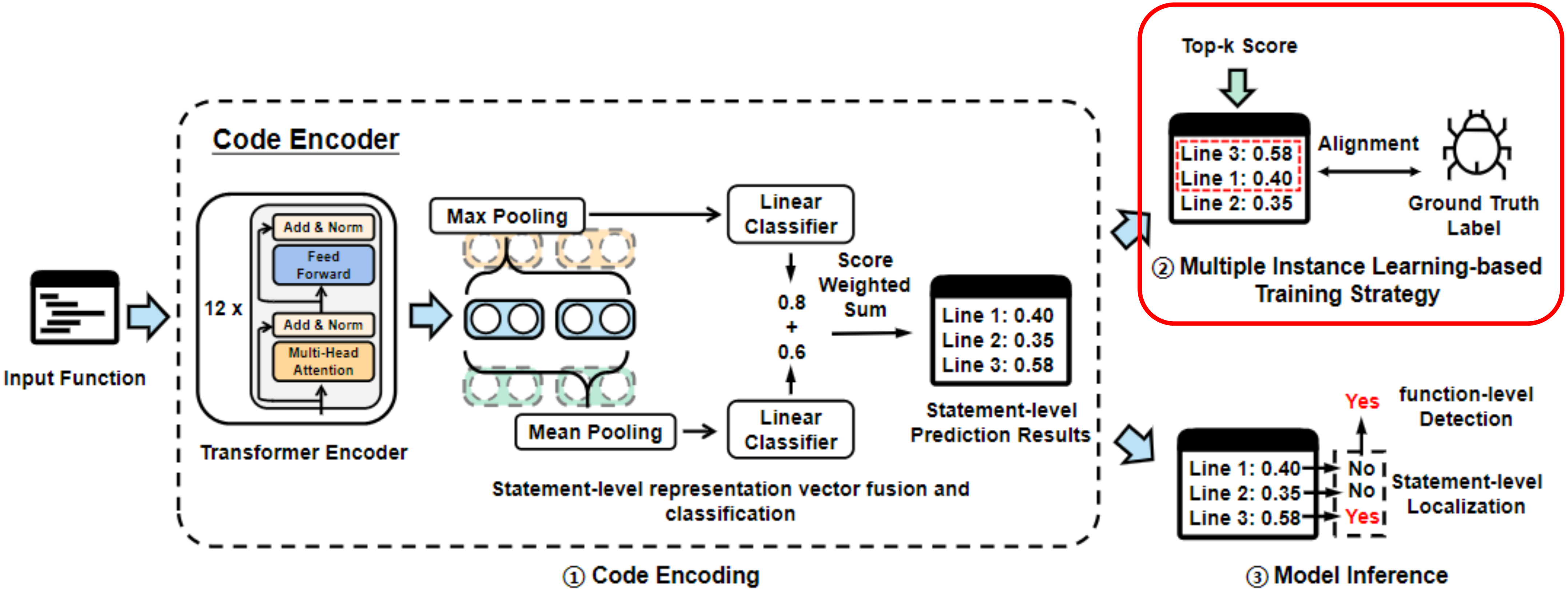

Human Labeling
expensive!
Detection Model
Non vulnerable function
Vulnerable function
Can we design a model which only need function-level data but can localize the vulnerable statements?

Overview



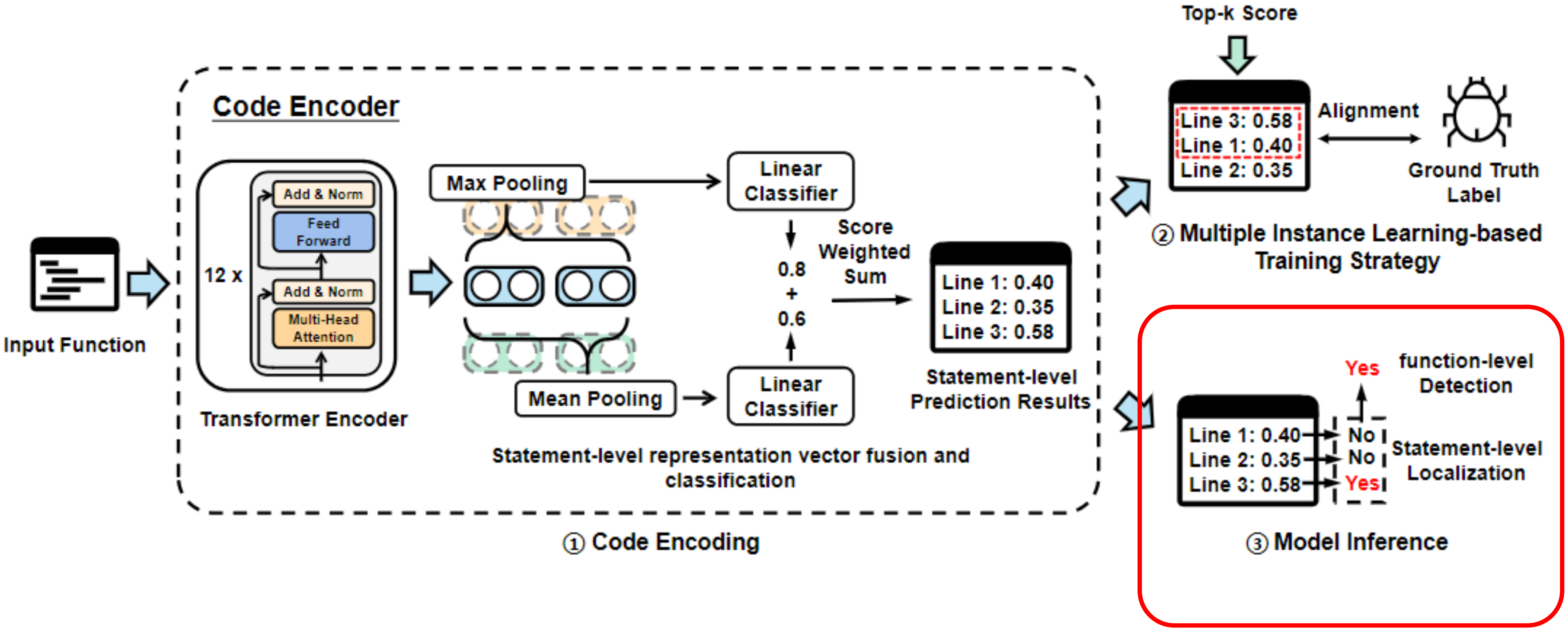
Code Encoder
Input Function
12 x
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Transformer Encoder
Max Pooling
Mean Pooling
Linear Classifier
Linear Classifier
Score Weighted Sum
0.8
+
0.6
Line 1: 0.40
Line 2: 0.35
Line 3: 0.58
Statement-level Prediction Results
Statement-level representation vector fusion and classification
① Code Encoding
Top-k Score
Line 3: 0.58
Line 1: 0.40
Line 2: 0.35
Alignment
Ground Truth Label
② Multiple Instance Learning-based Training Strategy
Yes
function-level Detection
Line 1: 0.40
Line 2: 0.35
Line 3: 0.58
No
No
Yes
Statement-level Localization
③ Model Inference

Overview



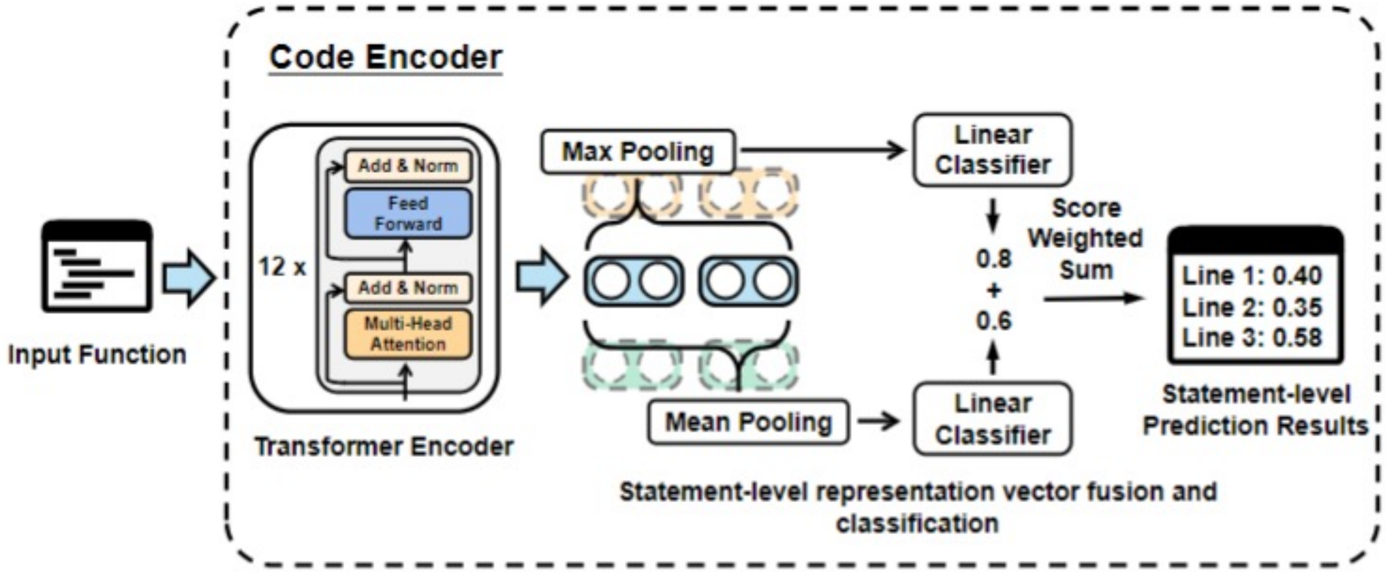
Code Encoder
12 x
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Transformer Encoder
Input Function
Max Pooling
Mean Pooling
Linear Classifier
Linear Classifier
0.8
+
0.6
Score Weighted Sum
Line 1: 0.40
Line 2: 0.35
Line 3: 0.58
Statement-level Prediction Results
Statement-level representation vector fusion and classification
① Code Encoding
Top-k Score
Line 3: 0.58
Line 1: 0.40
Line 2: 0.35
Alignment
Ground Truth Label
② Multiple Instance Learning-based Training Strategy
Yes
function-level Detection
Line 1: 0.40
Line 2: 0.35
Line 3: 0.58
No
No
Yes
Statement-level Localization
③ Model Inference

Overview



Code Encoder
Input Function
12 x
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Transformer Encoder
Max Pooling
Mean Pooling
Linear Classifier
Linear Classifier
0.8
+
0.6
Score Weighted Sum
Line 1: 0.40
Line 2: 0.35
Line 3: 0.58
Statement-level Prediction Results
Statement-level representation vector fusion and classification
① Code Encoding
Top-k Score
Line 3: 0.58
Line 1: 0.40
Line 2: 0.35
Alignment
Ground Truth Label
② Multiple Instance Learning-based Training Strategy
Yes
function-level Detection
Line 1: 0.40
Line 2: 0.35
Line 3: 0.58
No
No
Yes
Statement-level Localization
③ Model Inference

Model Design



① Code Encoding

- Feed the target function into Transformer-based encoder
- Adopt max pooling and mean pooling to generate the statement-level vector

Mean Pooling

- Mean Pooling is suitable for the detection of some vulnerabilities like overflow

```
// Variable for sales revenue for the quarter
float quarterRevenue = 0.0f;

short JanSold = getMonthlySales(JAN); /* Get sales in January */
short FebSold = getMonthlySales(FEB); /*
short MarSold = getMonthlySales(MAR); /*

// Calculate quarterly total
short quarterSold = JanSold + FebSold + MarSold;
```

Exists a potential risk of integer overflow if the sum exceeds the maximum value allowed for the short int primitive type.

Max Pooling

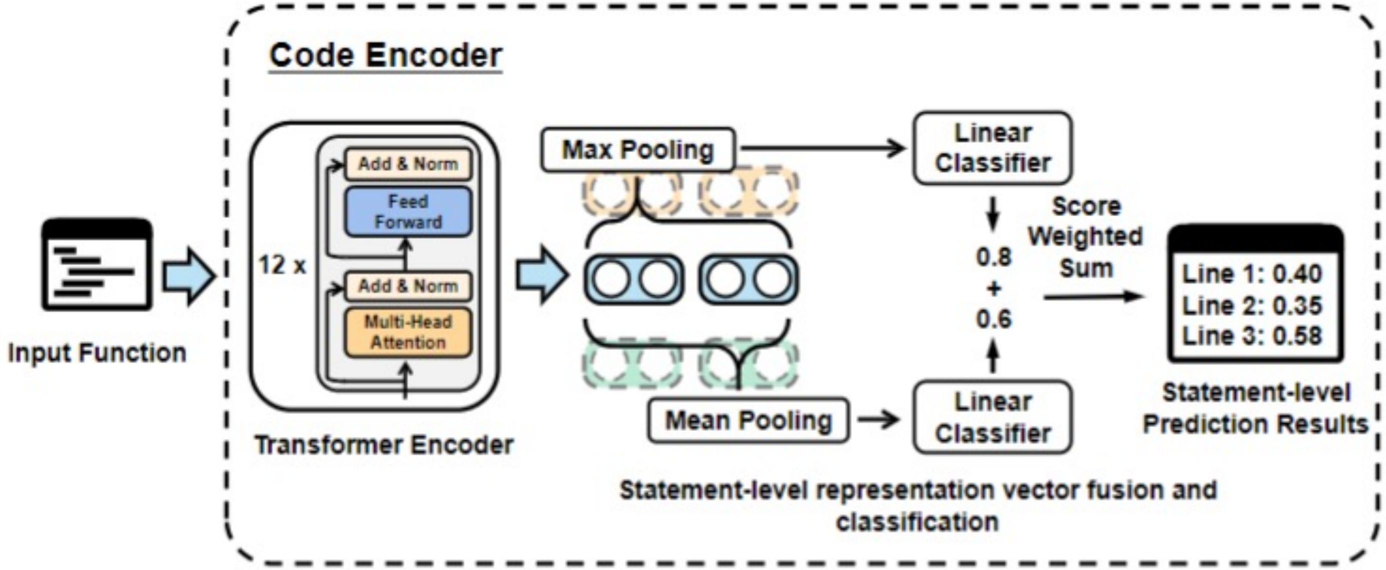
- Max Pooling is suitable for the detection of the vulnerabilities arise from variable or API misuse

```
// copy input string to a temporary string
char message[length+1];
int index;
for (index = 0; index < length; index++) {
      message[index] = strMessage[index];
}
message[index] = '\0';

// trim trailing whitespace
int len = index-1;
while (isspace(message[len])) {
    message[len] = '\0';
    len--;
}
```

Invoking the isspace() API on an address outside the limits of the local buffer if the input consists solely of whitespaces

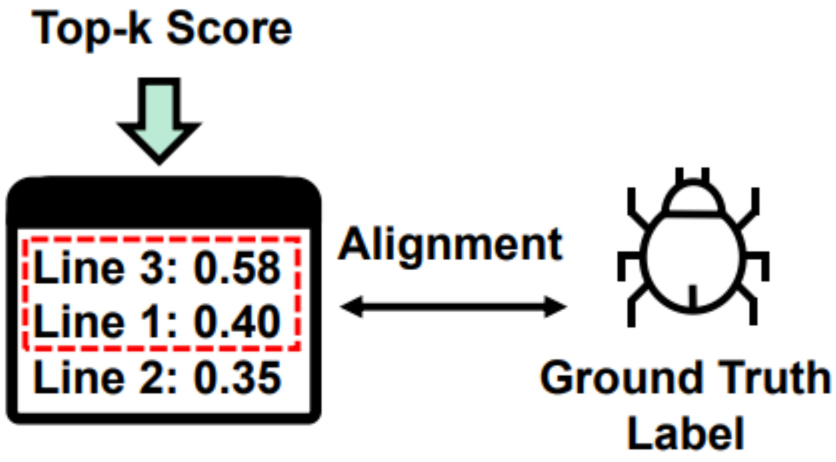
Model Design



① Code Encoding

- Feed the target function into Transformer-based encoder
- Adopt max pooling and mean pooling to generate the statement-level vector
- Adopt two linear classifier to output the statement-level prediction score and weighted sum of two scores

Training Strategy



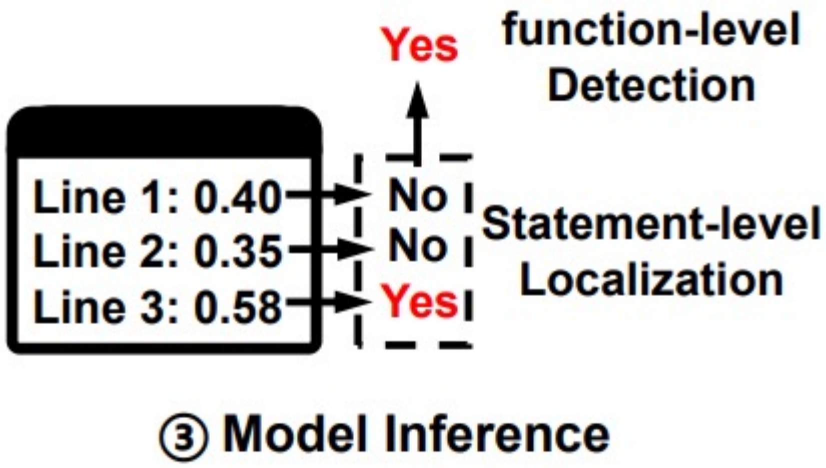
② Multiple Instance Learning-based Training Strategy

- Sort the statement in a desending order of prediction score

- Assign the function-level label as the pseudo statement level label for the top-k statements:

$$\mathcal{L}(\theta) = \frac{1}{N \cdot k} \sum_{i} \sum_{j}^{k} -\left[Y_i \log(p_{ij}) + (1 - Y_i) \log(1 - p_{ij})\right]$$

Model Inference



- Function-level vulnerability detection:

$$Y = \max\{y_1, \ldots, y_n\}$$

- Statement-level vulnerability localization:
  - Return the statement predicted as yes
  - Ruturn the top-k statement

Experiments

- Dataset:
  - FFMPeg+Qemu: released by Yaqin Zhou et al. in NeurIPS 2019
  - Reveal: released by Saikat Chakraborty et al. in TSE
  - Fan et al.: released by Jiahao Fan et al. in MSR 2020
- Baselines
  - VulDeePecker, SySeVR, Devign, Reveal, IVDetect, LineVul
- Metrics:
  - Accuracy (Acc), Precision (P), Recall (R), F1, Top-1, Top-3, Top-5, Mean First Ranking (MFR), Mean Average Ranking (MAR), Initial False Alarm (IFA)

Experiments

Our proposed method can achieve the comparable performance on the function-level vulnerability detection

TABLE XI
COMPARISON RESULTS ON FUNCTION-LEVEL VULNERABILITY. THE BEST RESULTS ARE HIGHLIGHTED IN **BOLD** FONT.

Model	Fan *et al.*				Reveal				FFMPeg+Qemu			
	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1
VulDeePecker	0.913	0.155	0.146	0.150	0.763	0.211	0.131	0.162	0.496	0.461	0.326	0.381
SySeVR	0.904	0.129	0.194	0.155	0.743	0.401	0.249	0.307	0.479	0.461	0.588	0.517
Devign	0.957	0.257	0.143	0.184	0.875	0.316	0.367	0.339	0.569	0.525	0.647	0.580
Reveal	0.928	0.270	**0.661**	0.383	0.818	0.316	**0.611**	0.416	**0.611**	**0.555**	0.707	0.622
IVDetect	0.696	0.073	0.600	0.130	0.808	0.276	0.556	0.369	0.573	0.524	0.576	0.548
LineVul	0.972	0.632	0.436	0.516	0.847	0.248	0.519	0.335	0.541	0.496	**0.909**	**0.642**
WIDLE	**0.977**	**0.724**	0.522	**0.607**	**0.922**	**0.471**	0.394	**0.429**	0.589	0.530	0.812	0.641

Experiments

Our proposed method can achieve the state-of-the-art performance on the statement-level vulnearbility localization

TABLE XII
COMPARISON RESULTS ON FUNCTION-LEVEL VULNERABILITY AND STATEMENT-LEVEL VULNERABILITY LOCALIZATION. THE BEST RESULTS ARE HIGHLIGHTED IN **BOLD** FONT.

Model	Acc	P	R	F1	MAR	MFR	IFA	Top-1	Top-3	Top-5
LineVul	N/A	N/A	N/A	N/A	9.49	7.17	6.17	0.005	0.252	0.375
WIDLE	0.983	0.183	0.338	0.237	**9.08**	**6.46**	**5.46**	**0.283**	**0.484**	**0.609**

Ablation Study

The combination of two pooling channels can enhances the ability of both function-level detection and statement-level localization

TABLE XIII
RESULTS OF THE FUNCTION-LEVEL VULNERABILITY DETECTION PERFORMANCE COMPARISON WITH DIFFERENT CHANNELS. THE BEST RESULTS ARE HIGHLIGHTED IN **BOLD** FONT.

Model	Fan et al.				Reveal				FFMPeg+Qemu			
	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1
$\text{WIDLE}_{\text{max}}$	0.976	0.721	0.500	0.591	**0.933**	**0.574**	0.375	**0.453**	0.567	0.513	**0.816**	0.630
$\text{WIDLE}_{\text{mean}}$	0.973	0.660	0.446	0.532	0.922	0.470	0.385	0.423	0.560	0.509	0.752	0.608
WIDLE	**0.977**	**0.724**	**0.522**	**0.607**	0.922	0.471	**0.394**	0.429	**0.589**	**0.530**	0.812	**0.641**

TABLE XIV
RESULTS OF THE STATEMENT-LEVEL VULNERABILITY LOCALIZATION PERFORMANCE COMPARISON WITH DIFFERENT CHANNELS. THE BEST RESULTS ARE HIGHLIGHTED IN **BOLD** FONT.

Model	Acc	P	R	F1	MAR	MFR	IFA	Top-1	Top-3	Top-5
$\text{WIDLE}_{\text{max}}$	**0.984**	**0.188**	0.299	0.231	9.17	6.66	5.66	0.268	0.481	**0.617**
$\text{WIDLE}_{\text{mean}}$	0.977	0.155	**0.416**	0.226	9.78	7.27	6.27	0.224	0.443	0.586
WIDLE	0.983	0.183	0.338	**0.237**	**9.08**	**6.46**	**5.46**	**0.283**	**0.484**	0.609

Experiments

Our method has comparable detection abilities for most types of vulnerabilities but has better performance in Integer overflow and double free

TABLE XV
DETECTION RESULTS FOR DIFFERENT CWE VULNERABILITIES WITH OUR PROPOSED WIDLE.

CWE-ID	Description	Rank	TPR	Proportion
CWE-787	Out-of-bounds Write	1	50.0%	7/14
CWE-416	Use After Free	4	61.5%	8/13
CWE-20	Improper Input Validation	6	56.6%	61/108
CWE-125	Out-of-bounds Read	7	54.2%	13/24
CWE-476	NULL Pointer Dereference	12	55.6%	5/9
CWE-190	Integer Overflow or Wraparound	14	77.8%	14/18
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	17	56.0%	70/125
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	21	52.4%	11/21
CWE-284	Improper Access Control	N/A	37.5%	3/8
CWE-189	Numeric Errors	N/A	52.6%	10/19
CWE-732	Incorrect Permission Assignment for Critical Resource	N/A	57.1%	4/7
CWE-254	7PK - Security Features	N/A	44.4%	4/9
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	N/A	42.9%	12/28
CWE-415	Double Free	N/A	71.4%	5/7
CWE-399	Resource Management Errors	N/A	48.7%	19/39
Total			54.8%	246/449

Summary

- Propose a novel approach which can localize vulnerabilities without additional labeling
- Integrate various pooling modules to capture code features for different types vulnerabilities
- The experiment results domenstrate its superior performance on both function-level and statement-level

CONTENTS

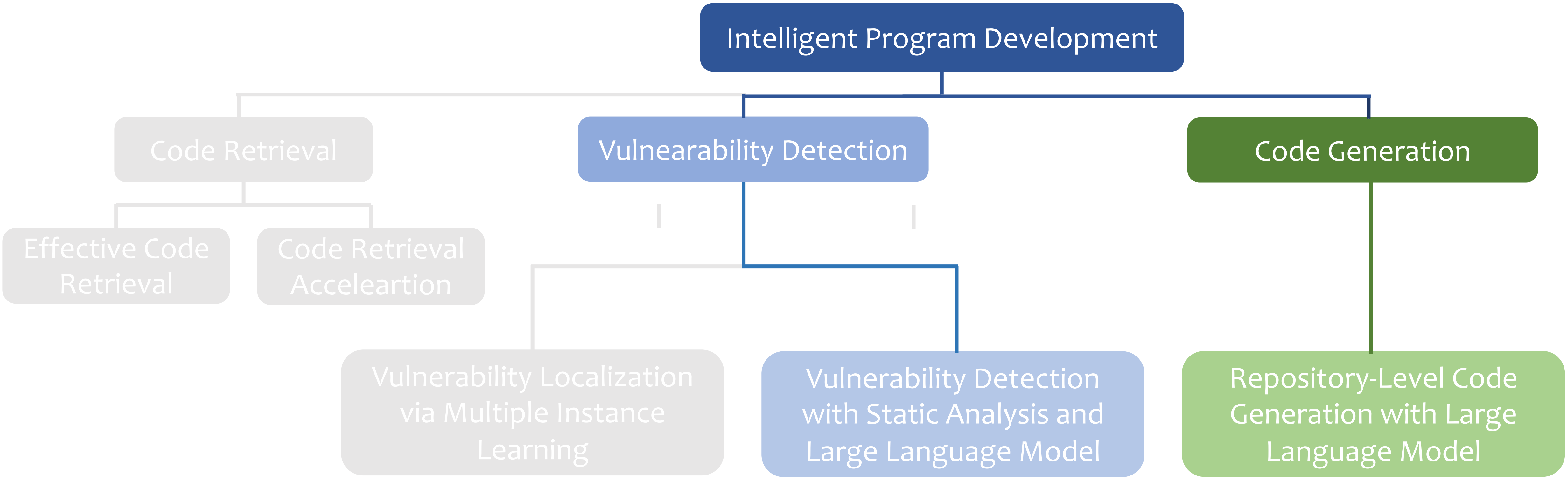
5

Conclusion and Future Work

Conclusion

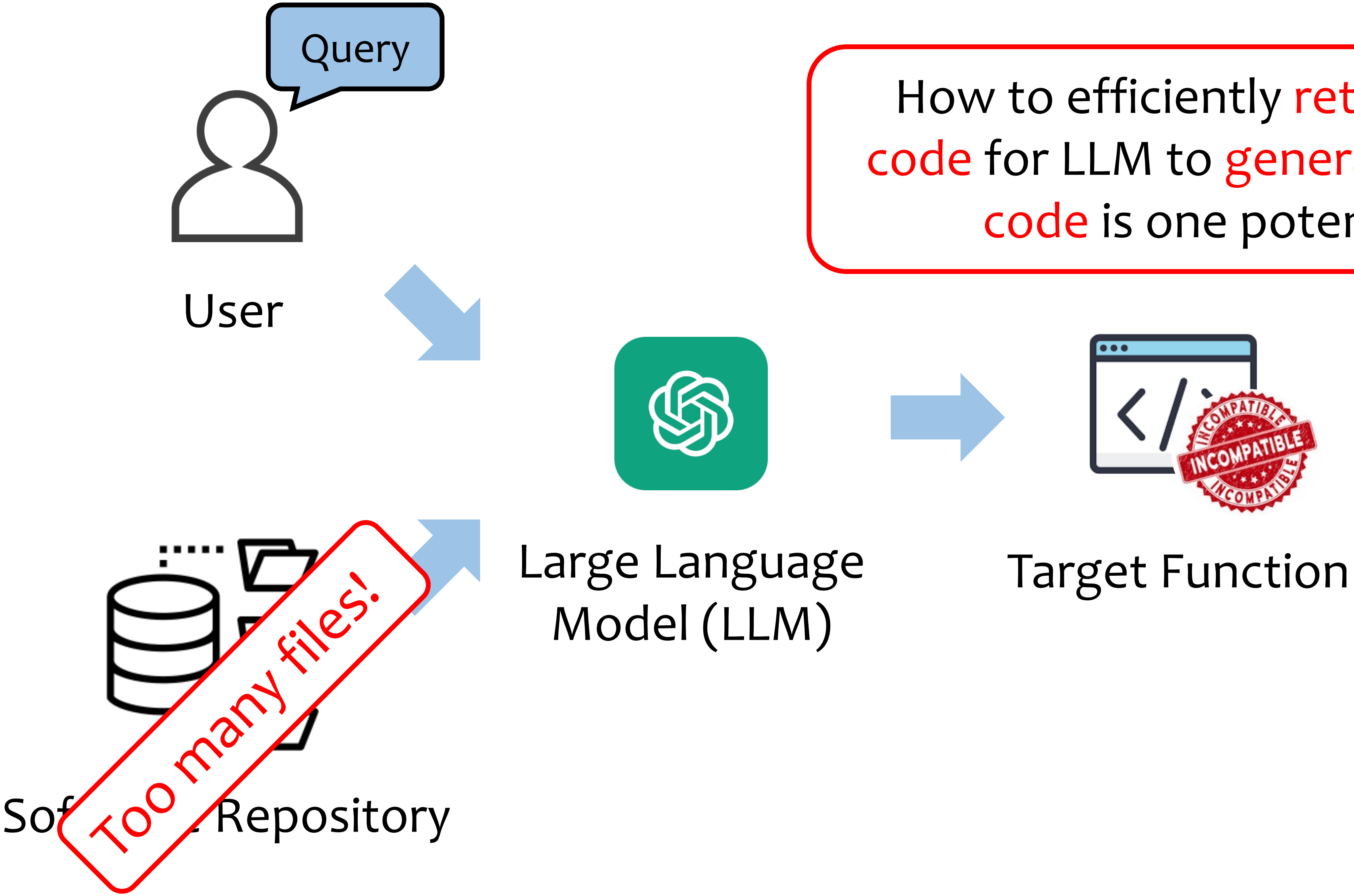
- Intelligent Program Development
  - Effective Code Retrieval
    - Semantic Dependency Learning Based Code Retrival
      - Integrate the dependency and semantics information for code retrieval
      - Evaluate the effectivenss of proposed method
  - Code Retrieval Acceleartion
    - Code Retrival Acceleration with Deep Hashing
      - Integrate code classification and deep hashing
      - Evaluate the effectivness and efficiency of proposed method
    - Code Retrival Acceleration via Segmented Deep Hashing
      - Convert the long hash code into segmented hash codes to improve retrieval efficiency
      - Propose strategies to increase the hash collision ratio and reduce the false positive ratio
  - Vulnearability Detection
    - Vulnerability Localization via Multiple Instance Learning
      - Propose a weakly supervised method for vulnerability detection and localization
      - Integrate various pooling modules for different types of vulnerabilities

Future Work

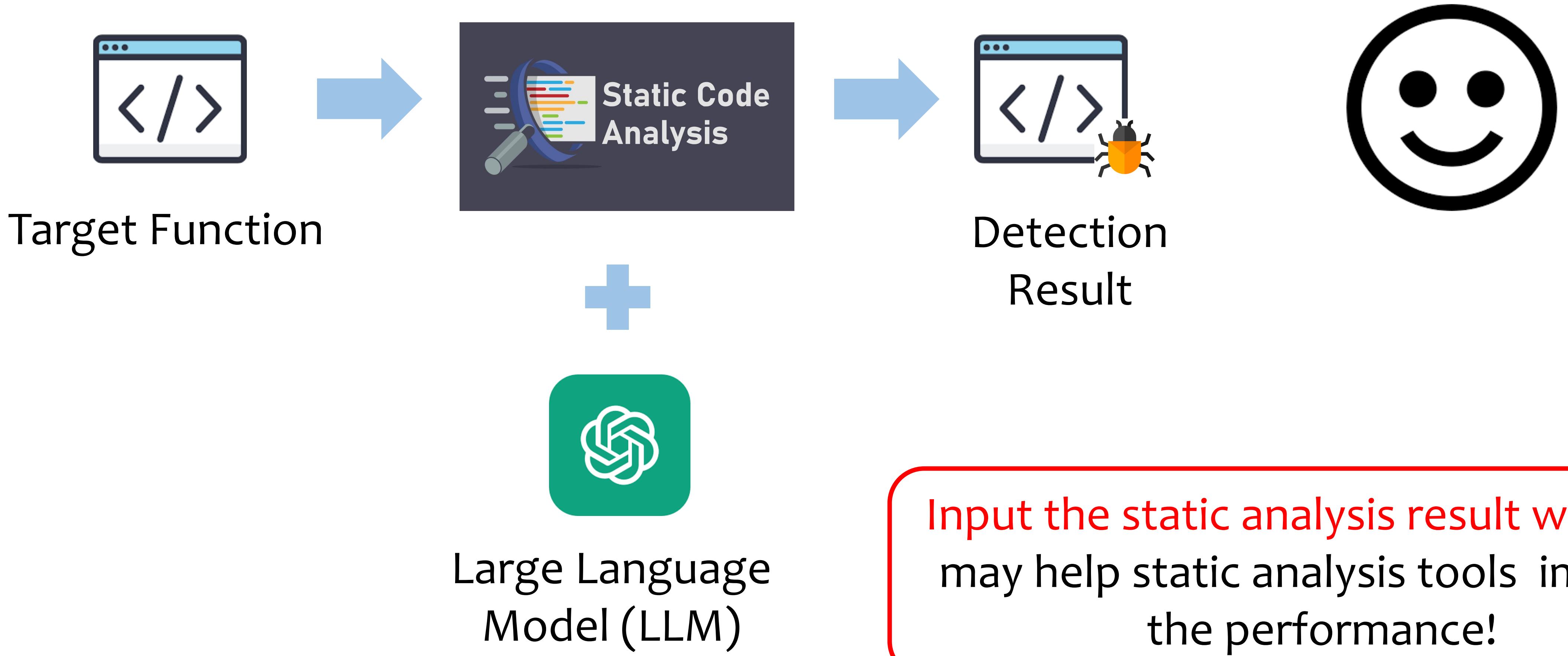

Intelligent Program Development
Code Retrieval
Vulnearability Detection
Code Generation
Effective Code Retrieval
Code Retrieval Acceleartion
Vulnerability Localization via Multiple Instance Learning
Vulnerability Detection with Static Analysis and Large Language Model
Repository-Level Code Generation with Large Language Model

Future Work

How to efficiently retreived the related code for LLM to generate repository-level code is one potential direction!



Future Work



Input the static analysis result with LLM may help static analysis tools improve the performance!

Publications

- **Wenchao Gu**, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, Michael R. Lyu, "Accelerating code search with deep hashing and code classification", 60th Annual Meeting of the Association for Computational Linguistics
- **Wenchao Gu**, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, Michael R. Lyu , "CRaDLe: Deep code retrieval based on semantic dependency learning", Neural Networks
- Weizhe Zhang*, **Wenchao Gu***, Cuiyun Gao, Michael R. Lyu, "A Transformer-based Approach for Improving App Review Response Generation", Software: Practice and Experience, 2023
- Ensheng Shi, Yanlin Wang, **Wenchao Gu**, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, Hongbin Sun, "Enhancing Semantic Code Search with Multimodal Contrastive Learning and Soft Data Augmentation", International Conference on Software Engineering 2022
- Zi Gong, Cuiyun Gao, Yasheng Wang, **Wenchao Gu**, Yun Peng, Zenglin Xu, "Source Code Summarization with Structural Relative Position Guided Transformer", 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering
- **Wenchao Gu**, Zongyi Lyu, Yanlin Wang, Hongyu Zhang, Cuiyun Gao, Michael R. Lyu, "SPENCER: Self-Adaptive Model Distillation for Efficient Code Retrieval", IEEE Transactions on Software Engineering (major revision)
- **Wenchao Gu**, yupan Chen, Yanlin Wang, Hongyu Zhang, Cuiyun Gao, Michael R. Lyu, "Weakly Supverised Vulnerability Detection and Localization via Multiple Instance Learning", ACM Transactions on Software Engineering Methodology (under review)
- **Wenchao Gu**, Ensheng Shi, Yanlin Wang, Lun Du, Shi Han, Hongyu Zhang, Meidong Zhang, Michael R. Lyu, "Accelerating Code Search via Segmented Deep Hashing", 40th IEEE International Conference on Data Engineering (under review)

Thank you!

ARISE
Automated Reliable Intelligent Software Engineering

香港中文大學
The Chinese University of Hong Kong

Error Analysis

Query: Convert directly the matrix from Cartesian coordinates (the origin in the middle of image) to Image coordinates (the origin on the top-left of image)

```
def transform_matrix_offset_center(matrix, y, x):
    o_x = (x - 1) / 2.0
    o_y = (y - 1) / 2.0
    offset_matrix = np.array([[1, 0, o_x], [0, 1, o_y],
    ↪   [0, 0, 1]])
    reset_matrix = np.array([[1, 0, -o_x], [0, 1, -o_y],
    ↪   [0, 0, 1]])
    transform_matrix = np.dot(np.dot(offset_matrix,
    ↪   matrix), reset_matrix)
    return transform_matrix
```

Query: Get successor to key, raises KeyError if a key is max key or key does not exist

```
def succ_key(self, key, default=_sentinel):
        item = self.succ_item(key, default)
        return default if item is default else item[0]
```