

# AI for Program Analysis Tasks

Wei Le

November 3, 2023

# From Big code to AI models: a "fast" history

- ▶ *big code*: billions of tokens of code and millions of instances of data from open source software systems: such as Linux, MYSQL, Django, Ant and OpenEJB, including code, changes, commits
- ▶ data-driven software engineering: combining machine learning and statistics
  - ▶ calculate statistical distributional properties
  - ▶ build probabilistic models of source code
  - ▶ code representation learning
- ▶ industry and academia interests:
  - ▶ DARPA: Mining and Understanding Software Enclaves (MUSE)
  - ▶ Startup: Deep Code - automatically write code
  - ▶ Microsoft and ABB: using data to drive software development decisions

# From Big code to AI models: a "fast" history

AI models:

- ▶ DNN: deep neural networks
- ▶ CNN: convolution neural networks (mostly for images)
- ▶ RNN: recurrent neural networks (mostly for natural languages)
- ▶ GAN: generative adversarial networks

People still use the following a lot for programming analysis tasks:

- ▶ GNN: graph neural networks
- ▶ LLM: large language models
  - ▶ transformer architecture
  - ▶ encoder + classification header
  - ▶ encoder + decoder
- ▶ LLM-extension: toolformer

# Applications: recommender systems/code generation in software engineering

Based on the language model built from copra and based on the current context of code, comments ...

- ▶ code completion: API sequence, next few lines
- ▶ synthesize the code for repair
- ▶ comment completion
- ▶ improve the names of methods

Challenge: the code generated may not be correct

## **GitHub Copilot**

# Applications: translations in software engineering

- ▶ Java programs to C and vice versa
  - ▶ Java to C# is easier
  - ▶ Challenges: mapping of API calls
- ▶ code to text (code summarization)/pseudo code
- ▶ text to code: automatically writing code

# Applications: code review and vulnerability detection

- ▶ Detecting which functions are vulnerable or buggy
- ▶ Detecting which lines are responsible for root causes
- ▶ Many datasets prepared: Devign, Reveal, D2A, MSR, ...
- ▶ Graph models and transformer models

Challenge: low accuracy 60-70 F1 score

**Amazon, CodeGuru**

# Applications: mining patterns

- ▶ loop idioms
- ▶ code convention, anomaly code
- ▶ code clone and repetitive code

# Applications: information retrieval and code understanding

- ▶ code search: given a natural language query, can you find the code snippets of interest
- ▶ semantic classification: classify software based on their functionalities

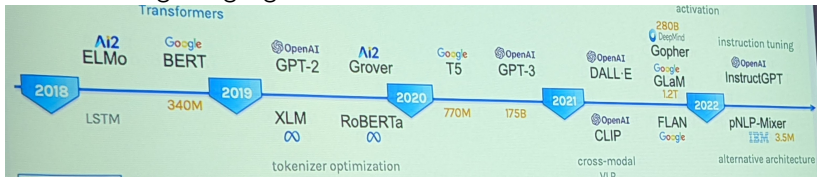


# Code Representation Learning, Source Code Modeling

- ▶ *Program embedding*: represent programs in vectors, each method/code snippet turns into a fixed-length vector
- ▶ Goal: enable neural network to solve problems for programs
- ▶ Input:
  - ▶ source code (code2seq, code2vec)
  - ▶ traces (DYPORO)
  - ▶ graph representations of code such as AST, CFG and PDG (Devign)
  - ▶ a mixture of code and traces (TRACED)
  - ▶ a mixture of source code and natural language
  - ▶ the output from lightweight code analysis
- ▶ Output: vector representation
- ▶ Approaches: supervised (task specific), self-supervised (general purpose)

# CodeBERT

- ▶ Reference paper: CodeBERT: A Pre-Trained Model for Programming and Natural Languages
- ▶ Pretrained large-language models



- ▶ CodeBERT, GraphCodeBERT, UnixCoder, CodeT5

# CodeBERT pre-training

Train the model to understand of mapping between code to natural language

```
def _parse_memory(s):  
    """  
    Parse a memory string in the format supported by Java (e.g. 1g, 200m) and  
    return the value in MiB  
  
    >>> _parse_memory("256m")  
    256  
    >>> _parse_memory("2g")  
    2048  
    """  
  
    units = {'g': 1024, 'm': 1, 't': 1 << 20, 'k': 1.0 / 1024}  
    if s[-1].lower() not in units:  
        raise ValueError("invalid format: " + s)  
    return int(float(s[:-1]) * units[s[-1].lower()])
```

Figure 1: An example of the NL-PL pair, where NL is the first paragraph (filled in red) from the documentation (dashed line in black) of a function.

# CodeBERT pre-training

- ▶ self-supervised pretraining: *MLM (masked language modeling)*, *RTD (replaced token detection)*
  - ▶ MLM: randomly select positions for masking PL or NL tokens
  - ▶ RTD:
    - ▶ an NL generator and PL generator, both for generating plausible alternatives for the set of randomly masked positions
    - ▶ The discriminator is trained to determine whether a word is the original one or not, which is a binary classification problem.
- ▶ CodeBERT trained on Java, Python and many more

# CodeBERT for software engineering tasks

- ▶ Load the model with pretrained weights
- ▶ Supervised training for the down stream tasks, e.g., code clone detection, code classification, vulnerability detection

# CodeBERT results

MODEL	RUBY	JAVASCRIPT	GO	PYTHON	JAVA	PHP	OVERALL
SEQ2SEQ	9.64	10.21	13.98	15.93	15.09	21.08	14.32
TRANSFORMER	11.18	11.59	16.38	15.81	16.26	22.12	15.56
ROBERTA	11.17	11.90	17.72	18.14	16.47	24.02	16.57
PRE-TRAIN W/ CODE ONLY	11.91	13.99	17.78	18.58	17.50	24.34	17.35
CODEBERT (RTD)	11.42	13.27	17.53	18.29	17.35	24.10	17.00
CODEBERT (MLM)	11.57	14.41	17.78	18.77	17.38	24.85	17.46
CODEBERT (RTD+MLM)	<b>12.16</b>	<b>14.90</b>	<b>18.07</b>	<b>19.06</b>	<b>17.65</b>	<b>25.16</b>	<b>17.83</b>

Table 4: Results on Code-to-Documentation generation, evaluated with smoothed BLEU-4 score.

# Design: training data

Vulnerability detection: a binary classification problem

- ▶ training dataset: ffmpeg, wireshark, linux, QEMU
- ▶ CVE and manual labeling (600 manual hour)
- ▶ 50% vulnerable, 50% not vulnerable data points
- ▶ label whether a function is vulnerable

# Design: code representation

## Code Property Graph: CFG + AST + Dependency graph

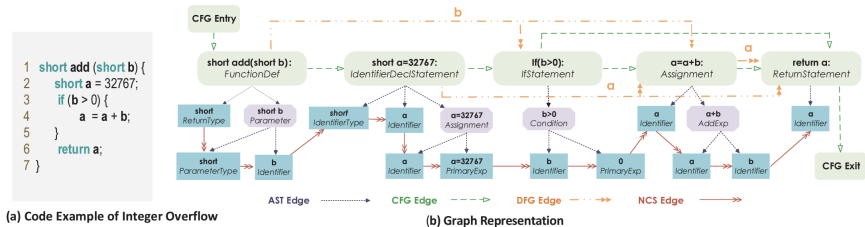


Figure 2: Graph Representation of Code Snippet with Integer Overflow



# Design: Graph Learning Architecture

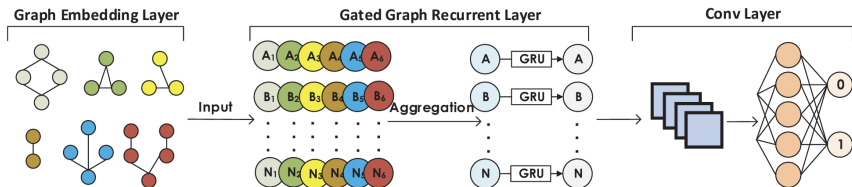


Figure 1: The Architecture of *Design*

1. Graph embedding: every node  $v$  has two attributes, Code and Type
  - ▶ Code contains the source code represented by  $v$  (pre-trained word2vec)
  - ▶ The type of  $v$  denotes the type attribute
2. Gated Graph Recurrent Layers: aggregate numbers information
3. Conv Layer: graph readout function

# Devign: Results

Method	Linux Kernel		QEMU		Wireshark		FFmpeg		Combined		Max Diff		Avg Diff	
	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1
Metrics + Xgboost	67.17	79.14	59.49	61.27	70.39	61.31	67.17	63.76	61.36	63.76	14.84	11.80	10.30	8.71
3-layer BiLSTM	67.25	80.41	57.85	57.75	69.08	55.61	53.27	69.51	59.40	65.62	16.48	15.32	14.04	8.78
3-layer BiLSTM + Att	75.63	82.66	65.79	59.92	74.50	58.52	61.71	66.01	69.57	68.65	8.54	13.15	5.97	7.41
CNN	70.72	79.55	60.47	59.29	70.48	58.15	53.42	66.58	63.36	60.13	16.16	13.78	11.72	9.82
<i>Ggrn</i> (AST)	72.65	81.28	70.08	66.84	79.62	64.56	63.54	70.43	67.74	64.67	6.93	8.59	4.69	5.01
<i>Ggrn</i> (CFG)	78.79	82.35	71.42	67.74	79.36	65.40	65.00	71.79	70.62	70.86	4.58	5.33	2.38	2.93
<i>Ggrn</i> (NCS)	78.68	81.84	72.99	69.98	78.13	59.80	65.63	69.09	70.43	69.86	3.95	8.16	2.24	4.45
<i>Ggrn</i> (DFG_C)	70.53	81.03	69.30	56.06	73.17	50.83	63.75	69.44	65.52	64.57	9.05	17.13	6.96	10.18
<i>Ggrn</i> (DFG_R)	72.43	80.39	68.63	56.35	74.15	52.25	63.75	71.49	66.74	62.91	7.17	16.72	6.27	9.88
<i>Ggrn</i> (DFG_W)	71.09	81.27	71.65	65.88	72.72	51.04	64.37	70.52	63.05	63.26	9.21	16.92	6.84	8.17
<i>Ggrn</i> (Composite)	74.55	79.93	72.77	66.25	78.79	67.32	64.46	70.33	70.35	69.37	5.12	6.82	3.23	3.92
<i>Devign</i> (AST)	<b>80.24</b>	84.57	71.31	65.19	79.04	64.37	65.63	71.83	69.21	69.99	3.95	7.88	2.33	3.37
<i>Devign</i> (CFG)	80.03	82.91	74.22	70.73	79.62	66.05	66.89	70.22	71.32	71.27	2.69	3.33	1.00	2.33
<i>Devign</i> (NCS)	79.58	81.41	72.32	68.98	79.75	65.88	67.29	68.89	70.82	68.45	2.29	4.81	1.46	3.84
<i>Devign</i> (DFG_C)	78.81	83.87	72.30	70.62	79.95	66.47	65.83	70.12	69.88	70.21	3.75	3.43	2.06	2.30
<i>Devign</i> (DFG_R)	78.25	80.33	73.77	70.60	80.66	66.17	66.46	72.12	71.49	70.92	3.12	4.64	1.29	2.53
<i>Devign</i> (DFG_W)	78.70	84.21	72.54	71.08	80.59	66.68	67.50	70.86	71.41	71.14	2.08	2.69	1.27	1.77
<i>Devign</i> (Composite)	79.58	<b>84.97</b>	<b>74.33</b>	<b>73.07</b>	<b>81.32</b>	<b>67.96</b>	<b>69.58</b>	<b>73.55</b>	<b>72.26</b>	<b>73.26</b>	-	-	-	-

# Devign: Results

Method	Cppcheck		Flawfinder		CXXX		3-layer BiLSTM		3-layer BiLSTM + Att		CNN		<i>Devign</i> (Composite)	
	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1
Linux	75.11	0	78.46	12.57	19.44	5.07	18.25	13.12	8.79	16.16	29.03	15.38	69.41	<b>24.64</b>
QEMU	89.21	0	86.24	7.61	33.64	9.29	29.07	15.54	78.43	10.50	75.88	18.80	89.27	<b>41.12</b>
Wireshark	89.19	10.17	89.92	9.46	33.26	3.95	91.39	10.75	84.90	28.35	86.09	8.69	89.37	<b>42.05</b>
FFmpeg	87.72	0	80.34	12.86	36.04	2.45	11.17	18.71	8.98	16.48	70.07	31.25	69.06	<b>34.92</b>
Combined	85.41	2.27	85.65	10.41	29.57	4.01	9.65	16.59	15.58	16.24	72.47	17.94	75.56	<b>27.25</b>





# DeepDFA: the Most Recent Work on Vulnerability Detection - 2024 ICSE to appear

see Slides

# Code2vec: Learning Distributed Representations of Code

```
String[] f(final String[] array) {  
    final String[] newArray = new String[array.length];  
    for (int index = 0; index < array.length; index++) {  
        newArray[array.length - index - 1] = array[index];  
    }  
    return newArray;  
}
```

## Predictions

<b>reverseArray</b>		77.34%
<b>reverse</b>		18.18%
<b>subArray</b>		1.45%
<b>copyArray</b>		0.74%

- ▶ Training data: code snippets, and their labels
- ▶ Goal: predicting a name of a method (semantic labeling of code snippets)
- ▶ Demo: <https://code2vec.org/>

# Overall Approach

1. obtain a path from ast
2. map a bag of paths to a real-value vector
3. learn the model
4. predict the name based on the model and the vector

# Obtain an *AST path*

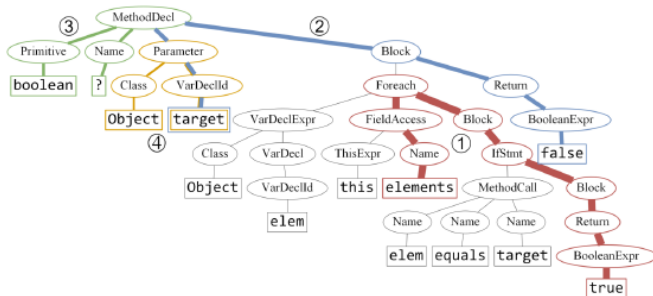


Fig. 3. The top-4 attended paths of Figure 2a, as were learned by the model, shown on the AST of the same snippet. The width of each colored path is proportional to the attention it was given (red ①: 0.23, blue ②: 0.14, green ③: 0.09, orange ④: 0.07).

(elements, Name↑FieldAccess↑Foreach↓Block↓IfStmt↓Block↓Return↓BooleanExpr, true)

# Distributed Representation of a Method

- ▶ a bag of path-contexts for a method

**Definition 3** (Path-context). Given an AST Path  $p$ , its path-context is a triplet  $\langle x_s, p, x_t \rangle$  where  $x_s = \phi(\text{start}(p))$  and  $x_t = \phi(\text{end}(p))$  are the values associated with the start and end terminals of  $p$ .

That is, a path-context describes two actual tokens with the syntactic path between them.

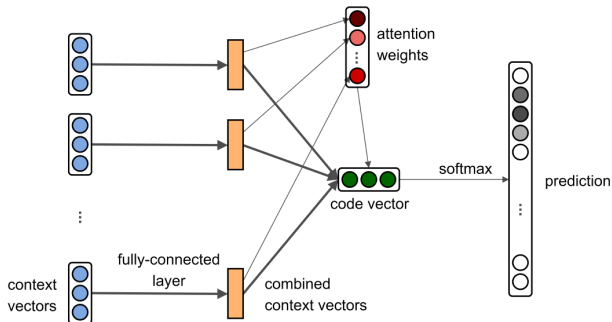
*Example 3.1.* A possible path-context that represents the statement: “ $x = 7;$ ” would be:

$$\langle x, (\text{NameExpr} \uparrow \text{AssignExpr} \downarrow \text{IntegerLiteralExpr}), 7 \rangle$$

- ▶ the weight is initialized with random values, then learned from the training data



# Neural Network Architecture



**Fig. 4.** The architecture of our path-attention network. A *full-connected layer* learns to combine embeddings of each path-contexts with itself; attention weights are learned using the combined context vectors, and used to compute a *code vector*. The code vector is used to predicts the label.

## Experimental Evaluation - Dataset

	Number of methods	Number of files	Size (GB)
Training	14,162,842	1,712,819	66
Validation	415,046	50,000	2.3
Test	413,915	50,000	2.3
Sampled Test	7,454	1,000	0.04

**Table 2.** Size of data used in the experimental evaluation.

# Experimental Evaluation - Results

Model	Sampled Test Set (7454 methods)			Full Test Set (413915 methods)			prediction rate (examples / sec)
	Precision	Recall	F1	Precision	Recall	F1	
CNN+Attention [Allamanis et al. 2016]	47.3	29.4	33.9	-	-	-	0.1
LSTM+Attention [Iyer et al. 2016]	27.5	21.5	24.1	33.7	22.0	26.6	5
Paths+CRFs [Alon et al. 2018]	-	-	-	53.6	46.6	49.9	10
<b>PathAttention (this work)</b>	<b>63.3</b>	<b>56.2</b>	<b>59.5</b>	<b>63.1</b>	<b>54.4</b>	<b>58.4</b>	<b>1000</b>

**Table 3.** Evaluation comparison between our model and previous works.

# Probabilistic models of code: How to model big code via machine learning (optional)

- ▶ *generative models*: defines a probabilistic distribution over code by stochastically modeling the generation of smaller and simpler parts of code, e.g., what is the probability of deriving to this sub-tree as the left child of the root
- ▶ *representational models*: conditional probability distribution over code element properties, e.g., what is the probability of types X, Y and Z for this variable a (based on, e.g., surrounding context of a)
- ▶ *pattern mining models*: unsupervised learning to infer a latent (not observable) structure within code

# Generative Models

How to generate code in different context:

- ▶ Input: training data  $D$
  - ▶ A output code representation:  $c$
  - ▶ A possibly empty context:  $C(c)$
  - ▶ Probabilistic distribution:  $P_D(c|C(c))$
- 
- ▶ language model:  $C(c) = \emptyset$
  - ▶ code-generative multimodal model:  $C(c)$  is a non-code modality, e.g., natural language
  - ▶ transducer model:  $C(c)$  is also code

# Generative Models

How to represent the code in generative models

- ▶ "a bag of words": a set of tokens
- ▶ a sequence of tokens: n-gram
- ▶ abstract syntax trees
- ▶ graphs

# Structure prediction: Predicting program properties from big code

```
function chunkData(e, t) {  
  var n = [];  
  var r = e.length;  
  var i = 0;  
  for (; i < r; i += t) {  
    if (i + t < r) {  
      n.push(e.substring(i, i + t));  
    } else {  
      n.push(e.substring(i, r));  
    }  
  }  
  return n;  
}
```

(a) JavaScript program with minified identifier names

```
/* str: string, step: number, return: Array */  
function chunkData(str, step) {  
  var colNames = []; /* colNames: Array */  
  var len = str.length;  
  var i = 0; /* i: number */  
  for (; i < len; i += step) {  
    if (i + step < len) {  
      colNames.push(str.substring(i, i + step));  
    } else {  
      colNames.push(str.substring(i, len));  
    }  
  }  
  return colNames;  
}
```

(e) JavaScript program with new identifier names and types

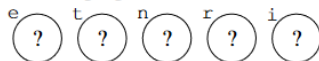
# Predicting program properties from big code

- ▶ code as a variable dependence network
- ▶ represent each variable as a node, the property of some node we know while the property of some node we don't know
- ▶ represent variable interaction as *conditional random field (CRF)* – a type of probabilistic graphical models that represent the conditional probability of labels  $y$  given observations of  $x$   $P(y|x)$
- ▶ train CRF to predict names and types of variables for Javascripts



# Predicting program properties from big code

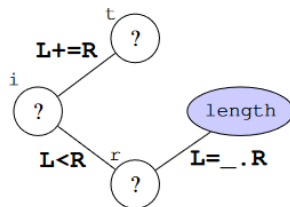
Unknown properties (variable names):



Known properties (constants, APIs):

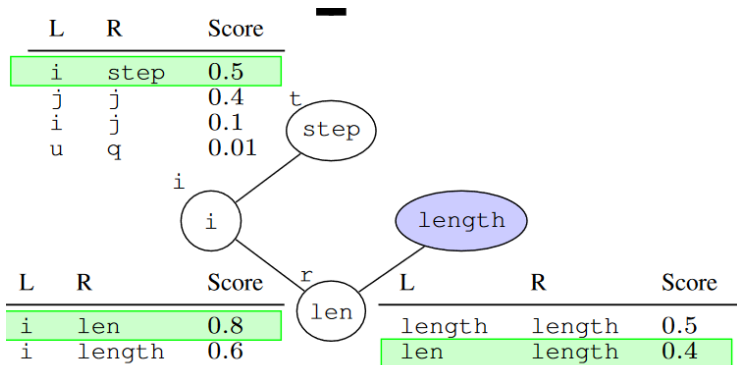


(b) Known and unknown name properties



(c) Dependency network

# Predicting program properties from big code



# Pattern mining models

- ▶ Pattern mining models aim to discover a finite set of human-interpretable patterns from source code, without annotation or supervision
- ▶ Probabilistic pattern mining models of code infer the likely latent structure of a probability distribution

# Pattern mining models

- ▶ Examples of pattern mining models: frequent pattern mining – it is not a probabilistic model but counting based
- ▶ Probabilistic model can find interesting models, sometimes frequent occurred patterns may not be interesting
- ▶ Most probably grouping of API elements

# Summary: AI for SE

- ▶ Program analysis to get and represent information from code
- ▶ Code embedding: represent code using vectors
- ▶ AI modeling of code
  - ▶ Source code as tokens
  - ▶ AST, CFG, DPG property graphs
  - ▶ Comments
  - ▶ Traces
  - ▶ ...
- ▶ Applications:
  - ▶ predicting developers intent: recommender system
  - ▶ predicting anomaly code
  - ▶ code translation
  - ▶ clone and idiom mining
  - ▶ predict names
  - ▶ test case generation

## Further Readings

- ▶ Learning to represent programs with graphs, ICLR 2018
- ▶ Code2seq: Generating sequences from structured representations of code, ICLR 2019
- ▶ Path-based Function Embedding and Its Application to Error-handling Specification Mining, FSE 2018
- ▶ Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces, FSE 2018
- ▶ Dynamic Neural Program Embedding for Program Repair, ICLR 2018
- ▶ Code2Vec: Learning Distributed Representations of Code, POPL 2019
- ▶ Blended, Precise Semantic Program Embeddings, PLDI, 2020
- ▶ COSET: A Benchmark for Evaluating Neural Program Embeddings, arXiv, 2019
- ▶ Predicting program properties from "big code"
- ▶ Deep Code
- ▶ Machine Learning for Programming
- ▶ A Survey of Machine Learning for Big Code and Naturalness (2018)
- ▶ Machine learning on source code