DeepBugs: A Learning Approach to Name-based Bug Detection

Presentation by Ben Meerovitch

“This paper presents DeepBugs, a learning approach to name-based bug detection, which reasons about names based on a semantic representation and which automatically learns bug detectors instead of manually writing them.”

Outline

- ✓ Intro + Examples
- ✓ Framework for learning to find name-related bugs (DeepBug)
- ✓ Name based bug detectors (built on-top of DeepBug)
- ✓ Results
- ✓ Related Work + Q&A

Introduction

Bug detection as a binary classification problem

Train a classifier that distinguishes correct from incorrect code.

Create likely incorrect code examples from an existing corpus of code.

A framework for learning-based and name-based bug detection

Three bug detectors built on top of the framework detect: swapped function arguments, incorrect binary operators, and incorrect operands in binary operations.

Applying on a corpus of 150,000 JavaScript files reveal 102 programming mistakes (with 68% true positive rate) in real-world code.

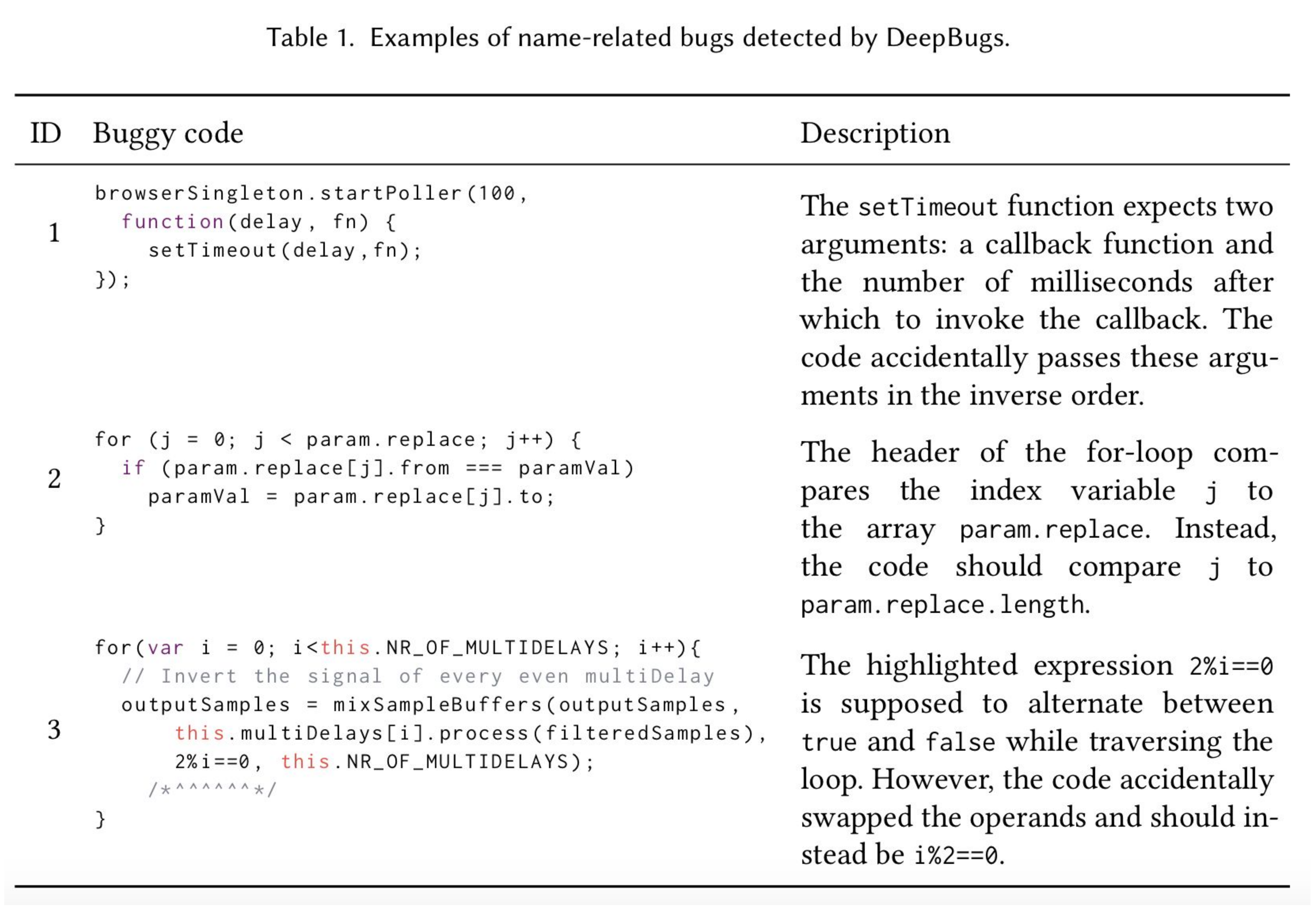


Table 1. Examples of name-related bugs detected by DeepBugs.

ID	Buggy code	Description
1	`browserSingleton.startPoller(100,` `function(delay, fn) {` `setTimeout(delay,fn);` `});`	The `setTimeout` function expects two arguments: a callback function and the number of milliseconds after which to invoke the callback. The code accidentally passes these arguments in the inverse order.
2	`for (j = 0; j < param.replace; j++) {` `if (param.replace[j].from === paramVal)` `paramVal = param.replace[j].to;` `}`	The header of the for-loop compares the index variable `j` to the array `param.replace`. Instead, the code should compare `j` to `param.replace.length`.
3	`for(var i = 0; i<this.NR_OF_MULTIDELAYS; i++){` `// Invert the signal of every even multiDelay` `outputSamples = mixSampleBuffers(outputSamples,` `this.multiDelays[i].process(filteredSamples),` `2%i==0, this.NR_OF_MULTIDELAYS);` `/*^^^^^^^*/` `}`	The highlighted expression `2%i==0` is supposed to alternate between `true` and `false` while traversing the loop. However, the code accidentally swapped the operands and should instead be `i%2==0`.

Angular.js

Angular-UI-Router project

DSP.js library

Name-Based Bug Detector: Challenges

Reason about the meaning of identifier names

Identifier names are inherently fuzzy and elude the precise reasoning that is otherwise common in program analysis.

Decide whether a given piece of code is correct or incorrect

Given an understanding of the meaning of identifier names, must yield a reasonably low number of false positives while detecting actual bugs.

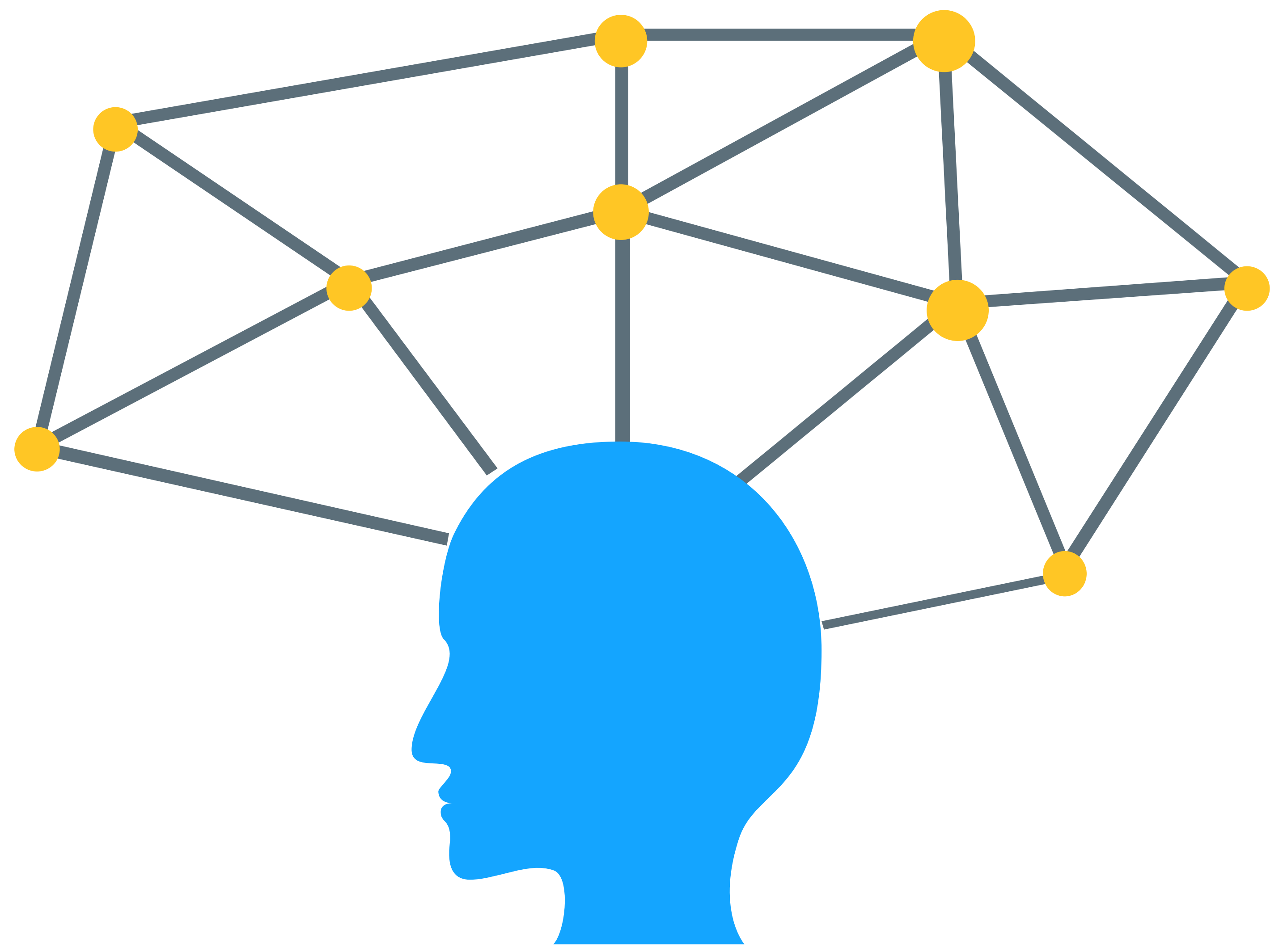
Name-Based Bug Detector: Solutions

Use a learned vector representation of identifiers.

This representation, called embeddings, preserves semantic similarities, such as the fact that *length* and *count* are similar.

Formulate the problem as binary classification

Train a model to distinguish correct from incorrect code.



DeepBug

A **framework** that supports different classes of name-related bugs

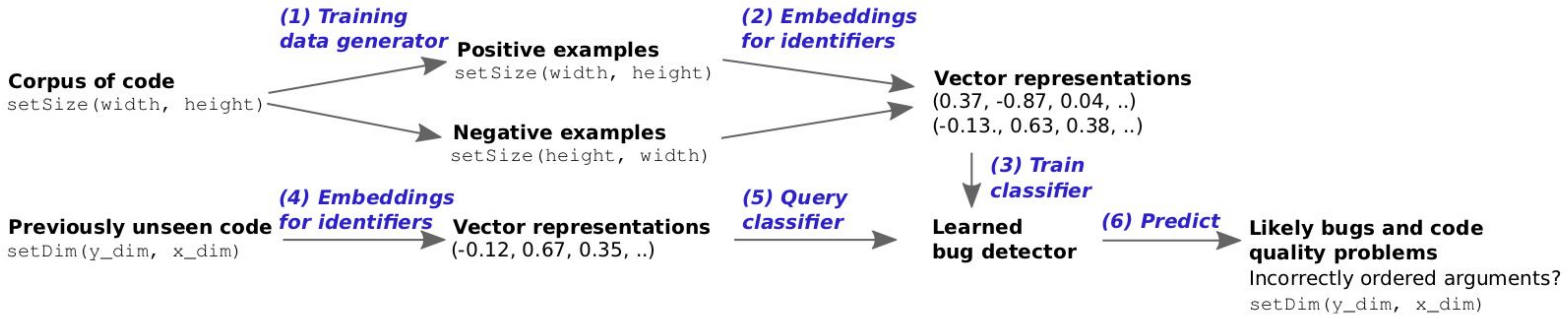


Fig. 1. Overview of our approach.

The framework extracts positive training examples from a code corpus, applies a simple transformation to also create large amounts of negative training examples, trains a model to distinguish these two, and finally uses the trained model for identifying mistakes in previously unseen code.

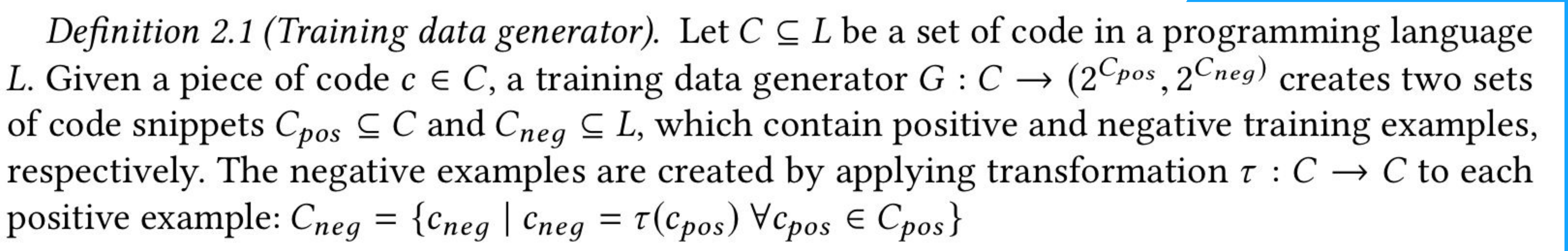
Drill-Down

(1) Extract and generate training data from the corpus.

(2) Represent code as vectors. (Word2Vec)

(3) Train a model to distinguish correct and incorrect examples.

(4) Predict bugs in previously unseen code.


Definition 2.1 (Training data generator). Let $C \subseteq L$ be a set of code in a programming language L. Given a piece of code $c \in C$, a training data generator $G : C \longrightarrow (2^{C_{pos}}, 2^{C_{neg}})$ creates two sets of code snippets $C_{pos} \subseteq C$ and $C_{neg} \subseteq L$, which contain positive and negative training examples, respectively. The negative examples are created by applying transformation $\tau : C \longrightarrow C$ to each positive example: $C_{neg} = \{c_{neg} \mid c_{neg} = \tau(c_{pos}) \, \forall c_{pos} \in C_{pos}\}$

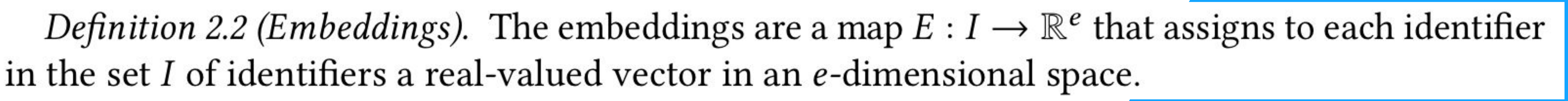
(1) Generating Training Data

Huge amount of existing code provides ample of examples of likely correct code. In contrast, it is non-trivial to obtain many examples of code that suffers from a particular bug pattern.

- Implementations of τ that apply simple AST-based code transformations will be covered later.

(2.1)

Embeddings: Identifiers and Literals



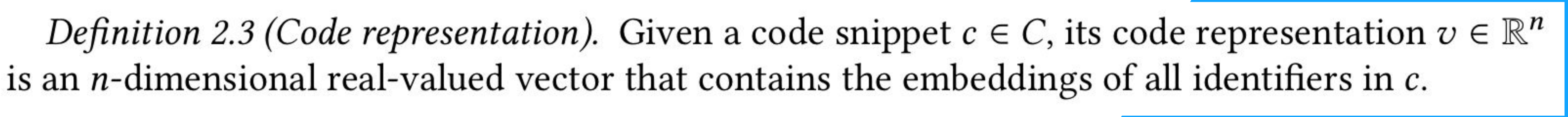
Definition 2.2 (Embeddings). The embeddings are a map $E : I \longrightarrow \mathbb{R}^e$ that assigns to each identifier in the set I of identifiers a real-valued vector in an e-dimensional space.

Semantic similarity does not always correspond to lexical similarity. Thus, a representation of identifiers that preserves semantic similarities is required.

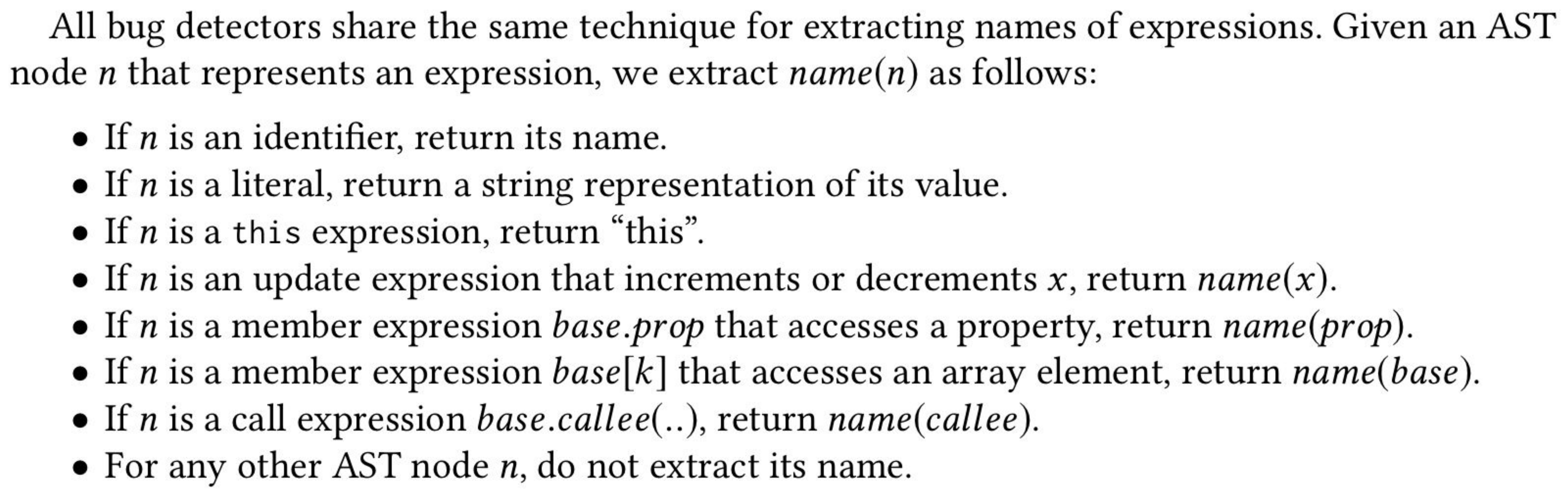
- consider literals in code, such as *true* and *23*.
- Word2Vec.

(2.2)

Vector Representations of Positive and Negative Code Examples



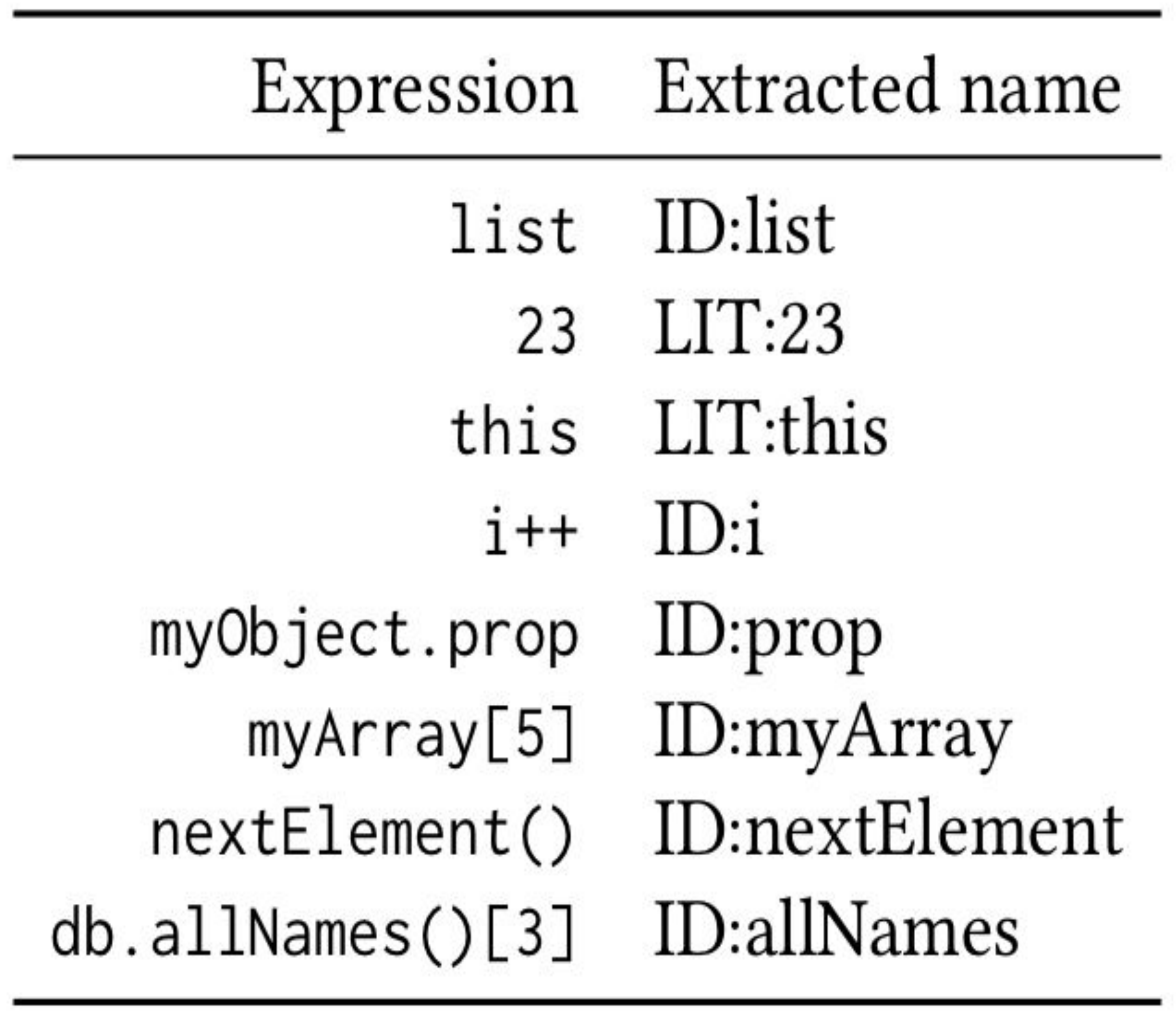
Definition 2.3 (Code representation). Given a code snippet $c \in C$, its code representation $v \in \mathbb{R}^n$ is an n-dimensional real-valued vector that contains the embeddings of all identifiers in c.



All bug detectors share the same technique for extracting names of expressions. Given an AST node n that represents an expression, we extract $name(n)$ as follows:

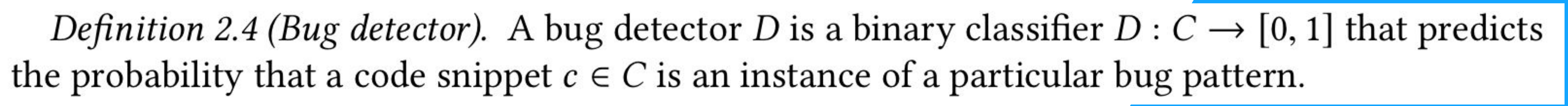
- If n is an identifier, return its name.
- If n is a literal, return a string representation of its value.
- If n is a `this` expression, return "this".
- If n is an update expression that increments or decrements x, return $name(x)$.
- If n is a member expression $base.prop$ that accesses a property, return $name(prop)$.
- If n is a member expression $base[k]$ that accesses an array element, return $name(base)$.
- If n is a call expression $base.callee(..)$, return $name(callee)$.
- For any other AST node n, do not extract its name.

Table 2. Examples of identifier names and literals extracted for name-based bug detectors.



Expression	Extracted name
`list`	ID:list
`23`	LIT:23
`this`	LIT:this
`i++`	ID:i
`myObject.prop`	ID:prop
`myArray[5]`	ID:myArray
`nextElement()`	ID:nextElement
`db.allNames()[3]`	ID:allNames

Given code snippets extracted from a corpus, our approach uses the embeddings for identifiers to represent each snippet as a vector suitable for learning.



Definition 2.4 (Bug detector). A bug detector D is a binary classifier $D : C \to [0, 1]$ that predicts the probability that a code snippet $c \in C$ is an instance of a particular bug pattern.

(3)

Training and Querying a Bug Detector

Based on the vector representation of code snippets, a bug detector is a model that distinguishes between vectors that correspond to correct and incorrect code examples, respectively.

- Feedforward neural network

NAME-BASED BUG DETECTORS

Three examples of name-based bug detectors built on top of the DeepBugs framework.

Examples

Accidentally swapped function arguments

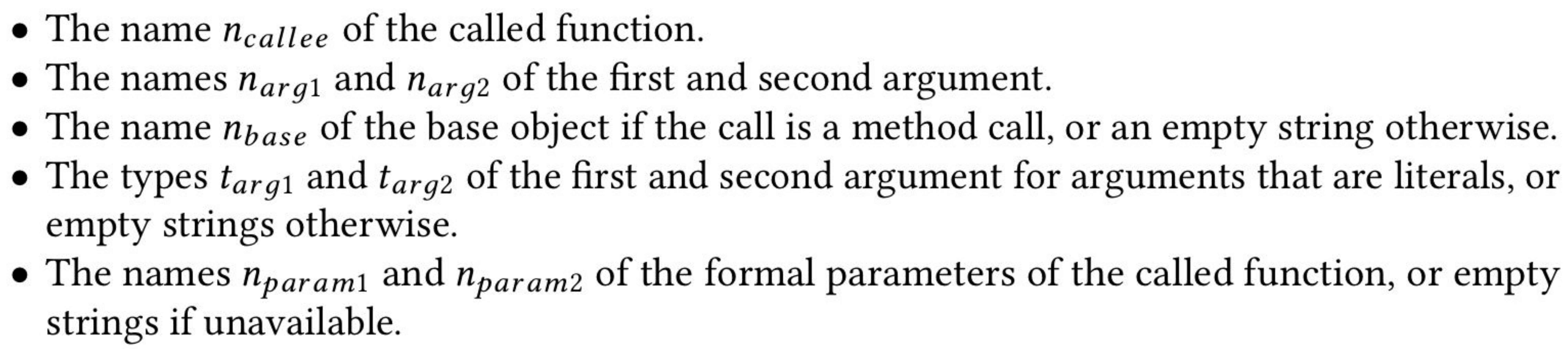
Incorrect binary operators

Incorrect operands in binary expressions

Swapped Function Arguments

Training Data Generator:

Create training examples from given code. Traverse the AST of each file in the code corpus and visits each call site that has two or more arguments. Extract the following:



- The name n_{callee} of the called function.
- The names n_{arg1} and n_{arg2} of the first and second argument.
- The name n_{base} of the base object if the call is a method call, or an empty string otherwise.
- The types t_{arg1} and t_{arg2} of the first and second argument for arguments that are literals, or empty strings otherwise.
- The names n_{param1} and n_{param2} of the formal parameters of the called function, or empty strings if unavailable.

Swapped Function Arguments

Training Data Generator:

Create for each call site a positive example

$$x_{pos} = (n_{base}, n_{callee}, n_{arg1}, n_{arg2}, t_{arg1}, t_{arg2}, n_{param1}, n_{param2})$$

and a negative example

$$x_{neg} = (n_{base}, n_{callee}, n_{arg2}, n_{arg1}, t_{arg2}, t_{arg1}, n_{param1}, n_{param2}).$$

To create the negative example, we simply swap the arguments w.r.t. the order in the original code.

Swapped Function Arguments

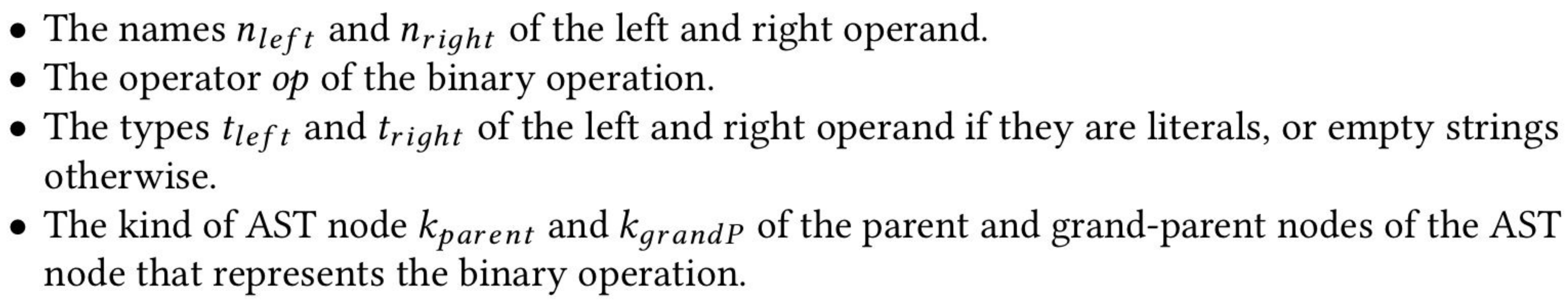
Code representation:

- Transform Xpos and Xneg from tuples of strings into vectors.
- Each name n is represented as E(n), where E is the learned embedding.
- Represent type names as vectors.
    - Define a function T that maps each built-in type in JavaScript to a randomly chosen binary vector of length 5.
- Compute code representation for Xpos or Xneg as the concatenation the individual vectors.

Wrong Binary Operator

Training Data Generator:

Create training examples from given code. Traverse the AST of each file in the code corpus and from each binary operation extract the following:



- The names n_{left} and n_{right} of the left and right operand.
- The operator *op* of the binary operation.
- The types t_{left} and t_{right} of the left and right operand if they are literals, or empty strings otherwise.
- The kind of AST node k_{parent} and k_{grandP} of the parent and grand-parent nodes of the AST node that represents the binary operation.

Wrong Binary Operator

Training Data Generator:

Create a positive and negative example

$$x_{pos} = (n_{left}, n_{right}, op, t_{left}, t_{right}, k_{parent}, k_{grandP})$$

$$x_{neg} = (n_{left}, n_{right}, op', t_{left}, t_{right}, k_{parent}, k_{grandP})$$

The operator op'≠op is a randomly selected binary operator different from the original operator likely to create incorrect code.

Wrong Binary Operator

Code representation:

- Create a vector representation of each positive and negative example by mapping each string in the tuple to a vector and by concatenating the resulting vectors.
- To map a kind of AST node K to a vector, we use a map K that assigns to each kind of AST node in JavaScript a randomly chosen binary vector of length 8.

Wrong Operand in Binary Operation

The intuition is that identifier names help to decide whether an operand fits another given operand and a given binary operator.

Training Data Generator:
The training data generator extracts the same information as in the last example, and then replaces one of the operands with a randomly selected alternative.

Wrong Operand in Binary Operation

Training Data Generator:
Create a positive example

$$x_{pos} = (n_{left}, n_{right}, op, t_{left}, t_{right}, k_{parent}, k_{grandP})$$

and a negative example

$$x_{neg} = (n'_{left}, n_{right}, op, t'_{left}, t_{right}, k_{parent}, k_{grandP})$$

or

$$x_{neg} = (n_{left}, n'_{right}, op, t_{left}, t'_{right}, k_{parent}, k_{grandP}).$$

To create negative examples that a programmer might also create by accident, use alternative operands that occur in the same file as the binary operation. For example, given *bits << 2*, the approach may transform it into a negative example *bits << next*.

Wrong Operand in Binary Operation

Code representation:

- Same as previous example.

Results

1. Experimental Setup
2. Extraction and Generation of Training Data
3. Warnings in Real-World Code
4. Accuracy and Recall of Bug Detectors
5. Efficiency
6. Usefulness of Embeddings and Vocabulary

Main Research Questions

- How effective is the approach at distinguishing correct from incorrect code?
- Does the approach find bugs in production JavaScript code?
- How long does it take to train a model and, once a model has been trained, to predict bugs?
- How useful are the learned embeddings of identifiers compared to a simpler vector representation?

1

Experimental Setup

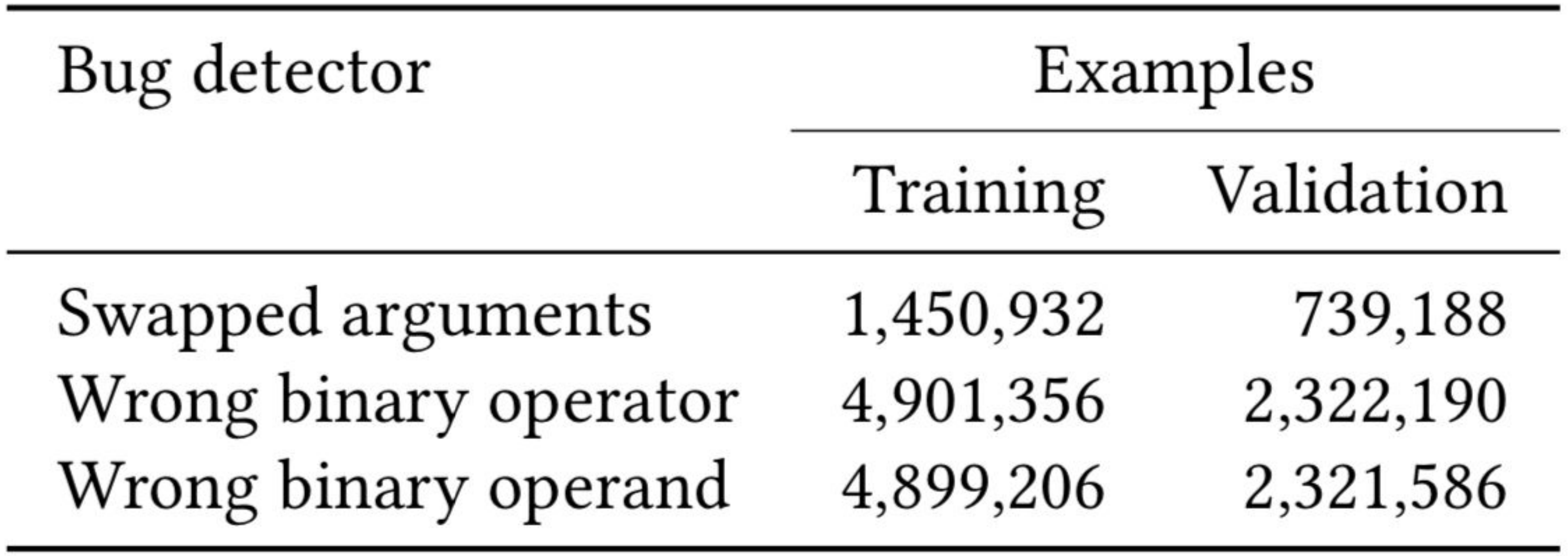
- Used 150,000 JavaScript files provided by the authors of earlier work as a corpus of code.
- Corpus contains 68.6 million lines of code.
- 100,000 files for training and the remaining 50,000 files for validation.
- All experiments are performed on a single machine with 48 Intel Xeon E5-2650 CPU cores, 64GB of memory, and a single NVIDIA Tesla P100 GPU

2

Extraction and Generation of Training Data

- Table 3 summarizes the training and validation data that DeepBugs extracts and generates for the three bug detectors.
- Half of the examples are positive and negative code examples, respectively.

Table 3. Statistics on extraction and generation of training data.



Bug detector	Examples	
	Training	Validation
Swapped arguments	1,450,932	739,188
Wrong binary operator	4,901,356	2,322,190
Wrong binary operand	4,899,206	2,321,586

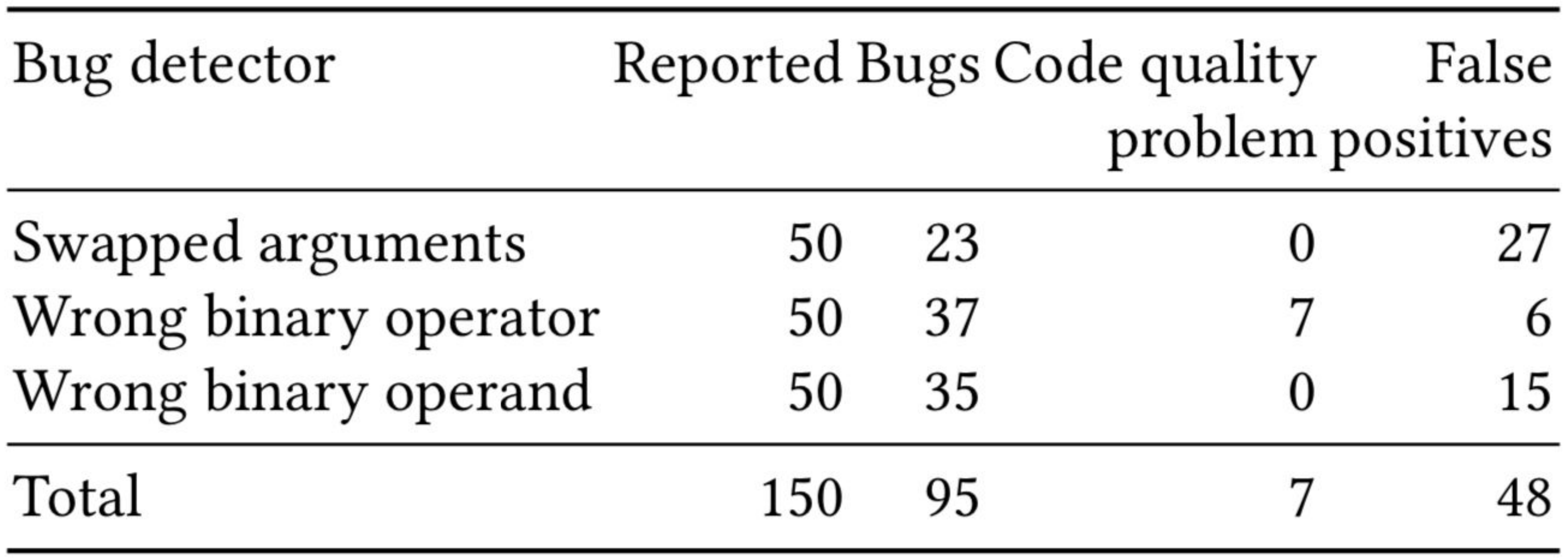
Warnings in Real-World Code

- Train each bug detector with the 100,000 training files, then apply the trained bug detector to the 50,000 validation files.
- Manually inspect code locations that the bug detectors report as potentially incorrect.
- Sort all reported warnings by the probability that the code is incorrect.
- Inspect the top 50 warnings per bug detector.
- Classify each warning in one of three categories.
    - Bug.
    - Code quality problem.
    - False positive.

Warnings in Real-World Code: Results

- Out of the 150 inspected warnings, 95 warnings point to bugs and 7 warnings point to a code quality problem, i.e., 68% of all warnings point to an actual problem.

Table 4. Results of inspecting and classifying warnings in real-world code.



Bug detector	Reported	Bugs	Code quality problem	False positives
Swapped arguments	50	23	0	27
Wrong binary operator	50	37	7	6
Wrong binary operand	50	35	0	15
Total	150	95	7	48

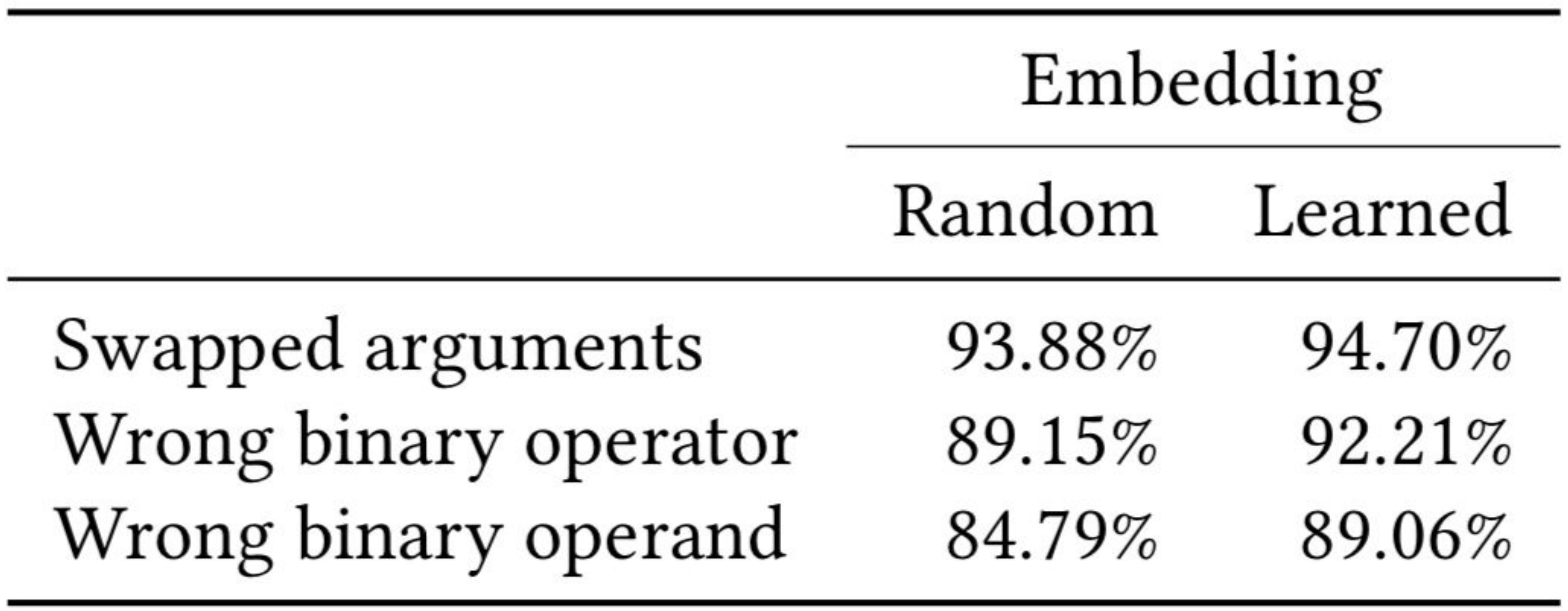
4

Accuracy and Recall of Bug Detectors

- Evaluate the accuracy and recall of each bug detector based on automatically seeded bugs.
- *Accuracy* - how many of the classification decisions that the bug detector makes are correct.
- *Recall* - how many of all bugs in a corpus of code that the bug detector finds.
- Use training data generator to extract correct code examples **Cpos** and to artificially create likely incorrect code examples **Cneg.**
- Query the bug detector D with each example c, which yields a probability D(c) that the example is buggy.
- Compute the accuracy: $accuracy = \frac{|\{c \mid c \in C_{pos} \wedge D(c) < 0.5\}| + |\{c \mid c \in C_{neg} \wedge D(c) \geq 0.5\}|}{|C_{pos}| + |C_{neg}|}$

Accuracy and Recall of Bug Detectors

Table 5. Accuracy of the bug detectors with random and learned embeddings for identifiers.



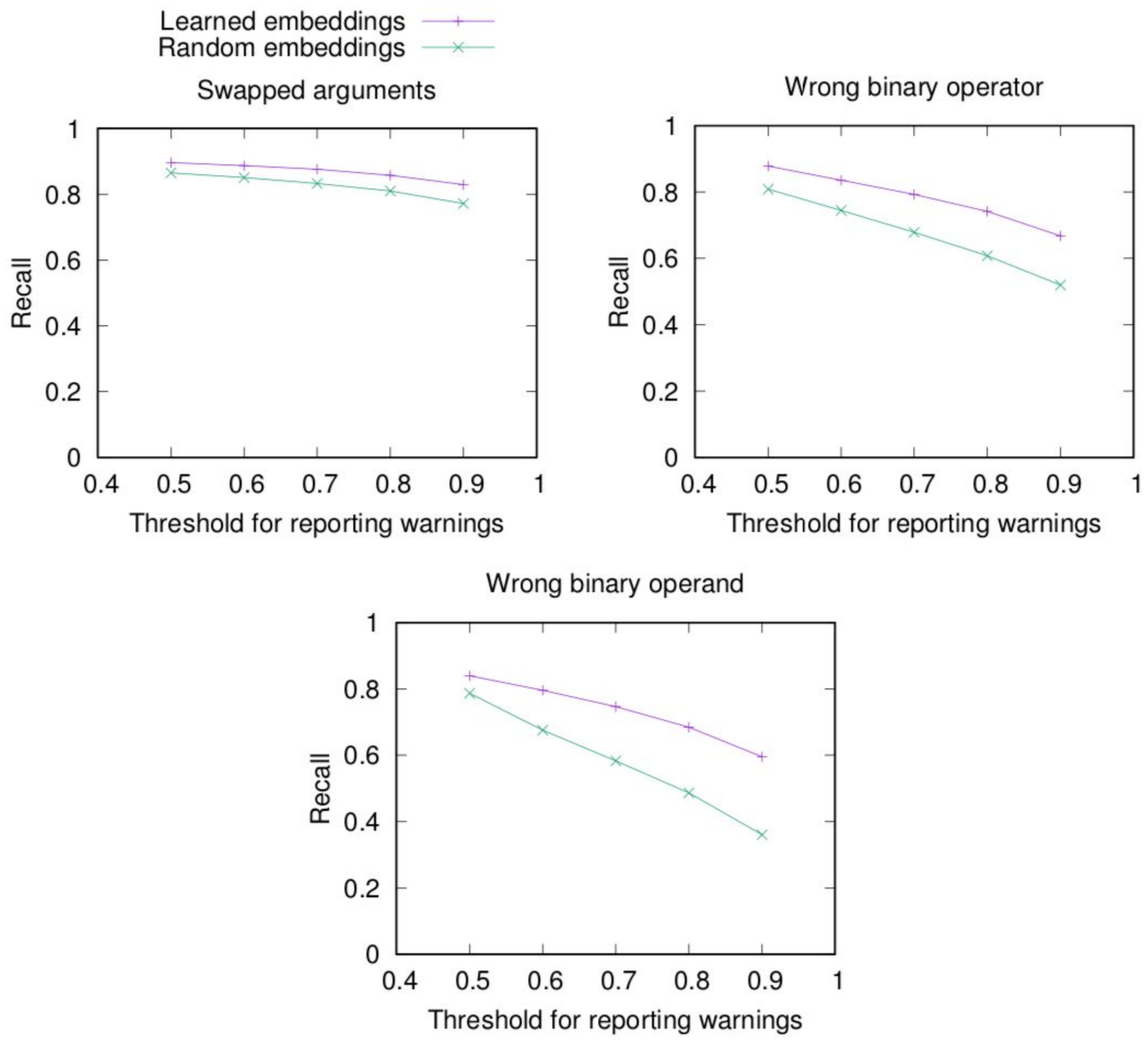
	Embedding	
	Random	Learned
Swapped arguments	93.88%	94.70%
Wrong binary operator	89.15%	92.21%
Wrong binary operand	84.79%	89.06%

Accuracy and Recall of Bug Detectors

- More warnings are likely to reveal more bugs, thus increasing recall, but are also more likely to report false positives.
- A developer inspects all warnings where the probability D(c) is above some threshold t.
- Model this process by turning the bug detector D into a boolean function $D_t(c) = \begin{cases} 1 & \text{if} \quad D(c) > t \\ 0 & \text{if} \quad D(c) \leq t \end{cases}$
- Compute recall: $recall = \frac{|\{c \mid c \in C_{neg} \land D_t(c) = 1\}|}{|C_{neg}|}$
- Measure the number of false positives: $\#fps = |\{c \mid c \in C_{pos} \land D_t(c) = 1\}|$

Accuracy and Recall of Bug Detectors

- Figure 2 shows the recall of the three bug detectors as a function of the threshold for reporting warnings.



Accuracy and Recall of Bug Detectors

- As expected, the recall decreases when the threshold increases
- For a threshold of t = 0.5, the three bug detectors report a total of 116,941 warnings, which corresponds to roughly one warning per 196 lines of code.
- For a threshold of t = 0.9, the number reduces to 11,292, i.e., roughly one warning per 2,025 lines of code
- In practice, we expect developers to inspect only the top-ranked warnings

5 Efficiency

- *Training time* consists of the time to gather code examples and of time to train the classifier.
- *Prediction time* consist of the time to extract code examples and the time to query the classifier with each example.
- Running both training and prediction on all 150,000 files takes between 36 minutes and 73 minutes per bug detector.
- The average prediction time per JavaScript file is below 20 milliseconds.

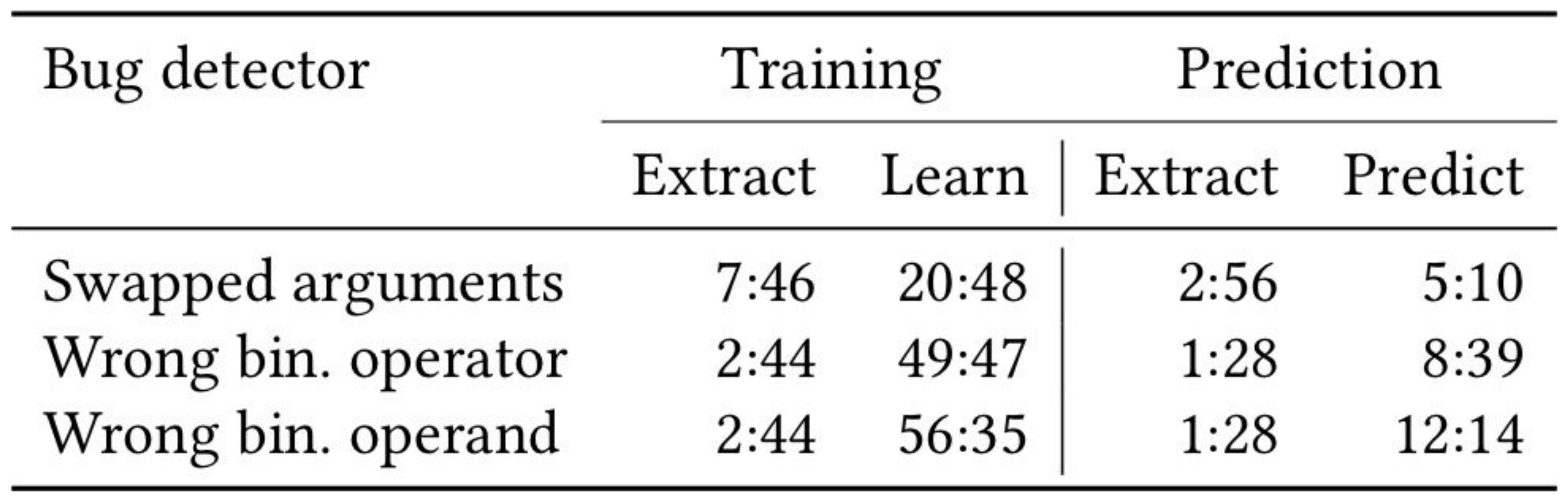


Table 6. Time (min:sec) required for training and using a bug detector across the entire code corpus.

Bug detector	Training		Prediction	
	Extract	Learn	Extract	Predict
Swapped arguments	7:46	20:48	2:56	5:10
Wrong bin. operator	2:44	49:47	1:28	8:39
Wrong bin. operand	2:44	56:35	1:28	12:14

6

Usefulness of Embeddings and Vocabulary

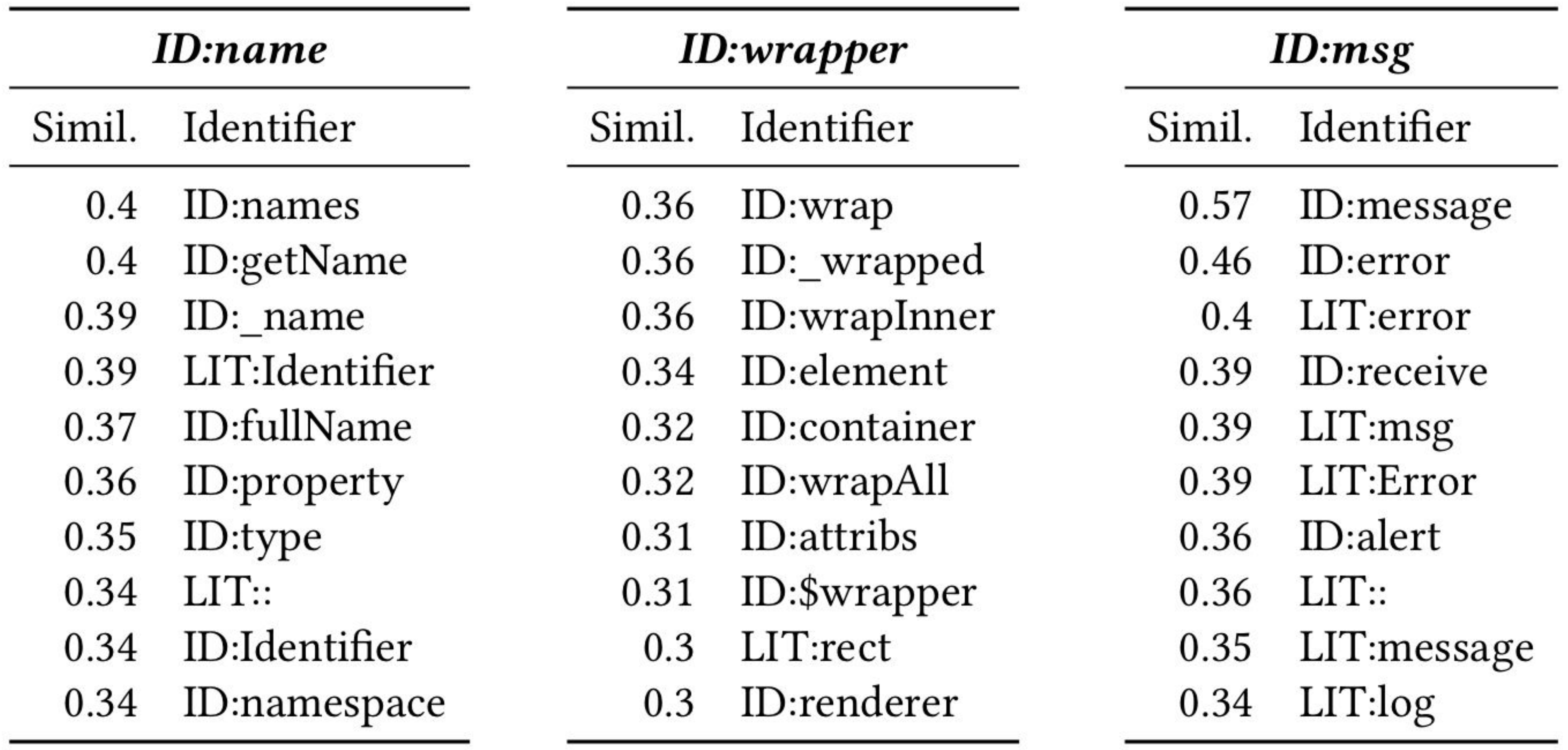
- Evaluate the usefulness of learned embeddings both quantitatively and qualitatively.
- Quantitative evaluation: compare DeepBugs with learned embeddings to a simpler variant of the approach that uses a baseline vector representation.
  - Compare the learned embeddings with the baseline w.r.t. accuracy and recall.

Findings:

- For all three bug detectors, the learned embeddings increase the recall.
- The bug detectors achieve relatively high accuracy and recall even with randomly created embeddings.

Usefulness of Embeddings and Vocabulary

- Qualitative assessment: show for a set of identifiers which other identifiers are the most similar according to the learned embeddings.
- Figure 3 shows the ten most similar identifiers for *name*, *wrapper*, and *msg:*



ID:name		***ID:wrapper***		***ID:msg***	
Simil.	Identifier	Simil.	Identifier	Simil.	Identifier
0.4	ID:names	0.36	ID:wrap	0.57	ID:message
0.4	ID:getName	0.36	ID:_wrapped	0.46	ID:error
0.39	ID:_name	0.36	ID:wrapInner	0.4	LIT:error
0.39	LIT:Identifier	0.34	ID:element	0.39	ID:receive
0.37	ID:fullName	0.32	ID:container	0.39	LIT:msg
0.36	ID:property	0.32	ID:wrapAll	0.39	LIT:Error
0.35	ID:type	0.31	ID:attribs	0.36	ID:alert
0.34	LIT::	0.31	ID:$wrapper	0.36	LIT::
0.34	ID:Identifier	0.3	LIT:rect	0.35	LIT:message
0.34	ID:namespace	0.3	ID:renderer	0.34	LIT:log

Fig. 3. Most similar identifiers according to the learned embeddings.

<u>FIndings:</u> The embeddings encode similarities between

- Abbreviated and non-abbreviated identifiers: *msg* and *message*
- lexically similar identifiers: *name* and *getName*
- lexically dissimilar identifiers such as *name* and *Identifier*, or *wrapper* and *container.*

Usefulness of Embeddings and Vocabulary

Vocabulary:

- Total number of unique tokens in the training data set is about 2.4 million.
- |V| = 10, 000, i.e., consider the 10,000 most frequent tokens.
- Figure shows that a small number of tokens covers a large percentage of the occurrences of tokens. Default vocabulary size covers over 90% of all occurrences of tokens.

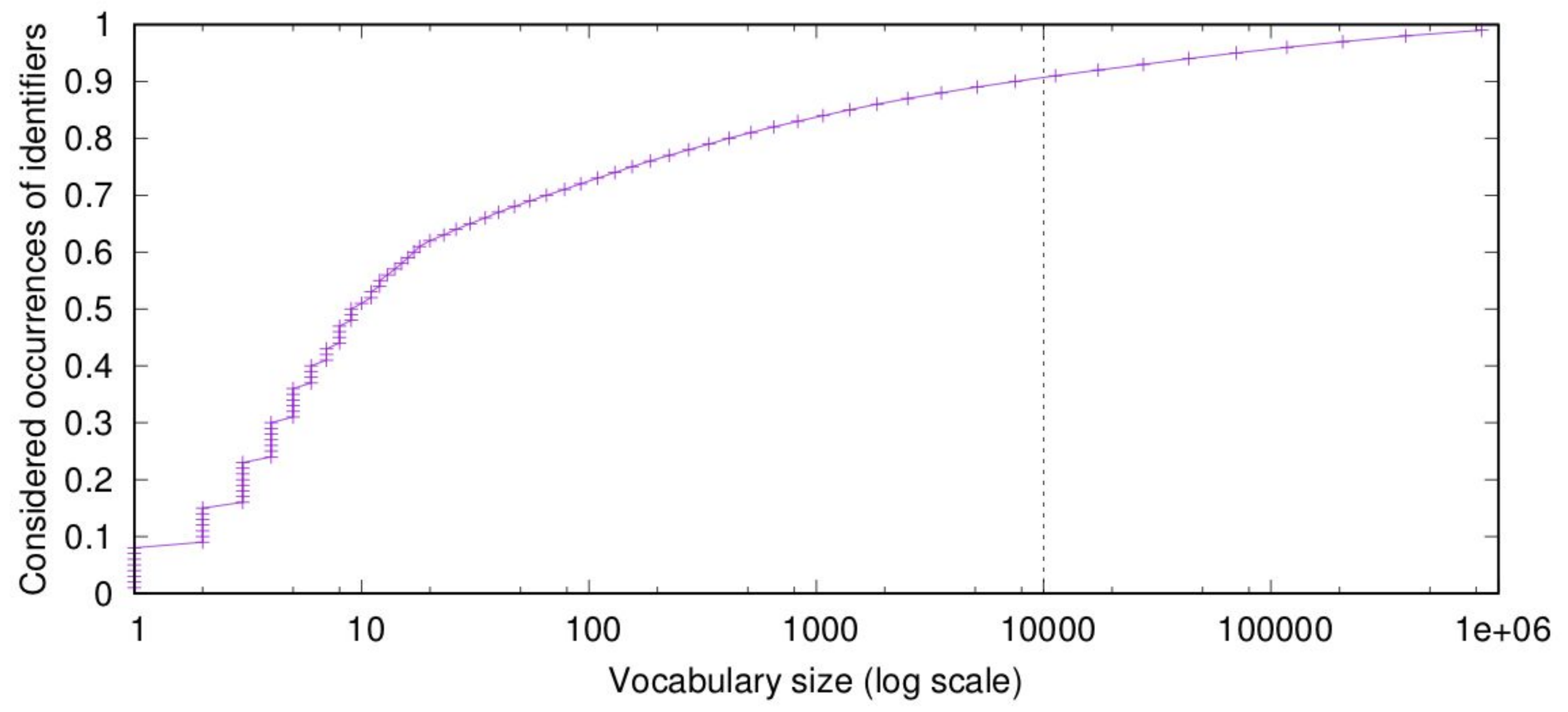


Fig. 4. Number of considered identifier occurrences depending on vocabulary size.

Conclusions

The key insights that enable the approach are:

(i) that reasoning about identifiers based on a learned, semantic representation of names is beneficial

(ii) that artificially seeding bugs through simple code transformations yields accurate bug detectors that are effective also for real-world bugs.

“Applying the framework and three bug detectors built on top of it to a large corpus of code shows that the bug detectors have an accuracy between 89% and 95%, and that they detect 102 programming mistakes with a true positive rate of 68%.”

In the long term, we envision our work to complement manually designed bug detectors by learning from existing code and by replacing some of the human effort required to create bug detectors with computational effort.

Thanks!

Any questions?

You can find me on: Piazza!