International Journal of Computer Applications (0975 - 8887)
Volume 187 - No.39, September 2025

Automatic Name-based Software Bug Detection via AST-Driven Static Analysis and Machine Learning

Dipu Dahal
Department of Electronics and Computer Engineering, Thapathali Campus, Institute of Engineering, Tribhuvan University Kathmandu, Nepal

Shirshak Acharya*
Department of Electronics and Computer Engineering, Thapathali Campus, Institute of Engineering, Tribhuvan University Kathmandu, Nepal

Yunij Karki*
Department of Electronics and Computer Engineering, Thapathali Campus, Institute of Engineering, Tribhuvan University Kathmandu, Nepal

Shashank Ghimire
Department of Electronics and Computer Engineering, Thapathali Campus, Institute of Engineering, Tribhuvan University Kathmandu, Nepal

Dinesh Baniya Kshatri
Department of Electronics and Computer Engineering, Thapathali Campus, Institute of Engineering, Tribhuvan University Kathmandu, Nepal

ABSTRACT

This paper presents a name analysis technique for statically typed languages to automatically classify and localize specific bugs in source code, eliminating the need for manually designed algorithms or heuristics. Name-based bug detection involves analyzing source code to detect potential bugs based on the names or labels used for variables, functions and other elements in the code. The Abstract Syntax Tree (AST) of the source code is utilized to automatically generate negative (buggy) samples due to the unavailability of a large set of negative samples. Approximately 720,000 code snippets of C language are collected from a large C code corpus and parsed into their corresponding ASTs using LibClang. Positive samples are extracted from AST and their contents are adjusted to generate negative samples. These samples are tokenized using a fine-tuned tokenizer and fed into a classification model for training to identify potential bugs. This paper describes techniques for detecting bugs related to swapped function arguments, wrong binary operators and wrong operator precedence, with a high F1 score between 83% and 95%. Moreover, the detection of new types of bugs can be easily accomplished by following similar steps taken in developing current bug detectors. The resulting system can automatically detect specific types of bugs in source code, serving as a tool that enhances code quality for software developers.

Keywords

Abstract Syntax Tree, C Language, Machine Learning, Hyperparameter, Embedding Vector, Bidirectional Encoder Representations from Transformers (BERT)

1. INTRODUCTION

Identifiers, such as variable and function names, are used by programmers to make the program more meaningful beyond the syntax provided by programming language and hence are crucial for program analysis. These names have been used by researchers to develop tools for identifier name recommendation [1], code recommendation [2] and bug detection in source code [3, 4, 5, 6]. However, most existing program analysis tools often ignore identifier names [7, 8] and sometimes replace them with less meaningful names [9], mainly because extracting the semantic meaning embodied in the identifiers is difficult.

Consider the three code snippets provided in Table 1 as examples of bugs detected by the trained models. Example 1 includes an operator precedence bug where the programmer wrongly assumes the execution order of logical operators and omits the parentheses. Example 2 shows a bug related to swapped function arguments where the function arguments are accidentally passed in reverse order. Example 3 shows a wrong binary operator bug where the programmer accidentally uses the subtraction operator (`-`) instead of the addition operator (`+`). The detection of these types of bugs would not have been possible without the use of identifier names.

Most existing name-based bug detectors [2, 3, 4, 6, 10] use lexical distance functions to calculate the similarity between

*These authors contributed equally to this work.

Table 1. : Samples of bugs detected using the proposed approach

S.N.	Part of C code with bug	Explanation

1	`int discountable(int age, int is_member)`
	`{`
	`  int res = age<18 || age>65 && is_member;`
	`  return res;`
	`}`
		Due to missing parentheses, precedence changes and the expression is executed in unintended order, as the precedence of `&&` is greater than `||`. The expression should instead be: `(age<18 || age>65) && is_member`

2	`void copyString(char *src, char *dest) {`
	`  if (src==NULL || dest==NULL) return;`
	`  strcpy(dest, src);`
	`}`
	`char source[50] = "Hi", destination[50];`
	`copyString(destination, source);`
		The `copyString` function requires a source string and a destination string. However, the arguments are passed in reverse order in the code.

3	`int array[] = {1,2,3,4}, size=4, sum=0;`
	`for (int i = 0; i < size; i++) {`
	`  sum = sum - array[i];`
	`}`
		The code incorrectly uses the subtraction operator (`-`) instead of the addition operator (`+`). The expression should instead be: `sum + array[i]`

identifiers and manually design rules for bug detection. Recent research on name-based bug detection [5, 11] has employed Deep Learning-based algorithms with datasets consisting of a dynamically typed object-oriented language. Instead, this paper uses Deep Learning on a dataset of C, a statically typed procedural language, where bug detection is more challenging due to the lack of class-based identifiers and fewer existing and synthetically producible buggy samples due to static type checking. Allamanis et al. [11] use a self-supervised approach for dataset generation and bug detection, but the localization accuracy of 85% or less for 6 out of 7 types of bugs significantly degrades the overall performance of bug detectors. In contrast, this paper's approach has 100% localization accuracy by locating the node of a probable bug in the Abstract Syntax Tree (AST) before extracting the data required for model input.

The proposed approach uses vector embeddings to reason about identifier names and trains a binary classification model to predict whether a given code snippet is correct or buggy. While datasets of correct code are abundant due to the presence of open-source repositories, assuming the code in them is mostly bug-free, buggy code samples are rarely available and difficult to create manually. To tackle this challenge, the proposed approach synthetically generates buggy code samples through simple transformations of correct code samples. This paper presents techniques for detecting bugs related to swapped function arguments, wrong binary operators and wrong operator precedence, with an F1 score ranging from 83 to 95 percent, where the detection of operator precedence bugs has never been done before.

The key contributions of this paper are:

(1) This paper presents a method to synthetically generate buggy data samples from correct samples of statically typed source code.

(2) This paper introduces a novel technique for the detection of operator precedence bugs with an accuracy of 94%.

(3) This paper presents a reusable system for bug detection, which can be used for training new bug detectors, especially for statically typed languages. The source code for this system is available as open-source at: `https://github.com/dipudl/deepscan`

2. RELATED WORKS

2.1 Machine Learning for Bug Detection

Initial bug detection techniques [2, 3, 4, 10] used a rule-based approach that needed to be manually designed and was less generalizable compared to Machine Learning (ML) based methods. Murali et al. [12] train a Recurrent Neural Network for detecting incorrect usage of Android APIs using only correct code samples. Wang et al. [13] create an n-gram-based model for detecting bugs in source code. Choi et al. [14] train a memory neural network to detect buffer overrun bugs from the raw form of source code. Devlin et al. [15] designed a custom neural network architecture for their use case of logical bug detection, where each node of the Abstract Syntax Tree (AST) is passed as input in the form of vector embeddings. In contrast to this paper's approach of using a small part of the source code as a sample, they synthetically inject bugs into the source code to create buggy samples of code and use the whole source code as a sample of data. DeepBugs [5] is the first reported tool to use ML for name-based bug detection, doing so on a dynamically typed language to detect the bugs related to swapped function arguments, wrong binary operators and wrong operands in binary operation. Allamanis et al. [11] use a self-supervised approach to create the training data, detect bugs and repair them. However, the localization accuracy of 85% or less for 6 out of 7 types of bugs seems to significantly degrade the overall performance of the bug detector.

2.2 Name-based Bug Detection

Name-based analysis for bug detection began with manually designed rules or heuristics [2, 4, 10] for identifying specific classes of bugs. Pardel and Gross [4] detect swapped function arguments in function calls using lexical similarity between arguments and

parameters. Lexical similarity (e.g., len and length) doesn't always apply to semantically related identifiers (e.g., length and count), so approaches that rely on such similarity often perform poorly. Rice et al. [10] use a similar approach but with additional heuristics to reduce false positives. They apply handpicked rules, such as excluding functions that intentionally send arguments in a different order (e.g., flip, invert, rotate) and removing common prefixes from function names. The present work is related to DeepBugs [5], which trains feed-forward neural networks separately for detecting each of the three types of name-based bugs in Javascript. Guangjie et al. [6] detect buggy return statements in functions using a sequence of heuristic rules and Jaccard distance to compute the lexical similarity between identifier names.

2.3 Other Approaches

The detection of syntax errors or compile-time errors [9, 16] is also being done using ML, but it is beyond the scope of this paper as it focuses on logical errors, mainly because syntax errors can be detected by the compiler and are easier to resolve. Rule-based vulnerability addition techniques [17, 18] focus on generating buggy code samples by injecting vulnerabilities into source code for the evaluation of bug detection tools, which differs significantly from this paper in terms of the method and purpose.

3. SYSTEM ARCHITECTURE AND METHODOLOGY

The system architecture for creating a name-based bug detector includes data preparation, code tokenization, embedding vector generation, model training, and finally, model inference for bug classification as shown in Figure 1. The core principle of this architecture is to train a classification model that can distinguish between correct code and buggy code samples. The steps involved in developing a bug detector using this system are explained in Sections 3.1, 3.2, 3.3 and 3.4.

3.1 Preparation of Positive and Negative Samples

Advanced deep learning-based bug detection models require a large amount, possibly hundreds of thousands to millions, of both positive and negative training data. Such a number of positive samples (correct code) can be collected from open-source repositories, assuming that the code snippets in them are mostly correct. However, there are insufficient or no buggy codes available for specific types of bugs, making it challenging to obtain them from bug reports and filtered commit histories of code repositories. To address this challenge, this paper automatically generates training data from a large C code corpus [19] consisting of approximately 722,366 C code files. The extraction of positive code examples involves generating an Abstract Syntax Tree (AST) from a C code file, filtering the required nodes of the AST and collecting a specific set of data from those nodes. Negative code samples are generated through swaps and adjustments in positive code samples. For example, a positive training sample for the operator precedence bug detection model is obtained by filtering out the root node of a unary or binary operator expression involving the required number of operands (at least three for expressions with only binary operators) and extracting the expression from that node. Meanwhile, the negative sample is generated by changing the precedence of the unary or binary operator expression through the insertion or removal of parentheses, whichever method is appropriate. The entire process of generating positive and negative samples, as well as the structure of these samples, are explained in detail in Section 4.

As shown in Figure 2, most of the C files in the dataset have less than a thousand lines of code and only 1,700 out of 722,366 C files have more than ten thousand lines of code. Since the AST generation and traversal times depend on code size, files with more than 10,000 lines of code are highly rare and excluded from dataset generation. However, model inference can accommodate any code size for bug classification and is independent of the code sizes used during model training.

3.2 Code Tokenization and Embedding Vector Generation

In the case of name-based bug detectors developed using machine learning (ML), semantically similar identifiers (e.g., count and size) can be distinguished from semantically different ones (e.g., count and country) based on the cosine distance of their vector embeddings, regardless of their lexical similarity. Vector embeddings of identifiers are generated after their tokenization. Tokenization in ML models refers to converting a sequence of text into a sequence of tokens that can be easily processed and analyzed by the ML model. In this project, code tokenization is done using the base version of the CodeT5 [20] tokenizer from Salesforce, which was fine-tuned on a C code corpus to create a new tokenizer with a vocabulary size of 20,000, well-suited to this type of C language dataset. Subword-level tokenization is used, which includes BPE (Byte-Pair Encoding) that merges the most frequent characters into one. For example, if the dataset consists of 'get', 'getting' and 'gets', then with byte pair encoding, it gives 'get' as one token and splits the other tokens as ('get', 'ting') and ('get', 's'). Hence, getting, gets and get will not be considered semantically different words in the vocabulary. Each token from the input sample is then grouped sequentially to generate the embedding vector of dimension 768 for each token using the base variant of the DistilBERT model [21].

In Transformer architecture [22] based models like DistilBERT, the input embedding for each token is generally calculated through the sum of token embedding and positional embedding. Positional embedding is used to represent the position of a token in the input sentence or code sample, as the meaning of an input sample highly depends upon the position of its tokens. The equation for calculating positional embedding for each token is as follows:

$$\begin{aligned} \text{PE}(\text{pos}, 2i) &= \sin\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \\ \text{PE}(\text{pos}, 2i+1) &= \cos\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \end{aligned} \tag{1}$$

where d_{model} is the dimensionality of the embedding (e.g., 768 for this paper), i is an integer in the range $\left[0, \frac{d_{\text{model}}}{2}\right]$ and PE is the matrix containing the positional encoding for each token.

3.3 Model Training and Hyperparameter Tuning

A binary classification model is trained for bug detection, which outputs the probability that the given code is buggy. Any model with binary classification capability can be used for bug classification. This paper trains DistilBERT for the detection of each type of bug, with a feed-forward neural network on top of it as a classification layer. DistilBERT is selected among the BERT

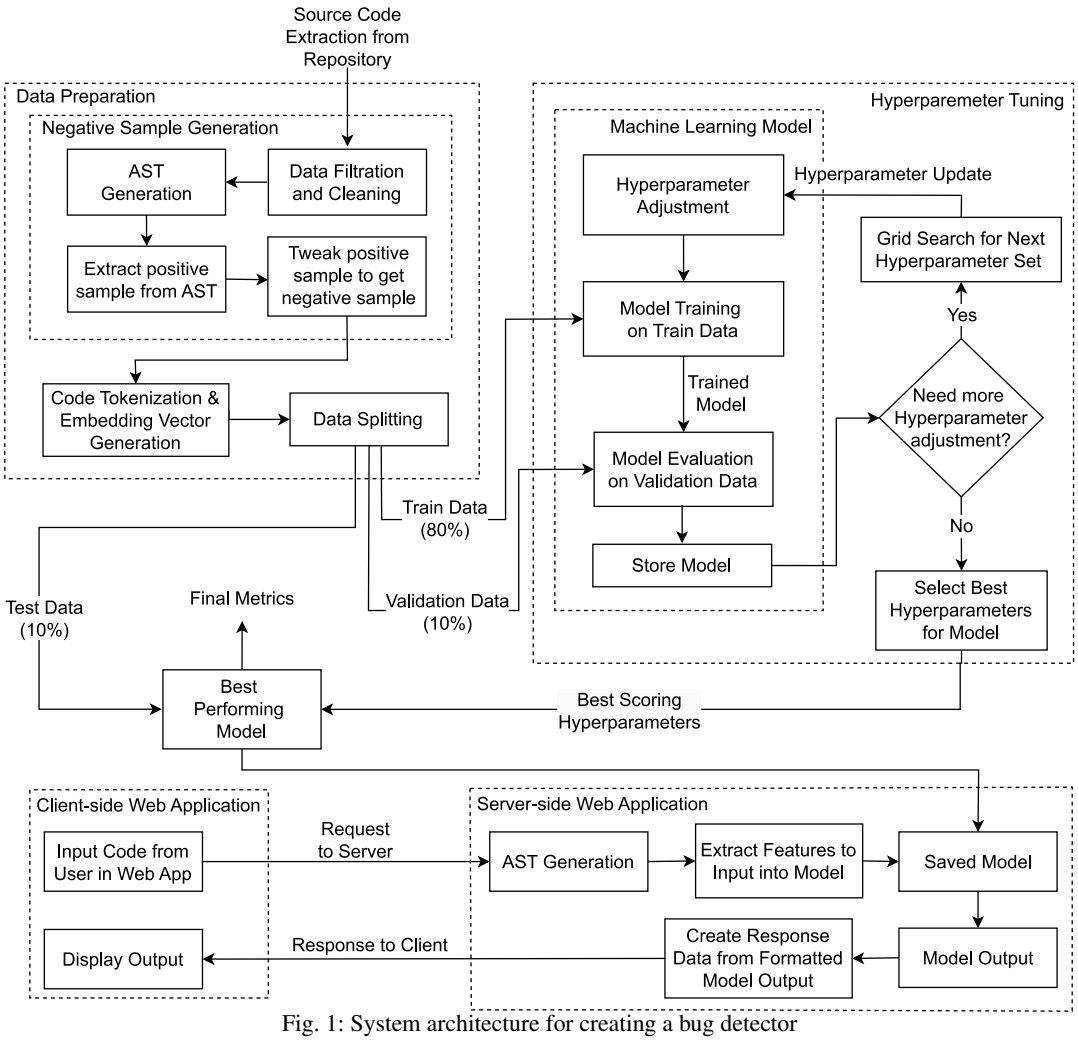

Fig. 1: System architecture for creating a bug detector

variants because it is a smaller, lighter and faster version, making it particularly suitable for resource-constrained situations like ours. DistilBERT can be trained to perform a wide range of natural language processing tasks and bug detection is one of them, as the identifiers and keywords in source code contain beneficial natural language information. The DistilBERT model used in this paper has the GeLU [23] activation function, the AdamW [24] optimizer with a weight decay of 0.01, a classification layer dropout of 0.2 and binary cross entropy as the loss function.

The data samples are split into training, validation and test sets before the models are trained. In this study, the data is divided as follows: 80% for training, 10% for validation and 10% for testing. The training set is used to train the bug detection model to classify input samples, the validation set is used for hyperparameter tuning and monitoring model performance during training, and the test set is used to compute unbiased evaluation metrics for the final trained model. Choosing appropriate values for hyperparameters is the most important step in model training, as they help accelerate the convergence of model loss and determine the final accuracy of the model. This paper tunes two hyperparameters: learning rate and batch size, as they are the most important hyperparameters for the model used. Four learning rate values (2×10^{-3}, 2×10^{-4}, 2×10^{-5} and 2×10^{-6}) and two batch sizes (32 and 64) are used for hyperparameter tuning, which requires training of eight models to determine the best values for these hyperparameters. A model is allowed to train up to 10 epochs, considering the resource constraints. However, more hyperparameters, such as weight decay and hidden layer dimension, can also be tuned and the number of epochs for training the model can be adjusted for convenience.

3.4 Model Inference for Bug Classification

The inference of a trained bug detection model involves using that model to make predictions on unseen code snippets. A model trained on sufficient samples of positive and negative code can learn patterns from them that allow it to perform well on unseen code snippets not included in its training set. A code snippet can contain multiple types of bugs and also multiple instances of a single type of bug, all of which are extracted and detected one by one from the appropriate model. Before the model inference, the input data for identifiable types of bugs are extracted from the AST of the given code snippet, as explained in Section 4. After that, the input samples are tokenized and fed to the respective models for vector embedding generation and classification to get the probability of

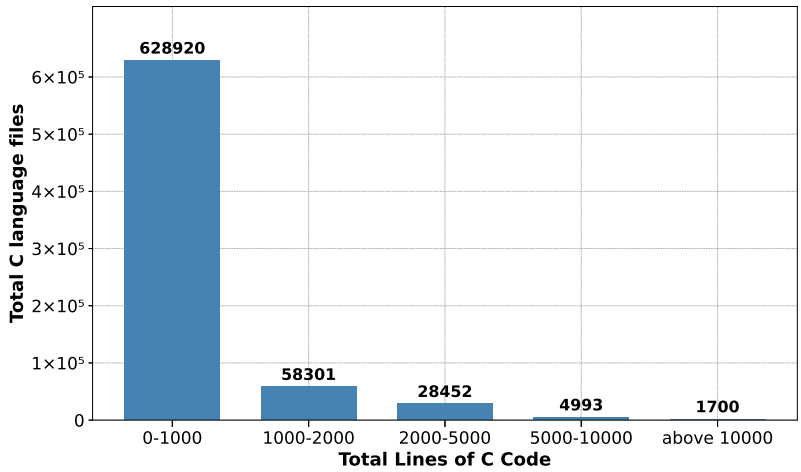

Fig. 2: Total lines of C code vs. total C code files in the dataset

bugginess for each input sample. Each input sample's location (start and end lines and character positions) is stored at the time of its extraction from the AST and displayed to the user along with the inference result.

As shown in Figure 1, this paper presents a complete web application to facilitate model inference for the end user. The client side of the web application contains an input section where users can provide code snippets to test, as well as a section for displaying results. The server side processes the code through a sequence of steps: AST generation, feature extraction, tokenization and model inference, to detect and localize bugs within the code.

4. IMPLEMENTATION OF BUG DETECTORS

This paper implements detectors for three types of name-based bugs: swapped function arguments, wrong binary operators and wrong operator precedence. The first two bug detectors have been implemented in some related papers using ML [5, 11] for dynamically typed languages; however, no prior work has been identified on detecting the third bug, namely the operator precedence bug. The system presented in this paper simplifies the development of new types of bug detectors and its application for creating new detectors is anticipated.

Creating a bug detector requires two steps: generating training data and training the bug detection model. Each bug detector has a separate code for dataset extraction, but the training code is similar and only varies slightly during data preprocessing. An AST of a source code file is required for training data extraction, which is generated through the Clang module in Python, providing access to the Clang[1] compiler using LibClang[2] as an interface. Positive training examples are generated through simple AST traversals and slightly transformed to create negative examples. The DistilBERT model from HuggingFace's transformer[3] library in Python, which primarily uses the Pytorch framework internally, is the base model for training bug detectors. The process of generating training data for each of the three types of bug detectors is described briefly in Sections 4.1, 4.2 and 4.3. After generating the training samples for a particular type of bug detector, each sample is tokenized using the finetuned CodeT5 tokenizer and fed to the DistilBERT model for embedding vector generation for each token and training for bug detection.

[1] `https://clang.llvm.org`

[2] `https://clang.llvm.org/doxygen/group__CINDEX.html`

[3] `https://huggingface.co/docs/transformers/index`

4.1 Swapped Function Arguments Bug

Bugs related to swapped function arguments occur in functions that require two or more arguments of the same type in statically typed languages like C, which is the focus of this paper. The compiler can detect the accidental swap of arguments of different types in a function call, so this situation is not considered while generating training samples and during model inference. Training examples are generated by traversing the AST of each file in the C code corpus to identify function calls that contain exactly two arguments of the same data type. However, the detection of this type of bug in functions with more than two equally typed arguments can be easily done during inference by creating multiple test samples from one function call through different combinations of two arguments in each sample. For example, if x_1, x_2 and x_3 are three equally typed arguments of a function, then three test samples i.e. (x_1, x_2), (x_2, x_3) and (x_1, x_3) can be created to independently query the bug detector.

The following information is extracted from each function call having two equally typed arguments:

$$\text{Positive example } (X_{\text{pos}}) = (fn, arg_1, arg_2, type, pm_1, pm_2)$$

$$\text{Negative example } (X_{\text{neg}}) = (fn, arg_2, arg_1, type, pm_1, pm_2)$$

where,

—fn: Name of the called function

—arg_1 and arg_2: Names of the first and second arguments passed to the function

—*type*: Data type of arg_1 and arg_2

—pm_1 and pm_2: Names of the first and second parameters of the function

The negative example (X_{neg}) is created by swapping the arguments from the positive example (X_{pos}) for this type of bug detection model. If an expression involving two or more identifiers is present as an argument in a function call, the full expression is taken as an argument without applying any heuristics for modifying it to keep

Table 2. : Example input sample for swapped function arguments bug detector

feature	value
fn	`copyString`
arg_1	`destination`
arg_2	`source`
$type$	`char *`
pm_1	`src`
pm_2	`dest`

the process simple and robust. However, if any argument exceeds 100 characters in length, the sample is not included for training the bug detector. Before feeding the input samples for training or testing the model, text pieces or features in the input sample are separated by a special token called a separator token (</s>). The separator token makes it easier for the model to distinguish between different parts of the input. For example, the final form of the positive sample (X_{pf}) looks like this:

$$X_{\text{pf}} = fn</s>arg_1</s>arg_2</s>type</s>pm_1</s>pm_2$$

The extraction of input samples for bug detection in unseen code is performed in the same way as the extraction of positive samples. For example, Table 2 presents the input sample for detecting the swapped function arguments bug in the second code example from Table 1.

4.2 Wrong Binary Operator Bug

The accidental use of a wrong binary operator leads to bugs that are quite difficult to figure out, e.g., `low && high` in place of `low & high`. Training examples are generated by traversing the AST of each file in the C code corpus to identify the root nodes of each binary operation. A positive and a negative sample are created from each binary operation.
The following information is extracted from each binary operation:

$$\text{Positive example } (X_{\text{pos}}) = (l, op, r, tl, tr, p, gp)$$

$$\text{Negative example } (X_{\text{neg}}) = (l, op', r, tl, tr, p, gp)$$

where,

—l and r: Names of the left and right operands in the binary operation

—op: Operator in the binary operation

—tl and tr: Data type of the left and right operands in the binary operation

—p and gp: Types of the parent and grandparent AST nodes of the binary operation

—op': A randomly selected binary operator from the list of possible swaps for the original operator (op) as given in Table 3.

The list of possible swaps for each binary operator in Table 3 contains the operators most likely to be accidentally replaced by the programmer, which helps create relevant negative examples. If the length of any operand exceeds 100 characters, the sample is not included for training the bug detector.

Table 3. : Binary operators with their corresponding possible swaps

Operator	Possible swaps separated by comma
$==$	$>, <, !=, >=, <=, =$
$<$	$>, ==, !=, >=, <=, <<, \wedge, >>$
$+$	$-, *, /, \%, +=$
$*$	$+, -, /, \%, *=$
$-$	$+, *, /, \%, -=$
$!=$	$>, <, ==, >=, <=, =$
$>$	$==, <, !=, >=, <=, <<, \wedge, >>$
$\&$	$\mid, \wedge, <<, >>, \&\&, \mid\mid$
$>=$	$>, <, !=, ==, <=, <<, \wedge, >>$
$/$	$+, -, *, \%, /=$
$<=$	$>, <, !=, >=, ==, <<, \wedge, >>$
$\&\&$	$\mid\mid, \&, \mid$
$<<$	$\mid, \&, \wedge, >>, \&\&, \mid\mid, <=, <$
$>>$	$\mid, \&, \wedge, <<, \&\&, \mid\mid, >=, >$
$\mid\mid$	$\&\&, \mid, \&, /$
$\%$	$+, -, *, /, <, >, >=, <=, !=, \%=$
$\mid$	$\&, \wedge, <<, >>, \&\&, \mid\mid, /$
$\wedge$	$\mid, \&, <<, >>, \&\&, \mid\mid, <, >$

The separator token is added between each feature in the input sample as described in Section 4.1. The extraction of input samples for bug detection in unseen code is done the same way as the extraction of positive samples. For example, Table 4 presents the input sample for detecting the wrong binary operator bug in the third code example from Table 1.

Table 4. : Example input sample for wrong binary operator bug detector

feature	value
l	`sum`
op	`-`
r	`array[i]`
tl	`int`
tr	`int`
p	`BINARY_OPERATOR`
gp	`COMPOUND_STMT`

4.3 Operator Precedence Bug

An operator precedence bug occurs when a programmer makes incorrect assumptions regarding the order in which operators are evaluated in an expression involving multiple operators with different precedence. For example, writing `reg & err == 0` instead of `(reg & err) == 0` leads to incorrect results because the precedence of the equality comparison operator (`==`) is higher than that of the bitwise AND operator (`&`), making the expression `err == 0` execute first. Since the operator precedence bug mainly arises from missed parentheses, only this case is considered during training data generation. The negative sample for this type of bug detector can be created by changing the execution order of positive expression in two ways:

(1) **Removal of parentheses from the positive sample**: If a unary or binary operation is enclosed in parentheses and at least one of its adjacent operators is of higher precedence, then removing the parentheses changes the execution order of the whole expression. For example, removing parentheses from `a / (b + c)` creates a bug because the operator enclosed in parentheses has lower precedence than its adjacent operator in the expression. But no bug can be created from the expression

Table 5. : Bug detector training and prediction metrics

Bug detector	Training time for one epoch	Training epochs	Prediction time per sample	Total data samples
Swapped function arguments	64 minutes	8	1.36 ms	531,140
Wrong binary operator	104 minutes	4	1.31 ms	1,111,986
Wrong operator precedence	96 minutes	5	2.01 ms	633,945

`a + (b / c)` by removing parentheses since the condition does not hold.

(2) **Insertion of parenthesis in a sub-expression of the positive sample**: If a unary or binary operator expression has lower precedence than its adjacent operators, it can be enclosed in parentheses to change the execution order of the whole expression. For example, inserting parentheses in the lower precedence operation of `a + b / c` creates `(a + b) / c`, which is a bug.

If the execution order of the positive expression cannot be changed by inserting or removing parenthesis in a sub-expression, it is discarded. Expressions containing only arithmetic operators with equal precedence (e.g., `+` and `-`) are also discarded because inserting or removing parenthesis in those expressions only changes the order of execution, not the final result. The training sample generation process for this bug detector considers all unary and binary operators except the assignment operators, as this type of bug is not prevalent in them. Training examples are generated by traversing the AST of each file in the C code corpus to identify the root nodes of each unary or binary operation containing multiple operators with different precedence. A positive and a negative sample are created from each of those expressions.
The exact expression in the code snippet is used as the input sample without modification. The extraction of input samples for bug detection in unseen code is done in the same way as the extraction of positive samples. For example, the input sample for detecting the operator precedence bug in the first code example of Table 1 is: `age < 18 || age > 65 && is_member`.

5. RESULTS

This section presents the metrics of the bug detection models and data samples and compares the performance of this project with similar projects.

5.1 Training Data and Training Time

The C code corpus [19], containing 722,366 C code files gathered from various open-source repositories, is used for generating the dataset for training and testing the three bug detectors. After the positive and negative samples are collected, duplicate samples are removed and the samples are divided into training, validation and testing sets containing 80%, 10% and 10% of the total samples respectively. The total number of data samples generated for each bug detection model is provided in Table 5. The training of bug detectors and their testing is done in parallel on two NVIDIA T4 GPUs of 16 GB each. The training and prediction times for each bug detector using the same GPU configuration are presented in Table 5. The prediction time for a sample ranges from 1.31 ms to 2.01 ms. The training time for each bug detector depends on the number of training samples and ranges from 64 minutes to 104 minutes for this project.

5.2 Metrics of Bug Detection Models

This subsection presents the F1 score and ROC (Receiver Operating Characteristic) curve of all bug detectors. The F1 score is the harmonic mean of precision and recall and is used to evaluate the performance of classification models. The F1 score is a balanced measure of a model's performance, considering false positives and negatives.

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2}$$

where,

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{3}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4}$$

The ROC curve is a plot of True Positive Rate vs. False Positive Rate at different classification thresholds, and AUC-ROC (Area Under the ROC Curve) measures the entire area under the ROC curve, ranging from 0 to 1, where an AUC-ROC of 1 represents a perfect classifier.
Model training is stopped either after the completion of 10 epochs or when the validation loss does not decrease by at least 0.01 for two consecutive epochs. The training of all three models seems to stop before reaching 10 epochs due to the same reason. Hyperparameter tuning is performed during model training and the best-performing hyperparameters and corresponding model are saved for inference. For example, the hyperparameter tuning metrics for the operator precedence bug detection model across different combinations of learning rate and batch size are presented in Table 7, where a learning rate of 2×10^{-5} and a batch size of 64 yield the best results. The highest F1 scores of the models range from 83.97% to 95.55%, as shown in Table 6. Similarly, the AUC-ROC values of bug detection models are quite good, ranging from 0.9274 to 0.9890, as shown in Table 6.

Table 6. : Metrics of bug detectors for testing set

Bug detector	AUC-ROC	F1 score
Swapped function arguments	0.9522	88.28%
Wrong binary operator	0.9274	83.97%
Wrong operator precedence	0.9890	95.55%

5.3 Comparison of Metrics with Similar Projects

The language of source code used in datasets varies among similar projects on name-based bug detection. Among the projects using datasets of the same programming language as this project, the dataset itself is different. Therefore, the direct comparison of metrics among the projects in Table 8 is invalid and the

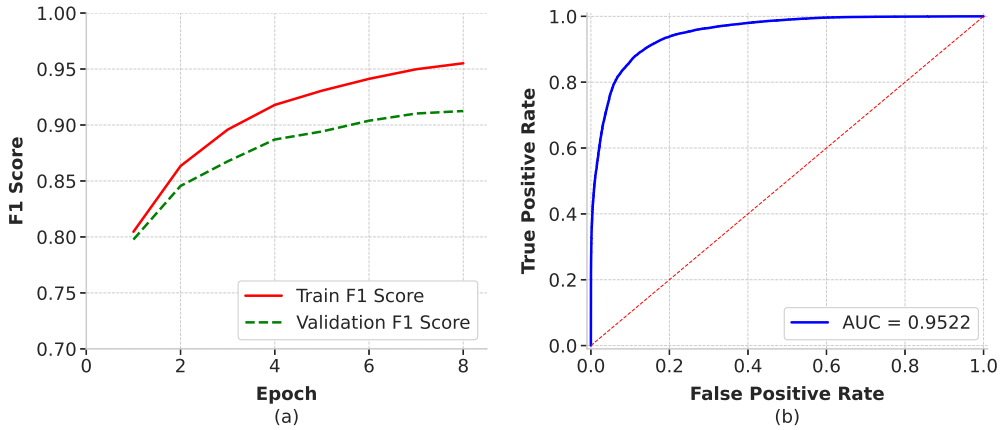

Fig. 3: Plot of swapped function args bug detector metrics: **(a)** F1 score vs epochs **(b)** ROC curve

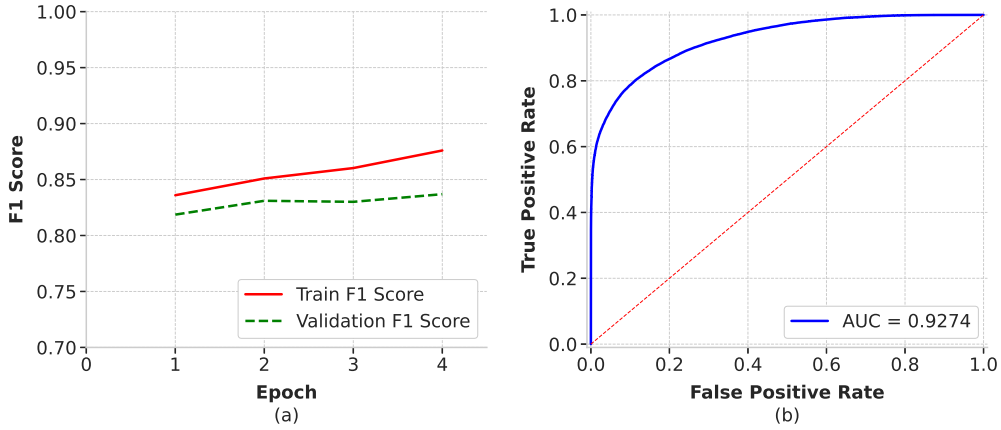

Fig. 4: Plot of wrong binary operator bug detector metrics: **(a)** F1 score vs epochs **(b)** ROC curve

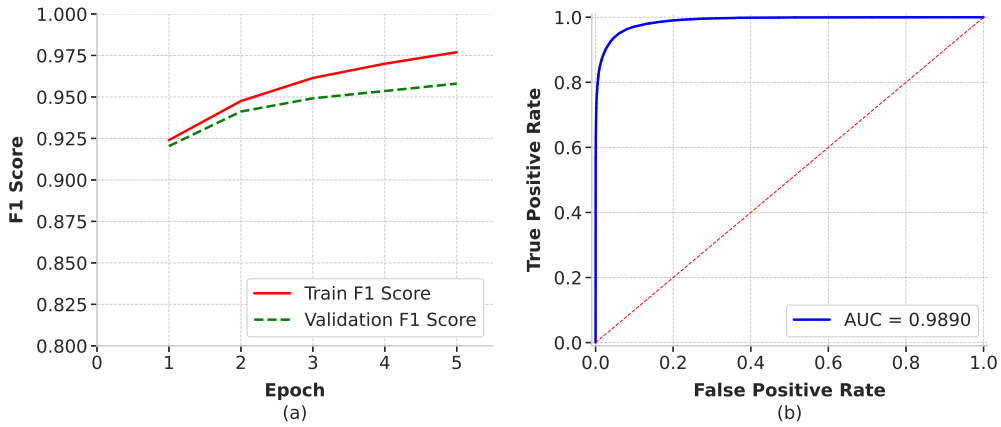

Fig. 5: Plot of operator precedence bug detector metrics: **(a)** F1 score vs epochs **(b)** ROC curve

metrics are presented only for qualitative study of the projects. Pardel and Gross [4], Rice et al. [10] and this paper use a dataset of the C language, while Liu et al. [2] and DeepBugs [5] use Java and Javascript datasets respectively. Direct comparison between bug detectors of different languages is not valid, but their accuracy and precision should not differ significantly. Among precision and accuracy, only one of the metrics is available in similar papers, so the unavailable metric is left blank in Table 8. Compared to the swapped function arguments bug detectors of the C language, this project has the highest precision of 88.69% and greater model training efficiency, as the other projects involve rule-based algorithms or hand-picked patterns. DeepBugs, which

Table 7. : Hyperparameter tuning for operator precedence bug detection model

Learning rate	Batch size	Validation loss	Total epoch	Accuracy	Precision	Recall	F1-score
2×10^{-3}	32	0.673336	3	0.599334	0.599334	1	0.749480
2×10^{-3}	64	0.673314	3	0.599334	0.599334	1	0.749480
2×10^{-4}	32	0.673450	3	0.599334	0.599334	1	0.749480
2×10^{-4}	64	0.673457	3	0.599334	0.599334	1	0.749480
2×10^{-5}	32	0.142283	4	0.944594	0.936895	0.973098	0.954653
2×10^{-5}	64	**0.134386**	5	0.949501	0.954414	0.961674	**0.958030**
2×10^{-6}	32	0.211942	7	0.905832	0.910915	0.934245	0.922433
2×10^{-6}	64	0.247917	7	0.889977	0.903224	0.914398	0.908776

Table 8. : Bug detector training and prediction metrics

Bug detector	Project	Dataset language	Precision	Accuracy
Swapped function arguments	Pradel and Gross	C	80%	-
	Liu et al.	Java	80%	-
	Rice et al.	C	85.47%	-
	DeepBugs	JavaScript	-	94.70%
	This Project	C	88.69%	88.33%
Wrong binary operator	DeepBugs	JavaScript	-	92.21%
	This Project	C	86.52%	84.43%
Wrong operator precedence	This Project	C	95.13%	94.60%

uses Javascript, has slightly higher accuracy for the first two bug detectors compared to this project, but a direct comparison is not feasible due to the difference in programming languages. The detector for the wrong operator precedence bug is implemented solely in this project and has a precision of 95.13% and an accuracy of 94.60%. In summary, this project has the highest precision among bug detectors of the same language and has comparable metrics to other projects using different languages.

6. CONCLUSION

This paper presents a machine learning-based method to detect bugs in source code through the analysis of identifier names, which is useful for both statically and dynamically typed programming languages. The automatic generation of buggy code samples through simple code transformations helps create hundreds of thousands of training data samples for bug detectors, which would have been impractical otherwise. This paper outperforms the previous name-based bug detectors for statically typed languages, at least in terms of training efficiency, by excluding heuristic rules and using advanced machine learning algorithms on top of more than half a million automatically generated training samples. The three bug detectors created through the proposed system, using a large corpus of source code, yield F1 scores between 83% and 95%, demonstrating the effectiveness of the system. In contrast to the training or use of Large Language Models (LLMs) for bug detection, this approach is cost-effective and resource-efficient for detecting specific but highly frequent types of bugs in source code. In the future, the system developed in this project is anticipated to be utilized in the development of new bug detectors and as an automatic data generation technique for training a variety of source code analysis tools.

7. REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 38–49, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 1063–1073, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Michael Pradel and Thomas R. Gross. Detecting anomalies in the order of equally-typed method arguments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 232–242, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Michael Pradel and Thomas R. Gross. Name-based analysis of equally typed method arguments. *IEEE Transactions on Software Engineering*, 39(8):1127–1143, 2013.

[5] Michael Pradel and Koushik Sen. Deepbugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[6] Guangjie Li, Yi Tang, Xiang Zhang, and Biyi Yi. Name-based approach to identify suspicious return statements. *Journal of Physics: Conference Series*, 1792(1):012018, feb 2021.

[7] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '04, page 132–136, New York, NY, USA, 2004. Association for Computing Machinery.

[8] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. Building useful program analysis tools using an extensible java compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23, 2012.

[9] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017.

[10] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. Detecting argument selection defects. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[11] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34 of *NIPS '21*, pages 27865–27876, Red Hook, NY, USA, 2021. Curran Associates, Inc.

[12] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Finding likely errors with bayesian specifications. *CoRR*, abs/1703.01370, 2017.

[13] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 708–719, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Min-Je Choi, Sehun Jeong, Hakjoo Oh, and Jaegul Choo. End-to-end prediction of buffer overruns from raw source code via neural memory networks. *CoRR*, abs/1703.02458, 2017.

[15] Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. Semantic code repair using neuro-symbolic transformation networks. *CoRR*, abs/1710.11054, 2017.

[16] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129, 2016.

[17] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[18] Jannik Pewny and Thorsten Holz. Evilcoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 214–225, New York, NY, USA, 2016. Association for Computing Machinery.

[19] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code != big vocabulary: open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1073–1085, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *CoRR*, abs/2109.00859, 2021.

[21] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, volume 30 of *NIPS'17*, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[23] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[24] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.