

A Structural and Behavioral Hybrid Framework for Detecting AI-Generated Code in Programming Education

Arockia Irudayaraja M.¹, Rexton George R.², Nishanth K.³, Sakthivel P.⁴

¹Assistant Professor, ^{2,3,4} Student, Department of Computer Science and Engineering, NPR College of Engineering and Technology, Dindigul, India.

E-mail: ¹arockiairudayaraja@nprcolleges.org, ²rextongeorger920822104085@nprcolleges.org,

³nishanthk920822104065@nprcolleges.org, ⁴sakthivelp920822104090@nprcolleges.org

Abstract

The Large Language Models (LLMs) for generating source code creates a need for code-aware detection mechanisms capable of analyzing characteristics beyond token-level statistics. This research proposes CodeGuardX, a hybrid framework of structure and behavior, which combines Abstract Syntax Tree inspection, runtime sandboxing, and lexical stylometry in the detection of AI-generated codes. A vector of 33 dimensions containing 17 structural, 6 behavioral, and 10 stylometric features is used for classification using XGBoost with probability calibration and threshold optimization based on F1 metric. Evaluation of the model is performed using 6,800 code samples from the domains of Python, Java, C, and C++; a test set of 1,200 code samples is used for testing purposes. CodeGuardX achieves an accuracy score of 93.8% and F1 score of 92.9%. Ablation study demonstrates increasing F1 scores of 77.8%, 87.3%, and 92.9%. The combination of source code obfuscation, the framework retains 89.4% F1-score, demonstrating the robustness to lexical transformations.

Keywords: AI-generated Code Detection, Abstract Syntax Tree Analysis, Semantic Execution Fingerprinting, Lexical Stylometry, Multidimensional Feature Fusion, XGBoost Classification, Source-Code Obfuscation.

1. Introduction

The LLMs in programming has enabled automatic generation of complete source code solutions based on natural language problem statements. Applications like ChatGPT and GitHub Copilot can generate code, documentation, and examples with minimal intervention on the part of the user [9], [15]. While these capabilities support software programming and learning, they create difficulties in programming assessment, where the source codes submitted are supposed to be an indication of programming skills. It has been necessary to establish reliable methods of recognizing AI-produced codes.

The detection of AI-generated source code is different from detecting type of AI-generated content because programs contains features like syntax, structure, conventions, and semantics that are executable. Techniques that focus on text cannot be said to have been developed with the intention of taking into consideration these aspects, and they are prone to tampering [1], [2]. Code authorship analysis techniques have demonstrated the applicability of structural and stylistic features to detect programming patterns [7], [12], [13]. Nevertheless, static features may not suffice to capture the semantics of the program, leading to the need to include runtime features with the source code analysis.

To address this limitation, the proposed AI code generator detection framework presents a combination of both structural and behavioral features of AI code generators, named CodeGuardX. The proposed framework uses AST-based structural code analysis, sandboxed behavioral analysis, and lexical stylometric code analysis techniques. A total of 33 attributes are combined and categorized via the calibrated XGBoost model, making it possible to conduct research in Python, Java, C, and C++. It is specifically focused on exploring whether the combination of structural layout, behavior execution, and coding style can serve as a strong indicator of artificial intelligence code generation.

2. Related Works

Detection of AI generated content mostly depend on the statistical nature of the generated sequence. The technique used by DetectGPT for detecting machine-generated content is based on probability and curvature while GLTR checks on token-likelihood to differentiate between generated and human written content [1], [10]. These methods are developed for natural language

generation and do not have syntactic structure and executability of code into account. Evaluation studies of AI detectors for programming have indicated the challenges of applying AI detection techniques for programming directly [2], [5].

The increasing use of generative AI in higher education is accompanied by concerns about the reliability of assessments and academic integrity [3], [4], [6], [9]. AI-based programming tools are capable of producing entire solutions, making traditional assessment methods ineffective if code ownership needs to be established. Watermarking has been considered as a potential method for detecting LLM-produced output, yet its practical applicability hinges on the integration with the generation process [8].

Source-code authorship analysis provides more program-specific detection signals. Convolutional neural networks and LSTM model structures have been explored for capturing source-code authorship features [7], [12], while code stylometry proves that programming style can differentiate the behavior of authorship [13]. They show the importance of source-code-specific representation but mainly focus on programmer attribution and not the detection of code produced by LLMs [11].

Overall, the existing research studies indicate a requirement for detection methods that move beyond textual statistics and independently analyzed authorship characteristics. Specifically, the use of the combination of structural features, lexical stylometry, and runtime behavior analysis in detecting multilingual AI-generated code is not sufficiently researched. CodeGuardX addresses this problem by integrating these two representations in a unified detection framework for programming education.

3. Proposed Methodology

3.1 CodeGuardX Framework Architecture

The CodeGuardX algorithm has been developed as a six-step source code detection pipeline which turns a provided piece of code into a multi-dimensional form in order to detect human versus machine-generated code. As illustrated in Figure 1, the first step of the pipeline starts with the validation of the programming language used in the provided code, as well as normalization of the code. The normalized code is then analyzed by three parallel streams. Behavioral analysis stream runs the program in an isolated environment and computes six

features related to runtime behavior, which together constitute the semantics-based execution fingerprint. The stylometric analysis stream computes ten features of the program in terms of lexical and programming style. The output of all three streams is merged to compute a fixed-size 33-dimensional feature vector.

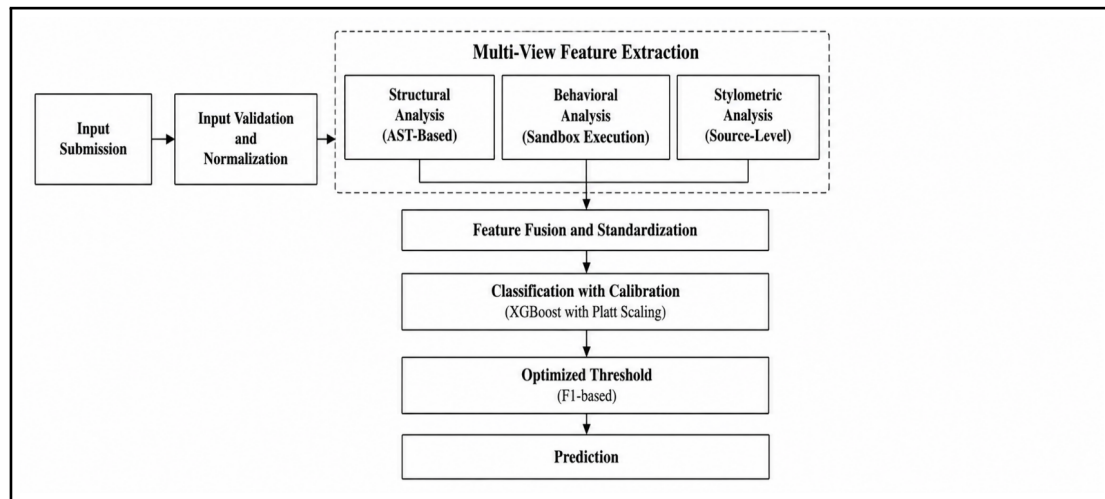


Figure 1. CodeGuardX framework for multidimensional AI-generated code detection

The fusion result is standardized according to the statistical data extracted from the training set, and finally classified by means of XGBoost algorithm. Further on, the output of the classifier is recalibrated by means of Platt probability calibration, and then the threshold for the decision-making process is determined. This architectural design enables CodeGuardX to leverage complementary information from static, runtime, and lexical properties without being dependent on only one property of the source code. This full design will therefore allow for a uniform representation where the source-level and execution-level evidences can be derived from the structural and stylometric, and behavioral properties respectively.

3.2 Dataset Description

CodeGuardX is evaluated using an artificial corpus consisting of human-written and AI-generated source codes from Python, Java, C, and C++. The dataset creation process adheres to the overall research paradigm where there exist separate classes of human-written and AI-generated code for source-code classification, as mentioned in [14]. The human-made programs were selected from open source codes, whereas AI-generated programs were generated using GPT-4 by providing it with specific prompts related to the assignment. Some examples of tasks included in the dataset are sorting, recursion, data structures, and graphs traversals. A

stratified partition approach was used in developing the models and evaluating their performances on the held-out test dataset. The structure of the entire dataset is shown in Table 1.

Table 1. Characteristics of the CodeGuardx dataset

Characteristic	Description
Total samples	6,800
Human-written	3,400
AI-generated	3,400
Programming languages	Python, Java, C, C++
Human-code source	Open-source repositories
AI-code source	GPT-4
Task categories	Sorting, recursion, data structures, graph traversal
Training / Validation / Testing	70% / 15% / 15%
Held-out test samples	1,200
Classification objective	Human-written vs. AI-generated

3.3 Input Validation and Code Normalization

Initially, the input handler checks if the source code provided is not empty and if its size does not exceed 100,000 symbols. Declared language should be in either Python, Java, C or C++. For language-dependent validation, programs in Python are validated via compile (), while in Java, validation includes checking whether there is a declaration of class, interface or enumeration. In C and C++, the input is accepted if there is an include statement or function definition. These validation steps ensure that malformed or unsupported inputs do not enter the feature extraction process.

Following validation, the source is normalized using indentation removal, line-level whitespace stripping, and blank-line collapsing. This preprocessing reduces formatting variation while preserving the program constructs required for structural, behavioral, and stylometric analysis. The normalized representation is then passed to the three feature-extraction streams of CodeGuardX.

3.4 AST-based Structural Feature Extraction

The structural analysis phase will transform the normalized source code to its syntactical representation in the target language before feature extraction. The parsing of Python codes is

done using the ‘ast’ library, Java using javalang and C and C++ using structural extractor based on regular expressions. This extracted representation has three facets of the organization of the programs: topology, programming constructs, and structural complexity.

In total, 17 structural features are obtained from the code, which include: depth of the tree, number of nodes, diversity of node types, number of functions, number of classes, number of loops, number of conditions, number of try-except clauses, number of lambdas, number of returns, number of imports/includes, number of assertions, number of comprehensions, average length of the functions, maximum length of the functions, average depth of nesting, and maximum depth of nesting. These features describe structure and complexity of the code irrespective of its formatting. Specifically, nesting-based features provide evidence of structural nature to differentiate programming structure of human vs. artificial intelligence-written code.

3.5 Sandboxed Behavioral Analysis and Semantic Execution Fingerprinting

The behavioral analysis phase assesses the execution properties of the normalized program in an isolated subprocess. The execution of the program is bound by a hard timeout of 5s, standard input stream is closed and standard output/error streams are captured. Execution of Python programs is restricted to prevent usage of dangerous functions like open, exec, eval, compile, import, breakpoint, input and memoryview. Programs in Java/C/C++ are compiled using javac/gcc/g++, respectively, with any intermediate files being cleaned up after execution.

The resulting execution trace is represented using six behavioral features: raises_exception, has_stdout, execution_time_ms, exception_type_code, output_lines, and timed_out. Exception code captures the typical runtime scenarios, whereas the output and timing features describe the execution-level characteristics that are impossible to extract only from the source structure. The aforementioned features comprise the semantic execution fingerprint, which supplements the structural and stylistic streams of evidence.

The behavioral representation is especially relevant to the detection task since assignment-related programs from students and generated code differ in the execution environment. The paper notes the existence of demonstration output and full execution artifacts as the behavioral properties that can help differentiate between generated programs and programs submitted for automatic assessment. Because renaming of variables, deletion of comments, and

addition of dead code do not always affect execution semantics, the behavioral evidence thus produced can still be valuable after those source code manipulations.

3.6 Lexical and Stylometric Analysis

Stylometric stream is the characterizes of the source code programming style that does not show up through the AST topology or run-time execution. The feature extraction uses AST-informed measurement techniques as well as lexical analysis and regular expressions to quantify the amount of documentation, formatting, identifiers and repetitions in the code. Ten features are derived: comments percentage, blank line percentage, line properties, long line percentage, identifier length, naming conventions, magic numbers, duplicated lines, docstrings and single letter variables.

These characteristics present supplementary evidence of systematized differences in code design. Documentation metrics indicate the presence of explanation comments and documentation strings that accompany the implementations. The identifier length and naming convention metrics define naming regularity. Line-based features reflect formatting and layout characteristics, whereas magic number and duplicate line features reflect implementation-level regularities. As such, the resultant stylometric feature set retains characteristics that are relevant to the source expression despite similarities at the structural level. This stream is then integrated with the other two streams as opposed to using it independently for decisions.

3.7 Multidimensional Feature Fusion and Standardization

The three analytical perspectives generate heterogeneous representations with disparate semantics and metrics. To integrate these complementary representations, the vectors are combined together into a single representation:

$$x = x_S \parallel x_B \parallel x_L \quad (1)$$

where x_S , x_B , and x_L denote the structural, behavioral, and stylometric representations, respectively. The fused vector consists of 33 features, including 17 structural, 6 behavioral, and 10 stylometric features. The canonical feature order is preserved during training and testing, ensuring that each feature maintains the same position within the vector.

Because the fused features have very different numerical ranges, direct classification might let high-range features dominate the classification process. Standardization is hence done before classification through the use of statistical measures from the training set:

$$\tilde{x}_i = \frac{x_i - \mu}{\sigma} \quad (2)$$

where x_i denotes the original feature value, while μ and σ represent the corresponding training-set mean and standard deviation, respectively. The same parameters obtained during training are used for validation and inference, ensuring that no information from the evaluation set affects feature scaling.

3.8 XGBoost-based Classification

The standardized 33-dimensional representation is fed into an XGBoost classifier that discovers nonlinear patterns within the three evidence dimensions. XGBoost is a suitable choice for the fused representation as the feature set includes a mix of numeric and binary features, where their interaction may provide more information than their individual contributions. The classifier employs 400 estimators, a maximum depth of 6, a learning rate of 0.08, row sampling of 0.8, and column sampling of 0.8. The logarithmic loss serves as the evaluation metric, , where as the scale_pos_weight is adjusted based on the class balance.

The objective of learning consists of the loss function for binary classification combined with the regularization term:

$$L = \sum_{i=1}^n l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega f_k \quad (3)$$

where $l(y_i, \hat{y}_i)$ denotes the binary classification loss, and Ωf_k controls the complexity of the k -th tree through its structural and leaf-weight penalties. This regularization technique ensures that the model does not become unnecessarily complex while permitting the interaction between structural, behavioral, and stylistic features to influence the decision boundary. Five-fold cross-validation is performed on the training set to assess the generalizability of the model before calibration and threshold tuning using the validation set.

3.9 Probability Calibration and Threshold Optimization

The XGBoost classifier outputs raw probability scores which are not necessarily equivalent to calibrated probabilities. CodeGuardX therefore applies Platt Scaling to convert the raw output score to calibrated probabilities as follows:

$$\tilde{p}_i = \frac{1}{1 + \exp(A\hat{p}_i + B)} \quad (4)$$

where $\hat{p}_i$ is the raw classifier output, and A and B are calibration parameters estimated from the validation samples. The parameters are estimated by minimizing the negative log-likelihood between the predicted probabilities and the validation-set labels.

After calibration, the threshold value is optimized as opposed to being set to the default value of 0.5. Possible threshold values are examined based on validation precision and recall values; the threshold with the highest F1 is selected:

$$\tau^* = \arg \max_{\tau \in [0,1]} \left(\frac{2 \text{Pr} \tau \text{Re} \tau}{\text{Pr} \tau + \text{Re} \tau} \right) \quad (5)$$

The experimental configuration produces $\tau^* = 0.9797$. The high decision threshold is appropriate for the intended academic integrity application because an inaccurate accusation that AI generated the text has more serious ramifications than an undetected instance.

3.10 CodeGuardX Inference

The inference stage applies the same preprocessing and feature extraction process that was done while training the model for consistency purposes. Following validation and normalization, the program submitted is run through the structural, behavioral, and stylometric analyzers whose output is then concatenated using the pre-defined order of 33 features and standardized using the previously learned standardization parameters. The output is then fed into the trained XGBoost model and calibrated using the previously learned Platt parameters.

The calibrated probability is finally compared with the optimized threshold τ^* . The

decision function is defined as:

$$\hat{y} = \begin{cases} 1, & \tilde{p} \geq \tau^*, \\ 0, & \tilde{p} < \tau^*. \end{cases} \quad (6)$$

where $\hat{y} = 1$ represents AI-generated code, and $\hat{y} = 0$ represents human-written code. At this final stage, the calibrated multidimensional data is translated into a binary decision by CodeGuardX, while the probabilistic result is retained for confidence-based interpretation.

3.11 Implementation Environment and Tools

CodeGuardX is implemented using Python as the primary development environment, and parsing and compilation language-dependent modules are employed for source code analysis across multiple languages. Syntax analysis for Python programs is done by using the built-in ast module, and for Java, the javalang module is used. For C and C++ programs, structure features are extracted by using the implemented regular expression-based module. The runtime behavioral analysis of the program takes place through subprocess isolation execution under the constraint of fixed execution. The programs javac, gcc, and g++ serve as compilers for Java, C, and C++, respectively. The extracted features from structural, behavioral, and stylometric analysis are integrated into the unified representation and normalized based on the parameters obtained only from the training dataset. The XGBoost classifier is used in this study, and then the output probabilities of the classifier are calibrated by the Platt scaling method and optimized via the threshold optimization approach.

4. Results and Discussion

4.1 Feature Importance Analysis

Fused feature representation is evaluated based on XGBoost gain scores. Figure 2 show the ten most important features extracted by the trained classifier. From the distribution of feature importance, one can conclude that structural as well as source level features make significant contributions to classification, whereas behavioral information from the output side adds to discrimination. The presence of nesting, identifier, function-length, and execution output are found to be among the top characteristics, it can be concluded that AI-generated code recognition depends on various aspects of program development, and not just one lexical aspect. These findings favor the inclusion of structural, behavioral, and stylistic representations in the

CodeGuardX framework.

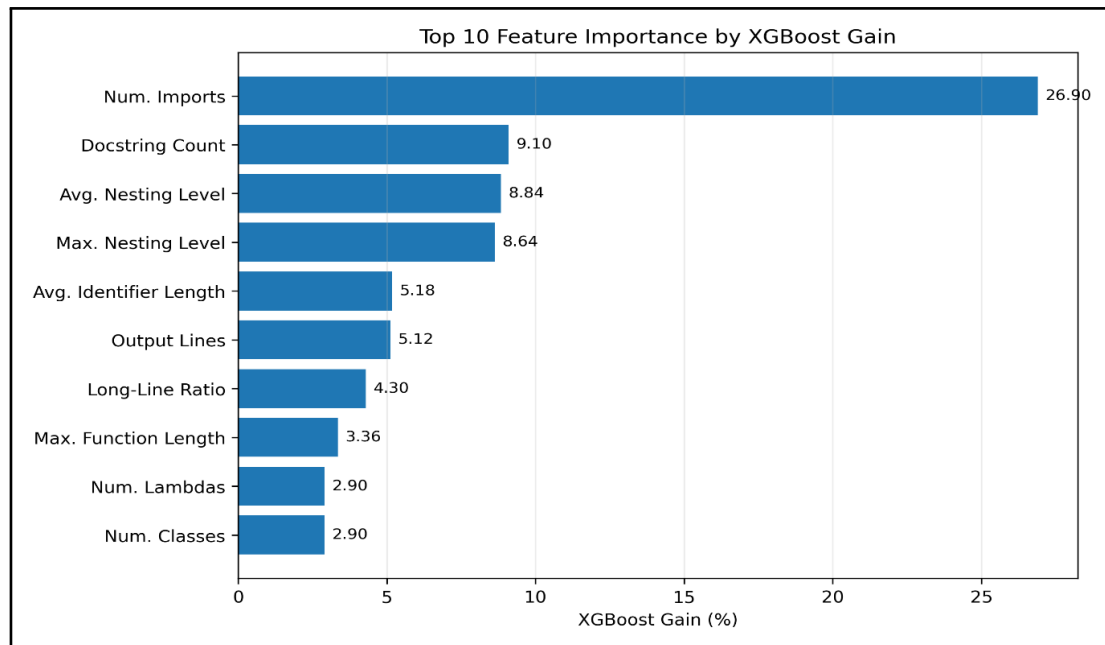


Figure 2. Feature importance by XGBoost gain

4.2 Performance of the Proposed CodeGuardX Configurations

The performance of the proposed CodeGuardX framework is evaluated progressively by activating its complementary feature dimensions. The Style-only configuration establishes the source-level stylometric representation, while the Structural + Style configuration incorporates AST-derived structural characteristics into the same representation. The complete configuration additionally integrates semantic execution fingerprints, allowing the effect of runtime behavioral evidence to be assessed within the proposed framework. Table 2 summarizes the performance obtained by these CodeGuardX configurations on the held-out test set.

Table 2. Performance of proposed CodeGuardX configurations on the held-out test set

CodeGuardX Configuration	Accuracy	Precision	Recall	Score
Style-only	78.3%	76.5%	79.1%	77.8%
Structural + Style	87.6%	86.2%	88.4%	87.3%
Full CodeGuardX	93.8%	92.6%	93.2%	92.9%

The results demonstrate a progressive improvement as additional analytical dimensions are incorporated into the proposed framework. The results demonstrate a progressive improvement

as additional analytical dimensions are incorporated into the proposed framework. The Style-only configuration captures source-level programming characteristics but does not represent the hierarchical organization of the program or its execution behavior. Incorporating structural features substantially strengthens the representation by introducing information related to program organization and complexity. The complete configuration of CodeGuardX performs best when behavior-based elements are added, which shows that runtime behavior offers more discrimination capabilities in addition to the static source code analysis. This improvement is consistent with the primary principle behind the design of CodeGuardX, which relies on multidimensionality rather than a feature-specific detection approach.

4.3 Cross-language and Cross-model Generalization

Table 3. Cross-language and cross-model generalization

Source / Language	Accuracy	Precision	Recall	Score
Python	96.5%	95.8%	96.4%	96.1%
Java	91.8%	90.6%	92.2%	91.4%
C++	91.1%	89.9%	91.5%	90.7%
C	90.6%	89.4%	91.2%	90.3%
GPT-4	93.8%	92.6%	93.2%	92.9%
GPT-3.5 Turbo	90.2%	89.1%	90.4%	89.7%
Claude 3.5 Sonnet	91.5%	90.8%	91.2%	91.0%
Gemini Pro 1.5	89.7%	88.5%	90.1%	89.3%
Real student code	94.0%	92.5%	95.2%	93.8%

The generalization analysis is examining whether CodeGuardX retains its detectability performance regardless of the programming language used and the source LLMs. The findings from the language-specific and cross-model test are illustrated in Table 3. The language-level performance reveals consistent performance across the four tested languages including Python, Java, C++, and C. The cross-model evaluation will assess whether the learned representation is associated with GPT-4 generation properties. Performance on GPT-3.5 Turbo, Claude 3.5 Sonnet, and Gemini Pro 1.5 is maintained within a comparatively small range, thus justifying the transferability of learned structural, behavioral, and stylometric properties. The real-student-code evaluation will further provide the evidence under realistic human-submitted code.

4.4 Obfuscation Robustness and Threshold Analysis

Robustness of the CodeGuardX algorithm is investigated by performing an analysis using increasing strength of source code transformations. Table 4 provides the values of metrics for the original program code and three different types of source code transformations, such as renaming of variables, deletion of comments, and insertion of dead code. No transformation corresponds to the original program code and is used as a baseline to evaluate the effect of each transformation method. It is evident from Table 4 that renaming of variables and deleting of comments do not affect F1 metrics significantly, indicating the independence of decision-making from identifiers and comments. Dead-code insertion similarly results in a small performance reduction because the added statements do not necessarily alter the underlying execution characteristics. When all three transformations are applied simultaneously, the reduction becomes more pronounced, reflecting the combined effect of modifying lexical, formatting, and source-level characteristics. Nevertheless, the retained performance demonstrates the contribution of the structural and behavioral representations to robust detection.

Table 4. Obfuscation robustness of CodeGuardX

Transformation	CodeGuardX F1 Metrics
No transformation	92.9%
Variable renaming	91.8%
Comment stripping	90.5%
Dead-code insertion	91.2%
All three combined	89.4%

The precision-recall curve shown in Figure 3 presents another metric for evaluating the operating point of the classifier. The presented framework achieves an AUC-PR equal to 0.96 and the optimal threshold for optimizing F1 score through validation is found as $\theta = 0.9797$. At this operating point, the manuscript states that the false-positive rate stays below 1.2% while recall is greater than 93%. In case of selecting a threshold equal to 0.90, the recall will be 96.1% whereas the precision will decrease to 81.3% and the false-positive rate will reach 4.7%.

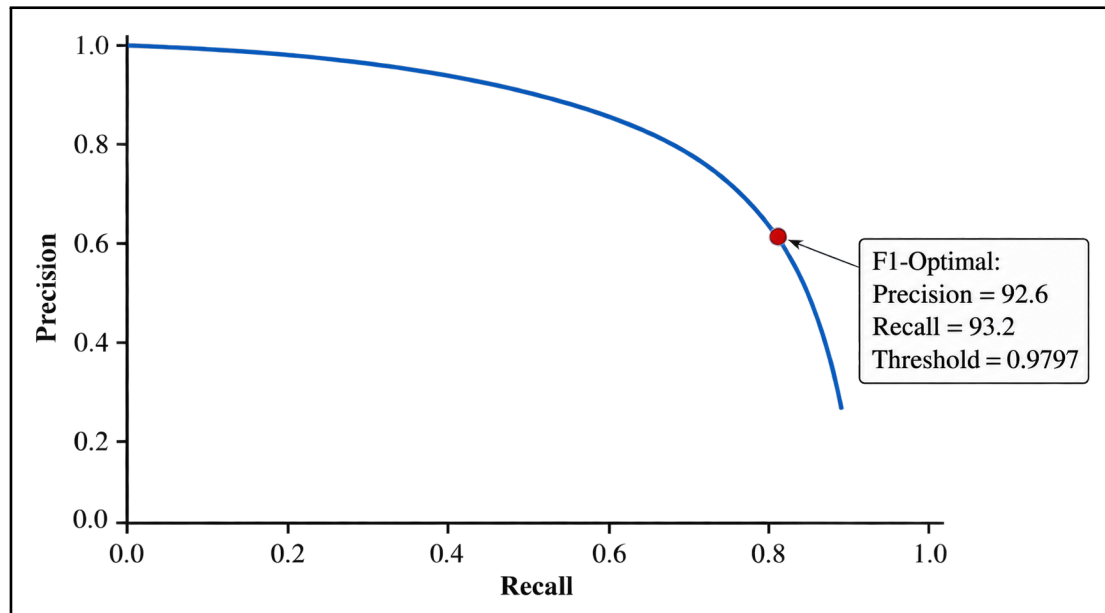


Figure 3. Precision-recall curve for CodeGuardX

5. Conclusion and Future Work

CodeGuardX provides a unified approach to AI-generated code detection through the combination of structural, behavioral, and stylistic clues into one classification system. The analysis of the AST provides insights about the program's structure and its complexity, while sandboxing offers semantic execution fingerprints for further analysis. Lexical and stylometric analysis also capture programming conventions, documentation practices, and source code regularities. Combining such an approach with XGBoost and probability calibration makes it possible to perform code-aware detection in different programming languages as well as transformations of source code. Interpretability is achieved with the help of feature-level analysis and decision-making under the probabilistic confidence measure. Future works will focus on expanding larger authentic code corpuses from students, enabling multi-file/project level analysis, improving behavioral analysis for more execution environments, and integrating new code generation paradigms emerging from newer LLMs. Domain adaptation and model updates will also be explored to ensure continued reliability in detecting programming generations as the methods evolve.

References

- [1] Mitchell, Eric, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. "Detectgpt: Zero-Shot Machine-Generated Text Detection using Probability Curvature." In

- International Conference on Machine Learning, Proceedings of Machine Learning Research, 2023, 24950-24962.
- [2] Pan, Wei Hung, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. "Assessing Ai Detectors in Identifying Ai-Generated Code: Implications for Education." In Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training, 2024, 1-11.
- [3] Farrelly, Tom, and Nick Baker. "Generative Artificial Intelligence: Implications and Considerations for Higher Education Practice." *Education Sciences*, 2023, vol. 13, no. 11: 1109.
- [4] Chan, Cecilia Ka Yuk. "A Comprehensive AI Policy Education Framework for University Teaching and Learning." *International Journal of Educational Technology in Higher Education*, 2023, vol. 20, no. 1: 38.
- [5] Otterbacher, Jahna. "Why Technical Solutions for Detecting AI-Generated Content in Research and Education are Insufficient." *Patterns*, 2023, vol. 4, no. 7.
- [6] Chaka, Chaka. "Detecting Ai Content in Responses Generated by Chatgpt, Youchat, and Chatsonic: The Case of Five Ai Content Detection Tools." *Journal of Applied Learning Teaching*, 2023, vol. 6, no. 2: 94-104.
- [7] Abuhamad, Mohammed, Ji-su Rhim, Tamer AbuHmed, Sana Ullah, Sanggil Kang, and DaeHun Nyang. "Code Authorship Identification using Convolutional Neural Networks." *Future Generation Computer Systems*, 2019, vol. 95: 104-115.
- [8] Kirchenbauer, John, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. "A Watermark for Large Language Models." In *International Conference on Machine Learning, Proceedings of Machine Learning Research*, 2023, 17061-17084.
- [9] Becker, Brett A., Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. "Programming is Hard-or At Least it used to Be: Educational Opportunities and Challenges of Ai Code Generation." In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V*, 2023, vol. 1: 500-506.

- [10] Gehrmann, Sebastian, Hendrik Strobelt, and Alexander M. Rush. "Gltr: Statistical Detection and Visualization of Generated Text." In Proceedings of the 57th Annual Meeting of The Association for Computational Linguistics: System Demonstrations, 2019, 111-116.
- [11] Chen, Tianqi, and Carlos Guestrin. "Xgboost: A Scalable Tree Boosting System." In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, 785-794.
- [12] Alsulami, Bander, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. "Source Code Authorship Attribution using Long Short-term Memory-based Networks." In European Symposium on Research in Computer Security, Cham: Springer International Publishing, 2017, 65-82.
- [13] Caliskan-Islam, Aylin, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. "De-Anonymizing Programmers Via Code Stylometry." In 24th USENIX Security Symposium, 2015, 255-270.
- [14] Boukili, G., El Garouani, S., and Riffi, J., "A Dataset for Human-Written and AI-Generated Code Source Classification," Data in Brief, 2025 vol. 65, 112527. Available: <https://data.mendeley.com/datasets/kjh95n54f8/1>
- [15] GitHub, "Github Copilot X: The AI-Powered Developer Experience," GitHub Research Report, 2023. Available: <https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/>