The Hidden DNA of LLM-Generated JavaScript: Structural Patterns Enable High-Accuracy Authorship Attribution

Norbert Tihanyi
Technology Innovation Institute
Abu Dhabi, United Arab Emirates
norbert.tihanyi@tii.ae

Bilel Cherif
Technology Innovation Institute
Abu Dhabi, United Arab Emirates
bilel.cherif@tii.ae

Richard A. Dubniczky
Eötvös Loránd University
Budapest, Hungary
richard@dubniczky.com

Mohamed Amine Ferrag
United Arab Emirates University
Al Ain, United Arab Emirates
mohamed.ferrag@uaeu.ac.ae

Tamás Bisztray*
University of Oslo
Oslo, Norway
tamasbi@ifi.uio.no

Abstract

In this paper, we present the first large-scale study exploring whether JavaScript code generated by Large Language Models (LLMs) can reveal which model produced it, enabling reliable authorship attribution and model fingerprinting. With the rapid rise of AI-generated code, attribution is playing a critical role in detecting vulnerabilities, flagging malicious content, and ensuring accountability. While AI-vs-human detection usually treats AI as a single category we show that individual LLMs leave unique stylistic signatures, even among models belonging to the same family or parameter size. To this end, we introduce `LLM-NodeJS`, a dataset of 50,000 Node.js back-end programs from 20 large language models. Each has four transformed variants, yielding 250,000 unique JavaScript samples and two additional representations (JSIR and AST) for diverse research applications. Using this dataset, we benchmark traditional machine learning classifiers against fine-tuned Transformer encoders and introduce CodeT5-JSA, a custom architecture derived from the 770M-parameter CodeT5 model with its decoder removed and a modified classification head. It achieves 95.8% accuracy on five-class attribution, 94.6% on ten-class, and 88.5% on twenty-class tasks, surpassing other tested models such as BERT, CodeBERT, and Longformer. We demonstrate that classifiers capture deeper stylistic regularities in program dataflow and structure, rather than relying on surface-level features. As a result, attribution remains effective even after mangling, comment removal, and heavy code transformations. To support open science and reproducibility, we release the `LLM-NodeJS` dataset, Google Colab training scripts, and all related materials on GitHub: https://github.com/LLM-NodeJS-dataset.

Keywords

AI-generated, code stylometry, LLM, authorship attribution

1 Introduction

Authorship attribution [22, 24], often referred to as stylometry, is the task of determining the author of a document, text, or piece of source code by analyzing stylistic and structural features. In most existing studies on human versus AI authorship, the AI-generated category is represented by a single model [9, 19, 29, 33] or, at best, a small set of models [8], thereby limiting the robustness of the resulting methods [31].

In human code attribution, typically only limited samples per author are available [1, 11], while *large language models (LLMs)* allow the generation of virtually unlimited, task-specific datasets for mapping their coding style. However, because many LLMs are trained on overlapping code corpora [25], they may inherit a broad mixture of stylistic and structural characteristics. With the growing presence of AI-generated content online, researchers have also raised concerns about recursive training and eventual style convergence among models [27, 35]. If such convergence occurs, model-level attribution, or even model family detection may become increasingly difficult.

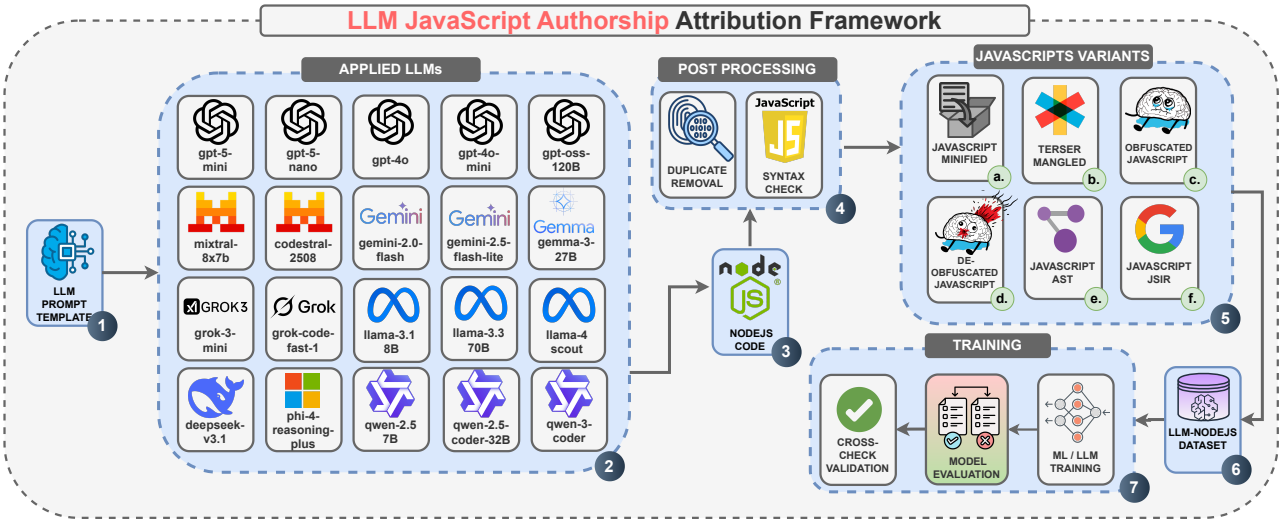
Big-tech leaders predict that AI will soon generate most of our source code [4, 5, 28]. Thus, robust attribution in LLMs is urgently needed for critical domains like software forensics, malware analysis, cybercrime investigations, and academic integrity [30]. Recent work by Bisztray et al. [6] shows that different LLMs leave distinguishable fingerprints when writing C code, achieving 94.5% accuracy in telling apart C code generated by five leading LLMs.

These findings highlight that attribution research must extend beyond the binary AI-vs-human setting to richer tasks such as model-level attribution, family-level attribution, co-authorship attribution, obfuscation-robust attribution, and malware/threat actor attribution. These tasks differ not only in dataset accessibility but also in their sensitivity to model evolution, as LLM fingerprints can change rapidly with the introduction of new models.

In [6], code stylometry focused on distinguishing C code among five LLM families (Claude-3.5, DeepSeek-v3.1, Gemini-2.5, GPT-4.1, LLaMA-3.3-70B). Extending this line of work, we focus on JavaScript—a flexible, dynamically typed language whose diverse styles and widespread tooling make attribution more challenging. We further investigate whether stylometric attribution is possible within the same model family (GPT-4o, GPT-4o-mini, GPT-4.1, GPT-5, gpt-oss) and explore scaling to larger classification tasks (10–20 classes). To this end, we introduce the first diverse AI-generated JavaScript dataset for authorship attribution and address the following research questions:

*Corresponding author

Figure 1: The seven-step methodology for LLM authorship attribution in JavaScript code.



RQ1: Which machine learning and transformer-based approaches achieve the most robust performance for black-box, model-level JavaScript source code attribution?

RQ2: What signals do deep-learning methods exploit to distinguish between different models?

To answer these questions, we follow the methodology shown in Figure 1. On a high level, our pipeline begins with prompt creation for back-end Node.js coding tasks and code generation from 20 state-of-the-art LLMs. The outputs are syntax-checked, deduplicated, and transformed into multiple JavaScript variants. These are then used to train both traditional *Machine Learning* (ML) models and transformer-based models for authorship attribution, which are subsequently evaluated and validated on an independent dataset. The detailed methodology is introduced in Section 3.

In this paper, we present the following key contributions:

- **LLM-NodeJS dataset:** We release a public dataset of 50,000 syntactically correct and executable Node.js back-end programs, expanded with four additional code variants (minified, mangled, obfuscated, deobfuscated), resulting in 250,000 JavaScript samples. Covering 20 state-of-the-art models and diverse web tasks, the dataset also includes two additional representations *JavaScript Intermediate Representation* (JSIR) and *Abstract Syntax Tree (AST)*, providing the first large-scale foundation for a wide range of JavaScript research, not limited to code authorship.
- **Classifier evaluation:** Using the LLM-NodeJS dataset, we benchmark traditional ML classifiers (Random Forest, XGBoost, KNN, Logistic Regression, Linear SVM) and fine-tuned transformers (i.e., BERT, CodeBERT, Longformer), including a custom *CodeT5-Authorship* model with a modified classification head and removed decoder layers. We show that JavaScript attribution is highly effective, achieving over 95% accuracy across five models within the same GPT family (GPT-4.1, GPT-4o, GPT-4o-mini, GPT-5-nano, GPT-5-mini, GPT-oss-120b) and nearly 90% for 20-class attribution.
- **Robustness to code variants:** Attribution remains accurate on minified (91%), terser-mangled (93%), and deobfuscated (85%) code, while obfuscation lowers accuracy due to token inflation. These results show that classifiers rely on deeper structural signals—such as AST structure and data-flow graphs—rather than shallow stylistic features explored in prior work
- **Open science:** To support reproducibility, we release the LLM-NodeJS dataset along with all Google Colab notebooks and training scripts on the project GitHub page: https://github.com/LLM-NodeJS-dataset.

Key Takeaway

The key insight of our work is that individual LLMs leave distinctive, structurally grounded fingerprints, enabling reliable model-level attribution beyond traditional AI-vs-human detection. **As AI-generated code becomes standard, the key question shifts from whether it was written by AI to which AI produced it**, highlighting the need to treat the "AI-generated" category as diverse rather than uniformly representing it with one model.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 outlines the methods used to construct the dataset and to train the classifiers. Section 4 presents experimental results, Section 5 discusses limitations and future work, while Section 6 concludes the paper.

2 Related Work

This section reviews authorship attribution methods for source code, organized by technique family. We then discuss benchmark datasets and tasks.

2.1 Authorship attribution architectures

2.1.1 Traditional Machine Learning. Authorship attribution has long been framed as supervised classification, mapping code features into vectors for classifiers such as logistic regression, naïve Bayes, SVMs, or tree ensembles [20]. These methods are cheap and interpretable but rely on feature engineering and lag behind deep learning in accuracy [6].

2.1.2 Deep Learning. Neural models learn directly from code, capturing stylistic hierarchies that manual features miss [42]. Abuhamad et al. achieved 92.3% accuracy on 8,903 authors in Google Code Jam with an RNN+Random Forest hybrid, also showing robustness to obfuscation [2]. Such models scale better to thousands of authors but require large labeled data.

2.1.3 Pre-trained and Generative Models. Pre-trained models (i.e., BERT, CodeBERT, RoBERTa) learn contextual embeddings from large code corpora, yielding state-of-the-art attribution with minimal manual features [13, 16, 41]. Fine-tuned CodeT5 variants achieve strong accuracy even across closely related authors [21].

Generative LLMs (e.g., GPT-4) support zero/few-shot attribution, reaching 65–69% on hundreds of authors [12]. Beyond human code, Bukhari et al. distinguished AI- from human-written C code at 92% [8], while Idialu et al. reported F1 ≈ 0.91 for GPT-4 vs. human Python [23], though most treat AI as a single-source class. Bisztray et al. extended this to the AI-vs-AI setting, with a modified *CodeT5* surpassing 95% accuracy across five LLMs from different model families [6].

Together, these results show both human and machine code leave detectable stylistic signatures.

2.2 Code Representation

2.2.1 Feature Engineering Approaches. Early work fingerprinted authors via lexical cues (keywords, punctuation), formatting habits, and software metrics [38]. Caliskan-Islam et al. introduced the *Code Stylometry Feature Set* (CSFS) [11]. While effective on human subjects, hand-crafted features degrade under obfuscation, adversarial edits, or IDE suggestions [26, 32, 36].

2.2.2 Graph and Structural Representations. Graphs capture structural and semantic style. ASTs encode syntax [17], while *Program Dependence Graphs (PDGs)* and *Control Flow Graphs (CFGs)* highlight dependencies and control flow. Bogomolov et al. used AST path–based classifiers (PbRF, PbNN) to reach 98% on curated sets but saw sharp drops on realistic corpora [7]. Graph-based features are more robust to formatting changes than stylometry [18, 20].

2.2.3 Obfuscated Code Attribution. Static stylistic features are fragile under obfuscation [26, 32, 36], but syntax and AST-based methods remain effective [11]. One Study achieved ~93% accuracy even on obfuscated code using deep learning [1].

2.2.4 Execution and Binary-Level Approaches. Dynamic analysis leverages runtime traces (system calls, memory, performance) to reveal algorithmic signatures resistant to superficial edits [40]. However, sandboxing is costly and mostly limited to malware forensics [14, 15, 37]. Attribution also extends to binaries: stylistic markers often survive compilation [20, 34], and despite compiler variability, accuracy remains above chance [15].

2.3 Datasets, Benchmarks and Tasks

Authorship attribution depends on diverse labeled corpora. Widely used datasets for human attribution include Google Code Jam, Codeforces, GitHub OSS, POJ-104, and BigCloneBench [1–3, 7, 11]. Yet, classifiers designed for natural language reportedly perform poorly on source code [31]. For AI vs. human attribution, most studies treat AI as a single-model category [8, 23]. Both studies focus exclusively on machine learning methods and report strong results on binary classification of AI vs. human-written C code (F1 ≈ 0.91 and Accuracy ≈ 92%). However, their evaluation is limited to GPT-4 as the sole source of AI-generated code. The only large-scale AI vs. AI study is by Bisztray et al. [6], who introduce the LLM-AuthorBench dataset (32k C programs). They achieve 95% accuracy in distinguishing five LLMs, with transformers consistently outperforming traditional ML methods.

3 Methodology

The first part of our methodology involves constructing a new supporting dataset, `LLM-NodeJS`, to address the lack of syntactically correct and labeled LLM-generated JavaScript code in existing literature.

3.1 Dataset Creation- The LLM-NodeJS dataset

We designed a set of 250 complex back-end Node.js tasks inspired by real-world scenarios encountered by professional developers. Formally, let

$$\mathcal{T} = \{T_1, T_2, \ldots, T_{250}\}$$

denote the set of tasks. Each task T_i is decomposed into a collection of smaller subtasks, $T_i = \{t_{i1}, t_{i2}, \ldots, t_{im_i}\}$, where m_i is the number of subtasks in task T_i. These subtasks represent functionally meaningful components (e.g., writing a GitHub uploader, creating a URL shortener service with a local SQLite database) and introduce substantial structural and stylistic diversity, ensuring that no two generated programs in the `LLM-NodeJS` dataset are identical.

All generated JavaScript implicitly uses strict mode and ES6 imports instead of `require`. For example, one task involved creating a JWT validator service that verifies RSA-signed tokens and returns a JSON status, comprising dozens of smaller subtasks. For each task $T_i \in \mathcal{T}$, we generated $k = 10$ independent implementations per model, yielding

$$|\mathcal{D}_m| = k \times |\mathcal{T}| = 10 \times 250 = 2{,}500$$

samples per model m. Across twenty LLMs spanning eight model families (c.f. Figure 1), this resulted in a total of 50,000 unique programs. Only JavaScript snippets that were syntactically correct and passed `"node -check"` command were included in the dataset. To assess the effect of obfuscation on attribution accuracy, we created six variants of each sample:

(1) **JavaScript Minified** – whitespace, comments, and formatting removed.
(2) **Terser Mangled** - identifiers shortened systematically via Terser.
(3) **Obfuscated JavaScript** - transformed using techniques such as control-flow flattening or string encoding.
(4) **De-Obfuscated JavaScript** - partially restored versions improving readability through identifier recovery and reformatting.
(5) **JavaScript JSIR** - intermediate representation for program analysis and attribution.

(6) **JavaScript AST** - structured tree representation capturing syntax and hierarchy.

The dataset includes the validated original programs plus four syntactically correct variants (minified, mangled, obfuscated, deobfuscated), comprising a total of 250,000 JavaScript samples, as well as two additional representations (JSIR and AST) that support a wide range of JavaScript analysis research.

3.2 Selected Models for Dataset Evaluation

The second step of our analysis focused on evaluating three different approaches: (i) traditional ML algorithms, (ii) transformer-based encoder-only models, and (iii) a custom model with a modified architecture. These methods were applied to the newly created dataset to assess their performance.

3.2.1 Traditional ML Algorithms. As a baseline, we evaluated five different traditional ML algorithms on stylometric representations of JavaScript code, including *k-NN, Logistic Regression, Random Forest, Linear SVM*, and *XGBoost*. The pipeline includes pre-processing, feature extraction, and training. Each program was vectorized using TF–IDF over tokens with a vocabulary cap of 400, emphasizing lexical and control-flow patterns known to capture stylistic cues [10, 11]. Performance was evaluated using accuracy, precision, F1 score, and training time. Since our dataset is balanced, recall does not provide additional information.

3.2.2 Transformer-based Models . We evaluate three encoder-only architectures [39], including BERT, CodeBERT, and Longformer. Encoder models are typically favored for classification, while decoder models are preferred for generation with long context.

All transformer fine-tuning experiments were optimized for NVIDIA A100 (80 GB) GPUs. Batch sizes, gradient accumulation, and mixed-precision settings were adjusted to fully utilize available memory while avoiding overflow. In particular, training was performed in `bf16` precision with fused AdamW optimization and cosine restarts for learning rate scheduling. For BERT, we used an effective batch size of 32 (achieved through 16 × 2 gradient accumulation), whereas Longformer leveraged its larger context window with a batch size of 32 and no accumulation.

3.2.3 Custom CodeT5-JSA Architecture. In [6], the authors transformed CodeT5+ (770M parameters) into an encoder-only model by removing the decoder layer and attaching a lightweight classification head (H1) consists of:

`Linear layer` → `GELU` → `Dropout(0.2)` → `Linear layer`

For the ten and twenty class scenario the above setting has started dropping accuracy, therefore here we revised the classification head:

`Dropout(0.2)` → `Linear layer` → `Tanh` → `Dropout(0.2)` → `Linear layer`

In both settings the last layer is followed by `Softmax` activation to produce class probabilities. Replacing the GELU activation layer with `Tanh` and introducing an initial dropout layer improved stability and overall accuracy. For example on the deobfuscated twenty class scenario accuracy jumped from 72% to 79.94%.

In terms of optimisation, if full FP32 is used, the training time is 958 minutes, whereas switching to bfloat16 reduces the 20-class training time to 111 minutes, with only a 1–3% accuracy drop, which is still acceptable given the substantial training time gain.

4 Experimental Results

To answer our research questions, we conducted our experiments in two stages. The first is geared towards research question 1, where we investigate which machine learning and transformer-based architectures perform best in attributing JavaScript code (along with the obfuscated versions) to the original generator model.

Next, we aim to investigate if attribution performance holds even when there are more classes, and whether there are certain models, that are too similar to each-other to be accurately distinguished.

4.1 Dataset diversity analysis

Before starting the classification process, we assessed the dataset's diversity. As discussed in the methodology section, each task is posed ten times to the same model, enabling us to measure variability in the generated JavaScript programs. To quantify this diversity—both within and across model families—we used three *CodeBLEU* components: **N-gram match** (lexical overlap on tokens, identifiers, and keywords), **Syntax match** (structural overlap via ASTs), and **Dataflow match** (semantic overlap via variable dependencies and data/control flow, e.g., Jaccard over dependency graphs). Table 1 reports *median* similarities for GPT-4o, GPT-5, and Gemini. We summarize intra–vs.–inter model separation by the average gap Δ ($\sum$ intra-model medians - $\sum$ inter-model medians). *Lower similarity indicate greater diversity in the code, whereas for* Δ *higher values reflect stronger discriminative power between models.*

Table 1: Median CodeBLEU Similarities and Intra–Inter Gaps for JavaScript Programs

Model Pair	N-gram Match	Syntax Match	Dataflow Match
Intra-Model Comparisons			
Gemini × Gemini	0.29	0.58	0.40
GPT-4o × GPT-4o	0.29	0.60	0.46
GPT-5 × GPT-5	0.16	0.60	0.35
Inter-Model Comparisons			
Gemini × GPT-4o	0.09	0.49	0.31
Gemini × GPT-5	0.03	0.43	0.16
GPT-4o × GPT-5	0.01	0.38	0.11
Intra-Model Avg.	0.25	0.59	0.40
Inter-Model Avg.	0.04	0.43	0.19
Gap (Δ)	**0.20**	**0.16**	**0.21**

4.1.1 Key Results. Dataflow similarity provides the strongest discrimination between same and cross-model pairs, representing the clearest model-specific signal at the semantic level. Programs compared within the same model are semantically aligned (e.g., GPT-4o × GPT-4o median 0.46), whereas cross-family pairs can be markedly divergent (GPT-4o × GPT-5 median 0.11). *N-gram* similarity shows a comparable gap but is sensitive to surface-level edits.

Syntax yields the smallest gap, not because it lacks discriminative power, but because structural patterns are relatively stable across tasks.

4.1.2 Implications for attribution. These results indicate that model attribution will be most reliable when grounded in deeper semantic and structural signals rather than surface-level cues. Even when stylistic features are obfuscated or lexical patterns altered, the underlying logic and structural organization of code often remain detectable. *Dataflow similarity* thus provides the most obfuscation-resilient indicator of model identity, capturing consistent reasoning and dependency patterns. *Syntax similarity* offers complementary stability, reflecting structural regularities that persist despite superficial transformations, while *n-gram similarity*—although highly discriminative in clean conditions—degrades rapidly under lexical variation.

4.1.3 Token length distribution. Figure 2 shows the distribution of tokenized sequence lengths in our dataset, which informs architecture choice: short to medium-length contexts fit well within encoder-only models.

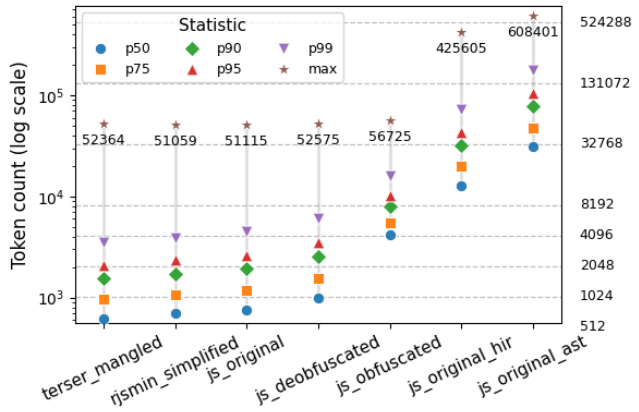


Figure 2: Tokenized Dataset Statistics, where PX marks that X percentage of samples fall below that token count.

In practice, trade-offs between context length, attention mechanism, and pretraining objectives determine how effectively stylistic signals are captured. Prior work [6] finds that representation quality can outweigh raw context window size. For example, a 512-token modified CodeT5 outperforms Longformer (which has a larger context-window).

4.2 JavaScript Authorship Attribution

Here, we present our key findings across different scenarios: (i) 5-class attribution, (ii) 10-class attribution, and (iii) 20-class attribution.

4.2.1 Five-Class Attribution for Different Model Families. We first replicate the setup of [6], where the base BERT model achieved an accuracy of 0.92 on a five-way attribution task assigning C program code to one of five generator LLMs from distinct model families.

Our selected models are `gpt-5-mini`, `gemini-2.5-flash-lite`, `qwen-2.5-coder-32b`, `llama-3.3-70b`, and `deepseek-v3.1`. This dataset consists of 12,500 JavaScript programs ($5 \times 2{,}500$), split into 80% training and 20% validation. In this setting, BERT_B achieved 97.9% accuracy, F1, and precision/recall, confirming that LLMs from different families exhibit distinct and learnable stylistic signatures in JavaScript code generation, just like in C. Since we already achieved very high accuracy in this scenario, we shifted our focus to a more challenging problem, where all five models belong to the same family.

4.2.2 Five-Class Attribution for Same Model Families. Next, we test five-class attribution with GPT variants only. Here, BERT_B still attains 90.2% accuracy, indicating that even closely related LLMs leave detectable stylistic fingerprints. For a complete comparison of model performance on this task, see the summary Table 2. Because we observe an approximate 8% drop in accuracy compared to the different model family classification task, we evaluated a range of models to determine the best achievable performance.

Figure 3 suggests that most errors arise from confusion between closely related variants of the same release (like GPT-4o vs. GPT-4o-mini, or among GPT-5 variants), while architecturally distinct versions can be distinguished more easily.

Variants of the same model likely share similar pre-training corpora and compute budgets. We *hypothesize* that their *Supervised Fine-Tuning* (SFT) and *Reinforcement Learning from Human Feedback* (RLHF) pipelines are also largely aligned. For example, GPT-4o and GPT-4o-mini may have been trained primarily on mid-2023 data, while GPT-5 variants likely incorporate early-2025 GitHub snapshots and more diverse coding examples, along with refined SFT and RLHF stages.

As AI-generated code becomes increasingly widespread, it remains uncertain whether model styles will converge into a uniform pattern or remain distinguishable through continued human-engineering interventions. We predict that as systems approach AGI, models will co-adapt and self-optimize, gradually blurring stylistic boundaries. Nonetheless, architectural and tokenization

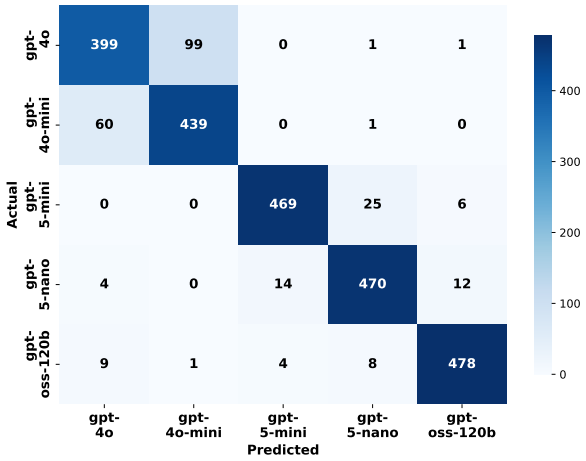


Figure 3: Confusion matrix for five-class attribution within the GPT family using BERT_B on js_original

Table 2: 5/10/20 class attribution by JavaScript Variant (different model families)

Model Name	Type	5-Class (GPT family only)				10-Class				20-Class				Parameters
		Acc(%)	Prec(%)	F1	Time	Acc(%)	Prec(%)	F1	Time	Acc(%)	Prec(%)	F1	Time	
ORIGINAL JAVASCRIPT RESULTS														
CodeT5-JSA	LLM	95.76	95.89	95.76	28:15	93.60	93.76	93.58	55:58	85.95	86.52	85:68	111:29	Layers:24, Token:512
Longformer$_B$	LLM	94.28	94.38	94.28	127:23	93.62	92.79	92.65	262:25	86.79	86.95	86.69	507:52	Layers:12, Token:2048
CodeBERT	LLM	94.28	94.40	94.27	07:00	91.88	92.11	91.86	13:15	85.87	86.22	85.74	25:08	Layers:12, Token:512
BERT$_B$	LLM	90.24	90.75	90.21	06:54	89.56	89.76	89.55	13:07	81.99	82.35	81.93	24:55	Layers:12, Token:512
XGBoost	ML	88.72	88.76	88.74	00:49	86.60	86.67	86.46	02:23	75.78	75.73	75.62	08:31	Estimators: 400
Random Forest	ML	86.24	86.27	86.25	00:35	81.20	81.85	80.75	01:15	68.25	68.28	67.19	02:56	Trees: 400
Linear SVM	ML	82.64	82.60	82.60	00:01	76.44	75.83	75.83	00:02	58.97	58.17	57.73	00:12	Max Iteration: 2000
Logistic Regression	ML	80.40	80.44	80.40	00:10	73.82	73.19	73.33	00:14	56.43	55.61	55.71	00:15	Max Iteration: 2000
KNN	ML	71.32	71.41	71.21	00:01	60.74	61.49	59.99	00:01	48.03	49.70	46.34	00:01	Neighbors: 5
MANGLED JAVASCRIPT RESULTS														
CodeT5-JSA	LLM	90.02	90.2	90.03	28:16	89.68	89.85	89.64	55:58	83.61	83.74	83.41	112:02	Layers:24, Token:512
Longformer$_B$	LLM	89.56	89.67	89.56	127:46	89.58	89.87	89.69	253:41	80.78	81.02	80.56	507:23	Layers:12, Token:2048
CodeBERT	LLM	89.28	89.54	89.26	06:55	89.36	89.89	89.40	13:00	81.28	81.49	81.13	25:26	Layers:12, Token:512
BERT$_B$	LLM	86.44	86.81	86.40	06:34	86.24	86.46	86.30	12:58	76.61	76.70	76.47	25:07	Layers:12, Token:512
XGBoost	ML	86.48	86.51	86.49	00:51	84.80	84.87	84.68	150.54	73.29	73.08	73.04	595.40	Estimators: 400
Random Forest	ML	83.04	83.05	82.99	00:31	79.72	80.55	79.31	68.30	66.93	66.88	65.98	168.78	Trees: 400
Linear SVM	ML	79.08	79.03	78.97	00:01	74.78	74.59	74.31	2.67	55.77	54.79	54.27	15.28	Max Iteration: 2000
Logistic Regression	ML	76.88	76.80	76.79	00:11	71.60	71.27	71.17	16.60	53.68	52.82	52.91	21.45	Max Iteration: 2000
KNN	ML	67.84	67.92	67.55	00:00	57.74	58.77	56.83	0.01	45.39	45.91	43.20	0.01	Neighbors: 5
DEOBFUSCATED JAVASCRIPT RESULTS														
CodeT5-JSA	LLM	89.92	90.40	89.86	28:12	89.78	90.00	89.78	56:08	79.94	79.79	79.60	112:44	Layers:24, Token:512
Longformer$_B$	LLM	88.32	88.42	88.32	127:14	88. 76	89.14	88.84	254:13	78.15	78.02	77.87	508:35	Layers:12, Token:2048
CodeBERT	LLM	86.76	86.78	86.74	06:41	84.50	84.60	84.33	13:17	75.46	75.33	75.03	25:16	Layers:12, Token:512
BERT$_B$	LLM	84.08	84.26	84.08	06:55	82.58	82.84	82.63	12:34	72.61	72.64	72.40	24:44	Layers:12, Token:512
XGBoost	ML	86.48	86.53	86.49	00:51	83.16	83.28	83.00	156.11	72.98	72.88	72.80	504.17	Estimators: 400
Random Forest	ML	83.44	83.47	83.41	00:30	79.22	80.06	78.78	65.69	66.35	66.51	65.42	166.82	Trees: 400
Linear SVM	ML	78.64	78.58	78.52	00:01	73.56	73.34	73.03	2.63	54.76	53.39	53.12	13.91	Max Iteration: 200
Logistic Regression	ML	77.76	77.72	77.67	00:08	71.40	71.10	70.96	14.59	53.43	52.44	52.66	19.42	Max Iteration: 2000
KNN	ML	70.28	70.30	70.06	00:00	60.40	61.47	59.64	0.01	47.63	48.05	45.63	0.01	Neighbors: 5

differences are expected to preserve some degree of distinguishability, if not through code style, then through higher-level moral, ethical, or policy biases imparted by their creators.

4.2.3 Ten and Twenty-Class Attribution. As the attribution task expands from five to ten and twenty classes, overall accuracy decreases across all models. Transformer-based encoders (`CodeT5-JSA`, `Longformer`$_B$, `CodeBERT`) consistently outperform classical ML baselines across all settings.

Among all models, `CodeT5-JSA` shows the best overall performance, maintaining 94% accuracy in the ten-class setup and above 88% in the twenty-class case for the original JavaScript. `Longformer`$_B$ achieves comparable results but at substantially higher computational cost. Traditional ML classifiers such as `XGBoost` and `Random Forest` remain competitive yet degrade more rapidly as class count increases, and linear methods (`SVM`, `LogReg`) fall below 60% at twenty classes. `XGBoost` consistently outperformed other ML models while remaining efficient, illustrating a favorable balance between accuracy and runtime. Surprisingly, on mangled and deobfuscated JavaScript code `XGBoost` managed to outperform the `Bert`$_B$ model on several occasions.

4.2.4 Different JavaScript Variants. To assess robustness, we evaluate attribution on terser-mangled and deobfuscated code (Table 2). Since performance on mangled and minified JavaScript was nearly identical, we report results for the mangled variant only. Although these transformations eliminate key stylistic cues—such as variable and function names—accuracy remains high. This indicates that surface-level features like naming patterns or token frequencies are not the primary drivers of attribution. Instead, the classifiers exploit deeper structural and semantic regularities that persist through code transformations. When code is deobfuscated, accuracy recovers substantially, confirming that restoring syntactic structure is sufficient to reestablish most discriminative patterns.

These results extend the observations of [6] from C to JavaScript, demonstrating that attribution remains feasible both when the number of models scale, and with different transformations. Even as lexical information is deliberately suppressed, model-specific signatures persist through structural and dataflow patterns. Overall, these findings confirm that attribution grounded in semantic and syntactic organization remains tractable and robust even under aggressive obfuscation and fine-grained class expansion.

4.3 Cross-check Validation

All JavaScript samples in the `LLM-NodeJS` dataset were generated through OpenRouter.ai. To ensure that this service does not watermark its outputs, we created a new dataset of 500 JavaScript samples corresponding to completely unseen tasks. These samples, which include Node.js–specific programs, were generated using the original OpenAI API—entirely independent from the OpenRouter.ai service—and do not overlap with the 50,000 samples used for training or evaluation.

We re-evaluated the five-class classifier using `CodeBERT` on the original JavaScript variant. It maintained high attribution performance, with only a 1–2% drop compared to the results observed in our experiments. This confirms that attribution generalizes well to previously unseen code distributions. The validation dataset is publicly released as `CROSS_CHECK_DATASET.json` on GitHub.

5 Limitations and Future Work

This study leaves several open questions that point to promising directions for future research:

- **Model scale and architectures:** Our experiments were limited to moderate-sized Transformer encoders such as BERT and CodeT5. Larger or extended-context architectures capable of processing JSIR and AST variants were not explored due to computational constraints. Future work should examine how model size, context length, and decoder-only architectures affect attribution accuracy. We did not conduct experiments on AST and JSIR representations, but made these variants available for the research community.
- **Language and data diversity:** The dataset is restricted to JavaScript, whereas prior work examined C [6]. Extending attribution to other languages (C++, Rust, Python, Java) or to cross-language setups—training on one language and testing on another—would help determine the generality of model fingerprints. Broader datasets covering varied coding domains and prompt styles would also strengthen external validity, as we only focused on web-deveopment.
- **Attribution generalization:** In the multi-class setting, the classifier assumes that all samples originate from the 20 studied LLMs. Future work should explore open-set recognition and continual learning to handle unseen models, as well as temporal drift studies to assess the stability of fingerprints as LLMs evolve.
- **Interpretability and robustness.** Attribution robustness should be tested not only under simple code transformations, but under adversarial modifications like transpilation, control-flow rewriting.
- **Ethical and practical considerations:** Attribution systems may be vulnerable to deliberate "style spoofing" or adversarial evasion. Future work should investigate these risks and develop safeguards for responsible use in code forensics and AI provenance detection.

6 Conclusion

We introduced the `LLM-NodeJS` dataset, a corpus of 250,000 validated Node.js programs comprising 50,000 original JavaScript programs, each with four variants (minified, terser-mangled, obfuscated, and deobfuscated), along with additional AST and JSIR representations for every sample. Using this dataset, we compared classical ML algorithms with fine-tuned Transformer encoders (e.g., BERT, CodeBERT, and Longformer) as well as a custom modified architecture based on the 770M-parameter CodeT5 model.

- **RQ1: Can we reliably attribute JavaScript programs to their generating LLM?**
  Answer: Yes. On original JavaScript, CodeT5-JSA attains **95.8%** accuracy in the five-way GPT-only setting, **94.6%** on the ten-class task, and **88.5%** on twenty classes (Table 2). Performance on terser-mangled, minified, and deobfuscated code remains close (typically single-digit drops), indicating that attribution does not depend primarily on surface naming or formatting.

- **RQ2: What signals do the models rely on?**
  Answer: Deeper structural and semantic cues are just as important as style. Our similarity analysis shows that *dataflow* provides the clearest same-vs.-cross model separation, along with *n-gram*, however, the absence of the latter still allows accurate attribution. These explain the robustness of attribution on mangled/minified code and the recovery after deobfuscation: classifiers exploit persistent logic, dependency structure, and control/dataflow rather than superficial lexical features.

Overall, our results demonstrate that LLM-generated JavaScript carries persistent, model-specific fingerprints that remain detectable across families, class scales (ten and twenty classes), and common code transformations. While accuracy naturally declines as stylistic boundaries narrow, semantically grounded signals continue to support reliable attribution.

References

[1] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. 2018. Large-Scale and Language-Oblivious Code Authorship Identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 101–114. doi:10.1145/3243734.3243738

[2] Mohammed Abuhamad, Tamer Abuhmed, David Mohaisen, and Daehun Nyang. 2021. Large-scale and Robust Code Authorship Identification with Deep Feature Learning. *ACM Trans. Priv. Secur.* 24, 4 (July 2021), 23:1–23:35. doi:10.1145/3461666

[3] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source Code Authorship Attribution Using Long Short-Term Memory Based Networks. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 65–82. doi:10.1007/978-3-319-66402-6_6

[4] Sam Altman. 2025. Sam Altman says AI is doing 50% of coding in companies. https://www.indiatoday.in/technology/news/story/sam-altman-says-

ai-is-doing-50-percent-coding-in-companies-warns-students-about-career-choices-2696937-2025-03-21. Accessed: 2025-09-09.

[5] Dario Amodei. 2025. Vibe Coding Is Coming for Engineering Jobs. https://www.wired.com/story/vibe-coding-engineering-apocalypse. Accessed: 2025-09-09.

[6] Tamas Bisztray, Bilel Cherif, Richard A Dubniczky, Nils Gruschka, Bertalan Borsos, Mohamed Amine Ferrag, Attila Kovacs, Vasileios Mavroeidis, and Norbert Tihanyi. 2025. I Know Which LLM Wrote Your Code Last Summer: LLM generated Code Stylometry for Authorship Attribution. *arXiv preprint arXiv:2506.17323* (2025).

[7] Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, and Timofey Bryksin. 2021. Authorship Attribution of Source Code: A Language-Agnostic Approach and Applicability in Software Engineering. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. Association for Computing Machinery, Athens, Greece, 932–944. doi:10.1145/3468264.3468606

[8] Sufiyan Bukhari, Benjamin Tan, and Lorenzo De Carli. 2023. Distinguishing AI- and Human-Generated Code: A Case Study. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (Copenhagen, Denmark) *(SCORED '23)*. Association for Computing Machinery, New York, NY, USA, 17–25. doi:10.1145/3605770.3625215

[9] Luana Bulla, Alessandro Midolo, Misael Mongiovì, and Emiliano Tramontana. 2024. EX-CODE: A Robust and Explainable Model to Detect AI-Generated Code. *Information* 15, 12 (2024). doi:10.3390/info15120819

[10] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. 2018. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, 15 pages. doi:10.14722/ndss.2018.23304

[11] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 255–270.

[12] Soohyeon Choi, Yong Kiam Tan, Mark Huasong Meng, Mohamed Ragab, Soumik Mondal, David Mohaisen, and Khin Mi Mi Aung. 2025. I Can Find You in Seconds! Leveraging Large Language Models for Code Authorship Attribution. doi:10.48550/arXiv.2501.08165 arXiv:2501.08165 [cs] version: 1.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. doi:10.48550/arXiv.2002.08155 arXiv:2002.08155 [cs].

[14] Alberto Ferrante, Eric Medvet, Francesco Mercaldo, Jelena Milosevic, and Corrado Aaron Visaggio. 2016. Spotting the Malicious Moment: Characterizing Malware Behavior Using Dynamic Features. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, Salzburg, Austria, 372–381. doi:10.1109/ARES.2016.70

[15] Jason Gray, Daniele Sgandurra, Lorenzo Cavallaro, and Jorge Blasco Alis. 2024. Identifying Authorship in Malicious Binaries: Features, Challenges & Datasets. *ACM Comput. Surv.* 56, 8, Article 212 (April 2024), 36 pages. doi:10.1145/3653973

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. doi:10.48550/arXiv.2009.08366 arXiv:2009.08366 [cs].

[17] Dixiao Guo, Anmin Zhou, Liang Liu, Shan Liao, and Lei Zhang. 2022. A Method of Source Code Authorship Attribution Based on Graph Neural Network. In *Proceedings of 2021 Chinese Intelligent Automation Conference*, Zhidong Deng (Ed.). Springer Singapore, Singapore, 645–657.

[18] Dixiao Guo, Anmin Zhou, Liang Liu, Shan Liao, and Lei Zhang. 2022. A Method of Source Code Authorship Attribution Based on Graph Neural Network. In *Proceedings of 2021 Chinese Intelligent Automation Conference*, Zhidong Deng (Ed.). Springer, Singapore, 645–657. doi:10.1007/978-981-16-6372-7_70

[19] Andrea Gurioli, Maurizio Gabbrielli, and Stefano Zacchiroli. 2025. Is This You, LLM? Recognizing AI-written Programs with Multilingual Code Stylometry . In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 394–405. doi:10.1109/SANER64311.2025.00044

[20] Xie He, Arash Habibi Lashkari, Nikhill Vombatkere, and Dilli Prasad Sharma. 2024. Authorship Attribution Methods, Challenges, and Future Research Directions: A Comprehensive Survey. *Information* 15, 3 (March 2024), 131. doi:10.3390/info15030131 Number: 3 Publisher: Multidisciplinary Digital Publishing Institute.

[21] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, Melbourne, Australia, 328–339. doi:10.18653/v1/P18-1031

[22] Baixiang Huang, Canyu Chen, and Kai Shu. 2025. Authorship Attribution in the Era of LLMs: Problems, Methodologies, and Challenges. doi:10.48550/arXiv.2408.08946 arXiv:2408.08946 [cs] version: 2.

[23] Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M Atlee, and Mei Nagappan. 2024. Whodunit: Classifying Code as Human Authored or GPT-4 generated-A case study on CodeChef problems. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 394–406.

[24] Patrick Juola. 2006. Authorship attribution. *Found. Trends Inf. Retr.* 1, 3 (Dec. 2006), 233–334. doi:10.1561/1500000005

[25] Kocetkov, Denis and Kumar, Divyanshu and Shoeb, Abu Awal Md and others. 2022. The Stack: 3 TB of Permissively Licensed Source Code. Hugging Face Dataset. https://huggingface.co/datasets/bigcode/the-stack Accessed: 2025-09-09.

[26] Christopher McKnight and Ian Goldberg. 2018. Style Counsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry. In *ACM Workshop on Privacy in the Electronic Society (WPES)*. ACM, Toronto, ON, Canada, 138–142.

[27] Melanie Mitchell. 2023. Data Contamination and Model Collapse in AI Systems. *AI Magazine* 44, 4 (2023), 358–364. doi:10.1002/aaai.12123

[28] Satya Nadella. 2025. As much as 30% of Microsoft code now written by AI, CEO Satya Nadella says. https://nypost.com/2025/04/30/business/microsoft-ceo-satya-nadella-says-30-of-code-now-written-by-ai. Accessed: 2025-09-09.

[29] Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2024. GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT. *Journal of Systems and Software* 214 (Aug. 2024), 112059. doi:10.1016/j.jss.2024.112059

[30] Paul W. Oman and Curtis R. Cook. 1989. Programming Style Authorship Analysis. In *Proceedings of the 17th Annual ACM Computer Science Conference (CSC '89)*. Association for Computing Machinery, New York, NY, USA, 320–326.

[31] Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. 2024. 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*. 1–11.

[32] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 479–496.

[33] Musfiqur Rahman, SayedHassan Khatoonabadi, Ahmad Abdellatif, and Emad Shihab. 2024. Automatic Detection of LLM-generated Code: A Case Study of Claude 3 Haiku. doi:10.48550/arXiv.2409.01382 arXiv:2409.01382 [cs] version: 1.

[34] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. 2011. Who Wrote This Code? Identifying the Authors of Program Binaries. In *Computer Security – ESORICS 2011*, Vijay Atluri and Claudia Diaz (Eds.). Springer, Berlin, Heidelberg, 172–189. doi:10.1007/978-3-642-23822-2_10

[35] Ilia Shumailov, Yarin Gal, Vitaly Shmatikov, Nicolas Papernot, and Ross Anderson. 2023. The Curse of Recursion: Training on Generated Data Makes Models Forget. *Nature Machine Intelligence* 5 (2023), 1411–1422. doi:10.1038/s42256-023-00782-5

[36] Lucy Simko, Luke Zettlemoyer, and Tadayoshi Kohno. 2018. Recognizing and Imitating Programmer Style: Adversaries in Program Authorship Attribution. In *Proceedings on Privacy Enhancing Technologies (PETS)*, Vol. 2018. 127–144. doi:10.1515/popets-2018-0007

[37] Qige Song, Yongzheng Zhang, Linshu Ouyang, and Yige Chen. 2022. BinMLM: Binary Authorship Verification with Flow-aware Mixture-of-Shared Language Model . In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 1023–1033. doi:10.1109/SANER53432.2022.00120

[38] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. 2024. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering* 30, 2 (Dec. 2024), 47. doi:10.1007/s10664-024-10590-1

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

[40] Ningfei Wang, Shouling Ji, and Ting Wang. 2018. Integration of Static and Dynamic Code Stylometry Analysis for Programmer De-anonymization. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security (AISec '18)*. Association for Computing Machinery, New York, NY, USA, 74–84. doi:10.1145/3270101.3270110

[41] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. doi:10.48550/arXiv.2109.00859 arXiv:2109.00859 [cs].

[42] Sarim Zafar, Muhammad Usman Sarwar, Saeed Salem, and Muhammad Zubair Malik. 2020. Language and Obfuscation Oblivious Source Code Authorship Attribution. *IEEE Access* 8 (2020), 197581–197596. doi:10.1109/ACCESS.2020.3034932