Distinguishing AI- and Human-Generated Code: a Case Study

Sufiyan Bukhari
University of Calgary
Calgary, AB, Canada
sufiyanahmed.bukhari@ucalgary.ca

Benjamin Tan
University of Calgary
Calgary, AB, Canada
benjamin.tan1@ucalgary.ca

Lorenzo De Carli
University of Calgary
Calgary, AB, Canada
lorenzo.decarli@ucalgary.ca

ABSTRACT

While the use of AI assistants for code generation has the potential to revolutionize the way software is produced, assistants may generate insecure code, either by accident or as a result of poisoning attacks. They may also inadvertently violate copyright laws by mimicking code protected by restrictive licenses.

We argue for the importance of tracking the *provenance* of AI-generated code in the software supply chain, so that adequate controls can be put in place to mitigate risks. For that, it is necessary to have techniques that can distinguish between human- and AI-generate code, and we conduct a case study in regards to whether such techniques can reliably work. We evaluate the effectiveness of lexical and syntactic features for distinguishing AI- and human-generated code on a standardized task. Results show accuracy up to 92%, suggesting that the problem deserves further investigation.

CCS CONCEPTS

• **Security and privacy → Software and application security**;
• **Software and its engineering → Risk management**.

KEYWORDS

supply chain security, AI code generation, program analysis

ACM Reference Format:
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 (SCORED '23), November 30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3605770.3625215

1 INTRODUCTION

Modern software is written compositionally by integrating software packages into a main codebase. Package managers—such as npm—automate dependency management, making it trivial to import third-party external code. The ease of integrating this external code enormously simplifies software development but complicates *software supply chain security*, i.e., the tasks of ensuring that externally-created software, when used in one's own code, does not generate security risks. As the amount of external code imported into a project grows, so does its attack surface [31].

SCORED '23, November 30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0263-1/23/11…$15.00
https://doi.org/10.1145/3605770.3625215

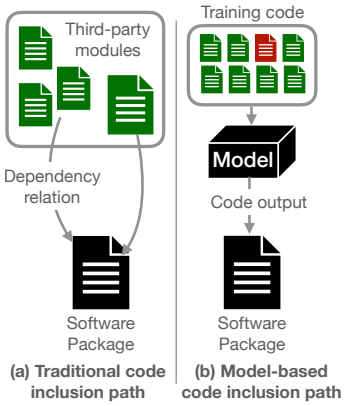

Figure 1: Traditional and model-based code inclusion paths

An emerging and poorly understood threat is the increasing use of AI assistants for code generation. These assistants are based on large language models (LLMs), such as OpenAI Codex [20] and SalesForce's CodeGen [24], which can readily generate code given a high-level outline, such as a function name or comment. As use of AI code assistants increases, so does the amount of model-generated code in software projects (a recent study suggests that, in files written with model assistance, up to 40% of code may be machine-generated [17].) Models are trained on a code dataset whose characteristics indirectly influence the generated code. Thus, from the point of view of the software supply chain, models are a novel *code inclusion path*, unlike traditional mechanisms based on explicit inclusion primitives such as import and forking (Figure 1).

Model-driven code generation is powerful and disruptive, potentially freeing programmers from low-level code development and enabling them to focus on high-level tasks. However, models also introduce potential security problems. As they learn to generate code from unvetted human-written samples, they are exposed to supply chain risks, as they ingest enormous amounts of potentially insecure source code as their training set. Evidence suggests that models may learn to generate insecure or incorrect code and generally perform worse—in terms of writing secure code—than humans [27, 28]. They may also be the target of *data poisoning*: an attacker may be able to affect model behaviors in specific situations by poisoning the training set with crafted software samples [30]. Finally, AI-generated code may cause inadvertent copyright violations [14]. Thus, it is important for projects to be able to determine whether their upstream dependencies include AI-generated code.

In this paper, we begin to consider the issue of restoring *code provenance* by determining whether a code segment was generated by an AI tool or a human. We expect that in the long term, as AI code assistants are further integrated into the supply chain, provenance information may be incorporated into the code generation process.

However, we also expect that AI code identification will constitute an important stop-gap measure until such processes solidify.

Our work is an initial exploration, asking whether simple lexical and syntactic code features (such as ones employed for exploit detection and code stylometry) can be used to distinguish human- and AI-generated samples *in ideal conditions*, i.e., on a cleanly labeled dataset and in the absence of confounding factors. We believe the answer can point to whether the problem is solvable and provide an initial assessment of the accuracy of AI code identification.

To answer our research question, we build a lexical and syntactic feature extraction pipeline, we apply it to a subset of the NYU Lost-at-C dataset [29] (which includes human- and AI-generated code samples on a standardized task), and we evaluate the effectiveness of a number of classification algorithms in distinguishing samples. Results show accuracy up to **92%** depending on the specific extraction/classification process. These results suggest that tracking AI code in the supply chain may be feasible and warrants further investigation.

Total Count	Human (Control)	AI (Autopilot)
Code Files	28	30
Functions	215	327
Lines of Code	3183	6716
Lines of Comments	445	337
Blank Lines	368	434

Table 1: Characterization of dataset

2 BACKGROUND

2.1 Supply Chain Security

Current software artifacts, rather than being monolithic, tend to incorporate a variety of external code as a set of *dependencies*. Typically, such code implements standardized functionality that would be time-consuming/complex to implement correctly, such as network communication, file parsing, ML/AI model building, etc. External dependencies are normally managed with the aid of automated tools—package managers—which take care of importing and updating external packages. This approach to software development has significant benefits in terms of time-to-market and costs. However, the automated nature of dependency management makes vetting dependencies non-trivial, opening the way to supply-chain attacks. In such an attack, a software project is compromised by an attacker who injects malicious code into one of its dependencies. Supply chain attacks have significant economic impact [15, 25], and the problem of supply chain security has attracted the attention of academia, industry, and government entities. Existing efforts involving producing best-practices [2, 8], detecting and containing attacks [7, 18, 35], and tracking provenance of dependency code [1].

2.2 AI Code Assistants

The term "AI code assistant" generally refers to an AI-based tool used to generate code on demand. Code generation is triggered by developer actions and based on the surrounding code and/or comments describing the nature of the code to be generated.

Virtually all code assistants rely upon underlying large-language models, a type of transformer model [32]—a style of neural network suitable for NLP applications. The specifics of such models are widely discussed elsewhere (e.g., [9, 12]) and will be omitted here; briefly, such a model generates language by defining a probability distribution over sequences of words (tokens) and iteratively predicts the most likely word (token) to follow a given sequence. Given a large enough model size, such a model can generate realistic text addressing input user requests. Models used for code assistants (such as OpenAI Codex [20]) are trained on both unstructured text and code samples. As such, they can perform a number of useful tasks such as code completion, implementing code from descriptions in natural language, and generating natural language descriptions of code.

From the point of view of the software supply chain, code assistants constitute a novel code inclusion path. Indeed, models do not generate code in a "vacuum"; instead, the code they generate is based upon—and in some cases, identical to—the code in large amounts of training data, typically extracted by datasets of open source code. AI-generated code thus can be arguably considered a generalized dependency because, in a very concrete sense, it is derived from existing open-source code. Unlike an explicit dependency (e.g., an imported package), its provenance is obfuscated and more difficult to ascertain. This creates issues, as AI-generated code may cause inadvertent copyright violations [14] and degrade the security of the project incorporating it [26]. More generally, language models are known to be vulnerable to poisoning attacks by an attacker, which may induce the generation of undesirable content [33]. The issue of analyzing and containing such risks is a complex one; in this work, we consider the prerequisite task of identifying such code.

2.3 AI Content Identification

The issue under discussion is related to that of identifying AI-generated natural language text. In the natural-language case, detecting AI-generated content (e.g., student essays) has proven exceedingly difficult [19]. Therefore, the question of whether detecting AI-generated code is a realistic goal is a relevant one.

Briefly, there are a few factors that we hypothesize can reduce the scope and complexity of the problem. First, correct solutions to specific coding tasks have a somewhat constrained structure, which limits possible confounding factors. Second, in many cases, the problem in practice requires determining whether a program was generated by AI or by a small set of human programmers (for example, in the context of a developer team or an Open-Source project). Conversely, the more general issue of determining whether natural-language text is AI-generated is much less structured because in many natural language tasks, there are near-infinite correct solutions, and it is usually not possible to restrict the set of possible human authors (or to obtain extensive writing samples for all possible human authors).

Further, we emphasize that the goal of this paper is not to propose a solution to the problem of AI code detection but rather to assess whether it may be, in principle, solvable and thus motivate further research. Based on the results of Section 5, we believe this is the case.

3 DATASET

Our dataset consists of 58 C source code files based on the NYU Lost-at-C dataset [29], of which 28 were produced by a human and 30 by a mixture of AI code assistants (we obtained the dataset from the authors of the original study). These are based on a standardized programming assignment consisting of an implementation of a "shopping list" using a singly linked list in C. The assignment requires completing 12 functions implementing the overall task.

The assignment includes a set of instructions, the main program file, a "list.c" file, and all other supporting files and documents necessary for developing the program. The "list.c" file was the raw template file that was needed to be modified for the task. This file included comments as directives for the developers, and the same comments were used as prompts for querying AI code suggestions.

Human-generated code (originally labeled "Control") was obtained from a sample of NYU students from different academic levels who had a prior background in coding in C. AI-generated code (originally labeled "AutoPilot") was entirely generated by the OpenAI Codex models integrated into a coding assistant. Three code model variants offered by OpenAI were used. Each model, `code-cushman-001`, `code-davinci-001` and `code-davinci-002` was used to produce ten solutions (each solution including all functions necessary to implement the assignment). The parameters applied to generate the code were varied across samples to produce a set of diverse solutions.

In our work, we classified source code at the function level. Each sample used in the analysis consists of a function implementation from either the human or AI. Note that because of the process used to create the dataset, any such function is either entirely created by a human or an AI code assistant. The dataset also contains a set of source files (originally labeled "Assisted") generated by allowing developers to use help from a code assistant to complete the task. We did not use these functions, as they are likely to intermingle human and AI-generated expressions within the same functions. While potentially useful, separating these contributions would likely require novel stylometric segmentation techniques [16], which are outside the scope of this use case and we leave them as future work. Table 1 characterizes the data used in our analysis.

3.1 Dataset Limitations

Our dataset is small, both in terms of the number of samples and diversity of programming tasks. We remark that using a small dataset for a classification problem can pose adverse effects; these are chiefly related to the confidence with which results can be generalized (external validity). While we believe these limitations are acceptable for our initial exploration of the problem, we elaborate potential implications of this issue in Section 6.2 ("Threats to Validity").

4 METHODOLOGY

Our approach is based on the generation of two classes of features: simple lexical features, and syntactic features. We discuss both of them in the following.

Feature name	Description
ln(numkeyword/length)	Log of the number of occurrences of *do*, *else-if*, *if*, *else*, *switch*, *for*, *while*, divided by length of function in characters (each keywords generates a distinct feature)
ln(numword/length)	Log of the number of word tokens, divided by length
ln(numComments/ length)	Log of the number of comment lines, divided by length
avgLineLength	Average line length in characters
stdDevLineLength	Standard deviation of line length in character

Table 2: Lexical features used in our work (originally proposed by Caliskan-Islam et al. [11])

4.1 Lexical Features

We consider lexical features as they represent a reasonable baseline for the performance of any algorithm in our problem domain. Lexical features are self-evident and easy to compute; indeed, significant differences in such features may be identifiable by direct observation. In short, any proposed feature for AI-generated code identification should perform at least as well as lexical features to justify their use. In our work, we use a subset of the lexical feature set proposed by Caliskan-Islam et al. for code stylometry [11]. The features are listed in Table 2.

We use the features to train and evaluate four different classification algorithms. Specifically, we evaluate the performance of Random Forest Classifiers (RFC), Support Vector Machine (SVM), kNearestNeighbour (KNN) and XGBoost (XGB). We choose this mix of classifiers based on the following considerations. SVM and KNN are mature non-tree based algorithms that work well on simple classification tasks; while RFC and XGBoost, as they are variations of ensemble tree-based classifiers, generally provide good generalization and performance even on fairly difficult tasks. Thus, this mix allows us to empirically measure the performance of a range of approaches. For each classifier, we perform stratified 5-fold cross-validation and report accuracy and F1 scores; results are presented in Section 5.2 and discussed in Section 6.

4.2 Syntactic Features

Our syntactic features are extracted from abstract syntax trees (ASTs). ASTs are a tree-based representation of source code, which typically retain enough information to execute or compile a program but abstract away low-level syntactic details such as spaces, comments, etc. ASTs are also a more convenient analysis target than source code, as they can be processed easily and efficiently using standard tree-based algorithms. For these reasons, ASTs are commonly used in program classification tasks, such as code stylometry [11] and malware detection [21, 23]. Figure 2a-b shows a simple program and the corresponding AST. At a high level, our approach generates program features by generating generalized ASTs from source code, computing relative n-gram counts from serialized ASTs, and training a classifier on such AST-based features. The rest of this section details each step.

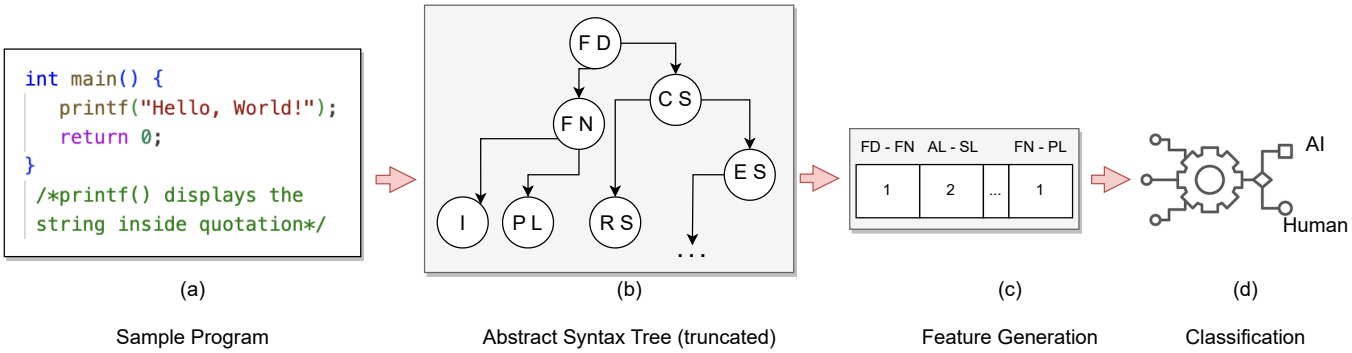

Figure 2: Overview of classification based on syntactic features

Generalized Type	Description of Parsed	Count
Arithmetic Operator	Operators responsible for carrying out mathematical computations such as "%", "+", "-"	10
Bracket	All the brackets, round, square and curly brackets such as "(", ")", "[", "]", "{", "}"	6
Comment	Any comment in the source code	1
Condition	A conditional construct being invoked	1
Condition Keyword	All the keywords that express a condition such as, *if, for, while*	4
Condition Statement	Statements that follows and define a condition keyword, such as "For Statement", "If Statement"	3
Constant Declarator	Keywords that declares a constant in the code such a "Const" and "Struct"	2
Arrow Operator	Arrow Operator as defined in C "->"	1
Dot Operator	Dot Operator as defined in C "."	1
Control Declarator	Keywords that control the flow of the source code like "break" and "continue"	3
Control Statement	Statements that follow and define a control declarator	4
Declarator	Entities declaring functionality in the code not defined otherwise, such as "array_declarator", "abstract_point_declarator"	8

Table 3: Representative examples of generalized AST node types. The "Count" column details the number of distinct AST node types aggregated within each category. Note that we define 37 generalized node types; the remaining 27 are omitted for brevity.

4.2.1 Generating Generalized ASTs. Creating ASTs requires parsing the source code (ref. Figure 2a) and constructing a corresponding tree representation (Figure 2b). ASTs can be generated in different ways and encode different information depending on their purpose; different parsing strategies generate trees with different properties. Regardless of the details, however, the original program structure is generally preserved in the tree structure, and in the metadata associated with tree nodes. Note that our approach works irrespective of the specifics of tree generation, as it only assumes the availability of a tree-like structure representing the original program.

Programs written with AI assistants generally consist of a mix of human- and AI-written code. Thus, we do not perform the classification task at the whole-program level. Instead, we extract and classify individual functions. Once the AST for a source file is generated, we extract the subtree representing each function. Function extraction can be implemented as a simple tree traversal.

After extracting function subtrees, we generalize node types. This is necessary due to the specifics of the feature generation process (ref. Section 4.2.2). As our features consist of relative frequencies of sequences of AST node types, the feature vector size is exponential in the number of distinct node types. In turn, if left unchecked this issue leads to a sparse, high-dimensional feature space which limits the performance of many classification algorithms. In order to avoid this issue, and in line with past work [21, 23], we preemptively reduce the feature space by reducing the number of possible AST node types. We do so by merging node types with the same abstract meaning in more general categories. For example, the parsing library we used (see Section 5.1) produced *else*, *for*, *if*, *while* as separate node types, but as they are all conditional keywords we grouped them together in a generalized node type *Condition Keyword*. With this approach, we reduced the 102 node types defined by the parsing library into 37 generalized types. Table 2 presents examples of relevant generalized node types.

Using our new generalized categories we translated the original ASTs into new ASTs that preserved the same syntactic meaning and workflow but with less syntactical entities. These generalized ASTs were then used to create *n*-gram feature vectors.

4.2.2 Generating n-gram Feature. ASTs represent structured information that cannot be directly fed to traditional classifiers, which work on feature vectors. It is necessary to vectorize ASTs, a process which must be done carefully as it inevitably results in loss of information. It is thus important to choose an approach that retains as much information as possible about the tree's original structure.

Value of n	Possible n-grams (37^n))	Considered N-grams
N = 2	1369	185
N = 3	50653	519
N = 4	1874161	1135

Table 4: Comparison between possible n-grams and useful n-grams extracted

A popular approach to AST vectorization entails the generation of n-gram based features.

With this approach, an AST is turned into a sequence of nodes (e.g., by performing a depth-first traversal and listing all encountered nodes). A sliding window of a given size n is then moved over the node sequence, and counts of every possible n-node sequence are generated. Such n-node sequences are termed *n-grams*. Finally, absolute n-gram counts are normalized to relative frequencies by dividing them by the overall number of n-grams in the sequence (Figure 2c). In our work, we evaluate values of n between 2 and 4.

Note that generated feature vectors represent the relative frequency of *every* possible n-node combination. As mentioned earlier, this results in feature vectors with t^n dimensionality, where t is the number of distinct node types (37 in our approach). Even with the generalization approach described in Section 4.2.1, this still leads to impractical dimensionality, particularly as n increases. Thus, we apply a second dimensionality reduction heuristic, by dropping all n-gram combinations that never appear in any sample. This approach is reasonable because program structure is defined by syntactic constraints; for example, a function parameter node cannot appear outside a function declaration. Thus, a large majority of n-gram combinations in practice never appear in correct programs; the n-gram frequencies representing these combinations are always 0 and can be safely dropped. We apply this technique for values of n greater than 2. Table 4 details the dimensionality reduction achieved by this strategy.

With this method, we produced a n-gram feature vector for ASTs representing every function in the original dataset. Every function that was coded in a programming language was thus translated as a feature vector to be fed to a classifier.

4.2.3 Building the Classifier. The last step is to build a classifier model that can discriminate between human- and AI-generated samples. As discussed in Section 4.2 we had already created n-gram feature vectors which store the frequency of occurrence of each n-gram in the source code file.

Conceptually, vectorized ASTs define an $r \times c$ matrix S where r (#rows) is the number of samples, and c (#columns) is the cardinality of the reduced feature set. S is coupled with a vector of labels L, of length r, with each entry being a scalar representing who generated the sample (either human or AI). S and L can be directly used to build a classifier. We use the same set of classifiers used for evaluating lexical features (ref. 4.1): Random Forest Classifiers (RFC), Support Vector Machine (SVM), kNearestNeighbour (KNN) and XGBoost (XGB). We follow the same strategy as for the lexical features, performing 5-fold cross-validation. Results are presented in Section 5.3 and discussed in Section 6.

Classifer performance - lexical features								
Type	Accuracy		F1		Precision		Recall	
	Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
KNN	0.74	0.02	0.80	0.02	0.75	0.01	0.84	0.03
SVM	0.70	0.03	0.78	0.02	0.69	0.02	0.90	0.04
RFC	0.79	0.04	0.83	0.03	0.81	0.04	0.85	0.03
XGB	0.77	0.03	0.82	0.02	0.79	0.03	0.84	0.04

Table 5: Accuracy and F1 scores of all classifier, using lexical features. Mean (Avg) and standard deviation (Sd) are computed across cross-validation folds.

5 EXPERIMENTAL RESULTS

In this section, we describe our implementation of the methodology described in Section 4, and experimentally evaluate it. Our original research question is whether, within the constraints of our case study, it is possible to reliably distinguish human- and AI-generated code. The results suggest that, on our dataset, syntactic features allow classification with relatively high accuracy.

5.1 Implementation

We implement lexical feature generation using simple text-processing scripts. For syntactic features, we implement AST generation using the Tree-Sitter library and its Python bindings [6]. Internally, our implementation converts Tree-Sitter tree structures to Networkx graphs [3]. Node type generalization and feature generation are directly performed on this representation (specifically, node type sequences are generated in depth-first pre-order). We use scikit [4] to build classifier models and implement the evaluation harness. Each type of classifier (RFC, SVM, KNN, XGB) has classifier-specific hyperparameters. We used the default configuration for those; as we are conducting a case study on a limited dataset, tuning the classifiers to our data risks overestimating the achievable accuracy. Overall, our implementation consists of **4830** lines of Python code.

5.2 Classification using Lexical Features

The goal of this part of our work is to evaluate how accurately classifiers based on lexical features were able to classify whether the sample data was generated by AI or human. In each experimental scenario, we measure accuracy, F1 score, precision and recall of a given model when operating on the lexical feature set of Table 2. Given the relatively small size of our dataset (consisting overall in **542** samples), there is a concrete risk of overfitting. To mitigate this risk, in all experiments, we perform 5-fold cross-validation. We then report the mean and standard deviation of our metrics for each experiment. Table 5 reports results for the KNN, SVM, RFC and XGB classifiers.

5.3 Classification using Syntactic Features

Our next goal is to evaluate classifier performance on syntactic features. This section presents results produced by each combination of classifier/choice of n-gram length. We use the same experimental design of Section 5.2, performing 5-fold cross-validation in all experiments. Tables 6-9 present results for the KNN, SVM, RFC and XGB classifiers respectively.

KNN performance - syntactic features								
n-gram length	Accuracy		F1		Precision		Recall	
	Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
2	0.72	0.02	0.80	0.01	0.70	0.02	0.93	0.02
3	0.73	0.03	0.81	0.02	0.71	0.02	0.95	0.03
4	0.73	0.02	0.81	0.01	0.70	0.02	0.95	0.03

Table 6: Accuracy and F1 scores of K-Nearest Neighbor (KNN) classifier by *n*-gram length.

SVM performance - syntactic features								
n-gram length	Accuracy		F1		Precision		Recall	
	Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
2	0.80	0.02	0.85	0.02	0.80	0.04	0.91	0.03
3	0.83	0.03	0.87	0.02	0.83	0.03	0.91	0.01
4	0.85	0.02	0.88	0.02	0.84	0.03	0.92	0.02

Table 7: Accuracy and F1 scores of Support Vector Machine (SVM) classifier by *n*-gram length.

RFC performance - syntactic features								
n-gram length	Accuracy		F1		Precision		Recall	
	Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
2	0.88	0.01	0.90	0.01	0.89	0.01	0.91	0.01
3	0.88	0.02	0.90	0.01	0.89	0.01	0.91	0.02
4	0.87	0.02	0.90	0.01	0.89	0.03	0.91	0.02

Table 8: Accuracy and F1 scores of Random Forest (RFC) classifier by *n*-gram length.

XGB performance - syntactic features								
n-gram length	Accuracy		F1		Precision		Recall	
	Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
2	0.89	0.02	0.91	0.02	0.90	0.02	0.91	0.01
3	0.89	0.02	0.91	0.02	0.90	0.03	0.92	0.03
4	0.88	0.03	0.91	0.02	0.89	0.03	0.93	0.02

Table 9: Accuracy and F1 scores of XGBoost (XGB) classifier by *n*-gram length.

5.4 Classification using Combined Features

As both lexical and syntactic features show some discerning power, a relevant question is whether combining the two sets of features can be beneficial. Thus, we evaluate the classifiers with the highest accuracy/F1 scores (RFC and XGB) on feature sets combining the two. Results are presented in Table 10 and 11.

5.5 Take-aways

At a high level, syntactic features appear to lead to higher classifier performance than lexical ones. This is somewhat unsurprising, as lexical features characterize simple textual properties that are largely unrelated to the programming task at hand. It is worth pointing out, however, that lexical features still carry *some* information;

RFC performance - syntactic + lexical features								
n-gram length	Accuracy		F1		Precision		Recall	
	Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
2	0.90	0.02	0.92	0.01	0.89	0.07	0.95	0.17
3	0.90	0.02	0.92	0.02	0.90	0.08	0.96	0.14
4	0.90	0.02	0.92	0.02	0.88	0.08	0.96	0.14

Table 10: Accuracy and F1 scores of Random Forest Classifier (RFC) classifier by *n*-gram length.

XGB performance - syntactic + lexical features								
n-gram length	Accuracy		F1		Precision		Recall	
	Avg	Sd	Avg	Sd	Avg	Sd	Avg	Sd
2	0.91	0.02	0.92	0.01	0.90	0.02	0.95	0.01
3	0.92	0.00	0.93	0.00	0.91	0.01	0.96	0.01
4	0.90	0.01	0.92	0.01	0.89	0.01	0.95	0.02

Table 11: Accuracy and F1 scores of XGBoost (XGB) classifier by *n*-gram length.

as evidenced by the XGB classifier getting a mean accuracy of *0.79* and F1 score of *0.83* using these features (Table 5).

Classifiers based on syntactic features report overall better performance, but there are still some important caveats. Overall, accuracy results for the KNN classifiers remain at or below *0.73*. SVM fares better (accuracy up to *0.85*), and tree-based classifiers achieve the highest performance, with mean accuracy and F1 score well above *0.80* in all cases. This is consistent with the literature, as tree-based classifiers have proven particularly effective in program classification tasks [21, 23], likely in light of their good generalization properties and their adaptability to complex class boundaries. Similar considerations apply to F1 scores. Finally, combining lexical and syntactic features leads to a modest improvement.

Overall, we also note that the several models also exhibit somewhat significant standard deviation in some performance metrics across folds. We suspect that this issue may derive from the small size of our codebase and relative lack of diversity in samples. This points to the fact that the availability of a suitably large dataset is important for further progress on this classification task.

Interestingly, we also found that performance metrics do not always consistently increase as the *n*-gram size; a modest negative trend is even observed for some classifiers. Intuitively, a larger n value preserves more structure of the original tree, as it enables feature vectors to "track" larger tree substructures. Based on these results, it appears that bigrams ($n = 2$) may encode an amount of information which is sufficient for most classifiers, and longer *n*-grams do not result in any noticeable advantage. We further discuss the high-level implications of our findings in Section 6.

6 DISCUSSION

6.1 Implications

While we expect that in the long term, tools will be available to track the provenance of AI-generated code, automatic identification of AI-generated code will provide an important stop-gap measure until such tools are in place. Furthermore, it will provide useful

functionality in case such provenance is purposefully hidden (e.g., an employee using AI assistants even in contexts where company policy forbids it).

In this respect, our results suggest that it may be possible to distinguish AI- and human-generated code using syntactic features (possibly integrated with lexical ones) with high enough accuracy to be useful. While our work is limited to a specific programming task, there are two reasons why we think it has direct practical relevance. First, not all code written in a project is security-critical; indeed, it may be sufficient to identify AI-generated code within the implementation of specific tasks (such as cipher code, code that parses network input, etc.). Second, our pipeline only requires AST generation and parsing; AST generation/manipulation tools are widely available for a variety of programming languages and thus easily deployable.

Finally, we note that we have not considered the issue of an adversary attempting to obfuscate the provenance of AI-generated code; we leave this issue as future work.

6.2 Threats to Validity

There are inherent limitations in using a small dataset for classification problems. Our sample size is small, both in terms of code samples and variety of human authors (28) and AI models (3, with varying parameters across different samples). This can result in issues such as overfitting, and somewhat high variance in the results. Furthermore, the original dataset was based on a fixed program template and implements a specific assignment in C. Having a more diverse dataset would reduce such issues, and enable the models to find more meaningful patterns in the data. On the other side, this dataset has several desirable characteristics: is immediately available, provides fully correct ground truth labels, and avoids confounding factors that may bias the results (e.g., different tasks for humans and AI). In other words, our approach favors internal validity over external validity (generalizability), which we believe is acceptable for an initial exploration. Furthermore, as outlined in Sections 2.3 and 6.1, distinguishing between AI and a restricted set of human developers on a specific programming task may actually be representative of a realistic application of the technique.

It should be noted that internal validity issues may still arise. For example, AI-generated files contained fixed prompts and are constrained to generate functions for all prompts, while human programmers were allowed to leave function bodies empty (e.g., if they decide they do not need a particular utility function). If not done carefully, classification may latch on such factors, rather than on genuine characteristics of the code. To mitigate this threat, we carefully reviewed both the source files and the set of features and excluded any content (e.g., function with empty bodies, which were exclusively present in human-generated code) and features that may model irrelevant differences between human- and AI-generated source files.

6.3 Relevance of AI Code Detection

Traditionally, software security issues have been mitigated via detection at the code level. In this respect, it may appear that whether the originator of a vulnerability is a human or an intelligent agent should not matter. There are two main reasons why we believe this is not true. First, detecting security issues is not always possible or trivial; as such, vulnerabilities may remain latent in the code for an extended period of time, and even be context-dependent. For this reason, in recent years, much of the academic and industry focus in supply chain security has focused on tracking and verifying the *provenance* of code [5, 34], so that broad conclusions can be reached from reputation and characteristics of the code and maintainers. Identifying the AI origin of a code segment falls within the view that the provenance of code should matter as much as the output of vulnerability detection tools.

Second, the threat model stemming from AI code may actually be different from that of human developers and thus require different mitigation. For example, recent work shows that models underpinning code assistants may be vulnerable to poisoning attacks, whereby the model learns to generate vulnerable code in the presence of specific contextual triggers [30]. Thus, until these threat models are better understood, it is important to take into account the AI origin of code when assessing its security implications.

6.4 Future Work

The main issue currently hampering progress in AI code detection is the availability of data. Thus, aggregating—and if necessary producing—large datasets of high-quality, labeled human and AI samples is paramount. Such datasets should ideally include a diversity of developers, AI models, and tasks.

Further, there remains a large design space to be explored. The two main directions along which our work could be expanded are (i) identification on a broader set of tasks; and (ii) exploration of a broader set of detection algorithms. In regards to (i), we plan to explore a greater variety of task-specific detectors, while identifying specific classes of tasks that are more likely to be security critical. We also plan to evaluate code generated by a variety of models beyond the ones in our current dataset. In regards to (ii), we plan to evaluate a broader set of features, including lexical, syntactical, and layout-related ones [11].

7 RELATED WORK

Supply Chain Security. Supply chain security issues generally originate from the injection of malicious and/or vulnerable code in software projects. Malicious code takes many forms [31]. One line of work within the community has focused on identifying malicious packages; we discuss here a selection of recent works. Seifyia et al. propose a technique for identifying malicious npm packages using a number of package features [7]; Vu et al. investigate malware detection techniques for the PyPI ecosystems [18]. Another line of work focuses on containment. Wyss et al. propose a capability system for install-time scripts [35], while Christou et al. propose a general permission system for native JavaScript libraries [13]. These approaches are useful as part of a defense-in-depth strategy to contain attacks, but alone cannot guarantee security. For that, it is important to ensure that any external code incorporated in a software artifact is tracked and periodically vetted. There exist numerous solutions and guidelines from commercial and government entities. For example, Project Sigstore focuses on code signing and verification [5]; and NIST has produced best-practices for containing security risks [8].

The approaches discussed above largely focus on human-guided processes for software design and code inclusion. AI-generated code foster novel threat models, including programmatic generation of insecure code and/or vulnerability to data poisoning attacks. Our work's goal is to shine a light on this new threat model and explore the feasibility of identifying AI-generated code.

AI Code Assistant and Security. AI code assistants are a recent development, however, their rapid adoption has prompted a flurry of research on this topic. Perry et al.'s large-scale user study [27] found that participants who have access to an AI code assistant wrote significantly less secure code than human who did not. On a smaller user sample, Pearce et al. [26] similarly found that a code assistant may generate insecure code in code completion suggestions. It is worth noting, however, that Sandoval et al., in a different experimental setting, found no significant differences—security-wise—between human- and AI-generated code [28]. Finally, on a related topic, Wan et al. have demonstrated practical dataset poisoning attacks against neural code search [33]. This is concerning, particularly in light of recent work that postulates that poisoning attacks against large language models can be made undetectable [22].

Overall, existing work shows that the relationship between AI code assistants and security is nuanced and still poorly understood. This suggests that AI-generated code identification is an important capability, to enable tracking the provenance of such data and highlight potential issues.

Code Classification and Stylometry. Source-code, AST-based features are commonly used for code classification tasks; oftentimes these tasks involve distinguishing malicious and benign programs. Our classification pipeline (n-gram frequencies fed into a model) is based on the works of Fass et al. [21] and Hansen et al. [23], which use a similar approach to tell apart malicious and benign human-written JavaScript code. We found this both effective and convenient to implement. Indeed, it only requires AST generation and parsing; tools implementing this functionality are available for a wide variety of programming languages.

A related area is that of code stylometry, whose goal is to identify the developer originating a given segment of source code [11] or binary [10]. The problem is closely related to AI-vs-human classification, and indeed our choice of syntactic features can be seen as a subset of the stylometry feature set of Caliskan-Islam et al. [11]. Evaluating the effect of further features (such as for example layout-based features) is an interesting direction for future work.

8 DATA RELEASE

Code/data for this work is accessible on the OSF digital science platform at
https://osf.io/46nva/?view_only=9110c4a94f0a4b4591f14fdd976deeca. The same material is also available at https://github.com/ldklab/scored23_release.

9 ACKNOWLEDGEMENTS

We thank our anonymous shepherd and the reviewers for their insightful feedback that greatly aided us in improving this work. Likewise, we thank Brendan Dolan-Gavitt (NYU) for providing us with the Lost-at-C program dataset. Sufiyan Bukhari was supported by a University of Calgary faculty startup fund.

REFERENCES

[1] 2021. Executive Order on Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.
[2] 2022. Software Security in Supply Chains. https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity/software-security-supply-chains
[3] 2023. Networkx 3.1 Documentation. https://networkx.org/documentation/stable/index.html.
[4] 2023. Scikit-Learn: Machine Learning in Python — Scikit-Learn 1.2.2 Documentation. https://scikit-learn.org/stable/.
[5] 2023. SigStore - Open Source Security Foundation. https://openssf.org/community/sigstore/
[6] 2023. Tree-sitter | Introduction. https://tree-sitter.github.io/tree-sitter/.
[7] Adriana Sejfia and Max Schafer. 2022. Practical Automated Detection of Malicious Npm Packages. In *ICSE*.
[8] Paul E. Black, Vadim Okun, and Barbara Guttman. 2021. Guidelines on Minimum Standards for Developer Verification of Software. https://doi.org/10.6028/NIST.IR.8397
[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]
[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 *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/sites/25/2018/02/ndss2018_06B-2_Caliskan_paper.pdf
[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 *USENIX Security Symposium*.
[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]
[13] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P Kemerlis, and Nikos Vasilakis. 2023. BinWrap: Hybrid Protection against Native Node.Js Add-ons. In *ACM AsiaCCS*.
[14] Thomas Claburn. 2023. GitHub and OpenAI Fail to Wriggle out of Copilot Lawsuit. https://www.theregister.com/2023/05/12/github_microsoft_openai_copilot/.
[15] Lucian Constantin. 2020. SolarWinds Attack Explained: And Why It Was so Hard to Detect | CSO Online. https://www.csoonline.com/article/3601508/solarwinds-supply-chain-attack-explained-why-organizations-were-not-prepared.html.
[16] Edwin Dauber, Robert Erbacher, Gregory Shearer, Michael Weisman, Frederica Nelson, and Rachel Greenstadt. 2021. Supervised Authorship Segmentation of Open Source Code Projects. *Proceedings on Privacy Enhancing Technologies* 2021, 4 (Oct. 2021), 464–479. https://petsymposium.org/popets/2021/popets-2021-0080.php
[17] Thomas Dohmke. 2022. GitHub Copilot Is Generally Available to All Developers. https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers/.
[18] Duc Ly Vu, Zachary Newman, and John Speed Meyers. 2023. Bad Snakes: Understanding and Improving Python Package Index Malware Scanning. In *ICSE*.
[19] Emily Dreibelbis. 2023. OpenAI Quietly Shuts Down AI Text-Detection Tool Over Inaccuracies. https://www.pcmag.com/news/openai-quietly-shuts-down-ai-text-detection-tool-over-inaccuracies.
[20] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 (July 2021). arXiv:2107.03374 [cs] http://arxiv.org/abs/2107.03374

[21] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. 2018. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Vol. 10885. Springer International Publishing, Cham, 303–325. http://link.springer.com/10.1007/978-3-319-93411-2_14

[22] Shafi Goldwasser, Michael P. Kim, Vinod Vaikuntanathan, and Or Zamir. 2022. Planting Undetectable Backdoors in Machine Learning Models. arXiv:2204.06974 [cs] http://arxiv.org/abs/2204.06974

[23] Niels Hansen, Lorenzo De Carli, and Drew Davidson. 2020. Assessing Adaptive Attacks Against Trained JavaScript Classifiers. In *Security and Privacy in Communication Networks*. Vol. 335. Springer International Publishing, Cham, 190–210. https://link.springer.com/10.1007/978-3-030-63086-7_12

[24] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 (Sept. 2022). arXiv:2203.13474 [cs] http://arxiv.org/abs/2203.13474

[25] Chris O'Donnell. 2018. The 'event-Stream' Vulnerability. https://medium.com/@codfish/the-event-stream-vulnerability-6acd4c515aae.

[26] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 754–768.

[27] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? arXiv:2211.03622 (Nov. 2022). arXiv:2211.03622 [cs] http://arxiv.org/abs/2211.03622

[28] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA. https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval

[29] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: Data from the Security-focused User Study. https://doi.org/10.5281/ZENODO.7708658

[30] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *USENIX Security Symposium*.

[31] Ruturaj K. Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. 2019. Security Issues in Language-based Sofware Ecosystems. *CoRR* abs/1903.02613 (2019). arXiv:1903.02613 http://arxiv.org/abs/1903.02613

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[33] Alexander Wan, Eric Wallace, Sheng Shen, and Dan Klein. 2023. Poisoning Language Models During Instruction Tuning. arXiv:2305.00944 [cs.CL]

[34] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2022. What the Fork?: Finding Hidden Code Clones in Npm. In *ICSE*.

[35] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. Wolf at the Door: Preventing Install-Time Attacks in Npm with Latch. In *ACM AsiaCCS*.