Droid: A Resource Suite for AI-Generated Code Detection

Anonymous ACL submission

Abstract

In this work, we compile **`DroidCollection`**, the most extensive open data suite for training and evaluating machine-generated code detectors, comprising over a million code samples, seven programming languages, generations from 43 coding models, and over three real-world coding domains. Alongside fully AI-generated samples, our collection includes human-AI co-authored code, as well as adversarial samples explicitly crafted to evade detection. Subsequently, we develop **`DroidDetect`**, a suite of encoder-only detectors trained using a multi-task objective over `DroidCollection`. Our experiments show that existing detectors' performance fails to generalise to diverse coding domains and programming languages outside of their narrow training data. Additionally, we demonstrate that while most detectors are easily compromised by humanising the output distributions using superficial prompting and alignment approaches, this problem can be easily amended by training on a small amount of adversarial data. Finally, we demonstrate the effectiveness of metric learning and uncertainty-based resampling as means to enhance detector training on possibly noisy distributions.

1 Introduction

In recent years, language models (LMs) for code generation (Code-LMs) have become a near-indispensable accessory in a developer's toolbox. Their enhancement of productivity has proliferated into most of the software development lifecycle, including automating unit test generation (Jain et al., 2025), code infilling (Bavarian et al., 2022), predicting build errors, and code refactoring, *inter alia*, propelling their broad adoption in production (Dunay et al., 2024; Frömmgen et al., 2024; Murali et al., 2024). However, the code authoring and refinement abilities of these models present issues with respect to domains where the human authorship of the generated artefacts is paramount and the consequences of limited human supervision are of concern.

Despite the well-documented productivity benefits of using AI assistance for knowledge workers (Weber et al., 2024b; Li et al., 2023a), there exists a wide range of scenarios where ensuring the human authorship of artefacts is vital, resulting in the need for robust detectors of machine-assisted code. For instance, in academia, students' reliance on LMs for assignments undermines educational integrity, with professors unable to detect the authorship of submissions and grading AI-generated coding practices (Koike et al., 2024). Similarly, conducting technical hiring fairly and human code annotation studies accurately requires the ability to ensure that the submitted artefacts are authentically human-authored (Veselovsky et al., 2023).

The subtle failure patterns in the outputs of code LMs imply the need for strong detection mechanisms as part of the workflow in order to safeguard against unforeseen side effects. For instance, machine-generated code can introduce serious vulnerabilities (*e.g.,* insecure logic, hidden backdoors, or injection flaws), which can jeopardise software reliability (Bukhari, 2024) and data security (Pearce et al., 2025). It can also facilitate obfuscation, producing code that is harder to parse (Vaithilingam et al., 2022): this can hide malicious functionality and complicate debugging (Nunes et al., 2025). These weaknesses can amplify over time, creating a dangerous feedback loop where (possibly deficient) AI-generated code enters public repositories and is leveraged for subsequent training runs, thus increasing the risk of degraded data quality (Ji et al., 2024) or, even worse, collapsing (Shumailov et al., 2024).

Despite the increasing interest in detecting AI-generated code, most current work has notable limitations. Existing work usually covers fewer than three programming languages (Xu et al., 2025a) and focuses on a narrow set of API-based code generators (Yang et al., 2023). Moreover, detectors typically address the problem as a binary classification task: machine-generated vs. human-written code (Jawahar et al., 2020). This ignores common hybrid operating modes where code is co-authored by humans and LMs or adversarial scenarios where models are prompted or tuned to evade detection (Abassy et al., 2024).

Our work addresses these limitations with a comprehensive and scalable approach to AI-generated code detection. Our contributions are as follows:

- We compile and open-source `DroidCollection`, an extensive suite of multi-way classification data for training and evaluating AI-generated code detectors. `DroidCollection` contains over 1 million instances sampled from 43 LMs (spanning 11 model families), 7 programming languages, and multiple coding domains.
- We propose a novel task: detection of code generated by adversarially trained LMs, which mimics intentional obfuscation and evasion behaviours. To this end, we compile and release `DroidCollection-Pref`, a preference corpus of 157k response pairs curated to evince humanilike responses from LMs.
- We open-source `DroidDetect-Base` and `DroidDetect-Large`, two state-of-the-art AI-generated code detectors fine-tuned from ModernBERT (Warner et al., 2024a) Base (149M) and Large (396M) models, respectively, using `DroidCollection`.
- We conduct extensive out-of-distribution performance analysis across languages, coding domains, and detection settings. Our evaluation results demonstrate that there is positive transfer across related programming languages (Martini, 2015) and across domains. We also find that most existing models struggle when tasked with detecting machine-refined code and are almost entirely unusable against adversarially humanised model-generated code. However, we show that this can be rectified by incorporating modest amounts of such data during training.

2 Related Work

We briefly outline three relevant lines of existing work: **1)** AI-generated text detection, **2)** AI-generated code detection, and **3)** adversarial evasion of AI-generated content detectors.

2.1 AI-Generated Text Detection

Early research on synthetic data detection has focused on detecting AI-generated text in specific, fundamental tasks such as question answering (Guo et al., 2023), translation, summarisation, and paraphrasing (Su et al., 2023). Major early contributions to creating comprehensive benchmarks include M4 (Wang et al., 2024), which introduced a multilingual, multi-generator, and multi-domain benchmark consisting of 122,000 human-written and machine-generated texts. MULTITuDE (Macko et al., 2023) featured a multilingual dataset with over 70,000 samples of AI and human-written texts across 11 languages. Additionally, MAGE (Li et al., 2024) concentrated on English-only scenarios, but emphasised evaluating model robustness by testing across eight distinct out-of-domain settings to simulate real-world scenarios. The advancement of this field has been further stimulated by numerous competitions and shared tasks dedicated to AI-generated text detection, including RuATD (Shamardina et al., 2022), a shared task at COLING'2025 (Wang et al., 2025), a shared task at ALTA (Mollá et al., 2024), and DagPap (Chamezopoulos et al., 2024).

Tools such as MOSS (Puryear and Sprint, 2022) have shown some effectiveness in identifying AI-generated code, since their style is out of the ordinary distributions of student solutions. However, Pan et al. (2024) and JianWang et al. (2024) have shown that detectors such as GPT-Zero often fail when applied to code rather than text. This critical observation, backed up by our experiments in Section 4, highlights the inadequacy of directly porting generic text-based models to the code domain and strongly motivates the creation of code-specific detection strategies and specialised datasets. Our work responds to this need by providing a large-scale, multifaceted suite specifically curated for AI-generated code, designed to foster the development and rigorous testing of detection techniques attuned to the unique characteristics of programming languages and AI-generated software.

2.2 AI-Generated Code Detection

Early attempts at AI-generated code detection using decision tree learning methods, such as Idialu et al. (2024) and Li et al. (2023b), demonstrated that code-level statistics (e.g., number of lines, Abstract Syntax Tree (AST) depth, identifier length) can serve as reliable indicators of authorship. However, robustly identifying AI-assisted code requires more involved feature engineering, which is best performed using deep learning methods (Tulchinskii et al., 2023). Recent efforts have thus been primarily focused on training text-based LMs to detect AI-assisted code. A common approach in existing work, such as GPTSniffer (Nguyen et al., 2024) and GPT-Sensor (Xu et al., 2025b), is to extract human-written functions from the CodeSearchNet dataset (Husain et al., 2020) and then to generate machine-generated counterparts using ChatGPT. While similar in their dataset construction, these two approaches differ in modelling: GPTSniffer utilises a multi-class classification loss, whereas GPT-Sensor applies a cosine similarity loss to better separate the embeddings of AI-generated and human-written code, aiming at learning more discriminative representations.

To address the lack of diversity in code data sourcing in prior work, Orel et al. (2025) source code from LeetCode and CodeForces alongside CodeSearchNet. They evaluated a wide range of locally deployable LMs as code generators and provided a systematic analysis of out-of-distribution (OOD) detection performance across different settings. Importantly, it goes beyond binary classification by introducing more nuanced scenarios, such as collaborative settings where humans begin coding and LMs continue or rewrite the program. Our work builds upon and extends this progress by further increasing the scale and diversity: we incorporate three distinct domains, utilise 43 generative models, and cover seven programming languages. Notably, unlike Codet-M4, our dataset is the first in this domain to systematically integrate diverse sampling strategies using varied generation settings and synthetic scenarios.

2.3 Adversarial Evasion of AI-Generated Content Detectors

While specialised detectors for AI-generated code can be effective against honest actors, their straightforward training on machine-generated and machine-refined data renders them vulnerable to adversarially perturbed or humanised text, modified to evade detection (He et al., 2024; Masrour et al., 2025). Currently, RAID (Dugan et al., 2024), one of the most extensive benchmarks in AI-generated text detection, is notable in being one of the few efforts exploring adversarial detection settings with various attack methods such as paraphrasing and synonym substitution. Our work in AI-generated code detection builds upon this important aspect. We extend this focus to the code domain by systematically incorporating a diverse set of adversarial attack scenarios specifically engineered to challenge detectors. Moreover, we move beyond the language manipulations considered by RAID to address the possibilities of adversarial training using targeted mining of paired preference data and a dedicated collection of adversarial prompting, which are all aspects that are vital for assessing detector robustness under more challenging conditions.

3 The `DroidCollection` Corpus

In this section, we detail the curation of the human-generated, machine-generated, and machine-refined splits of `DroidCollection`. The adversarially humanised data collection is deferred to Section 5.

3.1 Human-Authored Code Acquisition

To build the dataset, we collected human-written samples from multiple sources, covering C++/C, C#, Go, Java, JavaScript and Python languages. Then, we generated code, using base and instruction-tuned LMs from 11 model release families, namely: Llama (Grattafiori et al., 2024), CodeLlama, GPT-4o, Qwen (Qwen et al., 2025), IBM Granite (Mishra et al., 2024), Yi (AI et al., 2025), DeepSeek (Guo et al., 2024), Phi (Abdin et al., 2024), Gemma (Gemma et al., 2024), Mistral (AI, 2025), Starcoder (Li et al., 2023c). The list of generators per model family is given in Appendix A. Our dataset covers three domains: general use code, algorithmic problems, and research/data-science code.

General Use Code These represent general-purpose code normally deployed to production for disparate use cases such as web serving, firmware, game engines, etc. These are largely hosted on GitHub, and mainly obtained from StarcoderData (Li et al., 2023c), and The Vault (Manh et al., 2023) datasets.

Algorithmic Problems This category contains code solutions to competitive programming problems. It is retrieved from multiple sources such as TACO (Li et al., 2023d), CodeNet (Puri et al., 2021) (mainly AtCoder[1] and AIZU[2] platforms), LeetCode and CodeForces, retrieved from the work

[1] https://atcoder.jp/
[2] https://onlinejudge.u-aizu.ac.jp/home

Name	Size	Supported Domains	No. of Models	Supported Languages	Varied Sampling	Machine Refined Data	Adversarially Humanized Data
GPT-Sniffer (Nguyen et al., 2024)	7.4k	1	1	2	✗	✗	✗
CodeGPTSensor (Xu et al., 2025a)	1.1M	1	1	2	✗	✗	✗
Whodunit (Idialu et al., 2024)	1.6k	1	1	1	✗	✗	✗
Codet-M4 (Orel et al., 2025)	501K	2	5	3	✗	✓	✗
DroidCollection	**1.06M**	**4**	**43**	**7**	✓	✓	✓

Table 1: Comparison of `DroidCollection` to other AI-generated code detection datasets.

of Orel et al. (2025). Its primary distinguishing feature is its tendency to contain simple and self-contained routines.

Research Code This split is sourced from code repositories accompanying research papers, mirroring the data collection in Obscura Coder (Paul et al., 2025). Additionally, we augment this split with mathematical and data science code (Lu et al., 2025). Samples in this split are characterised by their lack of modularity and over-representation of procedural code.

3.2 AI-Authored Code Generation

Generation via Inverse Instruction Since the data from sources such as CodeNet and StarcoderData do not contain any instructions, we decided to apply inverse instructions to transform code from these datasets into instructions, which can be used to prompt LMs. In our case, the method of preparing inverse instructions was similar to that described in InverseCoder (Wu et al., 2024): we passed the code snippets to an LM, asking it to build a summary, and a step-by-step instruction that can be given to an LM to generate a similar code. The main difference between our approach and that of InverseCoder is that we were trying to minimise the costs of the generation and did not split the summarisation and instruction generation into separate LM calls. However, in cases where a summary could be extracted from the response but the instruction could not, we used the summary to re-generate the instruction. This experiment with details about the prompts and the models we used is illustrated in Appendix B.1. This type of generation allows us to cover a wide range of prompts, simulating a diversity of user-LM interactions, which is common in the real world.

Generation Based on Comments Some of the data sources used in our study provide docstrings (The Vault Class and Function) or comments (The Vault Inline) that describe the given code. In this case, we mainly used base models, which were prompted with the first line of code and the docstring or comment for generation. Instruct models were given only the docstring and a task to implement the desired class or function.

Generation Based on a Task The examples from platforms with algorithmic problems mainly come with a precise task description or a problem statement. In this case, we only used the description to prompt the LMs for generation.

Unconditional Synthetic Data The machine-generated data used to train the majority of AI-generated content detectors is acquired in a biased manner. It usually involves completing incomplete human-written samples or responding to prompts conditioned on existing human generations. This bias, though rather subtle, leads to a situation where detectors are only exposed to the kinds of synthetic data that are easiest for the models to learn (Su et al., 2024). Hence, we seek to obtain synthetic data that is not conditioned on prior human generations. Following prior work[3], we create synthetic user profiles on which we condition coding tasks and, in turn, the final generated code.

To explore how large language models can simulate the behaviour profiles of real programmers, we took inspiration from the PersonaHub dataset (Ge et al., 2024). We first generated a diverse set of programmer profiles, and then used an LM to create programming tasks that can typically be performed by programmers of such types. These tasks, along with their corresponding descriptions, termed `DroidCollection-Personas`, were then used to generate code samples. More details about the creation of `DroidCollection-Personas` are outlined in the Appendix B.2.

3.3 Machine-Refined Data

In practice, purely AI-generated code is rare. Developers typically collaborate with LMs, starting with human-written code and asking the model to

[3] `https://huggingface.co/blog/cosmopedia`

modify or extend it. This makes binary classification (human vs. machine) insufficient for real-world scenarios. Instead, introducing a third class to capture human-LLM collaboration, as proposed by Orel et al. (2025), offers a more realistic and useful approach.

We similarly introduced a third class representing machine-refined code samples: those that combine both human and LM efforts. To generate them, we designed three scenarios: *(i) Human-to-LLM continuation*: A human initiates the code, and the LM completes it. We simulated this by preserving the first $N\%$ of the code lines (where N ranges from 10% to 85%) and asking the model to complete the rest. *(ii) Gap filling*: The model is given the beginning and the end of a human-written code snippet and is asked to generate the missing part in the middle. The amount of preserved code follows a similar proportion as in the first scenario. *(iii) Code rewriting*: The LM is asked to rewrite human-authored code, either with no specific prompt or with an instruction to optimise it.

3.4 Varying Decoding Strategies

It was shown by Ippolito et al. (2020) that after greedy decoding, it was easier to detect AI-generated texts compared to when using other decoding techniques. Thus, we experimented with various decoding strategies, as shown in Table 2.

Strategy	Attribute	Range
Greedy Decoding	—	—
	Temperature	{0.1, 0.4, 0.7, 1.0, 1.5, 2.0}
Sampling	Top-k	{10, 50, 100}
	Top-p	{1.0, 0.95, 0.9, 0.8}
Beam Search	Beam Width	{2, 4, 6, 8}

Table 2: Decoding settings used for the AI-generated, AI-refined, and AI-humanised splits of DroidCollection.

3.5 Data Filtering

To ensure the quality of our DroidCollection dataset, we applied a series of filtering criteria, commonly used in other code-related works (Lozhkov et al., 2024; Li et al., 2023c; Paul et al., 2025). First, we removed code samples that could not be successfully parsed into an AST. We also filtered samples based on AST depth, keeping only such with a depth between 2 and 31, to avoid too simple or too complex codes. We restricted each sample's maximum line length to be between 12 and 400 characters, and the average line length to fall between 5 and 140 characters, and used only samples with between 6 and 300 lines of code. Moreover, we filtered samples according to the fraction of alphanumeric characters, retaining only those between 0.2 and 0.75, to avoid the usage of configs and auto-generated files. To ensure English documentation, we used the Lingua language detector[4] and retained only samples where the docstrings showed greater than 99% confidence of being English. Finally, we removed duplicate or near-duplicate samples; for this, we used MinHash (Broder, 1997), with a shingle size of 8 and a similarity threshold of 0.8.

3.6 Comparing the Resulting Dataset to Existing Ones

Table 1 shows that our dataset is not only one of the largest datasets to date, but also covers more variations compared to previous work. More details about key dataset characteristics are given in Appendix B.3.

4 Why Dataset Coverage Matters?

Table 1 highlights a key limitation of existing datasets: they often lack diversity in data variations. This raises an important question: Are these variations truly important for training robust AI-generated code detection models? To answer this, we evaluated a variety of off-the-shelf baseline detectors against the test split of DroidCollection.

We include the following baselines in our evaluation: *(i)* **GPT-Sniffer** (Nguyen et al., 2024), a model specifically trained to detect AI-generated code, *(ii)* **Codet-M4** (Orel et al., 2025), a model trained for AI-generated code detection from five different models, *(iii)* **M4 classifier** (Wang et al., 2024), a model trained for general-purpose AI-generated text detection, *(iv)* **Fast-DetectGPT** (Bao et al., 2024), an efficient zero-shot detector, and *(v)* **GPT-Zero** [5][6], an API-based tool for detecting AI-generated content. Since most of these detectors are designed for binary classification (human-written vs. AI-generated), when evaluating on the ternary classification setting, we convert our ternary labels (human-written, AI-generated, AI-refined) into binary targets by treating both fully LM-generated and LM-refined code snippets as AI-generated.

[4] GitHub: pemistahl/lingua-py

[5] `https://gptzero.me/`

[6] Owing to the cost structure of the paid API, we selected a representative sample of 500 code snippets for each label-language and label-domain pair

	Model	2-Class				3-Class			
		General	Algorithmic	Research/DS	Avg.	General	Algorithmic	Research/DS	Avg.
Zero-Shot Baselines	Fast-DetectGPT (Bao et al., 2024)	75.07	63.05	65.43	67.85	66.43	62.90	64.30	64.54
	Codet-M4 (Orel et al., 2025)	53.41	44.63	65.43	54.49	41.90	46.06	55.43	47.80
	M4 (Wang et al., 2024)	50.17	57.91	44.67	50.92	56.46	58.13	51.21	55.27
	GPTSniffer (Nguyen et al., 2024)	54.25	36.85	32.10	41.07	45.22	31.75	39.88	38.95
	GPTZero	54.05	71.96	44.73	56.91	50.56	66.13	30.62	49.10
OOD Evaluation	DroidDetect$_{CLS}$-Base$_{General}$	99.30	53.73	76.46	76.50	93.05	46.22	76.99	72.09
	DroidDetect$_{CLS}$-Base$_{Algorithmic}$	49.63	98.26	60.78	69.56	47.86	92.84	56.58	65.76
	DroidDetect$_{CLS}$-Base$_{Research/DS}$	47.01	48.02	72.55	55.86	47.86	38.73	59.97	48.85
Fine-Tuned Baselines	GCN	78.57	60.61	67.79	68.99	56.85	46.91	51.13	51.63
	CatBoost	89.69	87.29	77.21	84.73	78.86	74.01	64.07	72.31
Full Training	DroidDetect$_{CLS}$-Base	99.22	98.22	87.57	95.00	92.78	93.05	74.46	86.76
	DroidDetect$_{CLS}$-Large	**99.38**	**98.39**	**93.24**	**97.00**	**93.08**	**92.86**	**80.42**	**88.78**

Table 3: Comparison of models in 2-Class (human- vs machine-generated) and 3-Class (human- vs machine-generated vs machine-refined) classification setups across programming languages in terms of weighted F1-score. In the OOD section, we show models trained on each domain individually. The best results are shown in **bold**.

For a fairer comparison between alternative architectural choices, we fine-tuned some additional models using a multi-class classification objective on our dataset: *(i)* a simple **GCN model** (Kipf and Welling, 2017), trained on our dataset (details in Appendix C.1) and *(ii)* A **CatBoost classifier** (Prokhorenkova et al., 2018), trained following a procedure similar to the one used in the *Whodunit* paper (Appendix C.2). Moreover, in order to stress-test the backbone of the DroidDetect suite, we fine-tuned two encoder-only transformer models, ModernBERT-Base and ModernBERT-Large (Warner et al., 2024b): DroidDetect$_{CLS}$-Base and DroidDetect$_{CLS}$-Large(details in Section 6).

We also evaluated the DroidDetect$_{CLS}$-Base backbone in OOD settings under *language shift* and *domain shift* conditions. Comparing the multi-domain and the multi-lingual performance of the baselines to our backbone models trained on restricted data splits allows us *(i)* to uncover possible shortcomings in the training data curation of popular baseline models, as they can be compared head-to-head to both the split-specific and fully-trained variants of our backbone, and *(ii)* to assess the inherent ease with which models are able to transfer along these settings, by-proxy outlining the value of extensive training data curation. We selected the base version of the backbone for this restricted training scenario since it was comparable to most of the chosen baselines in terms of size.

Tables 3 and 4 simultaneously highlight the significant challenges that our test dataset poses for existing baseline models, along with the value of training on our extensive training split. These zero-shot baselines not only underperform compared to simpler fine-tuned baselines, such as GCN or CatBoost, but also fall well short with respect to models trained on specific OOD subsets. Table 4 further shows that, under restricted training conditions, models tend to generalise better to syntactically similar languages. For example, a model trained on C/C++ performs reasonably well on C# and Java. However, for topologically isolated languages such as Python, all models not trained specifically for it tend to struggle in this setting.

Among the baselines evaluated, Fast-DetectGPT consistently yields strong performance across both languages and domains, outperforming all other baselines. In contrast, pre-trained models usually perform well only on languages and domains that are closely aligned with their original training data. This highlights the limitations of previously collected datasets, which do not cover the diversity of generations in DroidCollection, and hence are far from being useful in real-life scenarios.

Unsurprisingly, the DroidDetect$_{CLS}$ models trained on the full training set achieve the highest performance, nearing ideal scores in both binary and ternary classification tasks, with the benefits of parameter count apparent in the domination of DroidDetect$_{CLS}$-Large across all settings. See Appendix D.2 for further OOD stress-testing of the DroidDetect$_{CLS}$ models.

5 Adversarial Samples

With the development of post-training techniques such as PPO (Schulman et al., 2017), DPO (Rafailov et al., 2023), and GRPO (Shao et al., 2024), it has become possible to set up the training in adversarial ways that enable LMs to evade AI-generated code detectors. Prior work by Shi et al. (2024) and Sadasivan et al. (2023) has shown that LM-generated content detectors are vul-

	Model	2-Class							3-Class						
		C/C++	C#	Go	Java	Python	JS	Avg.	C/C++	C#	Go	Java	Python	JS	Avg.
Zero-Shot Baselines	Fast-DetectGPT (Bao et al., 2024)	81.33	72.77	81.16	76.03	73.60	74.59	76.58	77.85	66.37	72.73	69.45	70.34	69.11	70.98
	Codet-M4 (Orel et al., 2025)	61.12	50.68	19.66	56.15	58.75	41.44	47.97	53.81	40.74	18.28	45.26	53.51	36.09	41.28
	M4 (Wang et al., 2024)	62.22	40.73	57.59	48.39	61.47	64.44	52.81	65.33	50.38	60.49	56.25	61.21	53.64	57.92
	GPTSniffer (Nguyen et al., 2024)	63.02	48.90	79.89	40.30	38.34	45.94	52.40	64.18	42.29	76.19	34.94	34.94	47.22	49.96
	GPTZero	58.32	45.69	13.64	74.65	73.19	63.16	54.81	61.00	50.38	28.89	61.25	52.63	54.78	51.48
OOD Evaluations	DroidDetect$_{\text{CLS}}$-Base$_{\text{C/C++}}$	98.98	96.59	67.32	96.97	74.45	91.15	87.58	92.62	81.67	56.43	79.45	56.43	69.72	72.72
	DroidDetect$_{\text{CLS}}$-Base$_{\text{C\#}}$	93.66	99.20	78.89	95.20	71.13	89.87	87.99	80.95	92.93	57.74	84.17	54.25	65.18	71.04
	DroidDetect$_{\text{CLS}}$-Base$_{\text{Go}}$	93.33	86.00	98.94	89.97	71.45	88.72	88.07	80.74	63.61	92.93	74.18	50.38	65.37	71.20
	DroidDetect$_{\text{CLS}}$-Base$_{\text{Java}}$	95.53	96.42	94.57	99.31	75.59	80.26	90.28	85.00	84.43	58.85	93.38	63.25	64.57	74.91
	DroidDetect$_{\text{CLS}}$-Base$_{\text{Python}}$	80.27	85.48	82.28	88.80	98.85	86.62	86.75	67.59	75.56	53.70	79.31	93.08	69.96	73.20
	DroidDetect$_{\text{CLS}}$-Base$_{\text{JS}}$	95.76	97.38	75.27	96.45	68.98	97.80	88.61	87.96	87.58	52.78	86.32	60.78	89.67	77.52
Fine-Tuned Baselines	GCN	79.06	78.33	84.33	80.04	72.49	69.69	77.32	65.97	58.03	65.20	60.13	55.22	54.72	59.88
	CatBoost	94.00	91.20	90.57	92.26	89.51	82.55	90.02	84.57	81.32	81.54	82.42	78.15	70.98	78.83
Full Training	DroidDetect$_{\text{CLS}}$-Base	99.29	99.33	99.32	99.45	98.87	98.38	99.11	94.43	94.06	93.98	93.93	93.95	90.99	93.56
	DroidDetect$_{\text{CLS}}$-Large	**99.31**	**99.51**	**99.32**	**99.45**	**99.11**	**98.67**	**99.23**	**94.24**	**93.87**	**94.42**	**94.05**	**94.13**	**91.27**	**93.66**

Table 4: Comparison of models in 2-class (human- vs. machine-generated) vs. 3-class (human- vs. machine-generated vs. machine-refined) classification setups across programming languages in terms of weighted F1-score. In the OOD section, we train on each programming language individually. The best results are highlighted in **bold**.

	FastDetectGPT	GPTSniffer	M4	Codet-M4	GPT-Zero	DroidDetect$_{\text{CLS}}$-Base	DroidDetect$_{\text{CLS}}$-Large
Human-written	0.84	0.65	0.40	0.38	0.53	0.93	**0.98**
Adversarial samples	0.48	0.49	0.73	0.63	0.10	**0.92**	**0.92**

Table 5: Recall for human-written vs. adversarial examples. The red cells show that despite having high recall on adversarial samples, M4 and Codet-M4 struggle to detect human-written code. The best results are in **bold**.

nerable to adversarial attacks and spoofing. This motivates the inclusion of adversarial samples in DroidCollection to improve model robustness.

To this end, we introduce two types of adversarial attacks: prompt-based attacks and preference-tuning-based attacks. In the prompt-based setting, we construct adversarial prompts by instructing the model to "write like a human" in multiple ways, relying on the models' parametric knowledge of how to produce outputs that mimic human-authored code and thus challenge detection systems. In the preference-tuning-based setting, we curate DroidCollection-Pref, a dataset of 157K paired examples consisting of human-written and LM-generated code responses to the same prompt. Using DroidCollection-Pref, we train LMs with up to 9B parameters –including LLaMA, Qwen, and Yi–, using LoRA(Hu et al., 2022) with rank 128 and DPO for two epochs. These models' output distributions are, in effect, steered towards preferring human-like code, making them less likely to contain the stylistic giveaways of machine-generated code. Once trained, the models are used to generate new "machine-humanised" code samples. We filter their generations as in Section 3.5 to keep only high-quality adversarial examples. As a result, we obtained a nearly 1:1 ratio of prompt-based vs. preference-tuning adversarial attacks.

Table 5 shows that these adversarial samples are difficult for existing detectors to identify. M4 and Codet-M4 exhibit high recall, but they also show low recall on human-written texts. GPT-Zero performs the worst, with a recall of only 0.10. In comparison, even DroidDetect-Base achieves a recall above 0.9.

6 Detector Training and Ablations

With the aim to optimise detector performance, we conducted a series of ablation experiments starting from our DroidDetect$_{\text{CLS}}$ backbone to systematically identify the most effective model architecture and training strategy.

We began by exploring whether incorporating the structural representation of code could improve the detector's performance. Specifically, we trained a 4-layer Graph Convolutional Network over the AST representation of codes to evaluate its ability to distinguish AI-generated from human-written code. The results are shown in Appendix C.1. We can see that while structural signals are informative, GCNs alone are not sufficient to achieve strong generalisation.

Next, we explored early fusion of textual and structural representations by combining a text encoder with a GCN encoder. For text encoding, we used ModernBERT (Warner et al., 2024b), a transformer-based model pre-trained on a mixture of natural language and code. We experimented with both the base (149M parameters) and the large (396M parameters) variants. This model was selected for the inference efficiency (Warner et al., 2024b), and suitability for code-related tasks. However, as shown in Appendix C.3, this fusion strategy yielded only a marginal improvement. Consequently, we decided to use a text-only encoder for the final model.

We then address the issue of class separability, which can arise because adversarial and refined code can intuitively be similar to human-written code. We explore training our models using triplet loss (Hoffer and Ailon, 2018) in a supervised contrastive (Khosla et al., 2020) setup using the class labels. This metric-learning approach encourages the model to place samples of the same class closer to each other in the embedding space while pushing dissimilar samples apart, and it has been demonstrably effective in other detection scenarios that require high precision (Deng et al., 2022; Li and Li, 2024). We refer to these models as $\texttt{DroidDetect}_{\texttt{SCL}}$. Table 6 shows how metric learning has a mild but consistent positive impact on performance across the board.

Finally, we addressed the problem of noisy and mislabelled training data. Despite extensive data filtering, it is possible that some code samples curated as human-written may have been generated by coding-copilots. The presence of such examples could negatively impact the training of our detector. To address this, we applied Monte Carlo Dropout (Hasan et al., 2022) to estimate the model uncertainty on the human-written portion of the test set. Specifically, we identified the top 7% most uncertain samples –those for which a pre-trained model exhibited low prediction confidence–, and resampled the dataset, removing them from the training set. We then retrained the model on the remaining data, thereby getting rid of the influence of potentially mislabelled or ambiguous samples. This manner of self-bootstrapping datasets has an extensive track record in image (Yalniz et al., 2019; Xie et al., 2020) and text (Wang et al., 2022) representation learning, relying on the tendency of neural networks to understand patterns in clean labels before they overfit to noisy data (Feldman and Zhang, 2020). Incorporating this filtering into our training yielded our final DroidDetect models, which, as outlined in Table 6, achieved the best performance across size and classification settings. We include our mined uncertainty metadata in DroidCollection to enable further analysis, filtering, or alternative labelling strategies in future work.

We trained all models for 3 epochs, using AdamW (Loshchilov and Hutter, 2019) optimiser, setting the top learning rate to 5e-5, and applying the linear warmup (proportion 0.1) with cosine decay learning rate scheduler. The batch size is 64 for DroidDetect-Base and 40 for DroidDetect-Large.

Model Variant	2-class		3-class		4-class	
	Base	Large	Base	Large	Base	Large
DroidDetect	**99.18**	**99.25**	**94.36**	**95.17**	**92.95**	**94.30**
- Resampling [$\texttt{DroidDetect}_{\texttt{SCL}}$]	99.15	99.22	93.86	94.43	92.52	93.14
- Triplet Loss [$\texttt{DroidDetect}_{\texttt{CLS}}$]	99.14	99.18	90.51	94.07	89.63	92.65

Table 6: Weighted F1-score for DroidDetect across training ablations. The best results are shown in **bold**.

7 Conclusion and Future Work

We have curated DroidCollection, a large and diverse suite of datasets that facilitate the training and the evaluation of robust AI-generated code detectors to support their most common modes of operation, i.e., completion and rewriting, along with potentially adversarial use cases. Among openly available corpora for training AI-generated content detectors, DroidCollection offers the most exhaustive coverage with respect to the number of generators, generation settings, programming languages, and domains covered. We further developed DroidDetect, a suite of AI-assisted code detection models, in two sizes (Base and Large), and conducted extensive ablation studies to evaluate which training strategies yield the most effective performance for this task.

In future work, we plan to enhance the coverage of DroidCollection and the robustness of DroidDetect. Specifically, we plan to incorporate code samples from additional closed-source API-based generators, thus broadening the diversity of the code samples. We further plan to incorporate generations from popular reasoning or thinking LMs in order to enhance the applicability of our detectors. Finally, we plan to expand language coverage to include languages such as PHP, Rust, and Ruby, thereby making our benchmarks more representative of the global programming landscape.

Limitations

Corpus Updates and Coverage Possessing a perfect coverage over all major models in the current fast-paced release environment is an intractable task. We acknowledge that the release of new model families with unseen output distributions presents a challenge for all AI-generated content detectors. Since we have mature pipelines for machine-generated, machine-refined and adversarially-humanised data acquisition, we plan to update `DroidCollection` with generations sourced from future model releases.

Cost Effectiveness Owing to cost realities, the majority of training samples in our study are sourced from locally deployable models. The high costs of API invocations are the primary reason why our study leaves data collection from recently released reasoning/thinking models such as Anthropic's `Claude 3.7`, DeepSeek `R1`, and Google's `Gemini 2.5` for future work. For similar reasons, our evaluation of API-based detectors such as GPTZero was limited to a subset of the test set.

Potential Data Contamination In spite of the thorough curation and extensive filtering undertaken for `DroidCollection`, we acknowledge the possibility that a small number of AI-generated or AI-assisted code samples may still be mislabeled as human-authored, due to the inherent nature of the data sources used for the dataset construction. Seeking to limit the negative effects of mislabeled or noisy data, our work explores uncertainty-based dataset re-sampling using a pre-trained classifier, which we show to be effective in improving the model's performance by identifying ambiguous samples to discard during training. In the released dataset, we include flags for code snippets identified as suspicious, enabling downstream users to apply additional filtering or analysis as needed.

Ethics Statement

The human-written code samples in our dataset are sourced exclusively from publicly available code corpora vetted for appropriate licensing and PII removal. Additionally, all code generation was conducted in compliance with the terms of use of the respective model providers.

`DroidDetect` and `DroidCollection` aim to promote transparency in code authorship, especially in academic and research settings. While there is a risk that they could be misused to train models to evade detection, we strongly discourage any malicious or privacy-invasive applications. We advocate for the responsible use in strictly legitimate research and educational contexts.

References

Mervat Abassy, Kareem Ashraf Elozeiri, Alexander Aziz, Minh Ngoc Ta, Raj Vardhan Tomar, Bimarsha Adhikari, Saad El Dine Ahmed, Yuxia Wang, Osama Mohammed Afzal, Zhuohan Xie, Jonibek Mansurov, Ekaterina Artemova, Vladislav Mikhailov, Rui Xing, Jiahui Geng, Hasan Iqbal, Zain Muhammad Mujahid, Tarek Mahmoud, Akim Tsvigun, and 5 others. 2024. Llm-detectaive: a tool for fine-grained machine-generated text detection. *CoRR*, abs/2408.04284.

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, and 8 others. 2024. Phi-4 technical report. *Preprint*, arXiv:2412.08905.

01. AI, :, Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Guoyin Wang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, and 14 others. 2025. Yi: Open foundation models by 01.ai. *Preprint*, arXiv:2403.04652.

Mistral AI. 2025. Mistral small – a new balance of performance and efficiency. Online. Available at `https://mistral.ai/news/mistral-small-3` (Accessed: 1 April 2025).

Guangsheng Bao, Yanbin Zhao, Zhiyang Teng, Linyi Yang, and Yue Zhang. 2024. Fast-detectgpt: Efficient zero-shot detection of machine-generated text via conditional probability curvature. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *CoRR*, abs/2207.14255.

A.Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29.

Sufiyan Ahmed Bukhari. 2024. Issues in detection of ai-generated source code. *University of Calgary*.

Savvas Chamezopoulos, Drahomira Herrmannova, Anita De Waard, Drahomira Herrmannova, Domenic Rosati, and Yury Kashnitsky. 2024. Overview of the DagPap24 shared task on detecting automatically generated scientific paper. In *Proceedings of the Fourth Workshop on Scholarly Document Processing (SDP 2024)*, pages 7–11, Bangkok, Thailand. Association for Computational Linguistics.

Jiankang Deng, Jia Guo, Jing Yang, Niannan Xue, Irene Kotsia, and Stefanos Zafeiriou. 2022. Arcface: Additive angular margin loss for deep face recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 44(10):5962–5979.

Liam Dugan, Alyssa Hwang, Filip Trhlík, Andrew Zhu, Josh Magnus Ludan, Hainiu Xu, Daphne Ippolito, and Chris Callison-Burch. 2024. RAID: A shared benchmark for robust evaluation of machine-generated text detectors. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12463–12492, Bangkok, Thailand. Association for Computational Linguistics.

Omer Dunay, Daniel Cheng, Adam Tait, Parth Thakkar, Peter C. Rigby, Andy Chiu, Imad Ahmad, Arun Ganesan, Chandra Shekhar Maddila, Vijayaraghavan Murali, Ali Tayyebi, and Nachiappan Nagappan. 2024. Multi-line ai-assisted code authoring. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 150–160. ACM.

Vitaly Feldman and Chiyuan Zhang. 2020. What neural networks memorize and why: Discovering the long tail via influence estimation. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Alexander Frömmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamblin, Pierre-Antoine Manzagol, Marcus Revaj, Maxim Tabachnyk, Daniel Tarlow, Kevin Villela, Daniel Zheng, Satish Chandra, and Petros Maniatis. 2024. Resolving code review comments with machine learning. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2024, Lisbon, Portugal, April 14-20, 2024*, pages 204–215. ACM.

Kazuki Fujii, Yukito Tajima, Sakae Mizuki, Hinari Shimada, Taihei Shiotani, Koshiro Saito, Masanari Ohi, Masaki Kawamura, Taishi Nakamura, Takumi Okamoto, Shigeki Ishida, Kakeru Hattori, Youmi Ma, Hiroya Takamura, Rio Yokota, and Naoaki Okazaki. 2025. Rewriting pre-training data boosts llm performance in math and code. *Preprint*, arXiv:2505.02881.

Tao Ge, Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. 2024. Scaling synthetic data creation with 1,000,000,000 personas. *Preprint*, arXiv:2406.20094.

Team Gemma, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, and 89 others. 2024. Gemma: Open models based on gemini research and technology. *Preprint*, arXiv:2403.08295.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Biyang Guo, Xin Zhang, Ziyuan Wang, Minqi Jiang, Jinran Nie, Yuxuan Ding, Jianwei Yue, and Yupeng Wu. 2023. How close is chatgpt to human experts? comparison corpus, evaluation, and detection. *arXiv preprint arxiv:2301.07597*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

Md Mehedi Hasan, Abbas Khosravi, Ibrahim Hossain, Ashikur Rahman, and Saeid Nahavandi. 2022. Controlled dropout for uncertainty estimation. *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 973–980.

Xinlei He, Xinyue Shen, Zeyuan Chen, Michael Backes, and Yang Zhang. 2024. Mgtbench: Benchmarking machine-generated text detection. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 2251–2265. ACM.

Elad Hoffer and Nir Ailon. 2018. Deep metric learning using triplet network. *Preprint*, arXiv:1412.6622.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search. *Preprint*, arXiv:1909.09436.

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*, MSR

'24, page 394–406, New York, NY, USA. Association for Computing Machinery.

Daphne Ippolito, Daniel Duckworth, Chris Callison-Burch, and Douglas Eck. 2020. Automatic detection of generated text is easiest when humans are fooled. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1808–1822, Online. Association for Computational Linguistics.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. 2025. Testgeneval: A real world unit test generation and test completion benchmark. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Ganesh Jawahar, Muhammad Abdul-Mageed, and Laks V. S. Lakshmanan. 2020. Automatic detection of machine generated text: A critical survey. In *Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020*, pages 2296–2309. International Committee on Computational Linguistics.

Jessica Ji, Jenny Jun, Maggie Wu, and Rebecca Gelles. 2024. Cybersecurity risks of ai-generated code. Technical report, Center for Security and Emerging Technology. Center for Security and Emerging Technology.

JianWang, Shangqing Liu, Xiaofei Xie, and Yi Li. 2024. An empirical study to evaluate aigc detectors on code content. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 844–856, New York, NY, USA. Association for Computing Machinery.

Jonathan Katzy, Razvan Mihai Popescu, Arie van Deursen, and Maliheh Izadi. 2025. The heap: A contamination-free multilingual code dataset for evaluating large language models. *Preprint*, arXiv:2501.09653.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Ryuto Koike, Masahiro Kaneko, and Naoaki Okazaki. 2024. OUTFOX: llm-generated essay detection through in-context learning with adversarially generated examples. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 21258–21266. AAAI Press.

Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and Zhaoxiang Zhang. 2023a. Sheetcopilot: Bringing software productivity to the next level through large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Ke Li, Sheng Hong, Cai Fu, Yunhe Zhang, and Ming Liu. 2023b. Discriminating human-authored from chatgpt-generated code via discernable feature analysis. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023 - Workshops, Florence, Italy, October 9-12, 2023*, pages 120–127. IEEE.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023c. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023d. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Xianming Li and Jing Li. 2024. Aoe: Angle-optimized embeddings for semantic textual similarity. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 1825–1839. Association for Computational Linguistics.

Yafu Li, Qintong Li, Leyang Cui, Wei Bi, Zhilin Wang, Longyue Wang, Linyi Yang, Shuming Shi, and Yue Zhang. 2024. MAGE: Machine-generated text detection in the wild. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 36–53, Bangkok, Thailand. Association for Computational Linguistics.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2025. Mathcoder2: Better math reasoning from continued pretraining on model-translated mathematical code. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4768–4777, Red Hook, NY, USA. Curran Associates Inc.

Dominik Macko, Róbert Móro, Adaku Uchendu, Jason Samuel Lucas, Michiharu Yamashita, Matús Pikuliak, Ivan Srba, Thai Le, Dongwon Lee, Jakub Simko, and Mária Bieliková. 2023. Multitude: Large-scale multilingual machine-generated text detection benchmark. *CoRR*, abs/2310.13606.

Dung Nguyen Manh, Nam Le Hai, Anh T. V. Dau, Anh Minh Nguyen, Khanh Nghiem, Jin Guo, and Nghi D. Q. Bui. 2023. The vault: A comprehensive multilingual dataset for advancing code understanding and generation. *Preprint*, arXiv:2305.06156.

Simone Martini. 2015. Several types of types in programming languages. In *History and Philosophy of Computing - Third International Conference, HaPoC 2015, Pisa, Italy, October 8-11, 2015, Revised Selected Papers*, volume 487 of *IFIP Advances in Information and Communication Technology*, pages 216–227.

Elyas Masrour, Bradley Emi, and Max Spero. 2025. DAMAGE: detecting adversarially modified AI generated text. *CoRR*, abs/2501.03437.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, and 27 others. 2024. Granite code models: A family of open foundation models for code intelligence. *Preprint*, arXiv:2405.04324.

Diego Mollá, Qiongkai Xu, Zijie Zeng, and Zhuang Li. 2024. Overview of the 2024 ALTA shared task: Detect automatic AI-generated sentences for human-AI hybrid articles. In *Proceedings of the 22nd Annual Workshop of the Australasian Language Technology Association*, pages 197–202, Canberra, Australia. Association for Computational Linguistics.

Vijayaraghavan Murali, Chandra Shekhar Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, Nachiappan Nagappan, and Peter C. Rigby. 2024. Ai-assisted code authoring at scale: Fine-tuning, deploying, and mixed methods evaluation. *Proc. ACM Softw. Eng.*, 1(FSE):1066–1085.

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*, page 112059.

Henrique Gomes Nunes, Eduardo Figueiredo, Larissa Rocha Soares, Sarah Nadi, Fischer Ferreira, and Geanderson E. dos Santos. 2025. Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects. *CoRR*, abs/2502.02368.

Daniil Orel, Dilshod Azizov, and Preslav Nakov. 2025. Codet-m4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings. *Preprint*, arXiv:2503.13733.

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*, ICSE-SEET '24, page 1–11, New York, NY, USA. Association for Computing Machinery.

Indraneil Paul, Haoyi Yang, Goran Glavaš, Kristian Kersting, and Iryna Gurevych. 2025. Obscuracoder: Powering efficient code lm pre-training via obfuscation grounding. *Preprint*, arXiv:2504.00019.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Commun. ACM*, 68(2):96–105.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. Catboost: unbiased boosting with categorical features. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *Preprint*, arXiv:2105.12655.

Ben Puryear and Gina Sprint. 2022. Github copilot in the classroom: learning to code with ai assistance. *J. Comput. Sci. Coll.*, 38(1):37–47.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan

Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, and 25 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, volume 36, pages 53728–53741. Curran Associates, Inc.

Vinu Sankar Sadasivan, Aounon Kumar, Sriram Balasubramanian, Wenxiao Wang, and Soheil Feizi. 2023. Can ai-generated text be reliably detected? *CoRR*, abs/2303.11156.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

Tatiana Shamardina, Vladislav Mikhailov, Daniil Cherniankskii, Alena Fenogenova, Marat Saidov, Anastasiya Valeeva, Tatiana Shavrina, Ivan Smurov, Elena Tutubalina, and Ekaterina Artemova. 2022. Findings of the the ruatd shared task 2022 on artificial text detection in russian. In *Computational Linguistics and Intellectual Technologies*, page 497–511. RSUH.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300.

Zhouxing Shi, Yihan Wang, Fan Yin, Xiangning Chen, Kai-Wei Chang, and Cho-Jui Hsieh. 2024. Red teaming language model detectors with language models. *Transactions of the Association for Computational Linguistics*, 12:174–189.

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross J. Anderson, and Yarin Gal. 2024. AI models collapse when trained on recursively generated data. *Nat.*, 631(8022):755–759.

Dan Su, Kezhi Kong, Ying Lin, Joseph Jennings, Brandon Norick, Markus Kliegl, Mostofa Patwary, Mohammad Shoeybi, and Bryan Catanzaro. 2024. Nemotron-cc: Transforming common crawl into a refined long-horizon pretraining dataset. *CoRR*, abs/2412.02595.

Zhenpeng Su, Xing Wu, Wei Zhou, Guangyuan Ma, and Songlin Hu. 2023. Hc3 plus: A semantic-invariant human chatgpt comparison corpus. *CoRR*, abs/2309.02731.

Eduard Tulchinskii, Kristian Kuznetsov, Laida Kushnareva, Daniil Cherniavskii, Sergey I. Nikolenko, Evgeny Burnaev, Serguei Barannikov, and Irina Piontkovskaya. 2023. Intrinsic dimension estimation for robust detection of ai-generated texts. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA. Association for Computing Machinery.

Veniamin Veselovsky, Manoel Horta Ribeiro, and Robert West. 2023. Artificial artificial artificial intelligence: Crowd workers widely use large language models for text production tasks. *CoRR*, abs/2306.07899.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. 2022. Text embeddings by weakly-supervised contrastive pre-training. *CoRR*, abs/2212.03533.

Yuxia Wang, Jonibek Mansurov, Petar Ivanov, Jinyan Su, Artem Shelmanov, Akim Tsvigun, Chenxi Whitehouse, Osama Mohammed Afzal, Tarek Mahmoud, Toru Sasaki, Thomas Arnold, Alham Fikri Aji, Nizar Habash, Iryna Gurevych, and Preslav Nakov. 2024. M4: Multi-generator, multi-domain, and multi-lingual black-box machine-generated text detection. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1369–1407, St. Julian's, Malta. Association for Computational Linguistics.

Yuxia Wang, Artem Shelmanov, Jonibek Mansurov, Akim Tsvigun, Vladislav Mikhailov, Rui Xing, Zhuohan Xie, Jiahui Geng, Giovanni Puccetti, Ekaterina Artemova, Jinyan Su, Minh Ngoc Ta, Mervat Abassy, Kareem Ashraf Elozeiri, Saad El Dine Ahmed El Etter, Maiya Goloburda, Tarek Mahmoud, Raj Vardhan Tomar, Nurkhan Laiyk, and 7 others. 2025. GenAI content detection task 1: English and multilingual machine-generated text detection: AI vs. human. In *Proceedings of the 1stWorkshop on GenAI Content Detection (GenAIDetect)*, pages 244–261, Abu Dhabi, UAE. International Conference on Computational Linguistics.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. 2024a. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. *CoRR*, abs/2412.13663.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. 2024b. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. *CoRR*, abs/2412.13663.

Maurice Weber, Daniel Y. Fu, Quentin Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, Ben Athiwaratkun, Rahul Chalamala, Kezhen Chen, Max Ryabinin, Tri Dao, Percy Liang, Christopher Ré, Irina Rish, and Ce Zhang. 2024a. Redpajama: an open dataset for training large language models. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, and Sven Mayer. 2024b. Significant productivity gains through programming with large language models. *Proc. ACM Hum. Comput. Interact.*, 8(EICS):1–29.

Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. 2024. Inversecoder: Self-improving instruction-tuned code llms with inverse-instruct. *Preprint*, arXiv:2407.05700.

Qizhe Xie, Minh-Thang Luong, Eduard H. Hovy, and Quoc V. Le. 2020. Self-training with noisy student improves imagenet classification. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 10684–10695. Computer Vision Foundation / IEEE.

Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. 2025a. Distinguishing llm-generated from human-written code by contrastive learning. *ACM Trans. Softw. Eng. Methodol.*, 34(4).

Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. 2025b. Distinguishing llm-generated from human-written code by contrastive learning. *ACM Trans. Softw. Eng. Methodol.*, 34(4).

I. Zeki Yalniz, Hervé Jégou, Kan Chen, Manohar Paluri, and Dhruv Mahajan. 2019. Billion-scale semi-supervised learning for image classification. *CoRR*, abs/1905.00546.

Xianjun Yang, Kexun Zhang, Haifeng Chen, Linda R. Petzold, William Yang Wang, and Wei Cheng. 2023. Zero-shot detection of machine-generated codes. *CoRR*, abs/2310.05103.

Contents

1 Introduction 1

2 Related Work 2
- 2.1 AI-Generated Text Detection . . . 2
- 2.2 AI-Generated Code Detection . . 2
- 2.3 Adversarial Evasion of AI-Generated Content Detectors . . . 3

3 The `DroidCollection` Corpus 3
- 3.1 Human-Authored Code Acquisition 3
- 3.2 AI-Authored Code Generation . . 4
- 3.3 Machine-Refined Data 4
- 3.4 Varying Decoding Strategies . . . 5
- 3.5 Data Filtering 5
- 3.6 Comparing the Resulting Dataset to Existing Ones 5

4 Why Dataset Coverage Matters? 5

5 Adversarial Samples 6

6 Detector Training and Ablations 7

7 Conclusion and Future Work 8

A List Of Models Used 15

B Dataset Creation and Statistics 15
- B.1 Inverse Instructions Setup 15
- B.2 `DroidCollection-Personas` creation 16
- B.3 Dataset Statistics 16

C Detailed Architectural Ablations 17
- C.1 GCN Experiments 17
- C.2 CatBoost Experiments 17
- C.3 Does Structure-Based Late-Fusion Improve Robustness? 17

D `DroidDetect` Stress Tests 17
- D.1 Input Length Stress Tests 17
- D.2 Additional OOD Stress Testing . . 17

E Qualitative Examples 19
- E.1 Inverse Instructions Examples . . 19
- E.2 Dataset Samples 19

A List Of Models Used

Table 7 illustrates that we are using a diverse set of models from 11 model families, combining both instruct and base versions of models. We are also covering a diverse set of sizes: from 2B up to 72B, and use both open-weights and API-based models.

Model Family	Model
Yi	Yi-Coder-9B Yi-Coder-9B-Chat Yi-Coder-1.5B-Chat Yi-Coder-1.5B
GPT	GPT-4o-mini GPT-4o
Qwen	Qwen2.5-Coder-7B Qwen2.5-Coder-7B-Instruct Qwen2.5-Coder-1.5B-Instruct Qwen2.5-Coder-32B-Instruct Qwen2.5-72B-Instruct Qwen2.5-Coder-1.5B Qwen2.5-Coder-14B-Instruct
Gemma	codegemma-7b-it codegemma-7b codegemma-2b
CodeLlama	CodeLlama-70b-Instruct-hf CodeLlama-34b-Instruct-hf CodeLlama-7b-hf
Deepseek	deepseek-coder-6.7b-instruct deepseek-coder-6.7b-base deepseek-coder-1.3b-instruct deepseek-coder-1.3b-base
Granite	granite-8b-code-instruct-4k granite-8b-code-base-4k
Llama	Llama-3.1-8B-Instruct Llama-3.2-3B Llama-3.1-70B-Instruct Llama-3.3-70B-Instruct Llama-3.3-70B-Instruct-Turbo Llama-3.2-1B Llama-3.1-8B
Phi	Phi-3-small-8k-instruct Phi-3-mini-4k-instruct phi-4 Phi-3-medium-4k-instruct phi-2 Phi-3.5-mini-instruct
Mistral	Mistral-Small-24B-Instruct-2501
StarCoder	starcoder2-15B starcoder starcoder2-7b starcoder2-3b

Table 7: Model families and their selected models used in `DroidCollection`.

B Dataset Creation and Statistics

B.1 Inverse Instructions Setup

For inverse instruction creation, we applied 4 LLMs: GPT-4o-mini, Llama3.1 8B, Qwen2.5 7B, and Phi-3 small (7B). These models were given the code, and they were asked to generate their summary and a prompt which could result in an LLM generating it. The prompt is given on Listing 1.

Listing 1: Prompt for code analysis and LLM prompt generation

```
# Code Analysis and LLM Prompt
    Generation

You are an experienced software engineer
     using `{language}` programming
    language skilled in analyzing,
    summarizing, and writing code. When
    provided with code, you break it
    down into its constituent parts,
    summarize its functionality
    concisely, and create prompts to
    guide an LLM in replicating similar
    outputs.

## Your Tasks:
1. **Code Summary**: Analyze the given
    code and summarize its purpose,
    logic, and functionality. Enclose
    this summary within [SUMMARY] and [/
    SUMMARY] tags.
2. **Prompt Creation**: Write a clear
    and specific LLM prompt that, if
    provided to a language model, would
    generate code with similar
    functionality and structure. Enclose
     the LLM prompt within [LLM_PROMPT]
    and [/LLM_PROMPT] tags.

Interaction will be in the following way
    :

### INPUT:
[CODE]
{{code}}
[/CODE]

### OUTPUT:
[SUMMARY]
{{summary}}
[/SUMMARY]

[LLM_PROMPT]
{{prompt}}
[/LLM_PROMPT]
```

Examples of codes, and corresponding inverse instructions are in Tables 13 to 15.

B.2 `DroidCollection-Personas` creation

To generate `DroidCollection-Personas`, we started by identifying the main characteristics of a programmer. Our final list contains 9 features: Primary Programming Language, Preferred Frameworks, Field of Work, Code Commenting Style, Error-Proneness, Debugging Strategies, Code Aesthetics, Documentation Habits, Function Length Preference. The possible values for each feature are listed in Table 8.

Then we did a Cartesian product to combine all the possible combinations of these properties, and started generating the tasks, which could be performed by this programmer. For task generation, we used the GPT-4o model, and prompted it in the way shown in Listing 2.

Listing 2: Prompt for Persona's task generation

```
I have the following description
    of a programmer:
{description}
Write a non-trivial programming
    task
which matches what this person
    probably does at work,
you can ignore some of the person
    's traits. Return only the
    task.
```

After the tasks were generated, we deduplicated them using MinHash with the same parameters as for the dataset filtering. After that, the resulting tasks were used for code generation.

Property Name	Values / Options
Primary Programming Language	Python, Java, JavaScript, PHP, C, C#, C++, Go, Ruby, Rust
Field of Work	Web Development, AI/ML, Game Development, System Programming, Embedded Systems, Data Engineering, Research, Distributed Systems Developer, IoT
Code Commenting Style	Concise, Detailed, Minimal
Error-Proneness	High, Medium, Low
Debugging Strategies	Print Statements, Debugger, Logging
Code Aesthetics	Highly Readable, Functional, Minimalist, Hard to Comprehend
Documentation Habits	Detailed, Minimal, Occasional
Function Length Preference	Short, Medium, Long

Table 8: List of attributes and characteristics in `DroidCollection-Personas`.

B.3 Dataset Statistics

In this section, we present key statistics of our dataset and compare them with existing alternatives. As shown in Table 9, our dataset includes a broader class distribution and shows greater diversity in code structure, as reflected by higher AST depth percentiles and longer line lengths. It suggests that our dataset captures more complex and varied code patterns, making it a more challenging and real-life-oriented benchmark for evaluating AI-generated code detection models. The importance of varying code lengths and difficulties is also shown in Appendix D.1. We also show the number of samples per generator, and programming language (not considering the datasets with <= 2 languages or generators). Several qualitative examples of samples belonging to different classes in our dataset are shown in Tables 16 and 17.

C Detailed Architectural Ablations

C.1 GCN Experiments

We used a simple 4-layer Graph Convolutional Network (GCN) to evaluate how effectively a GCN can capture structural and semantic features of code. As input, we utilised AST representations of the code, treating them as graphs. To assess the impact of node-level information, we experimented with three types of node features:

- **Dummy features** – no meaningful features were provided at the node level;
- **One-hot encoded node types** – encoding the syntactic type of each AST node;
- **Node content embeddings** – textual embeddings derived from the string content of each node. To reduce computational overhead, we used the `HashingVectorizer`, which converts strings into sparse vectors by hashing tokens to fixed-dimensional indices without maintaining a vocabulary in memory.

As shown in Table 11, features based on the textual content of the node yielded the best performance, showing that the semantic information is important in distinguishing between human-written and AI-generated code.

C.2 CatBoost Experiments

Following the experimental methodology of Idialu et al. (2024) and Orel et al. (2025), we computed 733 statistical features that capture various structural properties of code. These include metrics such as the density of specific AST node types, average line length, whitespace ratio, and the number of empty lines, code maintainability index, among others.

These features were used to train CatBoost classifiers with automatically tuned hyperparameters. Figure 1 shows the top unique features ranked by SHAP (SHapley Additive exPlanations) values (Lundberg and Lee, 2017). Interestingly, the most informative features vary across the 2-, 3-, and 4-class classification tasks, suggesting that different granularities of classification are dependent on different aspects of code structure.

Nonetheless, some patterns persist across all setups. In particular, features related to the length of identifiers (variable names) and the density of comments consistently present as strong indicators for distinguishing AI-generated/Refined from human-written code.

C.3 Does Structure-Based Late-Fusion Improve Robustness?

To decide whether fusion is helpful for improving the detection, we combined the GCN from Appendix C.1 with our text-only classifier using early fusion of embeddings. We used OOD-based generalisation, and compared how well the models perform for 2, 3, and 4-class classification in OOD settings (since when trained directly, it is hard to measure the significance of the performance difference), and then compared in which scenarios each method provides a better weighted F1-score. Table 12 shows that there is no clear trend of one approach being better than another: in the binary classification task, there are more ties, fusion has a higher win-rate in 4-class classification, while the model without fusion performs best in the 3-class case. Then we compared how the difference in F1-scores between models compares to the interquartile range within the model's predictions. As shown in Figure 2, the interquartile range is much larger than the model difference, so both models with and without fusion perform nearly equally.

D `DroidDetect` Stress Tests

D.1 Input Length Stress Tests

Table 10 shows that while other approaches seem to work best on short code snippets, probably because they were trained on shorter code samples (mainly functions) as shown in Table 9, our models actually get better with longer code sequences. This matters because real code is not usually just a few lines long. Another important thing is how stable our models remain across different input lengths. When we cut the input from 512 to 128 tokens, `DroidDetect-Base` only drops 7.28 F1-score points (from 99.18 to 91.90), and `DroidDetect-Large` drops just 4.34 points (from 99.25 to 94.91). This consistency suggests the generalisability of our models to various inputs.

D.2 Additional OOD Stress Testing

To evaluate the generalisation ability of our models, we tested them on additional open-source datasets containing AI-generated code. Specifically, we sampled 15,000 examples from the Swallow-Code dataset (Fujii et al., 2025), a high-quality collection of Python code from The Stack v2 (Lozhkov

Metric	CoDet-M4	CodeGPTSensor	GptSniffer	DroidCollection
AST@75	**15.00**	12.00	**15.00**	**15.00**
AST@90	**18.00**	15.00	**18.00**	**18.00**
AST@99	23.00	20.00	23.15	**25.00**
Line@75	90.00	93.00	99.00	**107.00**
Line@90	113.00	112.00	117.00	**135.00**
Line@99	228.00	169.00	153.60	**314.00**
Class Distribution	AI - 50% Human - 50%	AI - 50% Human - 50%	AI - 90% Human - 10%	AI - 25% Human - 47% Refined - 13% Adv. - 15%
Avg. # of samples per language	**166,850**	-	-	148,491
Avg. # of samples per generator	**50,866**	-	-	8,458

Table 9: Comparison of AST depth percentile, line length percentile, class distribution, and average samples per language/generator between DroidCollection and existing datasets.

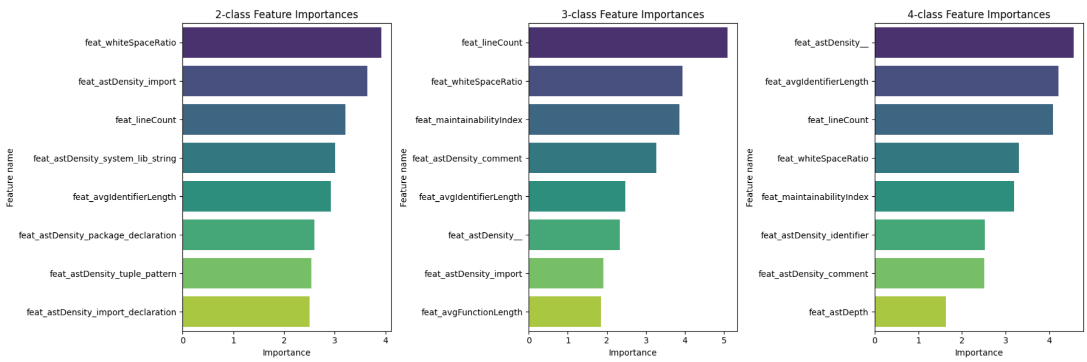

Figure 1: Feature importances

Model	Truncation Length		
	128	256	512
GptSniffer	57.05	57.20	56.64
M4	59.69	53.10	51.13
CoDet-M4	72.28	70.62	61.68
DroidDetect-Base	91.90	96.25	99.18
DroidDetect-Large	**94.91**	**98.31**	**99.25**

Table 10: Impact of input length truncation (measured using the ModernBERT tokeniser) on weighted F1-scores for binary classification. The most competitive numbers are highlighted in **bold**.

Features	2-class	3-class	4-class
Dummy	60.02	39.27	34.17
Node Type	50.12	39.54	33.12
Text	**76.67**	**59.10**	**51.14**

Table 11: Comparison of different feature types used as node-level features in a GCN, based on the weighted F1-score on the validation set. The most competitive numbers are highlighted in **bold**.

Classification	Tie (%)	With Fusion (%)	Without Fusion (%)
2-Class	60.0	40.0	0.0
3-Class	40.0	20.0	40.0
4-Class	20.0	60.0	20.0

Table 12: Comparative task-level win-rates of DroidDetect with and without GCN late-fusion aggregated over OOD classification tasks.

et al., 2024) synthetically refined by LLaMA3.3-70B-Instruct model. This dataset was concurrently released with our work and is highly unlikely to be part of the training distribution of any of our models, thus serving as a strong test for our models' recall on machine-rewritten code.

We also randomly selected 15,000 samples per programming language from The Heap (Katzy et al., 2025) dataset. This dataset contains illiberally licensed code with metadata about its presence in existing code-retraining corpora. We specifically filter for samples that are not exact- or near-duplicates with any sample in major pre-training corpora (Li et al., 2023c; Lozhkov et al., 2024; Weber et al., 2024a). Jointly, these ensure that our curated split is extremely unlikely to be seen by models during pre-training, thus constituting a stiff test of our detectors' recall on human-written code.

Both DroidDetect-Base and DroidDetect-Large models were tested on these datasets. On Swallow-Code, they achieved

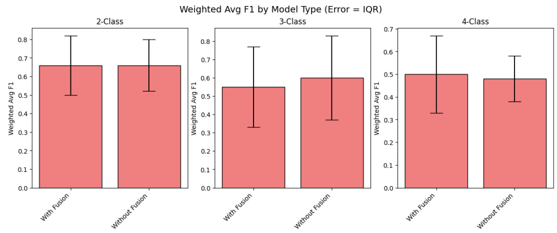

Figure 2: Weighted F1-score comparison between models with and without fusion

recall scores of 98.95% and 99.11% respectively. On The Heap, `DroidDetect-Base` reached 94.14%, while `DroidDetect-Large` achieved 96.28%. It shows that the models trained on our dataset can also work on other datasets robustly.

E Qualitative Examples

E.1 Inverse Instructions Examples

In Tables 13 to 15 we show examples of code with the corresponding inverse instructions.

E.2 Dataset Samples

In this appendix (Tables 16 and 17), we provide a small portion of code per class, written by different models in different languages. To check the diversity of our dataset, it is suggested to check the release repository[7].

[7] The release repository will be made public on acceptance.

<table>
<tr><th>Language</th><th>Instruction</th><th>Code</th></tr>
<tr><td>C++</td><td>Write a C-style function to retrieve the bus speed of a specific device, given its UART's baud clock divisor, base baud clock, and time base reading functionality. The function should set the DLAB, read the DLL and DLM, calculate the baud clock divisor, update the DLAB, and then loop to find the first edge of the baud clock, get the starting and ending time base values, and finally compute the bus speed from these values.</td><td>

```
static unsigned long __init
prpmc750_get_bus_speed(void)
{
unsigned long tbl_start, tbl_end;
unsigned long current_state, old_state, bus_speed;
unsigned char lcr, dll, dlm;
int baud_divisor, count;

/* Read the UART's baud clock divisor */
lcr = readb(PRPMC750_SERIAL_0_LCR);
writeb(lcr | UART_LCR_DLAB, PRPMC750_SERIAL_0_LCR);
dll = readb(PRPMC750_SERIAL_0_DLL);
dlm = readb(PRPMC750_SERIAL_0_DLM);
writeb(lcr & ~UART_LCR_DLAB, PRPMC750_SERIAL_0_LCR);
baud_divisor = (dlm << 8) | dll;

/*
* Use the baud clock divisor and base baud clock
* to determine the baud rate and use that as
* the number of baud clock edges we use for
* the time base sample. Make it half the baud
* rate.
*/
count = PRPMC750_BASE_BAUD / (baud_divisor * 16);

/* Find the first edge of the baud clock */
old_state = readb(PRPMC750_STATUS_REG) & PRPMC750_BAUDOUT_MASK;
do {
current_state = readb(PRPMC750_STATUS_REG) &
PRPMC750_BAUDOUT_MASK;
} while(old_state == current_state);

old_state = current_state;

/* Get the starting time base value */
tbl_start = get_tbl();

/*
* Loop until we have found a number of edges equal
* to half the count (half the baud rate)
*/
do {
do {
current_state = readb(PRPMC750_STATUS_REG) &
PRPMC750_BAUDOUT_MASK;
} while(old_state == current_state);
old_state = current_state;
} while (--count);

/* Get the ending time base value */
tbl_end = get_tbl();

/* Compute bus speed */
bus_speed = (tbl_end-tbl_start)*128;

return bus_speed;
}
```

</td></tr>
<tr><td>Python</td><td>Generate Python code that performs data reading, trimming, and datetime conversion for a set of files located in a specified directory. The files should be named in a pattern `met_sum_insitu_1_obop_hour_20<Year>.txt`, and the data should be trimmed to remove irrelevant columns, handle missing values, and create a datetime column using the existing year, month, day, and hour columns. The resulting data should be returned as a processed pandas DataFrame</td><td>

```
def metTrim():
# ---- initial reading of data
root = r'C:\Users\ARL\Desktop\MetData'
ext = list(range(12, 20)) # yearly extensions

colnames = ['na', 'yr', 'mo', 'dy', 'hr', 'dir', 'spd', 'steady', 'na', 'na', 'na', 'na', 'na', 'na']
met = pd.DataFrame(columns=colnames) # preallocate df
for yr in ext:
# read in data
data = pd.read_csv(root + r'\met_sum_insitu_1_obop_hour_20{}.txt'.format(yr), delim_whitespace=True,
header=None)
data.columns = colnames # apply col names
met = met.append(data) # append to list
print('Data Imported')

# ---- trimming data
met = met.drop('na', axis=1) # drop na cols
met = met.replace(-999.9, np.nan) # turn missing val to nan
met = met.replace(-9, np.nan)
met = met.replace(-999, np.nan)
met = met.replace(-99.9, np.nan)
met = met.dropna(axis=0, how='any') # remove rows with nan vals

# ---- convert date to datetime
metInt = met.applymap(int) # make sure values are ints
dates = createDatetime(metInt['yr'].values,
metInt['mo'].values,
metInt['dy'].values,
metInt['hr'].values)

met['datetime'] = dates # add it as a new column
met = met.drop(['yr', 'mo', 'dy', 'hr'], axis=1) # drop old date columns

return met
```

</td></tr>
</table>

Table 13: Code and Corresponding Inverse Instructions (C++, Python)

<table>
<tr><th>Language</th><th>Instruction</th><th>Code</th></tr>
<tr><td>Java</td><td>Given a file with blog entries and their respective dates , write a Java method to read the file , sort the entries in reverse chronological order , and separate them into lists for published and unpublished entries . The method should handle varying file existence conditions and log errors accordingly . Assume the blog entries are stored in lines of the file , with each entry consisting of a date in 'yyyyMMdd' format followed by the entry 's ID .</td><td><pre><code>private void readIndex(boolean published) {
File indexFile;
if (published) {
indexFile = new File(blog.getIndexesDirectory(), "blogentries-published.index");
} else {
indexFile = new File(blog.getIndexesDirectory(), "blogentries-unpublished.index");
}

if (indexFile.exists()) {
try {
BufferedReader reader = new BufferedReader(new FileReader(indexFile));
String indexEntry = reader.readLine();
while (indexEntry != null) {
indexEntries.add(indexEntry);

// and add it to the internal memory structures
Date date = new Date(Long.parseLong(indexEntry));
Day day = blog.getBlogForDay(date);

if (published) {
publishedIndexEntries.add(indexEntry);
day.addPublishedBlogEntry(indexEntry);
} else {
unpublishedIndexEntries.add(indexEntry);
day.addUnpublishedBlogEntry(indexEntry);
}

indexEntry = reader.readLine();
}

reader.close();
} catch (Exception e) {
log.error("Error while reading index", e);
}
}

Collections.sort(indexEntries, new ReverseBlogEntryIdComparator());
Collections.sort(publishedIndexEntries, new ReverseBlogEntryIdComparator());
Collections.sort(unpublishedIndexEntries, new ReverseBlogEntryIdComparator());
}</code></pre></td></tr>
<tr><td>JavaScript</td><td>Generate a JavaScript function named Teth , which is a constructor function , outside of its parent function . Teth should inherit all properties and methods from its parent function and add a new method - getChain . This method should return the string "teth" . The Teth function should attach its prototype to its parent 's prototype , moving it one level away . The code should utilize the Node.js environment and should include documentation to describe the purpose of the function and its newly added method .</td><td><pre><code>var Btc = require('./btc');
var bitcoin = require('bitcoinjs-lib');
var _ = require('lodash');

var Tbtc = function() {
// this function is called externally from BaseCoin
// replace the BaseCoin prototype with the local override prototype, which inherits from BaseCoin
// effectively, move the BaseCoin prototype one level away
this.__proto__ = Tbtc.prototype;
this.network = bitcoin.networks.testnet;
};

Tbtc.prototype.__proto__ = Btc.prototype;

Tbtc.prototype.getChain = function() {
return 'tbtc';
};

module.exports = Tbtc;</code></pre></td></tr>
</table>

Table 14: Code and Corresponding Inverse Instructions (Java, JavaScript)

Language	Instruction	Code

Go	Write a Go function that creates a set of constant labels for Prometheus metrics based on ISI cluster settings. The function should check if a site has been specified for the ISI cluster and create labels with both the cluster name and site name if available. Otherwise, it should create labels with only the cluster name. Include a log statement to display the created labels.	func CreateConstLabels() error {
		//Only create a const label for site if a site has been specified.
		if IsiCluster.Site != "" {
		ConstLabels = prometheus.Labels{"cluster": IsiCluster.Name, "site": IsiCluster.Site}
		} else {
		ConstLabels = prometheus.Labels{"cluster": IsiCluster.Name}
		}
		log.Debugf("ConstLables are %v", ConstLabels)
		return nil
		}

C#	Generate C# code that prompts the user for a string and then checks if it is a palindrome by comparing characters from the start and end of the string, moving towards the center. If the characters match, print a message indicating they are equal; otherwise, print a message indicating they are not equal. The code should handle strings of any length and input user characters until the string length is reached.	using System;
		
		namespace _76
		{
		class Program
		{
		static void Main(string[] args)
		{
		int cont = 0;
		
		Console.WriteLine("Quantos caracteres seu palindromo tem?");
		int max = int.Parse(Console.ReadLine());
		
		string [] caractere = new string [max];
		
		for (cont = 0; cont < max; cont++)
		{
		Console.WriteLine("Digite" + (cont + 1) + " caracteres da palavra");
		caractere[cont] = Console.ReadLine();
		}
		
		int inverso = cont;
		
		for (cont = 0; cont < max; cont++)
		{
		if (caractere[cont] == caractere[inverso])
		{
		Console.WriteLine("O " + (cont + 1) + " caractere " + (inverso + 1) + " caractere");
		}
		Console.WriteLine("O " + (cont + 1) + " caractere " + (inverso + 1) + " caractere ");
		inverso--;
		}
		}
		}
		}

Table 15: Code and Corresponding Inverse Instructions (Go, C#)

Language	Class	Generator	Code
Python	LLM-Refined (re-written)	Qwen2.5-72B	(see code below)
C	Human-written	Human	(see code below)

```python
from collections import defaultdict
class Solution:
MAXPRIME = 100001
isPrime = [0] * (MAXPRIME + 1)
isPrime[0] = isPrime[1] = -1
def __init__(self):
for i in range(2, MAXPRIME):
if isPrime[i] == 0:
isPrime[i] = i
for multiple in range(i * i, MAXPRIME + 1, i):
if isPrime[multiple] == 0:
isPrime[multiple] = i
def largestComponentSize(self, A):
label = defaultdict(int)
roots = {}
def find_root(key):
if key not in roots:
roots[key] = key
if roots[key]!= key:
roots[key] = find_root(roots[key])
return roots[key]
def merge_roots(k1, k2):
r1, r2 = find_root(k1), find_root(k2)
if r1!= r2:
r1, r2 = min(r1, r2), max(r1, r2)
label[r1] += label[r2]
roots[r2] = r1
return r1
for x in A:
root_id = None
prime_factors = set()
while self.isPrime[x]!= -1:
p = self.isPrime[x]
root_id = find_root(p) if root_id is None else merge_roots(root_id, p)
x //= p
label[root_id] -= 1
return -min(label.values())
```

```c
int
pack_white(char *ptr)
{
int cnt = 0;
char *tptr, ch;
if (ptr == NULL)
return (0);
tptr = ptr;
while (isspace(*tptr))
tptr++;
for (;;) {
while ((ch = *tptr) != '\0' && !isspace(ch)) {
*ptr++ = ch;
tptr++;
}
while (isspace(*tptr))
tptr++;
if (*tptr == '\0')
break;
*ptr++ = ' ';
cnt++;
}
*ptr = '\0';
return (cnt);
}
```

Table 16: Code samples by different models (Part 1)

<table>
<tr><th>Language</th><th>Class</th><th>Generator</th><th>Code</th></tr>
<tr><td>Go</td><td>Adversarial</td><td>Qwen2.5-Coder-7B-Instruct</td><td>

```
package main
import (
"context"
"fmt"
"golang.org/x/sync/errgroup"
)
func parallelExecute(ctx context.Context, runners ...func(context.Context) error) error {
g := new(errgroup.Group)
for _, runner := range runners {
runner := runner
g.Go(func() error {
return runner(ctx)
})
}
return g.Wait()
}
func main() {
ctx := context.Background()
runners := []func(context.Context) error{
func(ctx context.Context) error {
select {
case <-ctx.Done():
return ctx.Err()
case <-time.After(2 * time.Second):
return fmt.Errorf("runner 1 error")
}
},
func(ctx context.Context) error {
select {
case <-ctx.Done():
return ctx.Err()
case <-time.After(3 * time.Second):
return fmt.Errorf("runner 2 error")
}
},
}
err := parallelExecute(ctx, runners...)
if err != nil {
fmt.Println("First error encountered:", err)
}
}
```

</td></tr>
<tr><td>JavaScript</td><td>AI-Generated</td><td>Yi-Coder-9B</td><td>

```
class Vector2D {
#x;
#y;
constructor(x, y) {
this.#x = x;
this.#y = y;
}
setX(x) {
this.#x = x;
}
setY(y) {
this.#y = y;
}
getX() {
return this.#x;
}
getY() {
return this.#y;
}
add(vector) {
this.#x += vector.getX();
this.#y += vector.getY();
return this;
}
compare(vector) {
return this.#x === vector.getX() && this.#y === vector.getY();
}
}
```

</td></tr>
</table>

Table 17: Code samples by different models (Part 2)