
Ghost Marks in the Machine: A Critical Review of SynthID for Code Provenance Monitoring¹

Eve Sherratt-Cross* Theo Farrell* Sam Ogden* Oscar Ryley*

Durham University, Durham AI Safety Initiative

With
Apart Research

Abstract

AI-generated code is increasingly common in software and prone to security vulnerabilities. It is hence critical to monitor the origins of code used in secure applications. SynthID is a Google DeepMind method for watermarking AI-generated text, images and videos but there is currently no existing mechanism for code. We adapt various SynthID schemes to Python code, and analyse how effective they are using Bayesian detectors. We find that longer n-grams support more robust watermark detection, but the corresponding generated code is more prone to syntax and runtime errors. This work paves the way for critical future work, because code origin monitoring forms part of robust cyber defences against vulnerable or backdoored AI-generated code.

Code is available for reproducibility here:

<https://github.com/DurhamAISafety/ghost-marks-in-the-machine>

Keywords: AI security, safety infrastructure, software security

¹ Research conducted at the [Defensive Acceleration Hackathon](#), 2025

* Equal contribution

1. Introduction and Threat Model

With the growing popularity of large language models (LLMs) and other generative tools ([1], [2], [3]), there has been a growth in software containing AI generated code, with Microsoft CEO Satya Nadella suggesting as much as 30% of their code is now written by AI [1]. Recent studies have found between 50% and 70% of AI generated code contain security vulnerabilities ([2], [3], [4]), with potential causes including imperfect or flawed training data, iterative agentic development practices, and the underpinning nature of LLM architecture.

Current training methods introduce inherent insecurities to AI generated code as much of the training data is sourced from public open-source code platforms, with OpenAI's GPT-3, which served as a foundation for OpenAI's Codex model, including 54 million GitHub repositories in its training data [5]. This presents a significant threat to the security and quality of AI generated code when many repositories contain incomplete and insecure codebases. An NYU researchers study of GitHub Copilot's output found it produced buggy and insecure code ~40% of the time and this was largely attributed to unreliable, uncurated open-source code. LLMs also tend to generate code that is syntactically correct, but neglect code security unless otherwise prompted[2].

Agentic coding development methods also pose a significant security risk. Common agentic coding practice, particularly for individuals with minimal programming experience, is often characterised by rapid cycles of refinement without security concerned prompting, which will inevitably lead to insecure codebases. A counterintuitive phenomenon known as "feedback loop security degradation" occurs when initially secure code undergoes multiple rounds of AI-based "improvements" *without human intervention*. This process is associated with the introduction of new security issues[5]. Education of agentic coding practice is becoming ever important with the increasing prevalence of prompt injecting, and primarily for coding tasks, backdoored code present and immediate security risk.

Since AI-generated code is becoming increasingly common, flagging such code would be useful to help review it in more detail when used for security applications. We identify that while there is lots of existing research into safety evaluation of code, there is limited research on code origin monitoring / watermarking. Hence, we aim to contribute something to this literature.

Threat	Mitigation	Effect
AI generated code can introduce security vulnerabilities (whether due to malicious AI/actors or not)	Extend SynthID to work on AI generated code.	Flag GenAI content, for example to be reviewed in more detail.

Table 1. Our threat model.

SynthID [6] is a generative watermarking scheme for text, photos and videos which is used at scale in Google’s Gemini. It works by altering the underlying model’s output logits which increases the probability of different tokens being generated. This change is fixed to a secret key, which can also be used to detect the probability distribution in text without the LM. As such, any content generated by Google AI and pasted elsewhere (e.g. social media) can be pasted into a new Gemini instance which can use SynthID to confirm this. This is the only watermarking scheme used by any major LLM provider. However, SynthID has not been implemented for code, largely because code is low-entropy. That is, it is not as easy to replace a token in Python as it is in natural English, which contains many synonyms. We believe a SynthID-like solution for code watermarking could help monitor code origins.

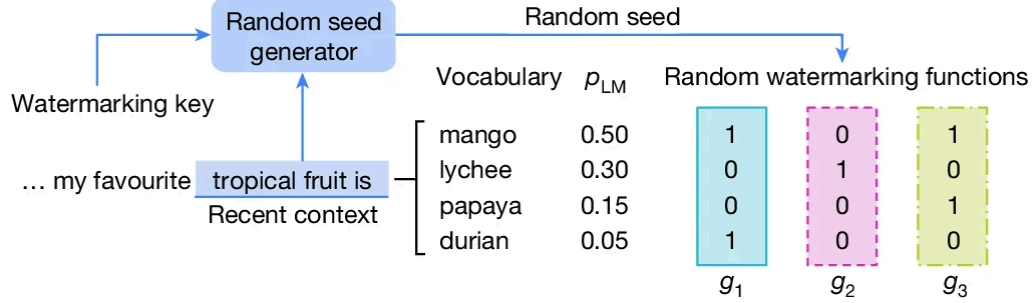
Our contributions are as follows:

- We adapt SynthID [5] to the setting of AI-generated code and demonstrate its application on Python code generation on a subset of the APPS dataset prompts [6].
- We provide a systematic analysis of the current capabilities and limitations of SynthID in this domain.
- We develop a CI/CD pipeline tool to enable continuous and automatic code provenance monitoring.
- SynthID NLP analysis to further ground our results.

2. Related Work

2.1 SynthID

LLM probabilities and random watermarking functions



Tournament sampling: over-generation with watermark-based iterative selection

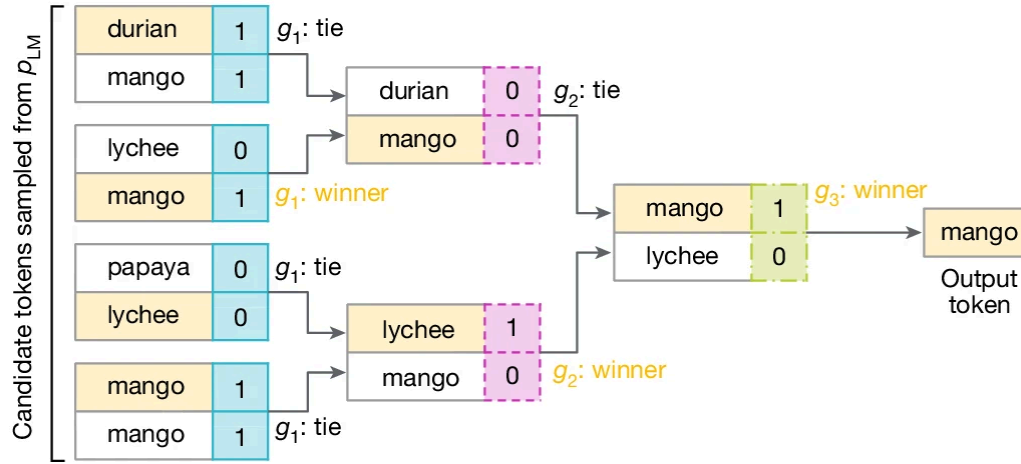


Fig 1. SynthID’s tournament-based watermarking

SynthID [5] is a generative watermarking scheme with three components: a random seed generator, a sampling algorithm, and a scoring function. To watermark a given token, the seed is defined as the hash of the preceding h tokens and a secret watermarking key. For every possible token in the vocabulary, the seed defines m ‘g-values’, which act as random deterministic scores. These g-values are stored in m ‘watermarking functions’, where each function has a g-value for each token. Tournament sampling is then used to select one of 2^m randomly sampled tokens from the language model (LM)’s original logits at that position. This works by pairing off the sampled tokens and selecting the one in each pair with the highest value in the first watermarking function. This means that tokens with higher g-values are favoured. The winners (ties are broken randomly) are then paired off and scored against the second watermarking function and so on until one token remains after m steps. This process is shown in Figure 1 for $m = 3$.

To detect watermarked text, only the watermarking key and LM-generated text are needed, where the text has had SynthID applied. These are input to a scoring function which recalculates the g-value for each token. Since SynthID favours tokens with high g-values, watermarked text will have a higher mean g-value

across tokens than unwatermarked text (which should have 0.5). Scoring can be achieved by simply comparing means, but Bayesian scoring gave better results.

3. Methodology

3.1 Setup

We use CodeGemma 7b IT [7] in our analyses. This provides good Python capabilities at manageable computational cost. We used Google DeepMind (GDM)’s SynthID GitHub repository [8] and the Hugging Face transformers library [9] for model loading and tokenisation. LM output is classified into 3 categories: passed and correct (runs and output correct), passed and incorrect (runs but incorrect output), and error (syntax or runtime error).

We use 1000 prompts from the APPS dataset [6], which contains 10,000 real interview-level coding questions for software engineers. We structured the prompt to ensure the model output was in python and encapsulated within a `solve()` function. We used a temperature of 0.7 and a Top-K value of 40.

3.2 Watermarking Implementation

We used GDM’s implementation of the SynthID watermarking scheme to modify the logit distributions based on tournament sampling, as described in previous sections. We configured the scheme to use a sample size of 2^{16} , a context window size of 1024 and a fixed set of watermarking keys, [101, 202, 303, 404, 505, 606, 707, 808, 909]. SynthID uses the preceding h tokens to generate a watermark. We tested 4 schemes with varying $h \in \{0, 2, 5, 10\}$ where $h = 0$ is no watermark applied. For standard text watermarking $h = 5$ is default [8] so we test above and below as well.

3.3 Watermark Detection

We analyse Bayesian scoring for watermark detection. Specifically, we train a detector for each $h > 0$ which computes the probability (i.e. confidence score) that the text is watermarked given the mean g-values. The detectors were trained with cross-entropy loss and L2 regularisation.

3.4 Evaluation Pipeline

To automate continuous code provenance monitoring, we integrate our code-focused SynthID classifier into a GitHub CI/CD pipeline triggered on pushes to repos and pull requests. This proof-of-concept demonstrates a practical approach for organizations tracking GenAI usage in their development workflows and poses particular value for companies with internal LLMs that could deploy their own SynthID infrastructure. This could also provide transparency for open-source projects whose contributors use AI tools.

Run test on changed files

```
1 ▶ Run python run_checks.py test.py
11 Checking: test.py
12 ✕ Watermark detected: code likely AI generated,
13 confidence: 0.82
14
15 =====
16 Results: 0/1 files passed
17
18 Failed files:
19 - test.py
20 Error: Process completed with exit code 1.
```

Run test on changed files

```
1 ▶ Run python run_checks.py test.py
11 Checking: test.py
12 ✓ Code unlikely AI generated,
13 confidence: 0.78
14
15 =====
16 Results: 1/1 files passed
```

Fig 2. GitHub code provenance monitoring workflow. The process fails if the code is detected as watermarked.

4. Watermark Detection in Code

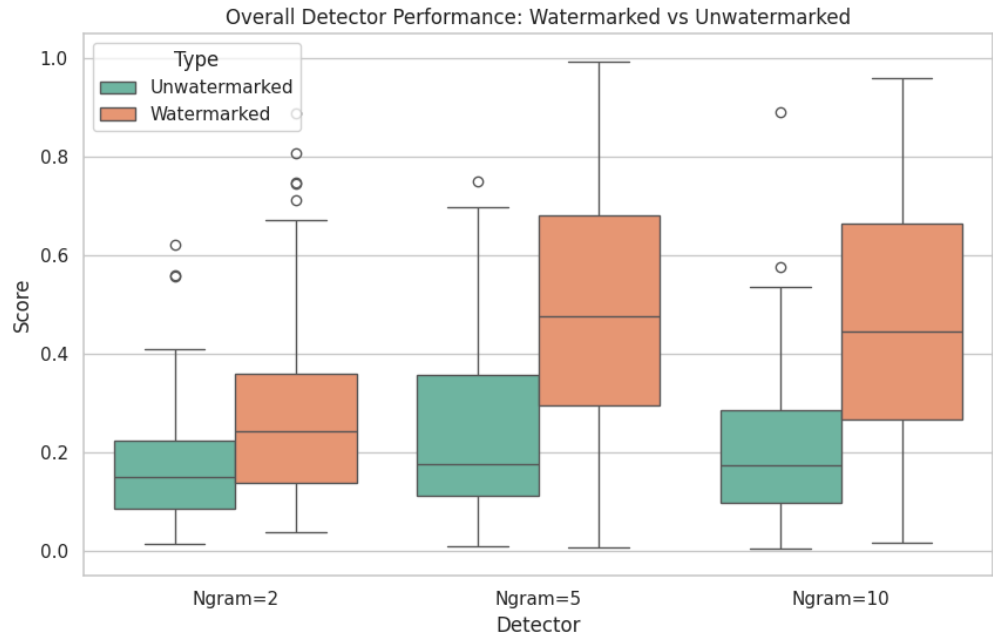


Fig 3. Overall Bayesian detector performance. Ngram refers to h in this paper.

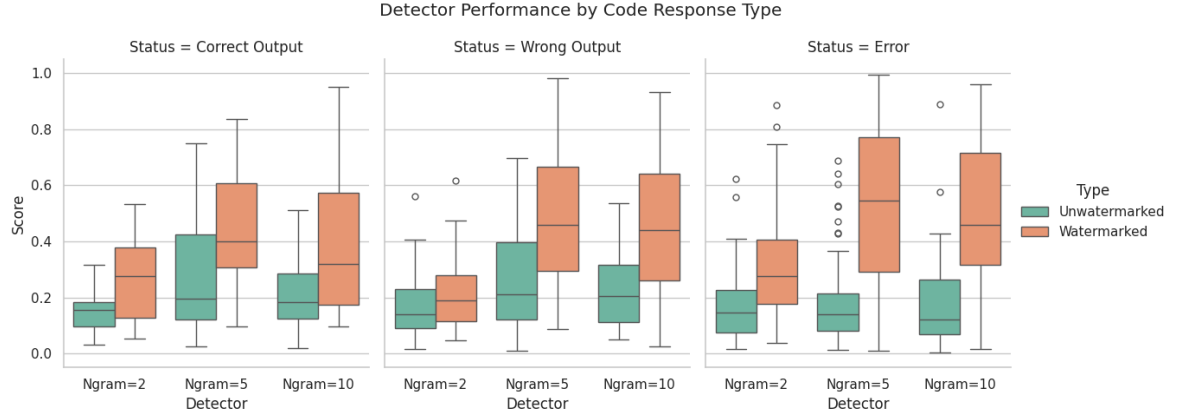


Fig 4. Bayesian detector performance by LM output category. The leftmost two are where the LM runs and gives an output, the right plot is when the model gives a runtime or syntax error.

The Bayesian detectors were somewhat effective at discerning watermarked code. Figure 1 shows the overall performance, and it is clear that larger h is generally more effectively watermarked. This is intuitive as the watermark is informed by more context. However, Figure 2 shows the performance by output category and $h = 10$ also correlates with more errors. We hypothesise that this is a causal relationship as larger h tends to introduce more artifacts. Additionally, these longer contextual relationships are more easily broken by refactoring, for example. Shorter contexts like $h = 2$ are more robust to formatting changes but are much harder to watermark, as evidenced by the lower scores for watermarked code.

5. Comparative N-Gram Analysis

To analyse the transferability of the SynthID method from regular text to code, we perform some NLP analysis.

Comparison of the frequency of n-grams on the watermarked and unwatermarked model responses from human eval data provided alongside the original Nature paper [5] revealed similarities and differences in watermarked vs unwatermarked data.

We also conducted TF-IDF [6] analysis on this data, which revealed minute differences in key word frequencies in text generated with or without SynthID methods (Fig 6).

More uncommon nouns are more likely to be distinctive to the watermarked language, whereas common and factual nouns like ‘time’ or structural phrases ‘combination of factors including’ or ‘play crucial role’ are less likely to be affected by having SynthID watermarks within them.

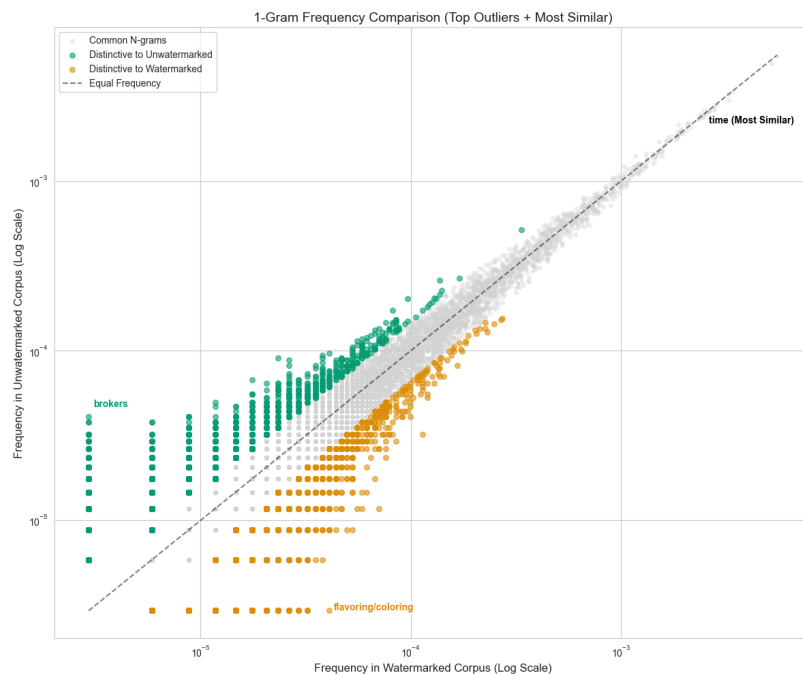


Fig 5. 1-gram/ word frequency comparison between SynthID watermarked vs unwatermarked natural language

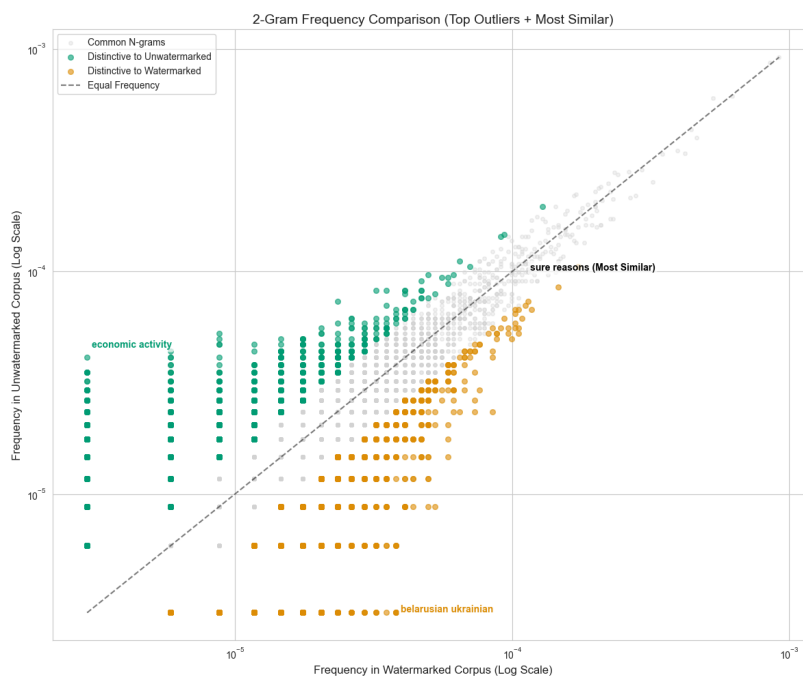


Fig 6. 2-gram frequency comparison between SynthID watermarked vs unwatermarked natural language

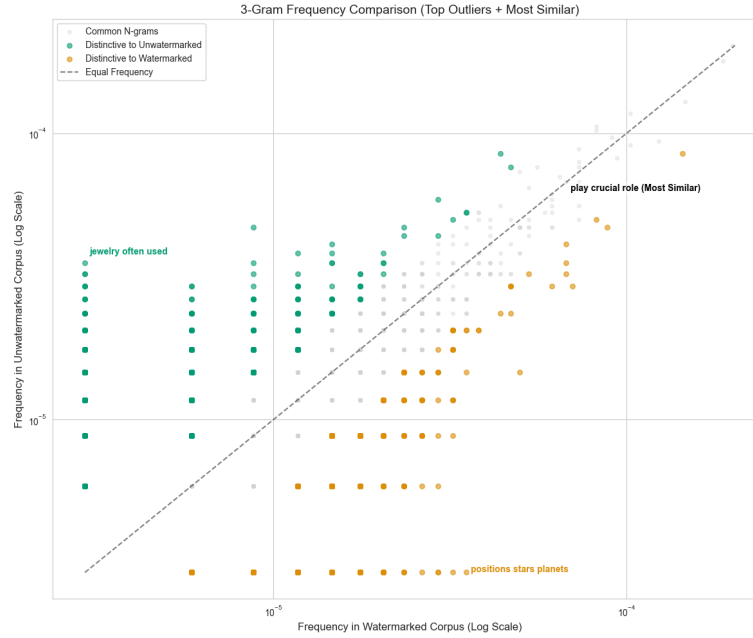


Fig 7. 3-gram frequency comparison between SynthID watermarked vs unwatermarked natural language

3-gram	Unwatermarked	Watermarked
('combination', 'factors', 'including')	68	67
('certain', 'medical', 'conditions')	60	63
('due', 'combination', 'factors')	43	49
('blood', 'sugar', 'levels')	28	48
('various', 'factors', 'including')	39	34
('number', 'factors', 'including')	31	41
('medical', 'conditions', 'certain')	35	27
('conditions', 'certain', 'medical')	34	27
('important', 'note', 'factors')	30	34
('may', 'vary', 'depending')	32	30
('consult', 'healthcare', 'professional')	27	31
('plays', 'crucial', 'role')	29	24

Table 2. Top 3-gram frequencies in watermarked vs unwatermarked data

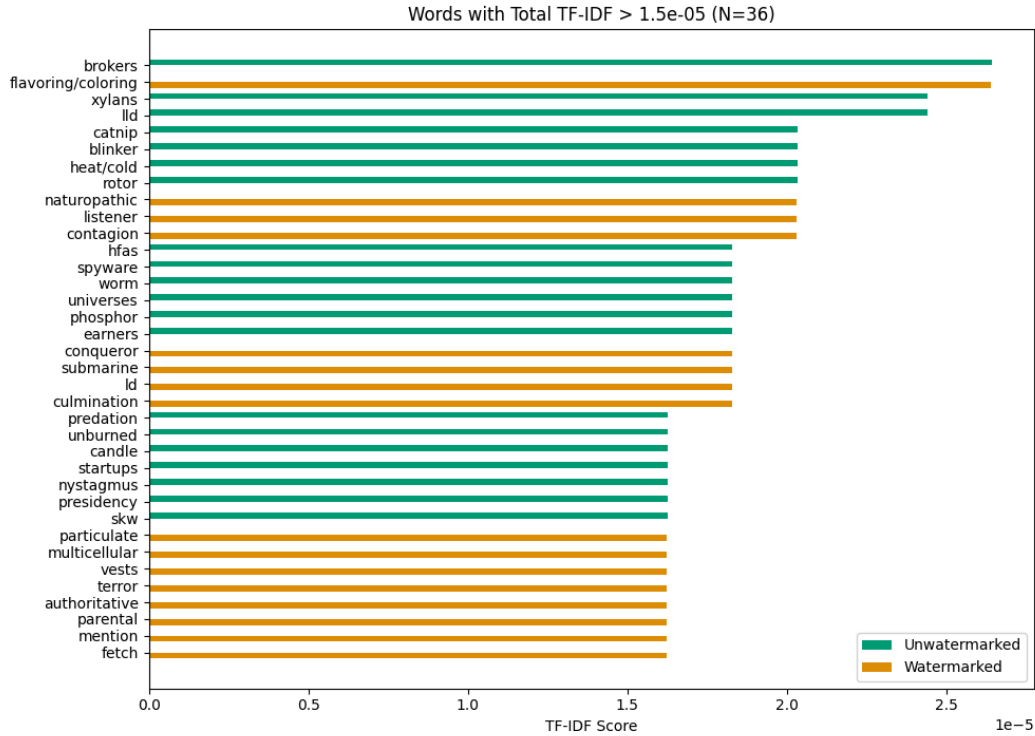


Fig 8. Term Frequency - Inverse Document Frequency of SynthID Watermarked vs Unwatermarked natural language model response

We adapt and utilise the same ngram frequency analysis to the generated code, comparing the different SynthID ngram lengths in watermarked model responses. This found similar patterns in the frequency comparisons.

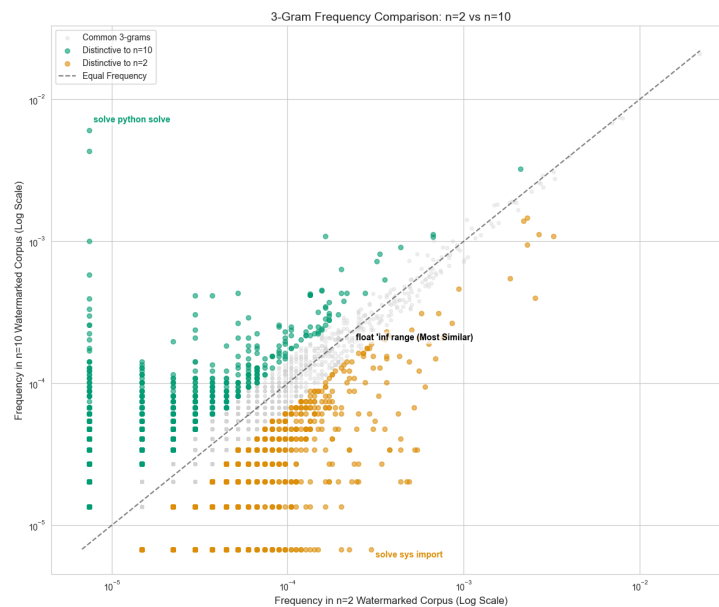


Fig 9. Comparison of 3-gram frequency given $n=10$ or $n=2$ SynthID ngram length

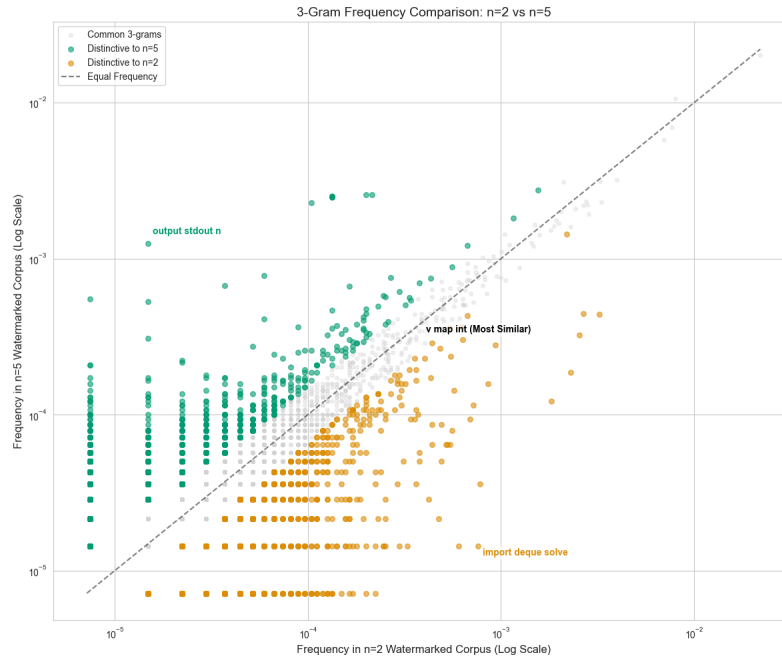


Fig 10. Comparison of 3-gram frequency given $n=5$ or $n=2$ SynthID ngram length

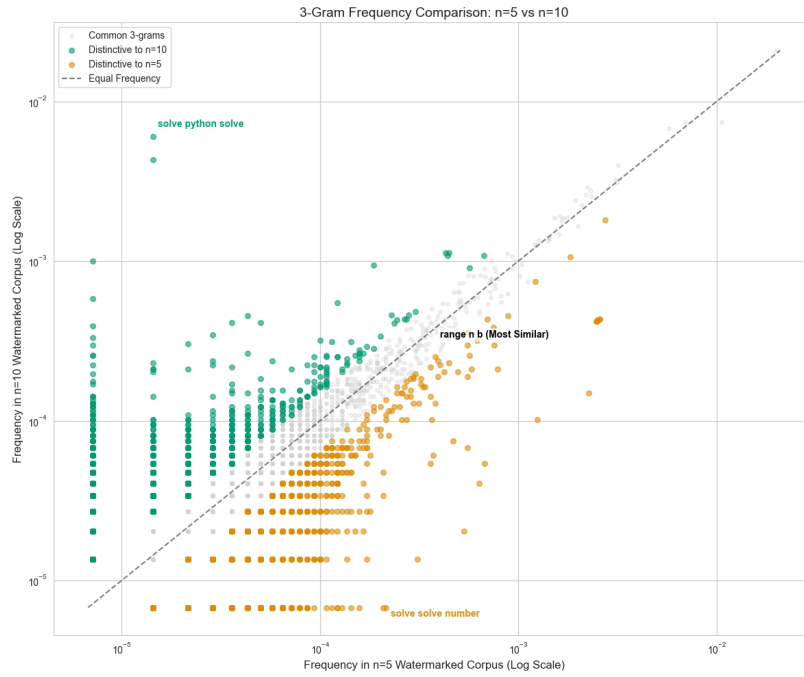


Fig 11. Comparison of 3-gram frequency given $n=10$ or $n=5$ SynthID ngram length

6. Discussion and Conclusion

Our findings provide a foundation for future work on watermarking AI-generated code. The trained Bayesian detectors were somewhat successful at detecting code that was watermarked by SynthID, which suggests that this approach is a promising avenue for research. Our continuous integration pipeline provides a practical approach to tracking AI-generated code in development workflows, which can be a valuable tool in a wider code quality monitoring infrastructure. However, this work is restricted to Python and a limited number of software engineering interview questions. We will explore different domains and programming languages in upcoming work.

Additionally, we do not rigorously test the robustness of watermarked code, although we performed some preliminary red-teaming. The adversarial attack strategies included variable and function name changing, reformatting, altering whitespace, and inserting comments. The aim was to mimic real-world attempts to evade detection or remove watermarks from code. The results suggested that longer, more complex code was more resistant to adversarial attack strategies as more complex codes dispersed the watermark across more locations and made it harder to remove.

Watermarking code would be a valuable tool for developers of secure applications and infrastructure. In open-source projects, it would be incredibly valuable to have watermarks on AI-generated pull requests that are not otherwise indicated as such. If code is flagged, then it can be put aside for manual or perhaps trusted LM review. In closed-source projects, watermarking can be used to ensure that employees are only using trusted and sanctioned LMs. We believe these problems will become even more common and that work on robust code watermarking will be a valuable defence and tool for resilience of cyber infrastructure.

7. Bibliography

- [1] Jordan Novet and Jonathan Vanian, ‘Satya Nadella says as much as 30% of Microsoft code is written by AI’, Apr. 29, 2025. [Online]. Available: <https://www.cnbc.com/2025/04/29/satya-nadella-says-as-much-as-30percent-of-microsoft-code-is-written-by-ai.html>
- [2] N. Tihanyi, T. Bisztray, R. Jain, M. A. Ferrag, L. C. Cordeiro, and V. Mavroeidis, ‘The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification’, in *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*, San Francisco CA USA: ACM, Dec. 2023, pp. 33–43. doi: 10.1145/3617555.3617874.
- [3] N. Tihanyi, T. Bisztray, M. A. Ferrag, R. Jain, and L. C. Cordeiro, ‘How secure is AI-generated code: a large-scale comparison of large language models’, *Empir. Softw. Eng.*, vol. 30, no. 2, p. 47, Mar. 2025, doi: 10.1007/s10664-024-10590-1.

- [4] ‘Cybersecurity Risks of AI-Generated Code’, Center for Security and Emerging Technology. Accessed: Nov. 23, 2025. [Online]. Available: <https://cset.georgetown.edu/publication/cybersecurity-risks-of-ai-generated-code/>
- [5] J. H. Klemmer *et al.*, ‘Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns’, in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, Dec. 2024, pp. 2726–2740. doi: 10.1145/3658644.3690283.
- [6] S. Dathathri *et al.*, ‘Scalable watermarking for identifying large language model outputs’, *Nature*, vol. 634, no. 8035, pp. 818–823, Oct. 2024, doi: 10.1038/s41586-024-08025-4.
- [7] D. Hendrycks *et al.*, ‘Measuring Coding Challenge Competence With APPS’, Nov. 08, 2021, *arXiv*: arXiv:2105.09938. doi: 10.48550/arXiv.2105.09938.
- [8] C. Team *et al.*, ‘CodeGemma: Open Code Models Based on Gemma’, June 19, 2024, *arXiv*: arXiv:2406.11409. doi: 10.48550/arXiv.2406.11409.
- [9] *google-deepmind/synthid-text*. (Nov. 23, 2025). Python. Google DeepMind. Accessed: Nov. 23, 2025. [Online]. Available: <https://github.com/google-deepmind/synthid-text>
- [10] T. Wolf *et al.*, ‘HuggingFace’s Transformers: State-of-the-art Natural Language Processing’, July 14, 2020, *arXiv*: arXiv:1910.03771. doi: 10.48550/arXiv.1910.03771.
- [11] [G. Salton and C. S. Yang, ‘ON THE SPECIFICATION OF TERM VALUES IN AUTOMATIC INDEXING’, *J. Doc.*, vol. 29, no. 4, pp. 351–372, Apr. 1973, doi: 10.1108/eb026562.](#) [ON THE SPECIFICATION OF TERM VALUES IN AUTOMATIC INDEXING’, *J. Doc.*, vol. 29, no. 4, pp. 351–372, Apr. 1973, doi: 10.1108/eb026562.](#)

8. Appendix

Security Considerations

We do not rigorously test the robustness of watermarked code through red-teaming, only performing preliminary analysis. This is a significant security consideration because we do not know how attackers might subvert SynthID for code, and how difficult this would be.

In future, we will:

- Extend the research to analyse how code reformatting (e.g. using Prettier) impacts SynthID detection.
- Extend analysis to the C programming language, for which security vulnerabilities are generally more common and severe than Python.
- Attempt further red-teaming to test the limits of SynthID on code.