Monthly Report
2026-7호

www.copyright.or.kr

저작권 기술 트렌드

COPYRIGHT TECH TREND

한국저작권위원회
KOREA COPYRIGHT COMMISSION

LLM 생성 코드의 출처 식별 기술 동향: 코드에 남은 흔적을 활용한 사후 탐지

뉴스 브리프

LLM을 활용한 코드 생성이 개발 현장으로 확산되면서, 특정 코드를 어떤 모델이 생성했는지 사후에 식별하는 기술의 필요성이 커지고 있다. LLM은 각기 미세하게 다른 코딩 스타일을 코드에 남기며, 이러한 흔적은 일종의 지문으로 작용해 생성 모델을 역추적하는 단서가 된다. DCAN은 소스코드에서 기능적 내용과 모델 고유의 스타일을 분리해 지문을 추출하는 기법으로, 네 개 프로그래밍 언어에서 높은 식별 성능을 나타냈다. GoCoMA는 소스코드와 컴파일 결과를 함께 분석하는 다중모드 방식으로 식별 성능을 개선했다. 두 기술은 통제된 평가 환경에서 높은 식별 성능을 보였지만, 코드 최적화·난독화와 같은 변형에 대한 강건성, 컴파일 환경 변화, 학습되지 않은 언어·모델에 대한 일반화는 아직 충분히 검증되지 않았다. 향후 강건성 확보, 신규 모델로의 확장, 접근 방식 다양화 등의 연구가 이어지며 활용 범위가 점차 넓어질 것으로 보인다.

뉴스 플러스

Ⅰ. 서론: LLM 코드 생성의 보편화와 출처 식별의 과제

• 바이브 코딩 확산과 LLM 코드 생성의 일상화

소프트웨어 개발 현장에서 대규모 언어 모델(LLM: large language model) 활용이 확산하고 있다. 개발자가 자연어로 요구사항을 기술하면 LLM이 이를 실행 가능한 코드로 변환하고, 디버깅과 테스트가 하나의 흐름으로 이어지는 방식이 자리 잡는 중이다. 이러한 개발 방식은 바이브 코딩(vibe coding)이라 불리며, 개발자에게 요구되는 역량 또한 코드를 직접 작성하는 능력에서 문제를 정의하고 구조를 설계하는 쪽으로 옮겨 가고 있다.[1)]

1) 김승기, "바이브 코딩 확산, '코드 없는 개발' 현실화…당근 과외까지 등장", 테크월드, 2026.04.23., https://www.epnc.co.kr/news/articleView.html?idxno=400792

이러한 변화는 기업과 교육 현장으로 이어지고 있다. 글로벌 IT 리서치 기업 가트너(Gartner)는 2028년까지 기업 소프트웨어 엔지니어의 90%가 AI 코드 어시스턴트를 사용할 것으로 전망했다. 초기 10%대에 머무르던 도입률이 2028년에는 크게 확대될 것이라는 관측이다. 개발 진입 장벽이 낮아지면서 비개발자까지 코드 작성에 참여하고 있으며, 국내에서는 중고 거래 플랫폼에 바이브 코딩 과외 공고가 등장하기도 했다.[2)]

LLM이 생성한 코드는 상용 소프트웨어와 오픈소스 저장소, 개인 프로젝트 등 다양한 영역에 걸쳐 유통되고 있다. 개발자가 직접 작성한 코드와 LLM이 생성한 코드가 하나의 프로젝트 안에 혼재하는 상황이 흔해지면서, 개별 코드의 기원을 확인하는 일이 새로운 기술적 과제로 대두하고 있다.

• LLM 코드 산출물의 출처 불분명성

LLM이 생성한 코드가 널리 활용되고 있으나 특정 코드가 어떤 모델에서 산출되었는지를 사후에 파악하기는 쉽지 않다. LLM의 코드 생성은 학습된 파라미터와 확률적 디코딩 과정을 거치므로, 완성된 코드만으로는 그 기원을 직접 확인하기 어렵다. 개발자가 여러 모델을 병행 사용하거나 LLM이 생성한 코드를 일부 수정해 프로젝트에 통합하는 경우, 이러한 불분명성은 한층 커진다.

출처 불분명성은 여러 영역에서 실무적 과제로 이어진다. 상용 소프트웨어에서 취약점이 발견되었을 때 해당 코드의 출처를 파악해 유사한 위험 코드를 선제적으로 점검하는 일, 시스템 장애나 보안 사고가 발생했을 때 문제가 된 코드의 출처를 규명하는 일이 대표적이다. 오픈소스 저장소에 유입된 코드가 특정 LLM에서 대규모로 생성된 것인지 확인하는 작업도 저장소의 품질 관리와 연관된다.[3)]

이러한 배경에서 대두한 것이 LLM 코드 출처 식별(LLMCSA: LLM code source attribution)이라는 과제이다. 기존 연구가 사람이 작성한 코드와 LLM이 작성한 코드를 구분하는 데 집중했다면, LLMCSA는 여러 LLM이 함께 사용되는 환경에서 특정 코드가 어느 모델에서 생성되었는지를 식별한다.[4)]

• 사후 탐지 접근법: 생성 지문 개념의 부상

LLM 코드 출처 식별 기술은 크게 두 방향으로 나뉜다. 코드 생성 시점에 식별 정보를 삽입해두는 워터마킹(watermarking) 방식과 이미 생성된 코드에서 흔적을 찾아내는 사후 탐지(post-hoc detection) 방식이다. 워터마킹은 모델 개발 단계에서 미리 도입하는 방식이라 이미 생성된 코드에는 적용되지 않는다. 반면 사후 탐지는 생성 과정에 접근하지 않고 이미 생성된 코드만으로 출처를 추정할 수 있어 실무적 활용 가능성이 있으며, 최근 관련 연구가 본격화되고 있다.

2) 김승기, "바이브 코딩 확산, '코드 없는 개발' 현실화…당근 과외까지 등장", 테크월드, 2026.04.23., https://www.epnc.co.kr/news/articleView.html?idxno=400792
3) Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212
4) Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212

사후 탐지 방식은 LLM마다 코드 생성 방식에 미세하지만 일관된 차이가 있다는 사실에서 출발한다. 동일한 프로그래밍 과제에 대해 서로 다른 LLM이 생성한 코드를 비교하면, 변수명을 짓는 방식, 반복문을 구성하는 구조, 주석을 작성하는 밀도와 형식 등에서 모델별로 일관된 경향이 나타난다. 이러한 경향성은 각 모델의 학습 데이터, 아키텍처, 정렬(alignment) 전략, 디코딩 방식의 차이에서 비롯되며, 코드 표면에 남는 통계적으로 구분 가능한 생성 지문(generative fingerprint)*으로 활용될 수 있다.[5)]

본 보고서는 이러한 생성 지문에 기반한 LLM 코드 출처 식별 기술의 원리와 연구 동향을 살펴본다. 먼저 코드의 기능적 내용과 모델 고유 스타일을 분리해 학습하는 프레임워크의 구조와 성능을 다룬다. 이어 소스코드와 바이너리 이미지(binary image)**를 결합한 다중모드(multimodal) 확장을 검토하고, 코드 수정과 난독화에 대한 강건성, 다국어 일반화 등 현재 기술의 한계를 짚는다. 마지막으로 이러한 한계를 보완하려는 최근 연구 흐름과 향후 전망을 정리한다.

* 생성 지문(generative fingerprint): LLM이 코드를 생성할 때 반복적으로 나타나는 모델 고유의 표현 패턴. 학계에서 아직 표준화된 용어는 아니며, 저자에 따라 스타일 지문(stylistic fingerprint), 코딩 개성(coding personality) 등으로도 불림.

** 바이너리 이미지(binary image): 컴파일된 코드의 이진 데이터(0과 1의 나열)를 픽셀 값으로 변환하여 시각적 이미지 형태로 표현한 자료. 코드가 실행 가능한 형태로 변환된 후의 저수준 구조 정보를 담고 있으며, 이미지 분석 기법을 적용해 코드의 특징을 추출하는 데 활용됨. 본론 2에서 다루는 BPEA 이미지가 이에 해당함.

II. 본론 1: 내용-스타일 분리 기반 LLM 코드 출처 식별

• 동일 과제, 다른 스타일: LLM별 코드 생성 특성

LLM 코드 출처 식별은 서로 다른 LLM이 생성한 코드에 모델별로 일관된 차이가 있다는 점에 기반한다. 최근 연구에서는 대표적인 LLM 네 종인 클로드(Claude), 챗GPT(ChatGPT), 딥시크(DeepSeek), 퀜(Qwen)에 동일한 프로그래밍 과제를 제시하고 생성된 코드의 특성을 정량적으로 비교했다. 실험은 C, 고(Go), 자바(Java), 파이썬(Python) 네 개 언어를 대상으로 하였으며, 주석을 포함한 경우와 포함하지 않은 경우를 나누어 조건별 스타일 차이를 살펴보았다.[6)]

실험에서 네 모델은 동일한 알고리즘 문제를 해결하는 경우에도 서로 구분되는 코드 작성 특성을 보였다. 예를 들어 같은 문제를 코드로 옮길 때, 한 모델은 조건 분기(if/elif)를 활용해 단계별 처리를 강조하는 방식을 선호하는 반면, 다른 모델은 스택 관련 연산(push/pop)이나 집계 함수(sum)를 활용해 문제를 해결한다. 변수명을 짓는 방식에서도 모델별 차이가 나타났다. 일부 모델은 의미가 드러나는 서술적 변수명(next_num 등)을 선호한 반면, 다른 모델은 짧은 이름을 사용하는 경향을 보였다.

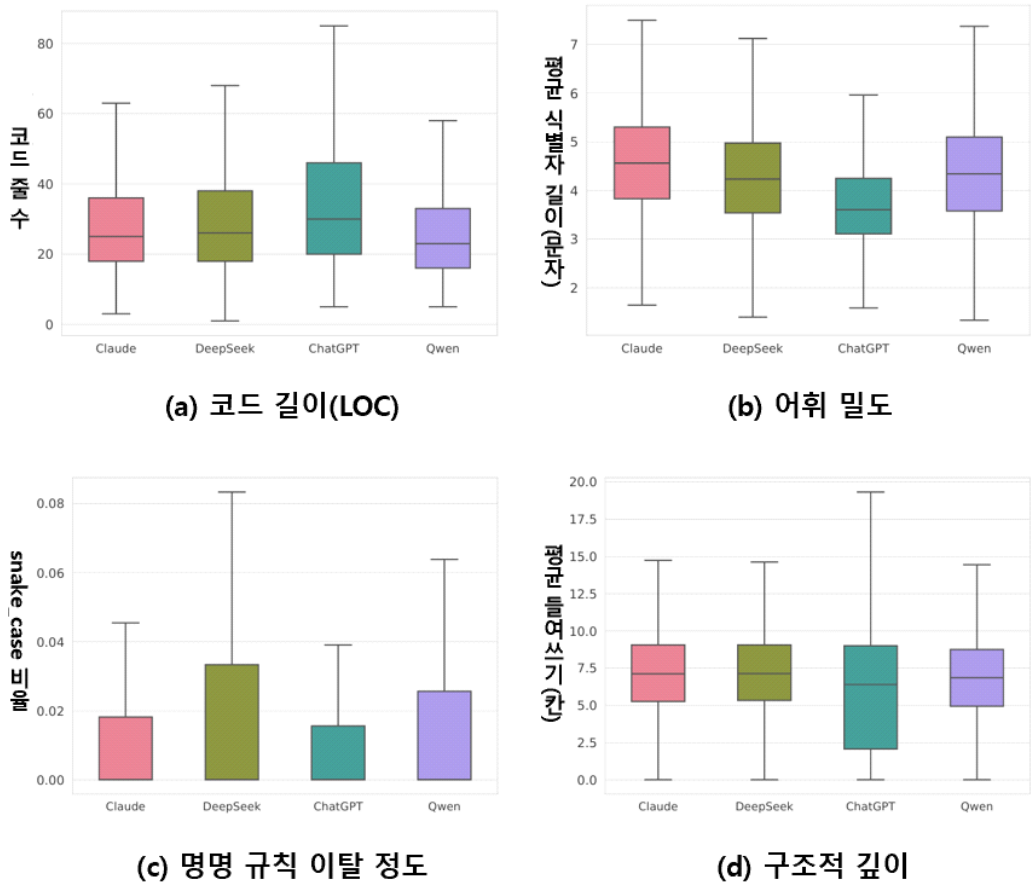
5) Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212
6) Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212

이러한 차이는 코드의 성능이나 정확성의 차이가 아니라 표현 방식의 차이이다. 네 모델 모두 주어진 과제를 올바르게 해결하는 코드를 산출하지만, 그 결과물에는 모델별 고유한 흔적이 남는다. 이는 각 모델의 학습 데이터와 훈련 방식이 서로 달라 형성된 코드 작성 관습이 산출물에 반영된 결과로 해석되며, 코드 표면에 남은 이러한 흔적이 모델을 구분하는 지문 역할을 할 수 있음을 시사한다.

• **정량 지표로 확인한 LLM별 스타일 차이**

앞서 정성적으로 살펴본 모델별 코드 스타일 차이는 몇 가지 정량 지표를 통해 구체적으로 나타난다. 연구진은 코드 길이, 어휘 밀도, 명명 규칙 이탈 정도, 구조적 깊이라는 네 가지 지표를 활용해 네 모델이 생성한 코드의 특성을 비교하였다.

[그림 1] LLM별 코드 스타일 지표 비교



(a) 코드 길이(LOC) (b) 어휘 밀도

(c) 명명 규칙 이탈 정도 (d) 구조적 깊이

출처: Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212 (논문 정보 기반 재구성)

첫째, 코드 길이는 하나의 함수를 구현하는 데 사용된 코드 줄 수(LOC: lines of code)로 측정된다. 챗GPT는 상대적으로 긴 코드를 작성하는 경향을 보인 반면, 퀜은 가장 간결한 코드를 작성했다. [그림 1] (a). 같은 알고리즘 문제라도 챗GPT는 각 처리 단계를 명시적으로 풀어쓰고, 퀜은 압축된 구현을 선호하기 때문이다.

둘째, 어휘 밀도는 코드에 등장하는 변수와 함수 이름, 즉 식별자(identifier)의 평균 문자 길이로 측정된다. 클로드는 다른 모델들보다 이름을 길게 짓는 경향을 보였다. 이름에 그 역할이나 의미를 풀어 담는 방식을 선호한다는 뜻이다. 반면 챗GPT는 상대적으로 짧은 이름을 자주 사용했다[그림 1] (b).

셋째, 명명 규칙 이탈 정도는 프로그래밍 언어마다 정해진 명명 방식에서 얼마나 벗어나는지를 측정한다. 예를 들어 자바에서는 nextNumber와 같은 낙타 표기법(camel case)*이 관례이지만, 딥시크와 퀜은 다른 두 모델에 비해 자바 코드에서도 next_number와 같은 뱀 표기법(snake case)을 자주 사용하였다[그림 1] (c). 이는 파이썬 등 다른 언어의 명명 관례가 학습 과정에서 영향을 미친 결과로 볼 수 있다.

넷째, 구조적 깊이는 코드의 평균 들여쓰기 수준으로 측정된다. 클로드와 딥시크, 퀜은 어떤 문제를 다루든 들여쓰기 깊이가 일정한 범위 안에 머무른 반면, 챗GPT는 문제에 따라 들여쓰기 깊이의 편차가 크게 나타났다[그림 1] (d). 이러한 정량 지표들은 모델별 스타일 차이가 특정 코드에 우연히 나타나는 것이 아니라 여러 코드에 걸쳐 일관되게 나타나는 경향임을 뜻한다. 이는 코드 스타일로 모델을 구분할 수 있는 근거가 된다.[7)]

* 뱀 표기법/낙타 표기법: 프로그래밍에서 변수나 함수 이름을 표기할 때 여러 단어를 연결하는 대표적인 방식. 낙타 표기법(camel case)은 이어지는 단어의 첫 글자를 대문자로 써서 연결하고(예: nextNumber), 뱀 표기법(snake case)은 밑줄 문자로 연결함(예: next_number). 자바에서는 낙타 표기법이, 파이썬에서는 뱀 표기법이 관례적으로 사용됨.

• 주석 스타일에 나타나는 모델별 특성

코드 스타일의 차이는 코드 구조뿐 아니라 주석에서도 나타난다. 주석은 프로그램의 실행에는 영향을 주지 않지만, 코드의 의도나 동작을 자연어로 설명하는 부분이다. LLM이 코드와 주석을 함께 생성하는 경우, 주석을 얼마나 상세하게 다는지, 어떤 형식으로 배치하는지에서 모델별 차이가 드러난다.

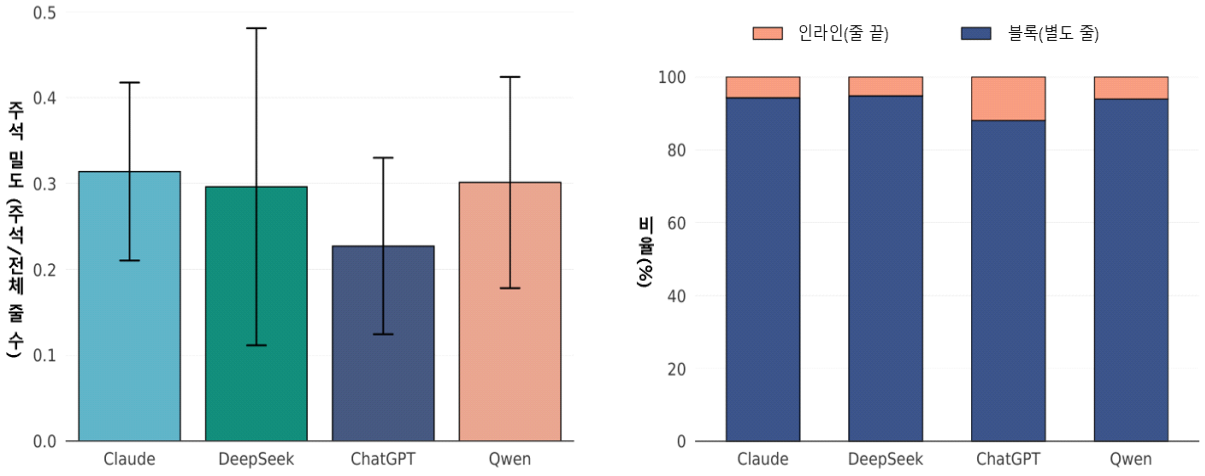
첫째, 주석 밀도는 전체 코드 줄 수 대비 주석의 비율로 측정된다. 챗GPT는 네 모델 중 가장 낮은 주석 밀도를 보였다[그림 2] (a). 이는 코드만으로 로직이 드러나도록 작성하며 부가적인 설명을 최소화하는 챗GPT의 경향을 반영한다. 반면 클로드와 딥시크, 퀜은 상대적으로 많은 주석을 덧붙였다.

둘째, 주석 형식에서도 모델별 차이가 나타난다. 주석은 배치 위치에 따라 두 가지로 나뉜다. 코드 줄 끝에 붙는 인라인 주석(inline comment)과 별도의 줄에 놓이는 블록 주석(block comment)이다. 네 모델 모두 블록 주석을 주로 사용하지만, 챗GPT는 인라인 주석 비율이 상대적으로 높게 나타났다[그림 2] (b).

셋째, 주석의 서술 방식에도 모델별 차이가 있다. 딥시크는 코드의 동작을 여러 단계로 나누어 상세히 서술하는 반면, 챗GPT는 상대적으로 간결한 표현을 사용한다. 예를 들어 같은 로직을 두고 딥시크는 "변환하고 자릿수를 채운다"와 같이 단계를 풀어쓰고, 챗GPT는 "각 숫자를 채운다"처럼 압축해 표현한다.

7) Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212

[그림 2] LLM별 주석 밀도와 형식 비교



(a) 주석 밀도 (b) 주석 형식

출처: Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212 (논문 정보 기반 재구성)

이러한 주석 스타일은 코드 구조와 함께 모델을 구분하는 또 다른 단서가 된다. 주석은 자연어로 작성되기 때문에 LLM이 학습 과정에서 형성한 언어적·표현적 경향이 반영될 수 있으며, 코드 구조만으로는 드러나지 않는 미묘한 차이까지 반영한다.[8)]

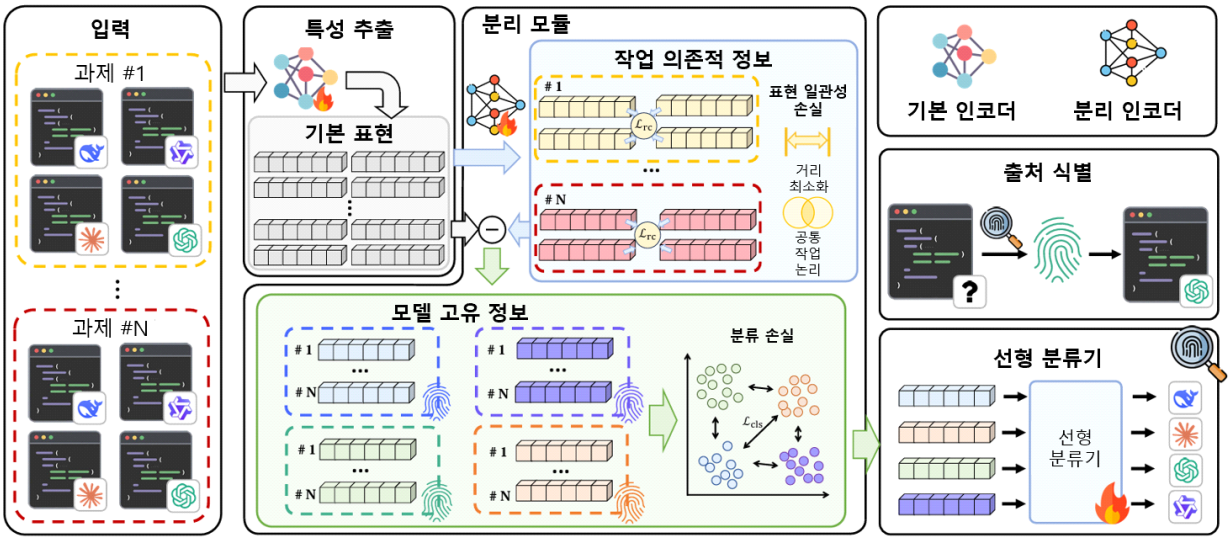
• **내용-스타일 분리 프레임워크의 구조**

앞서 살펴본 스타일 차이를 출처 식별에 활용하려면 코드에서 모델의 흔적만을 골라내는 기술이 필요하다. 그러나 코드에는 성격이 다른 두 종류의 정보가 뒤섞여 있다. 하나는 코드가 어떤 작업을 수행하는지에 관한 작업 의존적 정보(기능적 정보)이고, 다른 하나는 그 작업을 어떤 방식으로 표현했는지에 관한 모델 고유 정보(스타일 정보)이다. 예를 들어 정렬 알고리즘을 구현한 코드에는 정렬 작업 자체의 정보와 특정 LLM이 정렬을 표현한 고유한 방식이 동시에 반영된다. 출처 식별에 필요한 것은 모델 고유 정보이지만, 두 정보가 얽혀 있어 그대로는 분리하기 어렵다.

이 문제를 해결하기 위해 제안된 프레임워크가 DCAN(disentangled code attribution network)이다. DCAN은 코드에 담긴 정보를 두 부분으로 나누어 학습한다. 하나는 생성 모델과 관계없이 같은 작업에 공통으로 나타나는 작업 의존적 정보이고, 다른 하나는 특정 모델에서만 반복적으로 나타나는 모델 고유 정보이다. 작업 의존적 부분을 걸러내고 모델 고유 부분만 남기면, 이것이 모델을 구분하는 지문 역할을 한다.[9)]

8) Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212
9) Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212

[그림 3] DCAN 프레임워크의 구조



출처: Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212 (논문 정보 기반 재구성)

DCAN은 크게 세 단계로 구성된다[그림 3]. 첫 번째 단계에서는 사전 학습된 인코더(encoder)*를 활용해 입력 코드를 벡터 표현으로 변환한다. 이 벡터에는 작업 의존적 정보와 모델 고유 정보가 함께 담겨 있다. 두 번째 단계에서는 분리 모듈이 이 벡터에서 작업 의존적 부분과 모델 고유 부분을 나눈다. 세 번째 단계에서는 분리된 두 부분 중 모델 고유 부분만을 분류기(classifier)**에 입력해, 어느 LLM이 생성한 코드인지 판단한다.

분리 모듈은 감산 분해(subtractive decomposition)라는 방식으로 작동한다. 소리가 뒤섞인 녹음에서 배경 소음을 추정해 제거하면 목소리만 남듯, 전체 벡터에서 작업 의존적 부분을 추정한 뒤 별도의 신경망으로 빼면 남는 것이 모델 고유 부분이다.

이 분리가 제대로 이루어지려면 두 가지 학습 목표(training objective)***가 필요하다. 분류 손실(classification loss)은 분리된 모델 고유 부분만으로 어느 LLM이 만든 코드인지 정확히 맞히도록 유도한다. 표현 일관성 손실(representation consistency loss)은 여러 LLM이 같은 문제를 해결한 코드들에서 추출된 작업 의존적 부분이 서로 비슷해지도록 유도한다. 같은 문제라면 작업 정보는 어느 모델에서든 공통적이기 때문이다. 두 학습 목표는 모델 고유 부분이 모델별 특징을 더 많이 담고, 작업 의존적 부분이 공통 작업 정보를 더 잘 반영하도록 유도한다.

* 인코더(encoder): 입력된 텍스트나 이미지 등의 데이터를 수치 벡터 형태로 변환하는 신경망 구성요소. DCAN에서는 사전 학습된 코드 이해 모델인 UniXcoder를 활용함

** 분류기(classifier): 입력받은 데이터가 미리 정해진 여러 범주 중 어느 것에 해당하는지 확률적으로 판단하는 신경망 구성요소. DCAN에서는 모델 고유 성분을 입력받아 네 개 LLM 중 어느 것이 생성한 코드인지 판단하는 데 활용됨

*** 학습 목표(training objective): AI 모델을 학습시킬 때 성능의 기준이 되는 값. 모델의 출력이 정답에서 벗어난 정도를 수치화한 손실 함수(loss function)로 표현되며, 이 값이 작아지도록 모델을 조정하는 과정이 학습에 해당함

• DCAN의 식별 성능

DCAN이 LLM 코드의 출처를 식별할 수 있는지는 대규모 실험으로 확인되었다. 연구진은 네 개 언어(C, Go, Java, Python)를 대상으로 두 가지 조건에서 실험을 수행하였다. LLM이 주석 없이 코드만 생성한 결과를 대상으로 하는 순수 코드 설정(plain setting)과 주석이 포함된 코드를 대상으로 하는 주석 포함 설정(comment setting)이다. 성능 지표로는 F1 점수*를 사용하였다.

[표 1] 언어별 성능 비교 (F1 점수, %)

방법		C	Go	Java	Python	평균
순수 코드 설정 (plain setting)	CodeGPTSensor	82.48	81.78	75.89	65.37	76.38
	GPTSniffer	92.17	94.18	91.01	79.25	89.15
	DCAN	96.16	96.27	93.20	86.13	92.94
주석 포함 설정 (comment setting)	CodeGPTSensor	87.96	97.78	92.53	90.61	92.22
	GPTSniffer	96.48	97.01	96.18	93.48	95.79
	DCAN	97.99	99.07	98.25	98.25	98.38

출처: Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212 (논문 정보 기반 재구성)

순수 코드 설정에서 DCAN은 네 개 언어 평균 92.94%의 F1 점수를 기록했다. 이는 기존 방법인 GPTSniffer(89.15%)와 CodeGPTSensor(76.38%)를 상회하는 결과이다[표 1]. 주석 포함 설정에서는 성능이 한층 향상되어 평균 98.38%에 이르렀다. 코드 구조에 더해 주석에도 스타일 정보가 담겨 식별에 활용할 수 있는 단서가 늘어난 결과이다.

언어별로 살펴보면 뚜렷한 경향이 나타난다. 순수 코드 설정에서는 C(96.16%)와 Go(96.27%)에서 상대적으로 높은 성능을 보인 반면, 파이썬(86.13%)에서는 다소 낮게 나타났다. 파이썬은 다른 세 언어와 문법 구조가 크게 다르고 표현이 간결해 스타일 차이가 코드 표면에 덜 드러나기 때문으로 추정된다. 주석 포함 설정에서는 언어 간 격차가 좁혀졌으며 파이썬에서도 98.25%까지 향상되었다.

실험은 언어별 성능에 더해 문제 난이도와 언어 간 일반화도 함께 다루었다. 일반적인 분류 과제에서는 난이도가 높아질수록 성능이 저하되는 경향이 있으나, DCAN은 오히려 어려운 문제에서 더 높은 성능을 보였다. 쉬운 문제는 표준적인 해법이 있어 모델 간 코드가 비슷해지는 반면, 어려운 문제일수록 각 모델의 고유한 접근 방식이 드러나 지문이 뚜렷해지는 것으로 해석된다.[10)]

* F1 점수(F1 score): 분류 모델의 성능을 종합적으로 평가하는 지표. 해당 범주에 속하는 것을 놓치지 않고 맞히는 정도(재현율)와, 해당 범주로 예측한 것 중 실제로 맞는 비율(정밀도)을 함께 반영한 값. 백분율로 나타내며 값이 클수록 성능이 좋음.

10) Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212

Ⅱ. 본론 2: 다중모드 확장을 통한 식별 정확도 향상

• 소스코드 접근의 한계와 다중모드의 필요성

DCAN은 소스코드에 담긴 정보만을 활용해 출처를 식별한다. 코드의 구조, 명명 방식, 주석 표현 등 사람이 눈으로 읽을 수 있는 표면 정보를 학습해 지문을 추출하는 접근이다. 이러한 방식은 주석이 포함된 조건에서 98% 이상의 식별 성능을 보이지만 몇 가지 상황에서는 한계가 나타난다.

첫째, 소스코드는 다양한 방식으로 변형된다. 코드 최적화 도구는 변수명을 짧게 축약하고, 난독화 도구는 코드 구조를 의도적으로 뒤섞어 원래의 지문을 훼손한다. 이 경우 코드 표면에 드러나던 스타일 특성이 사라져 식별 정확도가 저하되는 경향이 있다.

둘째, 소스코드에는 담기지 않는 저수준(low-level)* 정보가 존재한다. 프로그래밍 언어로 작성된 소스코드는 컴퓨터가 실행할 수 있는 형태로 변환되기 위해 컴파일(compile)** 과정을 거치는데, 이 과정에서 산출되는 결과물에는 소스코드 표면에는 드러나지 않는 배치와 구조 정보가 담긴다. LLM이 소스코드를 어떤 방식으로 구성했는지에 따라 컴파일 결과물의 형태에도 차이가 생기며 이 차이가 추가적인 지문이 된다.

이러한 한계를 개선하기 위해, 최근 연구에서는 소스코드에 컴파일 결과물을 결합하는 접근이 등장했다. 서로 다른 형태의 데이터를 함께 다루는 이러한 방식을 다중모드(multimodal)*** 접근이라고 한다. 소스코드 표면에서 드러나지 않는 저수준 정보까지 지문 추출에 반영할 수 있어, 단일 정보에 의존할 때보다 풍부한 단서를 얻는다. 이러한 다중모드 접근을 구현한 프레임워크가 GoCoMA이다.[11)]

* 고수준(high-level)/저수준(low-level): 컴퓨터가 정보를 표현하는 방식이 사람의 사고에 가까운지, 하드웨어의 실행 형태에 가까운지를 나누는 기준. 소스코드처럼 사람이 읽고 이해하기 쉬운 형태를 고수준이라 하고, 컴파일 결과물처럼 컴퓨터가 직접 실행하는 형태에 가까운 표현을 저수준이라 함

** 컴파일(compile): 사람이 작성한 소스코드를 컴퓨터가 직접 실행할 수 있는 형태로 변환하는 과정. 각 프로그래밍 언어마다 전용 도구인 컴파일러(compiler)를 통해 수행되며 그 결과물의 형태는 언어에 따라 다름

*** 다중모드(multimodal): 서로 다른 형태의 정보를 함께 활용하는 접근 방식. 각 정보는 하나의 대상을 서로 다른 관점에서 표현하며, 이를 결합하면 단일 정보만 사용할 때보다 더 풍부한 특징을 파악할 수 있음

• 소스코드에 컴파일 결과물을 더한 다중모드 식별

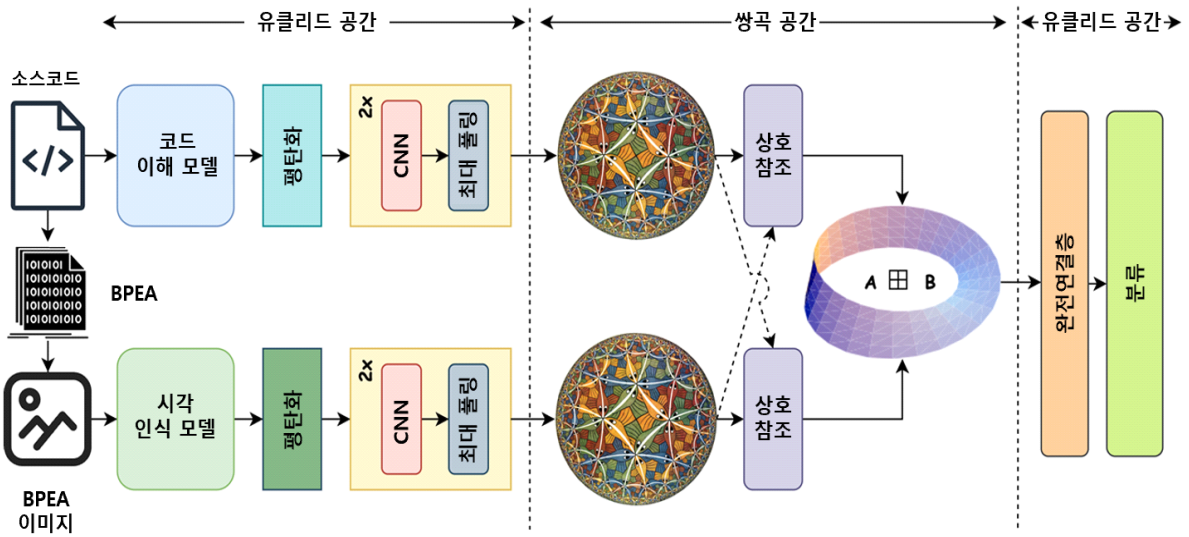
GoCoMA는 소스코드와 함께 컴파일 결과물을 활용해 출처를 식별한다. 컴파일 결과물은 언어에 따라 형태가 다르다. C·C++는 오브젝트 파일(.o), 자바는 클래스 파일(.class), 파이썬은 바이트코드 파일(.pyc)로 변환된다. GoCoMA는 이러한 결과물을 통칭해 바이너리 사전 실행 아티팩트(BPEA: binary pre-executable artifact)라 부른다.

BPEA는 0과 1로 이루어진 데이터이므로 그대로는 다루기 어렵다. 따라서 GoCoMA는 이를 한 장의 이미지로 변환한다. 데이터를 색깔 점으로 바꿔 바둑판처럼 늘어놓는 방식이다. 컴파일 결과물을 이미지로

11) Nitin Choudhury 외 3인, "GoCoMA: Hyperbolic Multimodal Representation Fusion for Large Language Model-Generated Code Attribution", arXiv, 2026.04.24., https://arxiv.org/abs/2604.16377

바꾸면 사진을 분류하듯 이미지 인식 기술로 그 안의 패턴을 읽어낼 수 있다. 이렇게 만든 이미지에는 컴파일 결과의 저수준 배치가 담기고, 사전 학습된 시각 인식 모델이 여기서 바이트 수준의 규칙성을 특징으로 추출한다. 컴파일이 되지 않는 경우에는 코드에서 주석·문자열·서식을 제거한 정규화된 이진 표현을 대신 사용한다.

[그림 4] GoCoMA 프레임워크의 구조



출처: Nitin Choudhury 외 3인, "GoCoMA: Hyperbolic Multimodal Representation Fusion for Large Language Model-Generated Code Attribution", arXiv, 2026.04.24., https://arxiv.org/abs/2604.16377 (논문 정보 기반 재구성)

GoCoMA는 두 종류의 정보를 각기 다른 모델로 처리한 뒤 결합한다. 소스코드는 사전 학습된 코드 이해 모델을 거쳐 임베딩(embedding)* 벡터로 변환되고, BPEA 이미지는 사전 학습된 시각 인식 모델을 거쳐 별도의 벡터로 변환된다. 코드 이해 모델은 변수명, 구문 구조, 주석 등 고수준의 언어적·구조적 특성을 담아내는 반면, 시각 인식 모델은 바이트 배치나 컴파일 최적화의 흔적 같은 저수준의 규칙성을 다룬다. 두 결과물은 동일한 프로그램을 서로 다른 방식으로 나타낸 것이므로, 이를 결합하면 어느 한쪽만 활용할 때보다 풍부한 지문을 얻는다.

다만 두 임베딩을 단순히 이어붙이는 방식으로는 한계가 있다. 소스코드의 고수준 정보와 BPEA의 저수준 정보는 대등한 관계가 아니라, 소스코드가 컴파일을 거쳐 BPEA로 변환되는 흐름에서 형성된 계층적 관계에 놓여 있다. GoCoMA는 이러한 계층 구조를 반영하기 위해 두 임베딩을 쌍곡 공간(hyperbolic space)**으로 옮긴 뒤, 그 안에서 서로를 참조하는 방식으로 결합한다. 결합이 끝난 표현은 다시 유클리드 공간(Euclidean space)***으로 돌아와 분류기에 입력되며, 최종적으로 어느 LLM이 생성한 코드인지 판정된다[그림 4].

공개된 두 표준 평가 데이터셋(CoDET-M4, LLM-AuthorBench)에서 GoCoMA는 소스코드만 활용한 단일모드 방식이나 두 정보를 유클리드 공간에서 단순 결합한 방식에 비해 일관된 성능 향상을 보였다. 특히 두 모델을 GoCoMA 방식으로 결합했을 때 가장 높은 식별 성능이 나타났다.[12)]

* 임베딩(embedding): 텍스트나 이미지 같은 다양한 형태의 데이터를 신경망이 처리할 수 있도록 수치 벡터 형태로 변환한 표현. 원래 데이터의 의미나 특성이 벡터 공간의 위치에 반영되며, 유사한 특성을 가진 데이터일수록 벡터 공간에서 가까이 위치함

** 쌍곡 공간(hyperbolic space): 일상적으로 사용되는 평면 공간과 달리, 중심에서 멀어질수록 공간이 급격히 넓어지는 성질을 지닌 특수한 표현 공간. 상하 관계가 있는 계층적 데이터를 표현하기에 적합함. GoCoMA는 쌍곡 공간의 한 모델인 푸앵카레 볼(Poincaré ball)을 사용함

*** 유클리드 공간(Euclidean space): 평면이나 3차원 공간처럼 일상적으로 접하는 좌표 공간. 쌍곡 공간과 달리 어느 지점에서도 공간의 성질이 같음

• 쌍곡 공간에서의 계층적 융합

GoCoMA는 두 임베딩을 쌍곡 공간에서 결합한다. 소스코드와 BPEA는 대등한 관계가 아니라 컴파일을 거쳐 이어지는 계층적 관계에 놓여 있는데, 이 관계를 살린 채 결합하기에는 유클리드 공간이 적합하지 않기 때문이다. 쌍곡 공간에서는 상위 개념일수록 중심에 가깝게, 하위 개념일수록 바깥쪽에 놓여 계층 구조가 자연스럽게 담긴다. 소스코드의 고수준 정보는 상위에, BPEA의 저수준 정보는 하위에 대응하므로, 이러한 배치가 두 정보의 관계를 유지하며 결합하는 데 적합하다.

유클리드 공간에서 두 임베딩을 단순히 이어붙이면 이러한 계층 관계가 평평하게 눌린다. 상위 정보와 하위 정보가 같은 평면에 나란히 놓여 둘 사이의 상하 관계가 흐려지기 때문이다. 반면 쌍곡 공간은 바깥으로 갈수록 급격히 넓어지므로, 하위의 세부 정보를 넓게 펼치면서도 상위 정보는 중심 가까이 유지해 계층을 살린 채 결합할 수 있다.

결합 과정에서는 코드 임베딩과 이미지 임베딩이 쌍곡 공간에서 서로 가까운 정도를 유사도로 환산하고, 유사도가 높은 쌍에 더 큰 비중을 두어 결합한다. 이러한 방식은 소스코드만으로는 놓치는 저수준 정보까지 지문에 반영해, 단순 결합 방식보다 높은 식별 성능을 보인다.[13)]

II. 본론 3: 기술의 한계와 적용상 과제

• 코드 변형에 대한 강건성

DCAN과 GoCoMA 두 기술은 모두 코드에 남은 흔적으로 출처를 식별한다. DCAN은 소스코드 표면의 스타일 특성을 활용하며, GoCoMA는 여기에 컴파일 결과물의 저수준 배치까지 결합한다. 이러한 흔적 기반 접근은 원본에 가까운 코드에서는 안정적으로 작동하지만, 코드가 변형되면 지문이 훼손되어 식별 정확도가 떨어지는 경향이 있다.

12) Nitin Choudhury 외 3인, "GoCoMA: Hyperbolic Multimodal Representation Fusion for Large Language Model-Generated Code Attribution", arXiv, 2026.04.24., https://arxiv.org/abs/2604.16377
13) Nitin Choudhury 외 3인, "GoCoMA: Hyperbolic Multimodal Representation Fusion for Large Language Model-Generated Code Attribution", arXiv, 2026.04.24., https://arxiv.org/abs/2604.16377

코드 최적화 도구는 속도나 용량을 개선하기 위해 코드를 자동으로 다듬는다. 이때 LLM이 남긴 명명 방식이나 구조적 특성이 함께 지워져 소스코드 표면의 지문이 흐릿해진다. 난독화 도구는 이보다 강하게 작용해 코드 흐름을 의도적으로 뒤섞고 무의미한 코드를 삽입하는 방식으로 원래의 스타일을 지운다.

GoCoMA는 소스코드가 변형되어도 BPEA 이미지의 저수준 지문으로 이러한 훼손을 일부 보완한다. 다만 BPEA 이미지 역시 컴파일 조건에 민감하기 때문에, 같은 소스코드라도 원본과 다른 설정으로 컴파일하면 형태가 달라지고 저수준 지문도 훼손된다. 결국 소스코드와 컴파일 결과가 모두 변형되면 두 기술 모두 안정적으로 작동하기 어렵다.[14]

• 툴체인 의존성과 다국어 일반화의 어려움

두 기술은 학습에 사용된 컴파일 환경이나 프로그래밍 언어를 벗어나면 성능이 저하된다. 툴체인(toolchain)* 의존성은 GoCoMA에서 두드러지게 나타난다. GoCoMA는 툴체인의 결과물인 BPEA 이미지를 활용하므로, 같은 소스코드라도 어떤 컴파일러와 옵션을 거쳤는지에 따라 BPEA 이미지가 달라진다. 그 결과 학습에 쓰인 것과 다른 툴체인으로 컴파일된 코드에서는 학습된 지문 패턴이 그대로 적용되지 않는다.[15]

다국어 일반화는 학습에 사용되지 않은 언어에서도 모델이 잘 작동하는지에 관한 문제로, 두 기술의 공통 과제이다. DCAN의 실험에서도 이러한 경향이 나타난다. 네 개 언어 중 세 개 언어로만 모델을 학습시킨 뒤 학습에서 제외한 나머지 언어로 평가했을 때, 문법 구조가 이질적인 파이썬에서는 순수 코드 설정 성능이 69.70%로 크게 하락했다. 다만 주석 포함 설정에서는 93.48%로 회복되었는데, 자연어 주석에 담긴 스타일 특성이 언어 차이를 넘어 식별 단서로 작용하기 때문이다.[16] GoCoMA 역시 새로운 언어에 대한 일반화가 아직 검증되지 않았으며, 컴파일 방식이 다른 언어로의 확장에는 추가적인 제약이 있다.

* 툴체인(toolchain): 소스코드를 실행 가능한 형태로 만드는 데 필요한 도구들의 집합. 컴파일러(소스코드를 실행 형태로 변환하는 도구), 링커(여러 코드 조각을 하나로 연결하는 도구), 패키징 도구 등이 포함됨.

• 사후 탐지의 본질적 한계와 향후 보완 방향

앞서 살펴본 두 기술은 모두 사후 탐지 방식에 속한다. 이미 생성된 코드에 남은 흔적을 분석해 출처를 되짚는 접근으로, 별도의 사전 조치 없이 유통 중인 코드에도 적용된다는 이점이 있다. 다만 사후 탐지 방식 자체에서 비롯되는 본질적인 한계도 지닌다.

첫째, 사후 탐지는 이미 생성된 코드를 분석하는 방식이기 때문에 코드가 생성되거나 유통되는 시점에 실시간으로 개입하기 어렵다. 둘째, 학습에 사용되지 않은 새로운 LLM에 대해서는 식별이 어렵다.

14) Nitin Choudhury 외 3인, "GoCoMA: Hyperbolic Multimodal Representation Fusion for Large Language Model-Generated Code Attribution", arXiv, 2026.04.24., https://arxiv.org/abs/2604.16377
15) Nitin Choudhury 외 3인, "GoCoMA: Hyperbolic Multimodal Representation Fusion for Large Language Model-Generated Code Attribution", arXiv, 2026.04.24., https://arxiv.org/abs/2604.16377
16) Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212

두 기술 모두 특정 LLM의 지문 패턴을 학습하기 때문에 새로운 LLM이 등장하면 재학습이 필요하다. 셋째, 코드와 그 컴파일 결과만 분석하므로, 코드가 실제로 실행될 때 드러나는 정보는 활용하지 못한다.

이러한 한계를 개선하기 위해 다양한 연구가 이어지고 있다. 코드 생성 시점에 식별 정보를 삽입하는 워터마킹 방식과 사후 탐지를 결합해 상호 보완하는 접근, 새로운 LLM의 지문 패턴을 학습에 추가하는 방식, 코드가 실행될 때 드러나는 정보를 함께 활용하는 방식 등이 검토되고 있다.

Ⅲ. 결론 및 전망

• 현재 기술 수준의 종합 평가

본 보고서는 LLM이 생성한 코드의 출처를 식별하는 두 기술을 중심으로 살펴보았다. 소스코드에서 내용과 스타일을 분리해 지문을 추출하는 DCAN은 네 개 언어에서 높은 식별 성능을 보였으며, 주석이 포함된 조건에서 성능이 더 높게 나타났다. 소스코드와 BPEA 이미지를 결합해 계층적 관계를 반영한 GoCoMA는 소스코드 단독 접근이나 유클리드 공간 결합 방식보다 나은 성능을 보였다.

두 기술은 접근이 다르지만 같은 출발점을 공유한다. 서로 다른 LLM이 코드를 생성하는 방식에는 미세하지만 일관된 차이가 있고, 그 차이가 코드 표면과 저수준 배치에 지문으로 남는다는 점이다. DCAN과 GoCoMA는 이 지문을 각기 다른 방식으로 분석해 출처를 식별한다.

다만 두 기술 모두 코드 변형에 대한 취약성, 툴체인 의존성, 다국어 일반화의 어려움 등의 한계를 안고 있으며, 사후 탐지 방식 자체에서 비롯되는 본질적 제약도 지닌다. 현재 기술은 통제된 조건에서는 안정적으로 작동하지만 실제 유통 환경의 다양한 변수에까지 완전히 대응하는 수준에는 이르지 못했다.

• 향후 기술 발전 방향

LLM 코드 출처 식별 기술은 이러한 한계를 개선하는 방향으로 발전하고 있다. 우선 코드 변형과 툴체인 변화에도 훼손되지 않는 강건한 지문을 확보하려는 시도가 이어진다. 최적화·난독화되거나 다른 환경에서 컴파일된 코드에도 안정적으로 작동하도록, 표면적인 스타일 이상의 지문을 추출하는 방식이 모색된다.

새로운 LLM에 대한 확장성도 지속적인 과제이다. LLM이 빠르게 개발되면서 학습 시점에 반영되지 않은 새 모델이 계속 등장한다. 이에 새 모델의 지문을 기존 학습에 편입하는 방식이 검토되고 있다. 언어 간 일반화도 이러한 확장성의 한 측면이며 문법 구조가 다른 언어까지 아우르는 학습 방식이 논의되고 있다.

사후 탐지 자체의 한계를 보완하기 위한 접근법 다양화도 나타난다. 코드 생성 단계에서 식별 정보를 삽입하는 워터마킹 방식과 사후 탐지를 결합하려는 접근이 대표적이다. 또한 코드 자체만 분석하지 않고 코드가 실행될 때 드러나는 정보까지 함께 활용하는 방식도 시도되고 있다. 단일 기술로 모든 상황에 대응하기보다 여러 접근법을 조합해 상호 보완하는 흐름이 뚜렷해지고 있다.

• LLM 코드 생성 시대의 출처 식별 전망

LLM을 활용한 코드 생성은 소프트웨어 개발 현장에 이미 일상적인 방식으로 자리 잡았다. 바이브 코딩의 확산에서 나타난 것처럼 앞으로도 LLM 생성 코드의 유통은 더욱 늘어날 전망이다. 유통 규모가 확대되는 만큼 개별 코드가 어떤 모델에서 산출되었는지를 사후에 확인하는 기술의 필요성도 커지고 있다.

본 보고서에서 살펴본 두 사례는 출처 식별 기술의 현재를 보여준다. DCAN은 소스코드 표면의 스타일만으로도 높은 식별 성능을 보였고, GoCoMA는 컴파일 결과의 저수준 정보를 더해 식별 근거를 넓혔다. 다만 이러한 성능은 통제된 평가 환경에서 확인된 결과이며, 실제 유통 환경에서의 검증은 아직 과제로 남아 있다.

LLM 코드 생성이 다양해지고 유통 방식이 복잡해질수록 출처 식별에 요구되는 조건도 까다로워진다. 그에 따라 강건성 확보, 새로운 모델에 대한 확장성, 접근법 다양화 등 여러 방향의 연구가 병행되고 있다. 다만 실제 활용의 폭은 강건성과 일반화 성능이 어느 수준까지 검증되느냐에 좌우된다. 이러한 검증이 뒷받침될 때, 출처 식별 기술은 LLM 생성 코드의 출처 확인을 보조하는 수단으로 활용 범위가 확대될 가능성이 있다.

참고문헌

- Jiaxun Guo 외 5인, "Code Fingerprints: Disentangled Attribution of LLM-Generated Code", arXiv, 2026.03.04., https://arxiv.org/abs/2603.04212
- Nitin Choudhury 외 3인, "GoCoMA: Hyperbolic Multimodal Representation Fusion for Large Language Model-Generated Code Attribution", arXiv, 2026.04.24., https://arxiv.org/abs/2604.16377
- 김승기, "바이브 코딩 확산, '코드 없는 개발' 현실화…당근 과외까지 등장", 테크월드, 2026.04.23., https://www.epnc.co.kr/news/articleView.html?idxno=400792