AI 시대의 코드 분석 기술*

오학주
고려대학교 컴퓨터학과

2026년 5월

1 들어가며

2026년 4월, 앤트로픽(Anthropic)은 'Claude Mythos Preview'를 공개했다 [1]. 발표에 따르면 새 모델인 미토스(Mythos)는 주요 오픈소스 소프트웨어에서 수천개의 신규 취약점을 발견했다. 그 중에는 보안성이 매우 견고한 것으로 알려진 OpenBSD에 27년간 잠복해 있던 결함과 YouTube 등 상용 서비스에서 광범위하게 사용되는 FFmpeg에 16년간 남아 있던 결함이 포함되어 있었다. 모두 전세계 보안 연구자들이 오랜 세월 검토해 온 소프트웨어들이다.

미토스는 단순히 취약점을 찾는 수준을 넘어, 그 취약점을 이용해 실제 공격을 수행하는 입력값(익스플로잇)까지 생성할 수 있음을 보여주었다. 리눅스 커널의 권한을 탈취하거나 브라우저의 보안 구역을 탈출하고, 멀리 떨어진 서버에서 코드를 실행하게 만드는 공격용 입력을 AI가 스스로 작성한 것이다. 앤트로픽은 이 모델을 일반에 즉시 공개하지 않고 제한된 파트너 그룹에만 우선 공개했는데, 모델 개발사가 공개를 스스로 미루는 결정 자체가 AI의 코드 분석 능력이 어느 수준까지 도달했는지를 단적으로 보여준다.

2 코드 분석 기술의 발전

코드 분석은 새삼스러운 기술이 아니다. 그 뿌리는 컴퓨터 과학의 시작과 맞닿아 있을 정도로 오래된 분야이다. 1936년 튜링은 임의의 프로그램이 종료할지 자동으로 판별하는 기계가 존재할 수 없음을 증명했고, 1953년 라이스는 이를 일반화하여 자명한 경우를 제외한 프로그램의 모든 의미적 성질은 자동으로 판별할 수 없음을 보였다. '이 프로그램에 결함이 있는가', '이 프로그램이 주어진 명세를 만족하는가', '이 프로그램이 항상 종료하는가' 등 코드 분석이 묻는 거의 모든 질문들이 오늘날에도 여전히 난제로 남아 있는 이유는 바로 이 이론적 한계 때문이다.

구체적으로 코드 분석은 안전성, 완전성, 자동화라는 세 가지 성질을 동시에 갖출 수 없다는 본질적 한계를 가진다. 결함이 없음을 완벽히 보장하려 하면(안전성) 허위 경보(False Positive)가 발생하기 쉽고, 탐지된 결함이 진짜임을 보장하려 하면(완전성) 실제 결함을 놓칠 위험이 커진다. 만약 안전성과 완전성을 모두 확보하고자 한다면, 자동화를 포기하고 사람의 개입이 필수적이다. 결국 지난 수십 년간 등장한 코드 분석 기술은 이 세 가지 성질 사이에서, 각자의 목적에 부합하는 실용적인 균형점을 찾아가는 방향으로 발전해 왔다.

2.1 동적 분석

동적 분석(Dynamic Analysis) 또는 테스팅은 안전성을 포기하는 대신, 완전성과 자동화를 확보하는 방향으로 발전해 왔다. 프로그램을 실제 환경에서 실행하며 결함을 찾기 때문에 탐지된

*삼성전자(DS부문) 기고 목적으로 작성됨

오류는 모두 실제 오류이며, 분석 결과에 허위 경보가 포함되지 않는다.

대표적인 기술로는 무작위 입력을 통해 결함을 유발하는 퍼징(Fuzzing), 그리고 프로그램의 실행 경로에 대한 조건을 누적하여 체계적으로 탐색하는 기호 실행(Symbolic Execution)과 콘콜릭 테스팅(Concolic Testing)이 있다. 이러한 동적 분석 기술은 실행되지 않은 경로의 오류를 놓칠 수는 있으나, 탐지된 결함에 대한 확실성 덕분에 산업 현장에서 성공적으로 활용되어 왔다.

한 예로 Microsoft는 콘콜릭 테스팅을 활용하여 윈도우즈 개발 과정에서 보안 결함의 약 1/3을 자동으로 발견하였다고 보고했으며 [5], Google은 퍼징을 통해 오픈소스 소프트웨어에서 수만 건의 결함을 찾아내었다 [6].

2.2 정적 분석

정적 분석(Static Analysis)은 완전성을 타협하는 대신, 안전성과 자동화를 확보하는 방향으로 발전해 왔다. 정적 분석의 핵심 아이디어는 요약 해석(Abstract Interpretation)이다. 프로그램을 실제 값(Concrete Value)으로 실행하는 대신, 값의 성질을 표현하는 요약 값(Abstract Value)을 가지고 프로그램을 실행시킨다.

예를 들어 $30 \times 12 + 11 \times 9$라는 연산에 대해 동적 분석이 459라는 실제 값을 계산한다면, 정적 분석은 이를 '정수', '홀수', 혹은 '400과 500 사이의 수'와 같이 결과값이 가지는 성질을 추론한다. 정적 분석의 강점은 이러한 요약 실행을 통해 프로그램의 무한한 실행 경로를 유한한 시간 내에 전수 조사할 수 있다는 점이다. 다만 요약으로 인해 발생하는 정보의 손실로 인해, 실제로는 오류가 아님에도 오류로 보고하는 허위 경보가 필연적으로 발생하게 된다.

동적 분석에 이어 정적 분석 또한 산업계에서 일상적으로 쓰이는 기술로 자리 잡았다. 대표적으로 Meta는 정적 분석기 Infer를 개발하여 1억 라인 규모의 방대한 코드베이스 내 결함을 개발 단계에서 실시간으로 탐지하고 있다 [4]. Google 또한 정적 분석 기술을 개발 파이프라인에 통합하여 소프트웨어 취약점을 사전에 탐지하는데 활용하고 있다 [9].

2.3 형식 검증

형식 검증(Formal Verification)은 분석의 안전성과 완전성을 모두 확보하는 대신, 자동화를 일부 포기하는 기술이다. 자동화를 제외하면 가장 강력한 형태의 코드 분석 기술이라 할 수 있다. 이 기술을 통해 검증된 프로그램은 모든 가능한 입력에 대해 문제 없이 실행된다는 것이 수학적으로 증명되었으므로, 별도의 테스팅이나 정적 분석을 거칠 필요가 없다. 따라서 소프트웨어의 안전성이 절대적으로 중요한 경우 가장 좋은 대안이 된다.

다만 이러한 검증 과정은 완전히 자동화하기 어렵다는 한계가 있다. 검증에 성공하기 위해 사람이 직접 코드가 가지는 성질(Invariant)이나 명세(Specification) 등의 근거를 수동으로 제공해야 하기 때문이다. 그러나 최근 실용적인 프로그램 검증 도구들이 등장하면서 산업계 내 활용 사례가 빠르게 증가하는 추세다. 한 예로, Amazon은 자체 시스템의 핵심 모듈에서 메모리 안전성을 검증하여, 기존의 동적·정적 분석으로는 탐지할 수 없었던 수십 개의 오류를 찾아내며, 형식 검증이 실제 대규모 시스템 보안에 활용될 수 있음을 보였다 [2].

2.4 AI 기반 분석

앞서 언급한 세 가지 방식이 전통적인 코드 분석 기술이라면, 최근 부상하고 있는 AI 기반 코드 분석은 LLM의 등장과 함께 비약적으로 발전하며 새로운 패러다임을 제시하고 있다.

가장 큰 변화는 수많은 전문가가 오랜 시간 축적해 온 직관과 노하우를 LLM이 내재화하여 분석에 활용하기 시작했다는 점이다. 기존 도구로는 '함수 이름에 담긴 코드의 직관적 의미'나 '경험적으로 파악된 코드 패턴과 오류의 상관관계'처럼 형식화하기 어려운 지식을 구현하는 데 한계가 있었다. 인간 전문가의 머릿속에 파편화되어 있던 이러한 휴리스틱과 통찰을 LLM이 하나로 통합하여 코드 분석의 영역으로 끌어올린 것이다.

그 결과, LLM은 전통적인 기술이 접근하기 어려웠던 영역에서 강점을 보인다. 단순히 코드 자체만 분석하는 것이 아니라 주석, 변수 이름, 커밋 메시지, 문서 등을 종합적으로 해석하여 '개발자의 의도'를 추론할 수 있기 때문이다. 덕분에 미리 정의된 규칙에 의존하던 기존 도구들이 놓치기 쉬웠던 오류들을 사람의 개입 없이 스스로 찾아낼 수 있다.

이러한 흐름은 실제 사례로도 증명되고 있다. 2024년 Google Project Zero의 'Big Sleep' 에이전트는 SQLite에서 메모리 안전성 취약점을 자율적으로 발견해냈으며 [7], 2025년 DARPA AI Cyber Challenge에서는 AI 시스템들이 대규모 오픈소스 코드의 취약점을 자동으로 탐지하고 패치하는 능력을 입증하였다 [3]. 앞서 소개한 미토스는 이러한 AI 기반 코드 분석 흐름의 최전선에 있는 기술 중 하나라고 할 수 있다.

3 향후 전망

3.1 AI 기반 코드 분석의 한계

AI 기반 분석 기술은 인상적인 성과를 보여주고 있으나, 모든 코드 분석 기법이 그러하듯 본질적인 한계로부터 자유로운 것은 아니다. 가장 근본적인 문제는 분석 결과의 신뢰성을 담보할 수 없다는 점이다. 안전성과 완전성 중 어느 한쪽도 이론적으로 보장할 수 없으며, LLM 특유의 환각(Hallucination) 현상으로 인해 논리적 오류를 범할 가능성이 상존한다. 또한, 동일한 입력에 대해서 분석 결과가 매번 상이하게 나타나거나 모델 업데이트에 따라 기존 결과가 재현되지 않을 위험도 존재한다. 이러한 불확실성은 AI 기반 분석 기술의 상용화에 있어 핵심적인 제약 요인으로 작용하게 된다.

경제적·보안적 측면의 제약 또한 무시할 수 없다. 대규모 코드베이스를 LLM으로 분석하면 막대한 토큰 비용과 추론 시간이 소요되며, 특히 대형 프로젝트의 정기 분석이나 지속적 통합(CI) 파이프라인에 도입 시 비용이 폭증할 우려가 있다. 아울러 외부 AI 서비스를 활용할 때 발생하는 소스코드 유출 리스크는 기업의 핵심 자산 보안과 직결된다. 코드 내의 알고리즘, 아키텍처 등 영업비밀은 물론 인증 정보와 취약점 단서까지 외부 인프라에 전송될 수 있고, 이는 곧 잠재적인 공격으로 이어질 수 있기 때문이다.

3.2 전통적 기법과의 결합

결국 코드 분석 기술은 AI 기반 기술과 전통적 기술이 상호 보완적으로 융합되는 방향으로 발전할 것으로 보인다. 전통적 기법은 신뢰성, 재현성, 비용 효율성 측면에서 탁월하지만 미리 정의된 규칙을 벗어난 영역을 탐지하는 데 한계가 있다. 반면 AI는 규칙 바깥의 비정형적 결함을 식별하는 데 강력한 성능을 발휘하나, 신뢰성과 비용 측면에서 취약하다. 따라서 향후 코드 분석은 각 기법의 장점을 극대화하고 약점을 보완하는 방향으로 나아갈 전망이며, 이미 초기 시도들이 등장하고 있다 [10, 8, 11].

먼저 전통적 분석 기술은 AI 분석의 고질적인 불확실성을 제거하고 결과물의 신뢰성을 담보하는 데 기여할 수 있다. 예를 들어, LLM이 결함 보고, 패치, 명세 등의 후보군을 대량으로 생성하면, 엄밀한 논리를 기반으로 하는 전통적 기법이 그중 타당한 결과만을 선별하고 분석 결과의 신뢰도를 검증하는 것이다. 또한, 정적 분석을 통해 취약점 의심 영역을 먼저 식별하고 필요한 곳에만 선별적으로 LLM을 호출함으로써 분석에 소요되는 토큰 비용을 절감할 수 있다.

역으로 AI는 전통적 분석이 수십 년간 해결하지 못한 난제들에 실마리를 제공할 수 있다. 예를 들어, 정적 분석의 허위 경보(False Alarm) 문제는 LLM이 코드의 의도와 맥락을 해석하여 위험도를 재평가함으로써 개선할 수 있으며, 동적 분석의 경로 폭발(Path Explosion) 문제 역시 LLM이 의미적으로 유망한 경로를 우선 탐색하도록 유도함으로써 완화할 수 있다. 나아가 전문가가 직접 작성해야 했던 명세 및 코드에 대한 성질을 LLM이 자동 생성하게 함으로써, 그간 자동화의 장벽에 부딪혔던 형식 검증 기술의 실용화를 앞당길 수 있다.

References

[1] Anthropic. Assessing claude mythos preview's cybersecurity capabilities. https://red.anthropic.com/2026/mythos-preview/, 2026.

[2] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Code-level model checking in the software development workflow. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020.

[3] DARPA. AI cyber challenge (AIxCC). https://aicyberchallenge.com/, 2025.

[4] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. *Communications of the ACM*, 62(8):62–70, 2019.

[5] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[6] Google. OSS-Fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz. Accessed: 2026.

[7] Google. From naptime to big sleep: Using large language models to catch vulnerabilities in real-world code. https://googleprojectzero.blogspot.com/2024/10/from-naptime-to-big-sleep.html, 2024.

[8] Yihe Li, Ruijie Meng, and Gregory J. Duck. Large language model powered symbolic execution. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):Article 385, October 2025.

[9] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Communications of the ACM*, 61(4):58–66, 2018.

[10] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Boosting static resource leak detection via LLM-based resource-oriented intention inference. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2905–2917. IEEE/ACM, 2025.

[11] Yupeng Yang, Shenglong Yao, Jizhou Chen, and Wenke Lee. Hybrid language processor fuzzing via LLM-based constraint solving. In *34th USENIX Security Symposium (USENIX Security 25)*, Seattle, WA, 2025. USENIX Association.