# BugSynth: 자연스럽고 믿을 수 있는 오류 벤치마크 자동 생성

# BugSynth: Automatic Synthesis of Realistic and Reliable Bug Benchmark


**요 약**

본 논문에서는 언어 모델과 프로그램 검증 기법을 결합하여 오류 프로그램을 자동 생성하는 도구 BugSynth를 제시한다. 오류 벤치마크는 오류 탐지기를 평가하고 언어 모델의 학습 데이터 부족 문제를 해결하는 데 있어 중요성이 커지고 있지만 기존에 제시된 벤치마크 구축 방법에는 한계점이 존재한다. 본 논문에서 제시하는 BugSynth는 0으로 나누는 오류의 위치가 명확한 프로그램을 원본 프로그램 없이 자동으로 생성한다. 실험 결과 BugSynth는 51%의 생성 성공 비율을 보였으며 벤치마크를 구축하여 정적 분석기를 평가함에 있어 사람이 작성한 벤치마크와 유사한 난이도를 가짐을 확인하였다.


## 1. 서 론

오류 벤치마크는 오류 탐지기를 평가하는 것뿐 아니라 최근 대두되는 언어 모델의 학습 데이터 고갈 문제[1]를 해결하는 측면에서 그 중요성이 커지고 있다. 오류 벤치마크를 구축하는 방법으로는 실제로 발견된 오류 프로그램을 수집[2]하거나 기존 프로그램에 오류를 삽입하여 오류 프로그램을 생성하는 방식[3] 등이 있다.

하지만, 기존에 제시된 오류 벤치마크의 구축 방법은 한계점이 존재한다. 발견된 오류 프로그램을 수집하는 방식의 경우 수동으로 오류의 존재를 검증해야 하기 때문에 데이터의 수집 및 추가가 어렵다는 문제가 있다. 기존 프로그램에 오류를 삽입하는 방식의 경우 프로그램의 의미 (Semantics)가 달라져 다른 지점에 의도하지 않은 오류가 발생할 수 있다. 이 경우 실제 정답 (Ground-truth)이 명확하지 않아 오류 탐지기를 올바르게 평가하기 어려워진다. 또한 위 방식들 모두 벤치마크 구축에 원본 프로그램이 필요하다는 단점이 있다.

본 논문에서는 언어 모델과 프로그램 검증 기법을 결합하여 오류 프로그램을 자동 생성하는 도구 BugSynth를 제안한다. BugSynth는 언어 모델이 코드를 생성하는 동안 실시간으로 프로그램 검증 기법을 통해 0으로 나누는 (Division-by-zero) 오류가 유일하게 존재하는 프로그램을 생성한다.

우리는 실험을 통해 BugSynth의 오류 프로그램 생성 능력을 평가하였고 이를 취합한 벤치마크로 정적 분석기를 평가하여 사람이 작성한 오류 벤치마크와의 유사성을 확인하였다. 실험 결과 BugSynth는 0으로 나누는 오류의 위치가 명확한 51개의 프로그램을 원본 프로그램 없이 생성할 수 있었고 사람이 작성한 오류 벤치마크와 유사한 난이도를 가짐을 확인하였다.

본 논문의 기여는 다음 2가지이다.

- 언어 모델이 생성한 합성 데이터로 자연스러운 오류 벤치마크를 구축하는 방법 제시
- 조건부 생성과 프로그램 검증 기법을 결합하여 생성되는 코드의 안전성을 보장하는 방법 제시

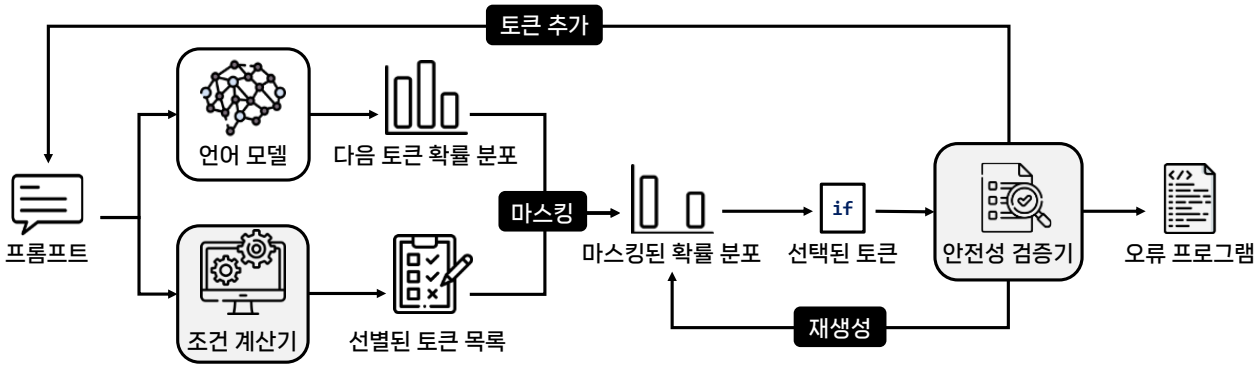
## 2. 방 법

### 2.1 전체 구조

BugSynth의 전체 구조는 그림 1과 같다. 가장 먼저 프롬프트 (Prompt)가 입력으로 들어오면 이를 언어 모델과 조건 계산기에 전달한다. 조건 계산기는 제약 조건을 절대 만족할 수 없는 토큰 (Token) 목록을 선별하며 이를 토대로 언어 모델이 출력한 확률 분포를 마스킹 (Masking)할 수 있다. 제약 조건의 만족 여부를 즉시 판단할 수 없는 경우, 안전성 검증기가 추가된 토큰들의 제약 조건 만족 여부를 사후에 검사하여 조건이 위배된다면 해당 토큰들을 삭제하고 재생성하도록 한다. 조건을 만족한다고 판단되는 토큰은 프롬프트에 추가되어 생성 흐름이 반복된다. 만약 생성 끝을 나타내는 토큰이 추가되었다면 반복을 종료하고 생성된 프로그램을 반환한다.

### 2.2 정의되지 않은 변수 (Undefined variable) 방지

BugSynth는 언어 모델이 생성하는 코드에 정의되지 않은 변수가 존재하지 않음을 보장한다. 매 순간 현재 생성 중인 범위 (Scope)에 정의된 변수 이외에 모든 변수 토큰을 마스킹한다. 예를 들어, 그림 2의 예시에서 3번째 줄 위치 (a)를 생성하고 있을 때 해당 범위에 정의된 변수는




**[그림 1]** BugSynth 흐름도

x, y, z, sum이다. 따라서 이 변수들을 제외한 나머지 변수 토큰을 마스킹하여 정의되지 않은 변수가 선택되지 않도록 한다.

### 2.3 0으로 나누는 오류 삽입

BugSynth는 0으로 나누는 오류를 유발하는 입력이 실제로 존재하도록 오류를 삽입한다. 먼저 무작위로 오류 유발 입력을 선정한다. 이 선정된 값을 입력으로 하여 지금까지 생성된 부분 프로그램을 인터프리터로 실행한다. 이후 현재 생성 중인 범위에서 정의된 변수 중 하나를 선택하여 0으로 나누는 오류가 발생하도록 나눗셈을 삽입한다. 예를 들어, 그림 2의 예시에서 오류 유발 입력이 x=4, y=3, z=2로 선정되었다면 6번째 줄 위치 (b)에 정의된 모든 변수의 값은 x=4, y=3, z=2, sum=9, avg=3이다. 이 중에서 변수 avg가 선택되었다면 나눗셈 '/(avg-3)'을 삽입하여 0으로 나누는 오류가 x=4, y=3, z=2일 때 발생함을 보장한다.

```
1   int average(int x, int y, int z){
2       int sum = x + y + z;
3       int avg = sum ⓐ / 3;
4
5       if(sum > 2){
6           int result = 2 / (avg - 3)ⓑ;
7           avg = result / (sum - 1)ⓒ
8       }
9   …
```

**[그림 2]** 예제 코드

### 2.4 기타 나눗셈의 안전성 (Soundness) 보장

BugSynth는 프로그램 검증 기법을 통해 목표 이외 지점에는 0으로 나누는 오류가 생성되지 않도록 보장한다. 구체적으로, 해당 위치에서 오류가 일어나는 조건을 CHC (Constrained Horn Clause) 논리식으로 표현한 후 그 논리식의 충족 가능성 (Satisfiability)을 확인하여, 안전하지 않은 나눗셈이라면 다시 생성하도록 한다. 예를 들어, 그림 2의 예시에서 7번째 줄 위치 (c)에 생성된 나눗셈은 다음과 같은 CHC 논리식으로 표현할 수 있다.

$$f(x,y,z) \wedge (sum = x+y+z) \wedge (avg = sum/3) \wedge (sum > 2)$$
$$\Rightarrow f.if(x,y,z,sum,avg)$$
$$f.if(x,y,z,sum,avg) \wedge (result = 2/(avg-3)) \wedge (sum - 1 = 0)$$
$$\Rightarrow BUG()$$

위 논리식에서 BUG로 도달하는 경우가 존재하지 않기 때문에 위치 (c)에 생성된 나눗셈은 안전하다.

만약 지금까지 생성된 코드에 반복문이 존재하는 경우 반복문의 불변식 (Invariant)을 계산하여 논리식으로 표현한다. 임의의 반복문의 의미를 정확하게 알아낼 수 없기 때문에, 반복문 내부의 명령문 (Statement) 대신 불변식을 논리식으로 표현하여 안전성을 보장한다.

## 3. 실 험

### 3.1 생성 통계

***RQ 1) BugSynth가 생성한 오류 프로그램의 품질은 어떠한가?***

**[표 1]** BugSynth의 생성 통계. 논리식 검증 제한시간 30초. Phi-2 모델[4], Top-p 샘플링(p=0.9, k=50, Temperature=1.0) 사용 RTX 2080 Ti × 4 GPU 환경에서 실험. 생성 시간 중앙값 34초.

| 생성 성공 여부 | 패턴/원인 | 개수 |
|---|---|---|
| 성공 (51개) | 안전한 나눗셈 존재 | 19 |
| | 오류만 존재 | 32 |
| 실패 (49개) | 시간 초과 | 29 |
| | 기타 생성 오류 | 20 |

BugSynth가 생성하는 오류 프로그램의 품질을 평가하기 위한 실험을 수행하였다. BugSynth로 100회 생성 시도하여 성공 및 실패 횟수를 측정하였고 생성 패턴과 실패 원인을 분석하였다.

실험 결과 BugSynth는 시도 횟수의 51%에 해당하는 51개의 오류 프로그램을 생성하는 데 성공하였다. 생성된 프로그램은 모두 오류 위치가 명확함을 감안하면 상당히 높은 비율로 생성에 성공했음을 확인할 수 있다. 또한, 오류 탐지기의 거짓 경보 (False alarm)를 평가할 수 있는 안전한 나눗셈이 존재하는 프로그램도 19개 생성되었다. 반면 실패한 사례는 49개 존재했으며 그 중에서는 논리식 검증에 시간이 오래 걸리거나 무한 루프가 생성되어 부분

프로그램 실행이 끝나지 않는 경우가 29개로 가장 많았다.

### 3.2 실제 벤치마크와의 비교

***RQ 2) BugSynth로 만든 벤치마크가 실제와 유사한가?***

BugSynth가 만든 51개의 오류 프로그램으로 벤치마크를 구축하여 사람이 작성한 벤치마크와 얼마나 유사한지 확인하는 실험을 수행하였다. 정적 분석기를 구현하는 수업 과제로 제출된 분석기 4개를 대상으로, 과제 채점용 기존 벤치마크 3종과 BugSynth 벤치마크에 대해 오류를 찾도록 했다.

**[표 2]** BugSynth 벤치마크와 사람이 작성한 벤치마크 3종 (A, B, C)으로 평가한 분석기 4개 (P, Q, R, S)의 정확도 (Precision)

| 요약 도메인 | 정적 분석기 | 벤치마크 | | | |
|---|---|---|---|---|---|
| | | **BugSynth** | **A** | **B** | **C** |
| Sign | P | 79.7% | 78% | 100% | 71.6% |
| | Q | 84.7% | 77.6% | 100% | 67.0% |
| | R | 79.7% | 78.2% | 75.0% | 65.7% |
| | S | 79.7% | 78.0% | 100% | 71.6% |
| | **평균** | **80.9%** | **78.0%** | **92.3%** | **68.9%** |
| Interval | P | 79.7% | 87.7% | 100% | 88.0% |
| | Q | 96.2% | 91.2% | 100% | 77.1% |
| | R | 79.7% | 84.9% | 75.0% | 73.4% |
| | S | 79.7% | 88.9% | 100% | 87.5% |
| | **평균** | **83.2%** | **88.1%** | **92.3%** | **81.2%** |

실험 결과 BugSynth 벤치마크는 전체 4종의 벤치마크 중 중간 정도의 난이도를 가졌으며, 특히 벤치마크 A와 유사함을 확인할 수 있었다. 두 벤치마크에서 정적 분석기의 정확도 차이를 비교한 결과, Sign 도메인에서는 3.7%, Interval 도메인에서는 5.9%의 비교적 낮은 정확도 차이를 보였다. 분석기가 출력한 경보를 확인한 결과, 두 벤치마크 모두 정답을 맞히는 것은 쉽지만 거짓 경보 개수에 의해 정확도가 달라지는 특징을 보였다.

BugSynth 벤치마크에서 분석기의 정확도가 비슷한 것은 두 요약 도메인에서의 분석 한계를 넘어서는 거짓 경보가 있었기 때문인 것으로 보인다. 따라서 평가 대상 분석기에 맞게 난이도를 조절하는 것이 BugSynth가 추후 개선해야 할 점으로 보인다.

## 4. 기술적 배경

### 4.1 언어 모델의 토큰 생성

언어 모델은 토큰을 순차적으로 생성해 나가는 방식으로 코드를 생성한다. 토큰은 언어 모델에서 다루는 문자열의 최소 단위이다. 따라서 언어 모델은 토큰의 나열을 입력으로 받으면 다음으로 올 토큰에 대한 확률 분포를 출력하는 함수로 생각할 수 있다. 확률 분포를 토대로 다음 토큰을 선택하는 방법으로는 가장 확률이 높은 토큰을 선택하거나 확률을 가중치로 하여 랜덤하게 샘플링(Sampling)하는 방법 등이 있다.

### 4.2 조건부 생성 (Constrained Decoding)

조건부 생성[5]은 언어 모델의 출력이 원하는 제약 조건을 만족하도록 제한하는 생성 전략이다. 그 실현 방법 중 하나로 토큰 마스킹 기법이 있다. 이는 모델이 출력한 확률 분포에서 제약 조건을 위배하는 토큰을 제거하여 선택되지 않도록 하는 기법이다. BugSynth에서는 토큰 마스킹 기법을 이용하여 의도되지 않은 오류 (정의되지 않은 변수 사용)를 방지하고 있다.

## 5. 결 론

본 논문에서는 언어 모델과 프로그램 검증 기법을 결합하여 오류 프로그램을 자동 생성하는 도구 BugSynth를 소개하였다. BugSynth는 0으로 나누는 오류의 위치가 명확한 프로그램을 51%의 비율로 생성에 성공하였다. 또한, BugSynth로 구축한 벤치마크는 사람이 작성한 벤치마크와 유사한 난이도를 가짐을 확인하였다. BugSynth는 기존 프로그램에 의존하지 않고 오류 프로그램을 생성하기 때문에 대규모 벤치마크를 쉽게 구축할 수 있다. 추후 이러한 신뢰성 있는 합성 데이터 생성 도구가 언어 모델의 학습 데이터 고갈 문제도 해결할 수 있을 것으로 기대한다.

## 6. 참고 문헌


[1] Villalobos, P., Sevilla, J., Heim, L., Besiroglu, T., Hobbhahn, M., & Ho, A. Will we run out of data? an analysis of the limits of scaling datasets in machine learning. *arXiv preprint arXiv:2211.04325*. (2022).

[2] Metzman, J., Szekeres, L., Simon, L., Sprabery, R., & Arya, A. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 1393-1403). (2021, August).

[3] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., ... & Whelan, R. Lava: Large-scale automated vulnerability addition. In *2016 IEEE symposium on security and privacy (SP)* (pp. 110-121). IEEE. (2016, May).

[4] Mojan, J., Sébastien, B., Marah, A., et al. Phi-2: The surprising power of small language models. In *Microsoft Research Blog*, from https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/. (2023).

[5] Hokamp, C., & Liu, Q. Lexically Constrained Decoding for Sequence Generation Using Grid Beam Search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1535–1546). Association for Computational Linguistics. (2017).