코드 생성 언어 모델의 코드 보안성 향상을 위한 프롬프트 튜닝

유미선 [1], 한우림 [1], 조윤기 [1], 백윤흥 [1]
[1] 서울대학교 전기정보공학부, 서울대학교 반도체 공동연구소

msyu@sor.snu.ac.kr, rimwoo98@snu.ac.kr, ygcho@sor.snu.ac.kr, ypaek@sor.snu.ac.kr

Prompt Tuning for Enhancing Security of Code in Code Generation Language Models

Miseon Yu[1], Yungi Cho[1], Woorim Han[1], Yunheung Peak[1]
[1]Dept. of Electrical and Computer Engineering and Inter-University Semiconductor Research Center (ISRC), Seoul National University

요 약

최근 거대 언어 모델의 발전으로 프로그램 합성 분야에서 활용되고 있는 코드 생성 언어 모델의 보안적 측면에 대한 중요성이 부각되고 있다. 그러나, 이를 위해 모델 전체를 재학습하기에는 많은 자원과 시간이 소모된다. 따라서, 본 연구에서는 효율적인 미세조정 방식 중 하나인 프롬프트 튜닝으로 코드 생성 언어 모델이 안전한 코드를 생성할 확률을 높이는 방법을 탐구한다. 또한 이에 따른 기능적 정확성 간의 상충 관계를 분석한다. 실험 결과를 통해 프롬프트 튜닝이 기존 방법에 비해 추가 파라미터를 크게 줄이면서도 보안률을 향상시킬 수 있음을 알 수 있었다. 미래 연구 방향으로는 새로운 조정 손실함수와 하이퍼파라미터 값을 조정하여 성능을 더욱 향상시킬 수 있는지 조사할 것이다. 이러한 연구는 보다 안전하고 신뢰할 수 있는 코드 생성을 위한 중요한 발전을 이끌 수 있을 것으로 기대된다.

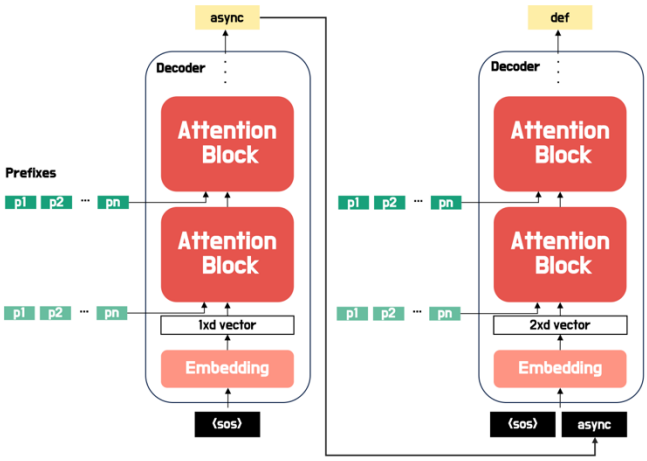
1. 서론

최근 거대 언어 모델 (Large Language Models, LLM)의 자연어 생성 능력이 발전함에 따라 LLM 을 다양한 영역에 적용하는 연구가 활발히 이루어지고 있다. 특히, 프로그램 합성 분야에서는 이러한 모델을 활용하여 코드를 자동 생성하는 연구가 진행되고 있다. GitHub 는 Copilot Chat 서비스를 통해 여러 소프트웨어 작업에 코드 생성 LLM 을 도입하고 있다. [1]에 따르면, Copilot 은 백만 명 이상의 개발자들과 5 천 개 이상의 기업에서 널리 사용되고 있다. 하지만, 이러한 코드 생성 LLM 이 보안적으로 취약한 코드를 작성할 수 있다는 연구 결과가 보고되었다. [2]에 따르면, Copilot 이 생성한 프로그램에서 40%가 중요한 보안 취약점을 포함하고 있었다. 이를 개선하기 위해 [3]은 사전 학습된(pre-trained) 모델에 지시학습(instruction tuning)을 통해 보안적 성능을 향상시키는 방안을 제시하였다. 하지만, 이는 전체 모델을 미세조정(fine-tuning)하기 때문에 상당한 시간, 자원 및 데이터셋이 필요하다. [4]는 이에 반해 효율적인 미세 조정 방법 중 하나인 프리픽스 튜닝(prefix-tuning)을 활용한다. 이는 모델 전체 파라미터의 약 0.1% 정도의 파라미터를 추가하고 이를 학습시킨다. 이를 통해 보안 작업에 특화되도록 모델을 조정할 수 있다. 그러나, 이러한 접근 방식은 보안적 성능을 강화하는 동시에 기존 모델이 보이던 기능적 정확성과의 상충 관계를 보이는 것으로 나타났다[3]. 본 논문은 이 보다 더 적은 파라미터가 사용되는 프롬프트 튜닝(prompt-tuning)을 통한 효율적인 조정 방식을 제안한다. 이에 대한 보안적 향상과 기능적 정확성 간의 상충 관계를 분석하여 프롬프트 튜닝의 적용 가능성을 평가한다.

2. 관련 연구

사전 학습된 LLM 이 생성하는 코드의 보안성을 강화하고자 하는 연구는 아직 초기 단계이다. [4]는 생성 코드의 보안성을 조절하는 SVEN 을 제안하였다. SVEN 은 프리픽스 튜닝을 활용하여 사전 학습 모델에 보안적 특성을 첨가하였다. 기존 공개 보안 데이터들을 학습에 적절하도록 전처리하고 이를 활용한

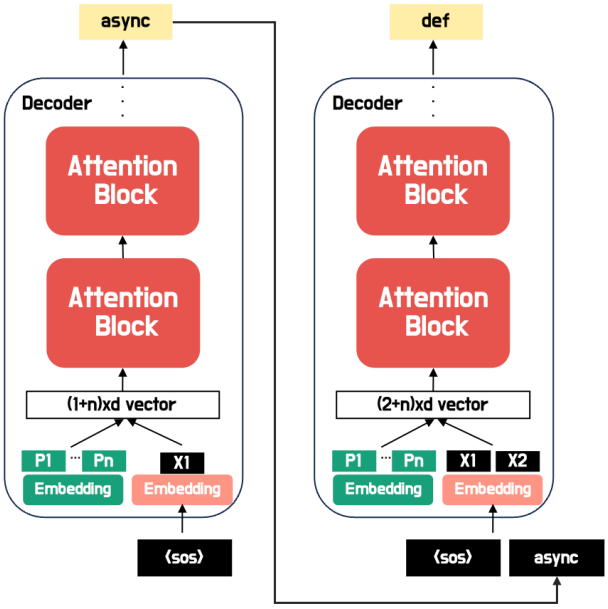
프리픽스 튜닝 및 추론 과정을 서술하였다. SVEN 은 이를 통해 모델의 파라미터 값 수정 없이 보안성을 강화 시켜주는 안전모드와 더 취약한 코드를 생성하는 취약모드를 사용자의 요구에 따라 조절할 수 있도록 하였다. [3]은 지시학습을 통해 생성 코드의 보안성을 강화하는 SafeCoder 를 제안하였다. SVEN 은 지시학습에 적용될 때, 보안과 기능성 사이의 교환이 내재적으로 제한되는데 반해, SafeCoder 는 두 가지 측면에서 모두 뛰어남을 보인다. 하지만, 지시 학습은 전체 모델의 파라미터 값을 수정하는 것이라 학습에 많은 시간과 자원이 소모되고 그만큼 많은 데이터가 필요하다. 본 연구는 지시 학습과 프리픽스 튜닝보다 더 적은 파라미터를 요구하는 프롬프트 튜닝을 적용하여 기존 LLM 의 보안성 강화 및 기능성 유지에 대해 분석하고자 한다.



(그림 1) 생성 LLM 에서의 프리픽스 튜닝

프롬프트 튜닝은 기존 모델의 성능을 향상시켜 특정 하류 작업(downstream task)을 수행할 수 있도록 하는 효율적인 훈련 메커니즘이다. 일반적으로 입력 임베딩 X앞에 일련의 토큰 P를 추가하여 수행된다. 여기서 모델이 정답인 Y를 출력할 가능성을 최대화하도록 토큰 P를 미세 조정하는 방식으로 학습이 진행된다[5]. [그림 1] 에서 볼 수 있듯이, 프리픽스 튜닝은 모든 트랜스포머 레이어에 추가되는 접두사 시퀀스를 학습하는 반면, 프롬프트 튜닝은 [그림 2]와 같이 입력 임베딩 앞부분에만 추가되어 파라미터 수를 효과적으로 줄인다. 기존 자연어 생성 모델에서는 특정 조건을 부여하는 controlled text generation 작업이 정의되어있다[6]. 이와 비슷하게 [4]에서는 코드 생성 모델이 보안적으로 강화된 코드를 생성하도록 하는 작업을 controlled code generation 으로 정의하였다. 기존 controlled text generation 에서는 프롬프트 튜닝을 적용하는 연구가 보고 되었으나[6] 해당 기술을 controlled code generation 에 적용하는 연구는 아직 보고된 적은 없고 본 논문이 첫 번째로 시도하고자 한다.

3. 안전한 코드 생성을 위한 프롬프트 튜닝



(그림 2) 생성 LLM 에서의 프롬프트 튜닝

본 논문에서 제안하는 프롬프트 튜닝의 개요는 [그림 2]과 같다. 자동회귀 언어모델(autoregressive language model)의 출력이 보안적 특성에 특화될 수 있도록 원래의 입력 앞에 n개의 임의의 연속적인 벡터 값을 가지는 토큰 임베딩(token embedding) P를 추가한다. 각 토큰 임베딩 $P_i(0 < i \leq n)$는 원래 인풋의 임베딩 차원 d의 크기를 가진다. 즉 추가된 P는 총 $n \times d$의 사이즈를 가진다. 이 추가된 P를 특정 작업의 라벨인 Y를 맞출 확률이 최대화 되도록 최적화 시키는 튜닝작업을 진행한다. 튜닝작업동안 사전 학습된 모델의 파라미터 값은 고정시킨다. 튜닝을 위한 훈련 데이터는 (x, m)쌍으로 되어 있고, x는 보안적인 코드, m은 원래 보안적이지 않던 코드와의 차이가 나는 지점을 1 로 기록한 마스크 매트릭스 이다. 그에 따른 손실함수는 다음과 같다.

$$L_{LM} = -\sum_{t=1}^{|x|} m_t \cdot \log P_r(x_t | h_{<t}) \qquad (1)$$

L_{LM}은 마스크 매트릭스 m과 함께 정의된다. 마스크 매트릭스 m은 코드에서 보안적으로 민감한 부분만 1 로 설정하여 그 부분에서의 손실(loss)만을 계산할 수 있도록 한다. x_t는 현재 t시점에서의 입력 토큰, $h_{<t}$는 t 시점 이전의 모든 히든 스테이트(hidden state)들

을 의미한다. P_r은 다음 토큰의 확률 분포를 의미한다. 추론 과정에서는 일련의 학습과정을 통해 얻은 토큰 임베딩 P의 고정된 값을 이용하여 이전에 없던 보안적 특성을 사전 학습된 모델에 첨가한다.

4. 실험 및 결과

4-1. 실험 세팅

본 논문은 실험을 위해 CodeGen[7] 350M, 2.7B 모델을 사용한다. 최대 토큰 길이를 1024 로 설정하였다. 실험은 3 개의 NVIDIA A6000 GPUs 에서 수행되었다. 보안 관련 훈련 및 평가를 위한 데이터셋으로는 MITRE top-25 에 포함되는 9 가지 CWEs 를 다루는 오픈 데이터셋을 사용하였다[3]. 선택된 9 가지 CWE 에 대해서는 [표 1]에서 볼 수 있다. 모두 Python 과 c/c++ 언어로만 이루어져 있고, 훈련 데이터는 720 개, 테스트 데이터는 83 개의 샘플로 나누었다. 프롬프트 튜닝을 위한 추가 토큰 임베딩 개수 n은 CodeGen-350M 의 경우 5, CodeGen-2.7B 의 경우 8 로 설정하였다. 모든 실험의 모델 샘플링 온도(sampling temperature)는 0.4 이다.

CWE id	취약점 이름	데이터 개수
022	path traversal	train: 51, val: 6
078	OS command injection	train: 95, val: 11
079	cross-site scripting	train: 45, val: 5
089	SQL injection	train: 184, val: 20
125	out-of-bound read	train: 130, val: 15
190	integer overflow	train: 38, val: 5
416	use after free	train: 57, val: 7
476	null pointer dereference	train: 70, val: 8
787	Out-of-bound write	train: 50, val: 6
	전체 데이터 개수	train: 720, val: 83

<표 1> 프롬프트 튜닝 및 보안률 평가를 위한 데이터셋의 9 가지 취약점 및 그에 따른 데이터 개수
(train: 훈련 데이터 개수, val: 테스트 데이터 개수)

4-2. 평가 항목

보안성 측정을 위해 보안률(security rate)을 계산한다. 보안률은 전체 유효한 생성 코드 샘플들 중에 보안적으로 안전한 코드의 비율이다. 각 평가 시나리오는 하나의 CWE 만을 대상으로 삼는다. 본 실험에서는 각 평가 시나리오 당 25 개의 샘플을 추출하고 이 중에서 유효한 샘플들 중 보안적으로 안전한 샘플의 비율을 측정한다. 안전한 샘플의 판단은 CodeQL[8]을 활용한다. CodeQL 은 특정 보안 취약점을 감지하기 위해 쿼리를 작성할 수 있는 오픈 소스 보안 분석 도구이다. 기능적 정확성을 측정하기 위해서는 HumanEval[9] 벤치마크에서 20 가지 시나리오를 사용한다. 이 모든 시나리오에 대해 평균 pass@k 점수를 측정한다. 전체 생성 코드에서 무작위로 k 개를 골랐을 때 1 개라도 모든 유닛 테스트(unit test)들을 통과하는 코드가 있으면 해당 모델이 문제를 풀었다고 정의한다. 따라서 pass@k 는 모든 k 개로 이루어진 그룹 중 문제가 풀어진 비율을 의미한다.

4-3. 실험 결과

CodeGen 350M, 2.7B 각각에 대해 측정된 보안률은 다음 [표 2]에서 볼 수 있다. 350M 모델의 경우 원래의 사전 학습 된 모델의 보안률은 약 62% 였다면, SVEN 의 경우는 약 84%로 약 23% 증가 하였지만, 프롬프트 튜닝이 적용된 우리 방식의 경우 약 66%로 약 4% 증가하였다. 그에 반해, [표 3]와 같이 pass@k 점수의 경우 SVEN 은 대부분의 경우에 대해서 값이 감소하는데 비해, 우리의 방식은 k 가 1 인 경우를 제외하고 원래의 LM 과 비슷하거나 더 높은 수치의 점수를 가진다. 이를 통해, 프롬프트 튜닝 방식은 SVEN 에 비해 보안성 강화는 약하지만 원래의 기능적인 부분은 유지되거나 향상되는 걸 확인할 수 있다. 또한, [표 4]에서와 같이 우리의 방법은 SVEN 보다 필요한 파라미터가 평균적으로 약 99% 적다. 그만큼 한정적인 데이터 셋으로 미세 조정할 수 있는 영향력이 적었다. 본 방법은 조금 더 적은 양의 시간 및 메모리 공간 사용으로 기존의 기능적 정확성을 유지시키면서 보안적 강화를 조정할 수 있는 방안이 될 것이다.

방법 / 모델	original	SVEN	ours
CodeGen-350M	62.10	85.40	66.30
CodeGen-2.7B	58.90	92.30	67.50

<표 2> 보안률 결과 (%)
(original: 원래 사전훈련 모델,
SVEN: [4]에서 제안한 방법으로 훈련된 모델,
ours: 본 논문에서 제안된 방법으로 훈련된 모델)

k / 방법	pass@1	pass@10	pass@50	pass@100
original	8.2	18.3	20.0	20.0
SVEN	8.3	11.4	14.4	15.0
ours	3.7	15.7	22.5	25.0

<표 3> CodeGen-350M 에 대한 pass@k 결과

	SVEN	ours	감소율(%)
CodeGen-350M	409,600	5,120	98.75
CodeGen-2.7B	2,621,440	20,480	99.22

<표 4> SVEN 과 우리 방법의 추가 파라미터 크기 비교

5. 결론 및 토의

본 연구는 안전한 코드 생성을 위해 프롬프트 튜닝을 활용하는 방법을 조사했다. 연속적인 값을 가지는 임의의 토큰 임베딩을 추가하고 조정함으로써 모델이 보안 코드 생성 작업에 적합하게 동작하도록 유도하고자 했다. 실험을 위해 CodeGen 의 350M, 2.7B 크기의 모델을 활용했고 각 모델에 대해 보안률과 기능적 정확성 수치를 통해 기존의 방법과 본 방법의 효과를 비교했다. 기본적인 프롬프트 튜닝의 방법을 사용해 추가 파라미터는 약 99% 감소시켰지만, 보안률의 상승폭이 기존 방법에 비해 크지 않았다. 본 연구는 기본적인 프롬프트 튜닝의 손실함수를 사용했다. 향후 연구에서는 보안 작업에 적합한 새로운 조정 손실함수를 적용하여 성능을 더욱 향상시킬 수 있는지 조사하고자 한다. 또한, 추가 토큰 수나 모델 샘플링 온도 등의 하이퍼파라미터 값을 조정하여 다양한 설정에서의 결과를 분석하고자 한다.

ACKNOWLEDGEMENT

이 논문은 2024 년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (RS-2023-00277326). 이 논문은 2024 년도 BK21 FOUR 정보기술 미래인재 교육연구단에 의하여 지원되었음. 이 논문은 2020 년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2020-0-01840,스마트폰의 내부데이터 접근 및 보호 기술 분석). 이 논문은 2023 년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (IITP-2023-RS-2023-00256081) 본 연구는 반도체 공동연구소 지원의 결과물임을 밝힙니다.

참고문헌

[1] Thomas Dohmke. 2023. GitHub Copilot X: the AI-powered Developer Experience. https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/.

[2] Hammond Pearce, et al. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. IEEE S&P. 2022.

[3] Jingxuan He, et al. Instruction Tuning for Secure Code Generation. arXiv. 2024.

[4] Jingxuan He, et al. Large language models for code: security hardening and adversarial testing. CCS. 2023.

[5] Brian Lester, et al. The Power of Scale for Parameter-Efficient Prompt Tuning. EMNLP. 2021

[6] Hanqing Zhang, et al. A Survey of Controllable Text Generation Using Transformer-based Pre-trained Language Models. ACM Comput. Surv. 56, 3, Article 64, 37 pages. 2024.

[7] Erik Nijkamp, et al. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. ICLR. 2023.

[8] 2023. CodeQL-GitHub. https://codeql.github.com.

[9] Mark Chen, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv. 2021.