AST 기반 정적 분석을 활용한 소프트웨어 취약점 탐지 성능 개선 연구

차우찬, 이선영*

순천향대학교

{chawoochan2, *sunlee}@sch.ac.kr

Enhancing Software Vulnerability Detection Performance Using AST-Based Static Analysis

Woo Chan Cha, Sun Young Lee*

Dept. of Information Security Enginnering, Soonchunhyang Univ.

요 약

소프트웨어 취약점으로 인한 보안 위협이 증가함에 따라, 취약점을 사전에 탐지하기 위한 정적 분석 기법의 중요성이 강조되고 있다. 특히 추상 구문 트리(Abstract Syntax Tree, AST)는 코드의 구조 정보를 반영할 수 있어 취약점 탐지 성능을 향상시킬 수 있다는 기대가 있으나, AST 파싱 정보의 실질적 효과에 대한 실증적 분석은 충분하지 않다. 본 연구는 AST 파싱이 소프트웨어 취약점 탐지 성능에 미치는 영향을 분석하기 위해, 동일한 학습 기반 코드 표현 모델 환경에서 입력 표현 방식만을 달리하여 각 접근법을 비교하였다. 원본 소스코드 만을 사용하는 Baseline과, AST 구조적 특징 만을 사용하는 AST 단독 방식, 원본 코드와 AST 특징을 결합한 통합 방식을 적용하여 성능을 평가하였다. 실험 결과, AST 단독 방식은 Baseline 대비 성능 저하를 보였지만, 통합 방식은 Baseline 대비 F1-Score 기준 3.6%의 성능 향상을 보였다. 이는 AST 구조 정보가 단독으로는 취약점 판단에 충분하지 않으며, 코드의 의미론적 표현과 결합될 때 효과적으로 활용될 수 있음을 나타낸다.

Ⅰ. 서 론

최근 소프트웨어의 복잡성이 증가함에 따라, 소프트웨어 취약점으로 인한 보안 사고의 위험성 또한 지속적으로 확대되고 있다. 이러한 취약점은 공격자에 의해 악용될 경우 심각한 시스템 침해로 이어질 수 있으며, 이에 따라 소프트웨어 개발 단계에서 취약점을 사전에 탐지하기 위한 정적 분석 기법의 중요성이 강조되고 있다.

기존의 정적 분석 기반 취약점 탐지 기법은 주로 소스코드의 토큰 정보나 패턴 매칭 방식에 의존하고 있다. 그러나 이러한 접근 방식은 코드의 구조적 특성과 문맥 정보를 충분히 반영하지 못하는 한계를 가지며, 그 결과 오탐이 발생하거나 복잡한 취약점 패턴을 정확히 탐지하지 못하는 문제가 존재한다. 이러한 한계를 보완하기 위해 학습 기반 코드 분석 기법과 구조적 정보 활용에 대한 연구가 활발히 진행되고 있다.

추상 구문 트리(Abstract Syntax Tree, AST)는 소스 코드의 문법적 구조를 계층적으로 표현할 수 있어, 코드의 구조 정보를 효과적으로 반영할 수 있는 대표적인 방법이다. 이에 따라 AST 파싱 정보를 활용한 취약점 탐지 기법은 성능 향상을 기대할 수 있는 접근 방식으로 제시되어 왔다. 그러나 AST 파싱 정보가 실제로 취약점 탐지 성능에 얼마나 기여하는지에 대한 실증적 분석은 충분하지 않으며, AST 정보의 단독 활용이 항상 긍정적인 효과를 보장하는지에 대해서는 명확히 검증되지 않았다.

본 연구에서는 AST 파싱이 소프트웨어 취약점 탐지 성능에 미치는 영향을 분석하는 것을 목적으로 한다. 이를 위해 동일한 학습 기반 코드 표현 환경에서 입력 표현 방식만을 달리한 세 가지 접근법을 비교한다. 구체적으로, 원본 소스 코드만을 사용하는 Baseline, AST 구조 특징만을 사용하는 AST-Only 방식, 원본 코드와 AST 특징을 결합한 Combined 방식을 적용하여 성능을 평가한다. 이를 통해 AST 기반 입력 표현의 효과와 한계를 분석하고, 구조적 정보가 취약점 탐지에서 효과적으로 활용될 수 있는 조건을 검증하려고 한다.

Ⅱ. 실험 방법

2.1. 데이터셋 및 샘플 구성

본 연구에서는 C/C++ 소프트웨어 취약점 탐지를 위해 DiverseVul 데이터셋을 사용한다. 해당 데이터셋은 실제 CVE를 기반으로 라벨링된 대규모 오픈소스 취약점 데이터셋으로, 취약 함수와 정상 함수 쌍으로 구성된다. 위험도와 발생 빈도를 고려하여 CWE-787(Out-of-bound Write), CWE-476(NULL Pointer Dereference), CWE-190(Integer Overflow)의 세 가지 취약점 유형을 선정하였고, 각 CWE 유형에 대해 취약 함수와 정상 함수를 동일한 수로 샘플링하였다. 총 2,040개의 함수 단위 샘플을 구성하였으며, 데이터는 CWE별 비율을 유지한 상태로 학습데이터 80%와 테스트 데이터 20%로 분류하였다.

2.2. AST 파싱 및 특징 추출

소스 코드의 구조적 정보를 추출하기 위해 Tree-sitter 기반 C/C++ 파서를 사용하여 AST를 생성하였다. AST로부터 총 64개의 구조적 특징을 추출하였으며, 이는 구조적 메트릭, 노드 타입 분포, 연산자 통계, 제어 흐름 패턴, 취약점 관련 함수 사용 여부의 다섯 범주로 구성된다. 모든 AST 특징은 학습 데이터 기준으로 정규화되어 사용된다.

2.3. 입력 표현 구성

AST 파싱의 영향을 분석하기 위해 입력 표현 방식만을 달리한 세 가지 접근법을 정의한다.

Baseline 방식은 원본 소스 코드를 학습 기반 코드 표현 모델에 입력하여 의미론적 임베딩 벡터를 생성하고, 이를 분류기의 입력으로 사용한다.

AST-Only 방식은 원본 소스 코드를 사용하지 않고, AST로부터 추출한 64개의 구조적 특징 백터만을 입력으로 사용한다.

Combined 방식은 원본 소스 코드로부터 생성된 의미론적 임베딩 벡터

와 AST 구조적 특징 벡터를 결합하여 하나의 입력 벡터로 구성한다. 이를 통해 코드의 의미 정보와 구조 정보가 동시에 반영된다.

2.4. 모델 및 학습 설정

코드의 의미론적 표현은 GraphCodeBERT를 사용하여 추출하며, 해당 모델은 특징 추출기로만 활용된다. 생성된 임베딩 벡터는 동일한 트리 기반 분류기를 통해 취약점 여부를 분류한다. 모든 실험은 동일한 분류기 설정을 사용하여 입력 표현 방식에 따른 영향만을 비교한다.

2.5. 평가 지표

모델 성능 평가는 Accuracy, Precision, Recall, F1-Score, AUC-ROC 지표를 기준으로 수행한다. Auccracy는 전체 샘플 중 정분류 비율이며, Precision은 취약으로 예측한 샘플 중 실제 취약의 비율로 오탐(False Positive) 억제 정도를 나타낸다. Recall은 실제 취약 샘플 중 탐지된 비율로 미탐(False Negetaive) 감소 정도를 의미한다. F1-Score는 Precision과 Recall의 조화 평균으로, 오탐과 미탐을 동시에 고려한 단일 지표이다. AUC-ROC는 분류 임계값 변화에 따른 성능을 평가하는 지표로, 취약 함수와 정상 함수를 얼마나 안정적으로 구분할 수 있는지를 나타내는 지표로 사용된다. 본 연구에서는 오탐과 미탐의 균형이 중요한 특성을 고려하여 F1-Score를 주요 평가 지표로 사용한다.

Ⅲ. 실험 결과

표 1. 입력 표현 방식에 따른 취약점 탐지 성능 비교

접근법	Accuracy	Preicison	Recall	F1-Score	AUC
Baseline	67.16%	66.36%	69.61%	67.94%	0.736
AST-only	65.20%	65.98%	62.75%	64.32%	0.701
Combined	70.10%	69.71%	71.08%	70.39%	0.747

표 1은 Baseline, AST-only, Combined 방식의 성능을 비교한 결과이다. 실험 결과, Combined 방식이 모든 평가 지표에서 가장 우수한 성능을 보였으며 F1-Score는 0.7039로 Baseline 대비 약 3.6% 향상되었다. 또한 AUC 역시 Combined 방식이 가장 높게 나타나, 전반적인 분류 성능이 개선되었음을 확인하였다. 반면, AST-Only 방식은 Baseline 대비 F1-Score가 약 5.3% 감소하였고, AUC 역시 낮아지는 결과를 보였다. 이는 AST 구조적 특징만으로는 취약점 탐지에 필요한 의미론적 정보가 충분히 반영되지 않음을 의미하며, AST 구조적 특징은 코드의 의미론적 표현과 결합될 때 효과적으로 활용될 수 있음을 보여준다.

Ⅲ. 결론

본 연구에서는 추상 구문 트리 파싱 정보가 소프트웨어 취약점 탐지 성능에 미치는 영향을 분석하였다. 이를 위해 동일한 학습 기반 코드 표현 환경에서 입력 표현 방식만을 달리한 세 가지 접근법을 비교하였다. 실험 결과, AST 구조적 특징만을 사용하는 방식은 Baseline 대비 성능 저하를 보인 반면, Baseline과 AST 구조적 특징을 결합한 방식은 가장 우수한 탐지 성능을 보였다. 이러한 결과는 AST 파싱 정보가 단독으로는 취약점 탐지에 필요한 정보를 충분히 제공하지 못하며, 코드의 의미론적 표현과 결합될 때 상호보완적으로 활용될 수 있음을 의미한다. 한편, 본 연구는 한정된 데이터셋을 대상으로 실험을 수행하였으며, 함수 수준의 이진 분류에 범위를 한정하였다. 따라서 다양한 CWE 유형 및 대규모 데이터셋에서의 추가 검증이 필요하며, CWE 다중 분류 수준으로 연구 범위를 확장하는 추가 연구가 필요하다

ACKNOWLEDGMENT

※ 본 연구는 정부(과학기술통신부)의 재원으로 한국연구재단의 지원을 받아 수행한 연구임(NO.RS-2024-00346749)

참 고 문 헌

[1] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, “DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection,” in Proceedings of the 26th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2023), ACM, 2023.

[2] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” in Findings of EMNLP 2020, pp. 1536-1547, 2020.

[3] D. Guo, S. Ren, S. Lu, A. Svyatkovskiy, D. Meijer, S. R. Chandra, M. Tufano, J. Duan, M. Gong, N. Deng, C. Fu, and M. Zhou, “GraphCodeBERT: Pre-training Code Representations with Data Flow,” in ICLR 2021, 2021.

[4] Z. Tian, B. Tian, J. Lv, Y. Chen, and L. Chen, “Enhancing Vulnerability Detection via AST Decomposition and Neural Sub-tree Encoding,” Expert Systems with Applications, vol. 238, 122204, 2024.

[5] R. Wang,, S. Xu, Y. Tian, X. Ji, X. Sun, and S. Jiang, “SCL-CVD: Supervised Contrastive Learning for Code Vulnerability Detection via GraphCodeBERT,” Computers & Security, vol. 143, 103994, 2024.

[6] X. Zhang, H. Li, Y. Liu, and Q. Wang, “SCL-CVD: Supervised Contrastive Learning for Code Vulnerability Detection via GraphCodeBERT,” Computers & Security, vol. 145, 103252, 2024.