AI 및 LLVM 을 활용한 정적 분석 기반 시큐어 코딩 점검 도구 연구

강인석 [1], 김보송 [2], 박솔빈 [3], 윤건우 [4], 조준형 [5], 서혁준 [6]
[1] 창원대학교 컴퓨터공학과 학부생
[2] 성신여자대학교 융합보안공학과 학부생
[3] 홍익대학교 자율전공 학부생
[4] 가천대학교 스마트보안학과 학부생
[5] 단국대학교 소프트웨어학과 학부생
[6] LG CNS

Research on static analysis-based secure coding tools using AI and LLVM

In-seok Kang[1], Bo-song Kim[2], Sol-bin Park[3], Geon-woo Yoon[4] , Jun-hyeong Cho[5], Hyuck-jun Suh[6]
[1] Department of Computer Engineering, Changwon National University
[2] Department of Convergence Security Engineering, Sungshin Women's University
[3] Department of free major, Hongik University
[4] Department of Smart Security, Gachon University
[5] Department of Software, Dankook University
[6] LG CNS

요 약

이 연구는 LLVM IR 을 활용한 보안 취약점 탐지의 새로운 접근 방식을 제시합니다. Juliet Test Suite 의 CWE-121 Stack-based Buffer Overflow 데이터를 사용하여 Word2Vec 으로 코드를 벡터화하고 LSTM 모델로 학습했습니다. 모델 성능은 정확도 90%, 정밀도 87%, 재현율 93%, F1 스코어 90%로 평가되었습니다. 향후 다양한 보안 취약점을 다룰 수 있는 다중 분류 모델로 확장 가능성을 제안합니다.

1. 서론

소프트웨어가 점점 더 복잡해지고 다양한 환경에서 활용됨에 따라, **시큐어 코딩(Secure Coding)**의 중요성은 그 어느 때보다 커지고 있다. 시큐어 코딩은 안전한 소프트웨어 개발을 위해 소스 코드 등에 존재할 수 있는 잠재적인 보안 취약점을 제거하고, 보안을 고려하여 기능을 설계 및 구현하는 등 소프트웨어 개발 과정에서 지켜야 할 일련의 보안 활동을 말한다.

전통적으로 C, Java 와 같은 고수준 프로그래밍 언어에서 이러한 보안 분석이 이루어졌으나, 본 연구에서는 중간 레벨 언어인 **LLVM IR(Low Level Virtual Machine)**을 기반으로 한 새로운 접근 방식을 제안한다. LLVM IR 은 프로그램의 실행 구조와 의미를 더 명확히 표현할 수 있어, 이를 학습한 모델이 얼마나 효과적으로 코드의 취약점을 탐지할 수 있는지를 평가하고자 하였다.

```
#include <stdio.h>
int main() {
  printf("Hello, World!\n"); return 0;
}
```

```
@.str = private unnamed_addr constant [15 x i8] c"Hello, World!\0A\00", align 1
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(i8* noundef getelementptr inbounds ([15 x i8], [15 x i8]* @.str, i64 0, i64 0))
  ret i32 0
}
```

<그림 1> C, LLVM 비교

2. 본론

A. 데이터 소개

Juliet 테스트 스위트(Juliet Test Suite)는 소프트웨어의 보안 취약점을 탐지하고 예방하는 연구와 도구 개발에 자주 사용되는 표준 데이터 세트입니다. 미국 국립표준기술연구소(NIST) 산하의 National Security Agency(NSA)에서 개발한 이 테스트 데이터들은 주로 C/C++ 및 Java 언어를 대상으로 하며, 다양한 유형의 보안 취약점이 포함된 코드를 제공합니다.

CWE-121 Stack Based Buffer Overflow **취약점 10,375 개**의 코드를 AI 모델 생성에 사용하였습니다.

B. 데이터 전처리

먼저, Juliet 코드에서 bad 와 good 코드를 따로 추출한 후 LLVM IR 코드로 변환합니다.

변환된 코드에서 디버깅 관련 메타데이터와 불필요한 선언문을 제거하여 분석에 필요한 코드만 남도록 처리하였습니다. 그 후, 변수명과 숫자를 일관성 있게 변환하여 학습 데이터를 정규화하였습니다. 마지막으로, 특수 문자를 기준으로 코드를 토큰화하여 분석에 적합한 형태로 변환하였습니다. 이러한 전처리 과정을 통해 코드의 의미론에 집중하면서, 학습의 효율성을 높이기 위한 데이터 전처리가 수행되

었습니다.
코드를 토큰 단위로 분할하였을 때, 토큰 개수가 100 개 이하인 데이터 함수 호출의 기능만 하는 짧은 코드는 제거하였습니다.

Preprocessing	Description
Juliet 코드에서 bad, good 코드 추출	Juliet 테스트 케이스에서 보안 취약점이 포함된 "bad" 코드와 안전한 "good" 코드를 추출한 후, 이를 LLVM IR로 변환합니다.
디버깅 및 메타데이터 관련 정보 제거	llvm.dbg.declare, llvm.dbg.label, llvm.loop, !dbg, ![숫자], #[숫자] 등 디버깅 관련 메타데이터를 제거하여 불필요한 정보를 삭제합니다.
선언문 제거	declare void @와 같은 함수 선언문을 제거하여 분석에 필요 없는 내용을 정리합니다.
변수명 변환	%로 시작하는 변수명을 VAR로 치환하여 일관성 있게 처리합니다.
숫자 토큰화	코드 내 숫자들을 각각의 숫자 단위로 나누어 토큰화합니다.
줄바꿈 문자 제거	코드 내 줄바꿈 문자(₩n)를 공백으로 치환하여 한 줄로 통합합니다.
특수 문자 처리 및 토큰화	특수 문자 및 구분자([^$#!@.%_a-zA-Z0-9*])를 기준으로 코드를 토큰화하여 개별 요소로 분리합니다.

<표 1> 전처리 단계

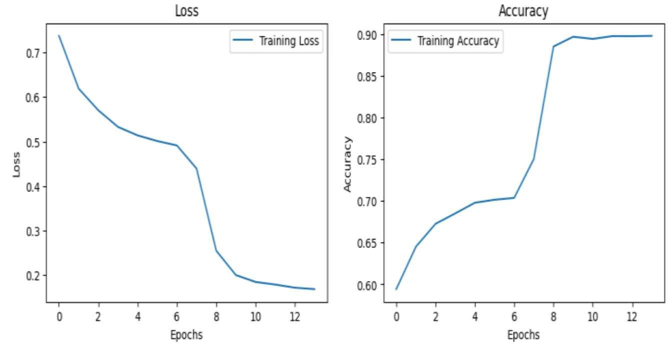
C. 모델 학습

A. Word2Vec

본 연구에서는 **Gensim 라이브러리**를 사용하여 Word2Vec 모델을 생성하였습니다. 텍스트 데이터를 토큰 단위로 나누어, 각 단어의 의미를 벡터로 표현하고 유사성을 학습하도록 하였습니다. Word2Vec 모델은 대규모 언어 모델(LLM)과는 달리, LLVM IR에 맞춘 임베딩을 생성할 수 있어 코드 문맥을 더욱 효과적으로 이해하고 유사도를 파악할 수 있습니다.
Word2Vec 모델 학습을 위해 파라미터는 벡터 크기(size)를 50, 윈도우 크기(window)를 10, 최소 빈도(min_count)를 1 로 설정하였으며, 중복된 토큰을 제외하고 총 543 개의 단어를 학습시켰습니다.

B. Deep Neural Networks

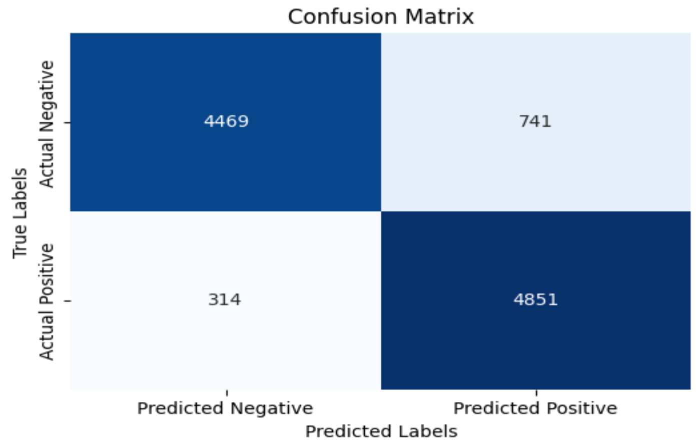
모델학습에는 LSTM(Long Short Term Memory)를 사용하였습니다. LSTM 모델은 3 개의 은닉층을 사용하였으며, 각각 64, 64, 128 개의 뉴런으로 구성하였습니다. 각 계층 사이에는 20%의 Dropout 과 BatchNormalization 로 과적합을 방지하였습니다. 각 LSTM 층에서 활성화 함수로 **하이퍼볼릭 탄젠트(tanh)**를 사용하였으며, 이 함수는 (-1, 1) 범위의 출력을 생성하여 기울기 소실 문제를 최소화하고 정보의 균형 있는 전달을 돕습니다.
특히, 본 연구에서는 기존의 단방향 LSTM 대신 양방향 LSTM(Bidirectional LSTM)을 사용하여 정방향과 역방향에서 동시에 문장을 학습하게 하였습니다. 이를 통해 문장의 전후 맥락을 보다 잘 반영할 수 있어 정확도가 향상되었습니다.
최적화 알고리즘으로는 Adamax **옵티마이저**를 사용하였고, 기본 학습률인 0.002 를 사용하였습니다. 또한, Epochs 는 20 으로 설정하였으며, validation_loss 기반으로 early stopping 을 적용하여 학습을 조기 종료하도록 하였습니다. 그 결과, 11 번째 Epochs 에서 가장 높은 **정확도(90%)**와 **손실률(15%)**을 기록하였습니다.



<그림 2> epochs 횟수에 따른 accuracy, loss

3. 결론 및 향후 연구 방향

본 논문에서는 Stack-based Buffer Overflow 코드의 테스트 데이터를 대상으로 정확도 90%, 정밀도 87%, 재현율 93%, F1 스코어 90% 결과를 보여줍니다. 정밀도가 비교적 낮기 때문에 긍정적 예측이 실제로는 부정적인 클래스로 판명하는 문제가 발생 가능성이 존재하지만, 전체적으로 일반화된 성능을 보이고 있습니다.
향후 연구 방향으로는 현재는 Stack-based Buffer Overflow 라는 취약점을 대상으로 하고 있지만, 다른 취약점을 가진 데이터를 학습하여 다양한 취약점을 분류할 수 있는 다중 분류 모델로 발전할 것입니다.



<그림 2> 모델 평가 지표

[1] Kvarnström, O. (2016). Static Code Analysis of C++ in LLVM.
[2] Gallagher, S. K., Klieber, W. E., & Svoboda, D. (2022). LLVM intermediate representation for code weakness identification. *Defense Technical Information Center, Tech. Rep.*
[3] McCully, G. A., Hastings, J. D., Xu, S., & Fortier, A. (2024). Bi-Directional Transformers vs. word2vec: Discovering Vulnerabilities in Lifted Compiled Code. *arXiv preprint arXiv:2405.20611.*

"본 논문은 과학기술정보통신부 대학디지털교육역량강화사업의 지원을 통해 수행한 ICT 멘토링 프로젝트 결과물 입니다"