논문 2022-2-1 http://dx.doi.org/10.29056/jsav.2022.12.01

흐름 그래프 정보의 기계학습을 활용한 보안 취약점 정적 탐지

모지환*, 홍성문*, 도경구**†

Static Detection of Security Vulnerabilities Using Machine Learning on Graph Information

Ji-Hwan Mo*, Sung-Moon Hong*, Kyung-Goo Doh**†

요 약

소프트웨어의 소스코드를 분석하여 보안 취약점을 정적으로 탐지하는 방법으로 오염분석이 널리 쓰이고 있다. 오염분석은 사용자 입력이 사용 지점에 도달하기 전에 위험 요소가 제거되는지 여부를 가능한 실행 흐름 경로를 모두 분석하여 탐지하는 분석 방법이다. 그런데 입력 데이터를 모르는 상황에서 실행 의미를 요약하여 분석할 수밖에 없어서, 감당할 수 있을 정도의 분석 비용으로 만족할 만큼 정확한 분석 결과를 얻기 힘든 단점이 있다. 이 논문은 그래프를 기계학습의 데이터로 활용할 수 있게 해주는 모델인 graph2vec을 사용하여, 소스코드의 흐름 그래프 데이터를 기반으로 기계학습 기술을 활용하여 보안 취약점을 탐지하는 방법을 제안한다. 소스코드의 흐름그래프를 구축한 다음, 벡터 데이터로 변환하여 기계학습을 진행한다. 프로그램 소스코드에 돌연변이 기법을 적용하여 충분한 데이터를 구축하여 학습을 시키고 평가를 진행한 결과, 약 99%의 정확도로 보안 취약점을 탐지함을 확인함으로써 기존의 재래식 보안 취약점 탐지 기술을 기계학습으로 대체할 수 있음을 확인하였다.

Abstract

Taint analysis is widely used in detecting security vulnerabilities in source code. However, it is difficult to obtain accurate results in reasonable amount of computing time due to the nature of static analysis. This paper proposes a new way of detecting security vulnerabilities using machine-learning technology utilizing an already established model, graph2vec. The flow graph of a program is transformed into a vector and then given to the model. The data set is prepared by first constructing simplified program and then applying mutation. The evaluation results show that the model detects security vulnerabilities with the accuracy of up to 99%, positively showing the possibility of applying machine-learning technology to the static detection of security vulnerabilities in source code.

한글키워드 : 오염 분석, 보안 취약점, 흐름 그래프, 기계학습, 그래프 임베딩

keywords : taint analysis, security vulnerability, flow graph, machine learning, graph embedding

* 한양대학교 대학원 컴퓨터공학과
** 한양대학교 소프트웨어학부
† 교신저자: 도경구(email: doh@hanyang.ac.kr)

접수일자: 2022.11.03. 심사완료: 2022.12.06.
게재확정: 2022.12.20.

1. 서 론

프로그램 분석 기법 중 오염 분석(taint analysis)[1,2]은 프로그램의 외부입력을 모두 믿을 수 없는 값으로 간주하고 이들의 데이터 흐름을 추적하여 프로그램에 어떤 영향을 주는지를 파악한다. 이러한 특징으로 인해 소스코드 특징 추출, 운영체제 동작 감시 등 여러 방면에 활용되고 있는데, 특히 SQL 명령 삽입, OS 명령 삽입, XSS(cross site scripting)과 같은 해킹 공격에 대한 소스코드의 보안 취약점[3] 탐지에도 널리 사용되고 있다.

그림 1은 이 중에서도 SQL 명령 삽입 공격에 취약한 코드의 예시이다. tableName과 name값을 사용자에게서 입력으로 받아 SQL 쿼리를 생성하고 있는데, 사용자의 입력에 따라 조작된 쿼리문 전달이 가능해지고 이로 인해 개발자의 의도와 달리 중요한 정보들이 유출될 수 있다. 이를 막기 위해서는 테인트 소스(taint source)와 테인트 싱크(taint sink) 사이에 개발자의 의도에 맞게 프로그램이 실행되도록 필터링하는 코드가 필요하다.

번호	소스 코드
1	String tableName = request.getParameter("tableName");⟵ Taint Source
2	String name = request.getParameter("name");⟵ Taint Source
3	String query = "select * from " + tableName + " where name = '" + name + "'"
4	stmt = con.createStatement(query);⟵ Taint Sink
5	rs = stmt.executeQuery();

번호	소스 코드
1	String tableName = request.getParameter("tableName");⟵ Taint Source
2	String name = request.getParameter("name");⟵ Taint Source
3	String query = "select * from " + tableName + " where name = '" + name + "'"
	Some Filtering Code
4	stmt = con.createStatement(query);⟵ Taint Sink
5	rs = stmt.executeQuery();

그림 1. SQL Injection 공격에 취약한 코드 예시
Fig 1. Code examples vulnerable to SQL Injection Attack

오염 분석은 이렇게 테인트 소스에서 시작하여 테인트 싱크로 빠져나가는 데이터의 흐름을 분석하여 소스코드의 보안 취약점 존재 여부를 판별할 수 있다. 데이터의 흐름을 분석하기 위하여 그래프 데이터를 이용하고 그래프 탐색을 하는 알고리즘에 따라 분석에 걸리는 시간과 정확도가 달라지며 기본적으로 반비례 관계를 갖는다.

본 논문에서는 기존의 그래프 탐색 방식이 아닌 기계학습을 활용한 새로운 보안 취약점 탐지 방법을 제시한다. 소스코드에서 그래프 데이터와 소스코드의 보안 취약점 유무를 판별하는 값(label)을 추출하고 이를 graph2vec[4]을 통해 임베딩 시킨 후 결과로 받은 벡터 데이터를 학습 데이터로 하여 기계학습을 통한 모델을 생성하고 평가한다.

2. 관련 연구

2.1 정적 프로그램 분석

프로그램 분석 방법은 크게 동적 프로그램 분석과 정적 프로그램 분석, 이렇게 두 가지로 나뉜다. 이 중 정적 프로그램 분석은 프로그램을 컴파일하고 실행하기 전에, 즉 실제로 실행하지 않고 프로그램을 분석하는 기법으로 당장 실행할 수 있는 애플리케이션뿐만 아니라 개발 초기의 프로그램에도 사용 가능한 기법이다.

정적 프로그램 분석은 소스코드의 전체적인 흐름과 문법 구조를 이해하고 찾고자 하는 특징을 찾아내어 분석하는 기법으로 특히 보안 취약점을 탐지하는 데 많이 사용되고 있다. 사용되지 않는 변수, 파라미터, 메소드나 불필요한 조건문, 중복 코드 등의 소스코드의 결함을 탐지할 수 있는 PMD[5], 자동으로 코드를 검사하여 프로그램 코딩 표준 컨벤션의 규칙에 어긋나는 코드를 알려주는 CheckStyle[6], 코드 클론을 검사하는 CPD[7] 등의 널리 알려진 정적 프로그램 분석 도구들이 존재하며, 최근에는 정적 프로그램 분

석에 기계학습을 활용하는 연구들이 많이 시행되고 있다.

2.2 그래프 임베딩

그래프 임베딩은 그래프 데이터를 벡터 또는 벡터의 집합으로 변환해주는 것을 말한다. 임베딩의 결과인 벡터 데이터는 그래프의 구조, 노드 간의 관계, 서브 그래프의 특징 등의 정보들을 나타낼 수 있어야 한다. 다양한 그래프 데이터의 성질들을 잘 파악할 수 있도록 하는 여러 가지 임베딩 기법들이 계속 연구되고 있다.

그래프의 한 노드에서 시작해서 임의의 이웃 노드로 정해진 횟수만큼 이동하면서 방문한 노드들을 나열한 랜덤 워크(random walk)를 사용하는 임베딩 기법[8], 노드를 방문할 확률을 도입하여 랜덤 워크를 생성하는 방법을 발전시켜 임베딩을 수행하는 기법[9], 랜덤 워크를 사용하지 않고 그래프 상에서 이웃 관계를 기반으로 노드와 노드 사이의 거리 정보를 학습함으로써 임베딩을 수행하는 기법[10] 등의 그래프의 어떤 정보를 중요한 정보로 판단하는지에 따라 다양한 연구가 진행되고 있다.

2.3 그래프 탐색

그래프 검색, 그래프 순회라고도 불리는 그래프 탐색은 그래프 데이터를 다루는 연산 중 가장 기본이 되는 연산으로 널리 알려진 깊이 우선 탐색(Depth First Search, DFS), 너비 우선 탐색(Breadth First Search, BFS)를 포함하여 다양한 알고리즘과 기법들이 존재한다. 특정 노드에서 다른 특정 노드로 가는 경로가 존재하는지, 그러한 경로 중 가장 빠른 경로는 어떤 경로인지, 경로 중 특정한 노드를 지나가는 경로는 어떤 경로인지, 그래프 상에 순환이 존재하는지 등의 그래프의 어떠한 특징을 찾아내는 것을 목적으로 사용되며 어떠한 특징을 찾아내고 싶을 때 그 특징을 찾아내기 위해 어떤 알고리즘 또는 기법을 사용하는지에 따라 성능과 실행 시의 비용은 천차만별이다. 이에 따라 다양한 그래프 탐색 알고리즘과 기법들이 연구되고 있을 뿐만 아니라, 특정 용도의 프로그램에 어떻게 적용하는가에 관한 연구[11] 또한 진행되고 있다.

3. 기계학습을 활용한 프로그램 분석 모델

이 절에서는 본 논문에서 기계학습 모델을 생성하기 위해 수행한 과정을 나타내는 전체적인 구상을 설명한다. 학습 데이터의 가장 기본이 되는 프로그램 소스코드에서 시작하여 먼저 기계학습을 하는 데 필요한 충분한 데이터의 확보를 위하여 돌연변이(mutation) 소스코드들을 생성한다. 이후 충분한 양의 데이터를 확보한 뒤 소스코드를 그래프 데이터로 변환하고, 그래프 임베딩을 통해 추출된 벡터 데이터를 학습시켜 모델을 생성한다. 그림 2는 이를 표현한 구상도이며, 각 데이터 변환 및 학습 과정에서 수행한 내용은 하나씩 차례대로 설명하도록 한다.

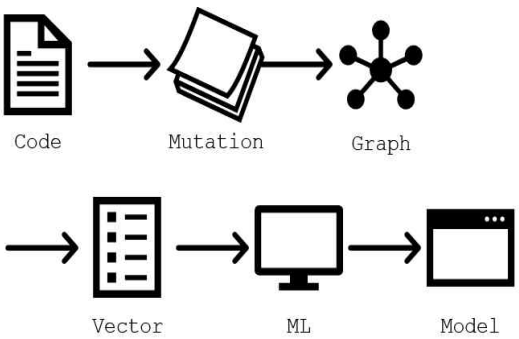

그림 2. 구상도
Fig 2. Schematic Diagram

3.1 학습 데이터의 특징

본 논문에서는 소스코드가 아닌 흐름 그래프

데이터를 통해 기계학습을 활용하기 때문에 소스코드를 흐름 그래프 데이터로 변환하는 데이터 전처리 과정이 필요하다. 하지만 실제 프로젝트의 소스코드의 경우 변환하는 과정이 복잡하고 많은 시간이 필요하다. 이에 본 연구에서는 그림 3과 같은 단순화한 데이터를 사용한다. 단순화한 학습 데이터의 특징은 다음과 같다.

- 모든 테인트 소스는 source()로 표현
- 모든 테인트 싱크는 sink()로 표현
- 모든 필터링 코드는 함수 filter()로 표현

번호	소스 코드
1	x := source();
2	sink (x)

Vulnerable Code

번호	소스 코드
1	x := source();
2	x := filter(x);
3	sink (x)

Invulnerable Code

그림 3. 단순화한 학습 데이터 예시
Fig 3. Examples of Simplified Learning Data

3.2 학습 데이터 확보

기계학습을 위한 필수적인 요소 중 하나는 학습 데이터를 확보하는 것이다. 3.1절에서 설명한 단순화한 학습 데이터로 기계학습을 수행할 정도의 많은 양의 학습 데이터를 순수하게 확보하는 것은 불가능에 가깝다. 이에 본 논문의 연구에서는 기본적인 학습 데이터에서 다음과 같은 간단한 돌연변이 변환을 통해 기계학습을 수행할 수 있는 학습 데이터를 확보하였다.

- 표현식의 변수 변경
- 표현식에 연산 및 변수 추가
- 표현식의 연산을 다른 연산으로 변경

이러한 변환을 거쳐 생성한 돌연변이 데이터를 포함하여 본 논문에서는 총 8198개의 소스코드 데이터를 확보하여 기계학습을 수행하였다. 그림 4는 이러한 수정을 거쳐 만든 돌연변이 데이터의 예시이다.

번호	소스 코드
1	x := source();
2	y := filter(x + 1);
3	z := 1;
4	sink (y)

번호	소스 코드
1	x := source();
2	y := filter(x + 1);
3	z := 1 + y;
4	sink (y + y)

번호	소스 코드
1	x := source();
2	y := filter(x + 1);
3	z := 1 * y;
4	sink (z / y)

그림 4. 돌연변이 소스코드 예시
Fig 4. Example of Mutation Source-code

3.3 그래프 데이터로 변환

소스코드 학습 데이터를 흐름 그래프 데이터로 변환하기 위한 모듈을 설계하였다. 흐름 그래프는 소스코드의 파싱을 어떤 기준으로 하는지에 따라 제어 흐름 그래프(control flow graph)와 데이터 흐름 그래프(data flow graph)로 나뉜다. 제어 흐름 그래프의 경우 문장(statement)을 기준으로 소스코드의 흐름을 나타내며, 데이터 흐름 그래프의 경우 표현식(expression)을 기준으로 소스코드의 흐름을 나타낸다. 본 논문의 연구에서는 이 두 가지를 모두 변환하는 모듈을 설계하였다. 그림 5는 소스코드 상에서 제어 흐름과 데이터 흐름을 화살표로 표현하고, 이를 실제 연구에서 사용한 모듈을 통해 그래프 데이터로 변환한 결과를 그림으로 출력한 것이다.

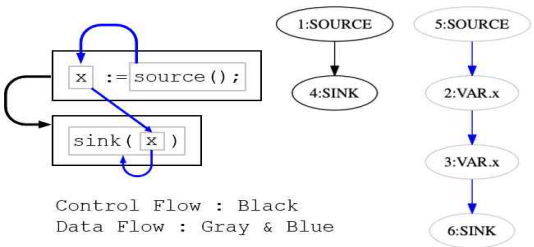

그림 5. 제어 흐름과 데이터 흐름 예시
Fig 5. Examples of Control Flow & Data Flow

3.4 그래프 임베딩을 통한 벡터화

graph2vec은 기존의 word2vec[12]과 doc2vec[13]의 방법에 착안하여 그래프에서 서브 그래프를 추출하고 서브 그래프들을 임베딩 시켜 비슷한 그래프들을 좌표상 비슷한 위치에 벡터화 하는 기법이다. 즉 그래프 분류 과제를 수행하기 위하여 개발된 모델로 본 연구의 주제인 그래프 상에서 보안 취약점의 존재 여부 또한 결국 그래프 분류 문제이므로 해당 모델을 사용하는 것이 적절하다고 판단하였다.

3.3절의 모듈을 통해 결과로 받은 흐름 그래프 데이터를 graph2vec의 입력 데이터로 주기 위해 전처리 과정이 필요하다. 그림 6은 그림 5의 흐름 그래프 데이터를 graph2vec의 입력 데이터 형식에 맞게 변환한 것이다. 모든 이음선 정보는 배열로, 각 노드의 번호와 특징(feature)값은 딕셔너리로 나타낸다.

번호	소스 코드
1	{"edges": [[1, 4], [2, 3], [3, 6], [5, 2]], "features": {"1": "SOURCE", "4": "SINK", "2": "VAR.x", "3": "VAR.x", "5": "SOURCE", "6": "SINK"}}

그림 6. graph2vec 입력 데이터 형식
Fig 6. Input Data Format of graph2vec

이렇게 모든 흐름 그래프 데이터를 변환하여 graph2vec의 입력 데이터로 넣어주면 임베딩된 벡터 데이터를 추출하여 결과로 내어준다. 벡터 데이터의 경우 하나의 흐름 그래프 데이터마다 1차원 배열로 형성되어있으며 몇 개의 원소로, 즉 몇 차원으로 임베딩 할지는 graph2vec 실행 시 인자 값을 함께 넘겨주어 설정할 수 있다. 본 논문의 연구에서는 128개의 값을 가지는 1차원 배열의 벡터 데이터를 추출하였다.

3.5 기계학습

기계학습의 학습 방법은 크게 지도학습(supervised learning), 비지도학습(unsupervised learning), 강화학습(reinforcement learning) 이렇게 3가지로 분류된다. 지도학습의 경우 학습 데이터의 정답(label)을 알려주며 학습을 시키는 것이며, 반대로 비지도학습의 경우 정답을 따로 알려주지 않고 비슷한 데이터들을 군집화하는 학습 방법이다. 강화학습의 경우 상과 벌의 보상(reward) 값을 주며 상을 최대화하고 벌을 최소화하도록 학습해나가는 학습 방식이다. 본 논문의 연구에서는 모델 평가를 위해 정답을 잘 맞혔는지 확인하기 용이하도록 지도학습 방식을 채택하였다.

또한, 기계학습 모델은 선형 회귀(linear regression), 비선형 회귀(nonlinear regression), 딥 러닝(deep learning) 등 다양한 모델들이 존재한다. 다양한 모델 중 SVM(Support Vector Machine)[14]은 분류 과제에 주로 사용하는 지도학습 모델이다. SVM은 결정 경계(decision boundary)를 정의하는 모델로, 주어진 데이터 점들이 두 가지 클래스에 속해있다고 가정했을 때 이 두 클래스 사이의 적절한 위치에 결정 경계를 긋고 새로운 데이터 점이 어떤 클래스에 속하는지 결정하는 것이다. 이러한 SVM의 특징은 보안 취약점이 존재하는 데이터와 존재하지 않는 데이터의 군집을 나눠 새로운 데이터에 대한 보안 취약점의 존재 여부를 판단하기에 적절하다. 이에 본 논문의 연구에서는 기계학습의 모델로 SVM을 채택하고 3.4절의 graph2vec의 결과로 받은 벡터 데이터를 통해 기계학습을 수행하였다.

4. 모델 평가

3.3절에서 설명했듯이 그래프 데이터는 제어 흐름 그래프와 데이터 흐름 그래프를 모두 생성하였다. 제어 흐름 그래프의 경우 확실한 보안 취약점 탐색은 불가능하지만, 소스에서 싱크로

향하는 경로는 존재하는 반면 모든 해당 경로에서 필터가 존재하지 않는 경우 한정적으로 판별할 수 있다. 이에 제어 흐름 그래프와 데이터 흐름 그래프가 각각 보안 취약점 탐지에 어느 정도의 기여를 보이는지 확인하기 위해 제어 흐름 그래프만을 학습시킨 경우, 데이터 흐름 그래프만을 학습시킨 경우, 그리고 두 가지의 그래프 모두 학습시킨 경우의 모델을 생성하여 모델 평가를 수행하였다.

기계학습 수행 시 3.2절에서 확보한 총 8198개의 소스코드 데이터의 70%를 학습 데이터로 활용하고, 나머지 30%의 데이터를 모델 평가를 위한 테스트 데이터로 활용하였다. 그림 7은 학습 데이터로 제어 흐름 그래프와 데이터 흐름 그래프만을 사용한 경우와 두 흐름 그래프 모두를 사용한 경우를 SVM의 커널을 달리하여 학습 데이터로 평가한 결과이다. SVM의 커널은 선형(linear), 다항식(polynomial), 방사 기저 함수(RBF:Radial Bias Function)이 있으며 선형의 경우 2차원의 데이터 점들을 가지고 결정 경계를 정의하며, 다항식은 3차원, 그리고 RBF의 경우 무한한 차원에서 결정 경계를 정의한다.

Kernel	Accuracy
Linear	0.6715447154471544
Poly	0.6626016260162602
RBF	0.5154471544715448

CFG

Kernel	Accuracy
Linear	0.9117886178861788
Poly	0.9719512195121951
RBF	0.9752032520325203

DFG

Kernel	Accuracy
Linear	0.9536585365853658
Poly	0.9869918699186991
RBF	0.9890243902439024

CFG+DFG

그림 7. 모델 평가 결과
Fig 7. Model Evaluation Results

제어 흐름 그래프만을 학습 데이터로 한 모델의 경우 앞서 설명했듯이 제어 흐름 그래프만으로는 정확한 판별이 불가능하여서 만족할 만한 정확도를 얻을 수 없었지만, 데이터 흐름 그래프를 학습시킨 모델과 두 가지 그래프 모두를 학습시킨 모델의 경우 약 97~99%의 정확도를 얻은 것을 확인할 수 있다. 또한, 데이터 흐름 그래프만을 학습시킨 모델보다 두 가지 그래프 모두를 학습시킨 모델의 평가가 좋은 것으로 보아 제어 흐름 그래프만으로는 제대로 된 학습이 불가능하지만, 제어 흐름 그래프도 소스코드의 보안 취약점을 탐지하는 데에 어느 정도 기여한다는 것을 확인할 수 있다.

또한, 그림 8은 그림 7에서 가장 높은 정확도를 보인 제어 흐름 그래프와 데이터 흐름 그래프를 모두 학습 데이터로 사용한 RBF커널의 SVM 모델이 탐지한 결과이다. 실제로 취약한 코드를 안전한 코드로 예측한 미탐(false negative)이 18개, 실제로 안전한 코드를 취약한 코드로 예측한 오탐(false positive)이 9개로 나머지 테스트 데이터에 대해서는 옳은 답을 예측해냈다. 이를 바탕으로 모델의 정밀도(precision)와 재현율(recall)을 계산하면, 각각 약 99.2%, 98.5%가 된다.

		모델 예측 결과	
		Vulnerable	Invulnerable
실제	Vulnerable	1174	18
	Invulnerable	9	1259

그림 8. 정확도가 가장 높은 모델 평가
Fig 8. Evaluation of Model with Highest Accuracy

그림 9는 각각 모델 평가 시 오탐의 결과를 얻은 소스코드, 그리고 해당 소스코드와 유사한 정탐(true positive)의 결과를 얻은 소스코드의 예시이다. 우측 소스코드의 경우 필터링이 되지 않은 11번 라인으로 인해 취약한 소스코드가 맞지만, 좌측의 소스코드의 경우 안전한 코드임에도 불구하고 취약하다고 예측되었다.

번호	소스 코드
1	x := source();
2	y := source();
3	z := source();
4	if x < 5 then
5	a := filter(z)
6	else
7	if y > 3 then
8	a := filter(z + 1)
9	else
10	if z < 10 then
11	a := filter(x)
12	else
13	a := filter(y)
14	fi
15	fi
16	fi;
17	sink(a * a)

False Positive

번호	소스 코드
1	x := source();
2	y := source();
3	z := source();
4	if x < 5 then
5	a := filter(z)
6	else
7	if y > 3 then
8	a := filter(z + 1)
9	else
10	if z < 10 then
11	a := x
12	else
13	a := filter(y)
14	fi
15	fi
16	fi;
17	sink(a + a)

True Positive

그림 9. 오탐 소스코드(좌), 정탐 소스코드(우)
Fig 9. False Positive Source-code(left),
True Positive Source-code(right)

마찬가지로 그림 10은 모델 평가 시 미탐의 결과를 얻은 소스코드, 그리고 해당 소스코드와 유사한 정탐(true negative)의 결과를 얻은 소스코드의 예시이다. 우측 소스코드의 경우 sink()로 내보내는 값이 필터링이 된 데이터인 a로 안전한 코드가 맞지만, 좌측의 소스코드의 경우 필터링이 되지 않은 z를 sink()로 내보내기 때문에 취약한 코드임에도 불구하고 안전하다고 예측되었다. 두 가지 예시 모두 기계학습 모델의 예측값으로 정확한 원인을 파악하는 것은 어렵지만, 학습 데이터를 확보하기 위한 돌연변이 데이터 생성의 여파로 유사한 소스코드의 그래프 데이터를 학습함으로써 나타난 현상으로 예측된다. 이는 향후 연구를 통해 단순화한 소스코드가 아닌 실전 프로젝트 소스코드를 분석한 결과를 관찰하면 원인을 파악할 수 있는 실마리를 찾을 수 있으리라 기대한다.

번호	소스 코드
1	x := source();
2	y := source();
3	z := source();
4	while z > 0 do
5	if z < 10 then
6	a := filter(x + y)
7	else
8	a := filter(x + z)
9	fi
10	done;
11	sink(z)

False Negative

번호	소스 코드
1	x := source();
2	y := source();
3	z := source();
4	while z > 0 do
5	if z < 10 then
6	a := filter(x + y)
7	else
8	a := filter(y + z)
9	fi
10	done;
11	sink(a)

True Negative

그림 10. 미탐 소스코드(좌), 정탐 소스코드(우)
Fig 10. False Negative Source-code(left),
True Negative Source-code(right)

5. 결 론

충분한 양의 소스코드 데이터가 확보되어 있다면, 그래프 생성 모듈과 기계학습을 활용하여 오염 분석으로 탐지할 수 있는 소스코드의 보안 취약점을 탐지할 수 있다는 가능성을 확인하였다. 기계학습을 통한 모델을 생성하여 이를 통해 적은 시간 비용을 가지고도 높은 정확도의 보안 취약점 탐지가 가능함을 확인하였기 때문에 향후 연구를 통해 실제 프로젝트 소스코드에도 이를 적용할 수 있다면 기존의 프로그램 분석 도구들을 대체할 가능성 또한 존재한다는 것을 알 수 있다.

본 논문에서는 3.1절에서 설명하였듯, 단순화한 소스코드 데이터를 활용하기 때문에 그에 맞는 AST를 통해 분석하는 모듈을 사용하였다. 이를 실제 프로젝트의 소스코드에도 호환될 수 있도록 AST를 확장하고, 기계학습을 수행할 수 있을 정도의 충분한 데이터를 확보하여 그래프 데이터로 변환한다면 모델을 생성하여 실제 프로젝트의 소스코드에서도 오염 분석을 통해 보안 취

약점을 탐지해내는 것을 기계학습 모델이 대신하는 것이 가능할 것이다.

"본 연구는 과학기술정보통신부 및 정보통신기획평가원의 SW중심대학지원사업의 연구결과로 수행되었음"(2018-0-00192)

참 고 문 헌

[1] T. Moon and H. Kim. Detecting Security Vulnerabilities in TypeScript Code with Static Taint Analysis. Journal of the Korea Institute of Information Security & Cryptology, 31(2), 263 - 277. (2021). DOI : 10.13089/JKIISC.2021.31.2.263.

[2] Avancini, Andrea & Ceccato, Mariano. Towards security testing with taint analysis and genetic algorithms. Proceedings - International Conference on Software Engineering, 65-71. (2010). DOI : 10.1145/1809100.1809110.

[3] S. Oh, T. Kim and H. Kim. Technology Analysis on Automatic Detection and Defense of SW Vulnerabilities. Journal of the Korea Academia-Industrial cooperation Society, 18(11), 94-103. (2017). DOI : 10.5762/KAIS.2017.18.11.94

[4] Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., & Jaiswal, S. graph2vec: Learning distributed representations of graphs. (2017). arXiv preprint arXiv:1707.05005.

[5] PMD (Programming Mistake Detector), 2022.10.11., https://pmd.github.io/

[6] CheckStyle, 2022.10.11., https://checkstyle.sourceforge.io/

[7] CPD(Copy/Paste Detector), 2022.10.11., https://pmd.github.io/latest/pmd_userdocs_cpd.html

[8] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: online learning of social representations. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 701 - 710. (2014). DOI : 10.1145/2623330.2623732

[9] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 855 - 864. (2016). DOI : 10.1145/2939672.2939754

[10] D. Wang, P. Cui and W. Zhu. Structural Deep Network Embedding. Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 1225 - 1234. (2016). DOI : 10.1145/2939672.2939753

[11] Qin, Yuehua & Cao, Han & Xue, Leyi. Research and Application of Knowledge Graph in Teaching: Take the database course as an example. Journal of Physics: Conference Series, 1607(1):012127. (2020). DOI : 10.1088/1742-6596/1607/1/012127.

[12] Mikolov, Tomas & Chen, Kai & Corrado, G.s & Dean, Jeffrey. Efficient Estimation of Word Representations in Vector Space. (2013). arXiv preprint arXiv:1301.3781.

[13] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. Proceedings of the International Conference on International Conference on Machine Learning, 32(2), 1188-1196. (2014). Available from https://proceedings.mlr.press/v32/le14.html.

[14] Evgeniou, Theodoros & Pontil, Massimiliano. Support Vector Machines: Theory and Applications. 2049. 249-257. (2001). DOI : 10.1007/3-540-44673-7_12.

저 자 소 개

모지환(Ji-Hwan Mo)

2021 한양대학교 ERICA 소프트웨어학부 학사
2021-현재 : 한양대학교 대학원 컴퓨터공학과 석사 과정
<주관심분야> 프로그래밍언어, 프로그램분석, ML4PL

홍성문(Sung Moon Hong)

2012 위덕대학교 컴퓨터공학과 학사
2014 한양대학교 컴퓨터공학과 석사
2014-현재 : 한양대학교 대학원 컴퓨터공학과 박사과정
<주관심분야> 프로그래밍언어, 프로그램분석, 소프트웨어 보안

도경구(Kyung-goo Doh)

1980 한양대학교 산업공학과 학사
1987 아이오와주립대학 컴퓨터과학 석사
1992 캔자스주립대학 컴퓨터과학 박사
1993-1995 일본 아이주 대학 교수
1995-현재 : 한양대학교 ERICA 소프트웨어학부 교수
<주관심분야> 프로그래밍언어, 프로그램분석, 소프트웨어 보안, 소프트웨어 공학