

SW 취약점 탐지에서 패치 기반 데이터셋의 함수 라벨링 한계와 트레이스 단위 데이터셋 구축 및 평가 (Function-Labeling Limitations of Patch-Based Datasets in Software Vulnerability Detection and Trace-Level Dataset Construction and Evaluation)

전세옥*, 최광훈**

(Se-Ok JEON, Kwanghoon Choi)

요약

딥러닝 기반 취약점 탐지 모델에서 성능 재현 실패가 보고되면서, 모델 구조뿐 아니라 데이터셋 구성 방식도 검토할 필요가 커졌다. 본 연구가 주목한 문제는 취약점 패치 기반 데이터셋의 함수 단위 라벨링이다. 함수 전체에 라벨을 부여하면 취약 흐름과 무관한 정상 코드가 같은 샘플에 포함될 수 있다. 이 경우 모델은 실제 취약 원인보다 함수 주변의 코드 특성에 의존해 학습할 가능성이 있다. 이에 본 연구는 함수 단위가 아니라 트레이스를 활용한 데이터셋을 제안한다. 트레이스는 입력 유입 지점에서 시작해 중간 전달 문장을 거쳐 보안 상 위험한 연산 지점에 도달하는 문장 리스트이다. 트레이스 단위 데이터셋의 적용 가능성을 확인하기 위해 딥러닝 모델과 LLM-RAG 기반으로 함수 단위 데이터셋과 트레이스 단위 데이터셋의 취약점 탐지 성능을 비교 평가하였다. 함수 단위 데이터셋인 Extended RealVul-FuncPair에서는 최고 F1 스코어가 딥러닝 0.67, LLM 0.64였다. 트레이스 단위 데이터셋인 SARD-Juliet TracePair에서는 최고 F1이 딥러닝 1.00, LLM 0.96으로 높게 나타났다. 다만 서로 다른 원본 데이터에 기반하므로, 이 차이를 트레이스 표현의 효과만으로 해석하기 어렵다. 10개 CVE 대상 TracePair에서는 딥러닝 0.56, LLM 0.67이었다. 이는 CVE 조건에서 Trace 추출 품질, 트레이스 길이가 성능을 제한할 가능성이 있음을 시사한다.

■ 중심어 : 취약점 탐지 ; 딥러닝 ; 패치 기반 데이터셋 ; LLM-RAG

Abstract

Performance reproduction failures in deep learning-based vulnerability detection highlight the need to examine dataset construction as well as model architecture, particularly function-level labeling in vulnerability-patch datasets. When a label is assigned to an entire function, non-vulnerable code unrelated to the vulnerable flow can be included in the same sample, causing the model to rely on surrounding code characteristics rather than the actual vulnerability cause. Accordingly, this study proposes a Trace-level dataset, where a Trace is a list of statements from the Source through Step statements to a security-sensitive Sink. To examine the applicability of the Trace-level dataset, we compared vulnerability detection performance with a function-level dataset using deep learning models and an LLM-RAG procedure. In the function-level dataset, Extended RealVul-FuncPair, the best F1-score was 0.67 for deep learning and 0.64 for LLM. In the Trace-level dataset, SARD-Juliet TracePair, the best F1-score was 1.00 for deep learning and 0.96 for LLM. However, because RQ1 and RQ2a use different source datasets, this difference cannot be attributed solely to the Trace representation. For TracePair targeting 10 CVEs, the best F1-scores were 0.56 for deep learning and 0.67 for LLM, suggesting that trace extraction quality and trace length may limit performance under real-world CVE conditions.

■ keywords : Vulnerability Detection ; Deep Learning ; Patch-based Dataset ; LLM-RAG

I. 서론

오픈소스에서는 코드가 여러 프로젝트에 복사되

* 준회원, 전남대학교 정보보호융합학과 석사과정

** 정회원, 전남대학교 정보보호융합학과 교수

이 논문은 2024년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구결과임
(RS-2024-00398993, 전기자동차 충전기 보안위협 대응 기술 개발)

이 논문은 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원-정보보호핵심원천기술개발사업의 지원을 받아 수행된 연구임
(RS-2025-25394739)

본 연구는 과학기술정보통신부 및 정보통신기획평가원의 인공지능융합혁신인재양성사업 연구 결과로 수행되었음
(IITP-RS-2023-00256629)

본 연구는 한국인터넷진흥원(KISA)-정보보안 특성화대학 지원사업의 지원을 받아 수행된 연구임

접수일자 : 2026년 07월 10일

게재확정일 : 2026년 08월 20일

수정일자 : 2026년 08월 09일

교신저자 : 전세옥, 최광훈 e-mail : sojeon@jnu.ac.kr

어 재사용된다. 이 과정에서 취약 코드도 함께 전파될 수 있으나, 원본 프로젝트의 패치가 파생 프로젝트에 자동으로 반영되지는 않는다. 전기차 충전 인프라 관련 오픈소스 사례에서도 유사한 취약 코드 전파 양상이 확인된다 [1]. zlib 1.2.5의 취약 코드는 libwebsockets 4.3.2에 복사되고 다시 open-ocpp에 포함되었으나, 원본 zlib의 1.2.9 패치는 파생 프로젝트에 반영되지 않았다. 따라서 패치가 누락된 취약점에 대한 탐지가 필요하다.

취약점 탐지 연구는 구문 기반 비교, 머신러닝 기반, 딥러닝 기반, LLM 기반 방법으로 전개되어 왔다. 구문 기반 기법인 VUDDY는 함수 코드를 토큰 리스트로 변환한 뒤 유사도를 비교한다 [2]. 머신러닝 기반 기법인 Tracer는 입력 지점에서 위험 지점까지 이어지는 트레이스에서 연산자, 위험 API, 경계값 조건의 출현 빈도를 특징 벡터로 사용한다 [3]. 딥러닝 기반 기법인 DeepWukong [4], LineVul [5], PDBERT [6]는 취약 코드 기반 데이터셋과 그래프 신경망(Graph Neural Network, GNN) 또는 CodeBERT 계열 모델을 사용한다. LLM 기반 기법

은 프롬프트, 검색 증강 생성(Retrieval-Augmented Generation, RAG), 파인튜닝을 활용한다 [7]. RAG는 유사 예시를 검색해 프롬프트에 함께 제공하는 절차를 뜻한다. 본 논문은 이 중 딥러닝 및 LLM 기반 방법에 초점을 둔다.

기존 딥러닝 기반 취약점 탐지 모델에서는 새로운 데이터셋에서 성능이 재현되지 않는 사례가 보고되었다. 그림 1에서 DeepWukong(2021) [4]과 LineVul(2022) [5]은 본인들이 사용한 벤치마크에서는 F1-score가 0.9로 높은 성능을 보고하였으나 2024년 이후 RealVul과 같이 QEMU 등을 포함한 새로운 데이터셋에서 그 모델들을 실험한 결과 F1-score가 0.02 수준으로 성능이 낮은 결과가 보고되었다 [8]. 이 결과는 성능 재현 실패의 원인을 모델 구조뿐 아니라 데이터셋 구성 방식에서도 검토해야 함을 시사한다.

성능 재현 실패의 가능한 원인 중 하나는 패치 기반 데이터셋의 함수 단위 라벨링 문제이다. 함수 내 패치 차이만으로는 취약과 정상을 구분하기 어렵고, 함수 전체 라벨은 취약 코드와 정상 코드를 함께 포함할 수 있다. 또한 이전에는 정상 라벨을 붙인 코드가 나중에는 취약하다고 시간 흐름에 따라 다르게 판별되어 충돌할 수도 있다. 이러한 조건은 모델이 실제 취약 원인과 무관한 코드 특성까지 함께 학습하게 만들 가능성이 있다.

그림 2의 libgd CVE-2016-8670 패치 전후 코드는 작은 경계값 수정만으로 취약 여부를 판단하기 어렵다는 점을 보인다. 핵심 변경은 11번 라인의

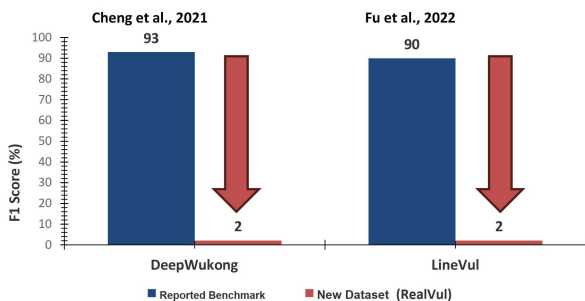


그림 1. 기존 딥러닝 기반 취약점 탐지 성능 재현 한계

```
01 static int dynamicGetbuf(gdIOCtxPtr ctx, void *buf, int len) {
    ...
06 dp = dcts->dp;
07 remain = dp->logicalSize - dp->pos;
08 if(remain >= len) {
09     rlen = len;
10 } else {
11     if(remain == 0) {
12         return 0;
13     }
14     rlen = remain;
15 }
16 memcpy(buf, (void *) ((char *)dp->data + dp->pos), rlen);
    ...
19 }
```

(a) Vulnerable Function Before Patch

```
01 static int dynamicGetbuf(gdIOCtxPtr ctx, void *buf, int len) {
    ...
06 dp = dcts->dp;
07 remain = dp->logicalSize - dp->pos;
08 if(remain >= len) {
09     rlen = len;
10 } else {
11     if(remain <= 0) {
12         return 0;
13     }
14     rlen = remain;
15 }
16 memcpy(buf, (void *) ((char *)dp->data + dp->pos), rlen);
    ...
19 }
```

(b) Non-vulnerable Function After Patch

그림 2. 패치 차이만으로 취약/정상 구분이 어려운 사례

'remain == 0'이 'remain <= 0'으로 바뀐 부분이다. 그러나 취약 여부를 판단하려면 'remain' 값이 'rlen'으로 전달되고 16번 라인의 'memcpy' 복사 크기에 영향을 주는 흐름까지 함께 해석해야 한다.

```

11  <Function-level Sample>
12  static int dynamicGetbuf(gdIOCtxPtr ctx, void *buf, int len) {
13      ...
14      dp = dctx->dp;
15      remain = dp->logicalSize - dp->pos;
16      if(remain >= len) {
17          rlen = len;
18      } else {
19          if(remain == 0) {
20              return 0;
21          }
22          rlen = remain;
23      }
24      memcpy(buf, (void *) ((char *)dp->data + dp->pos), rlen);
25      ...
26  }
  
```

Vulnerable

<Non-Vulnerable Code>

<Vulnerable Code>

그림 3. 취약 함수 내부에 정상 코드가 포함되는 사례

함수 전체에 취약 라벨을 부여하면 그림 3의 예처럼 취약 코드와 정상 코드가 같은 취약 샘플에 포함될 수 있다. 함수 단위 샘플은 실제 취약 흐름뿐 아니라 변수 대입, 분기, 반환문과 같은 주변 정상 로직도 같은 라벨로 포함한다. 이 경우 모델은 취약 점과 직접 관련 없는 코드까지 취약 패턴으로 학습할 가능성이 있다.

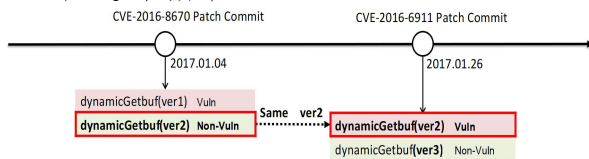


그림 4. 시간 흐름에 따른 정상/취약 라벨 충돌 사례

시간 흐름에 따라 동일 코드가 정상 라벨과 취약 라벨을 모두 받는 사례도 확인된다 [8]. 그림 4에서 'dynamicGetbuf'의 동일 코드는 CVE-2016-8670에서는 패치 후 정상 코드로 수집되었지만, 이후 CVE-2016-6911에서는 패치 전 취약 코드로 다시 수집되었다. 이러한 라벨 충돌은 모델이 동일 코드를 서로 다른 라벨로 학습하게 하여 취약 코드 패턴 학습을 어렵게 만들 가능성이 있다.

이를 위해 함수 전체가 아니라 Source에서 Sink까지 이어지는 트레이스 단위 데이터셋을 사용하여 함수 단위 라벨링의 한계를 완화할 수 있는지 평가한다. 트레이스 표현은 취약 판단에 필요한 흐름을 중심으로 샘플을 구성한다. 이 점에서 함수 전체를 하나의 샘플로 사용하는 방식과 구분된다.

평가 결과, 함수 단위 데이터셋인 Extended RealVul-FuncPair에서는 최고 F1-score가 딥러닝 0.67, LLM 0.64 수준으로 제한되었다. 트레이스 단위 데이터셋인 SARD-Juliet TracePair에서는 최고 F1이 딥러닝 1.00, LLM 0.96으로 높게 나타났다. 그러나 10개 CVE 대상 TracePair에서는 최고 F1이 딥러닝 0.56, LLM 0.67로 나타났다. 다만 RQ1과 RQ2a는 서로 다른 데이터셋을 사용하므로 관찰된 성능 차이를 함수 단위와 트레이스 단위의 차이만으로 해석하기 어렵다.

본 연구의 기여는 세 가지이다. 첫째, 함수 단위 라벨링 한계를 보완하기 위한 트레이스 단위 취약점 탐지 데이터셋을 제안한다. 둘째, 딥러닝 모델과 LLM-RAG를 이용한 트레이스 기반 취약점 재현 탐지 절차를 구현하고 실험한다. 셋째, SARD-Juliet TracePair와 10개 CVE 대상 TracePair를 구축하고, 비교를 위해 Extended RealVul FuncPair를 구성한다.

II. 관련 연구

딥러닝 기반 취약점 탐지 모델의 성능 재현 실패 문제는 본 연구에서만 제기된 것이 아니다 [9, 10]. RealVul 연구는 취약점 탐지 데이터셋의 구성 방식이 라벨 해석에 영향을 줄 수 있음을 지적하였다 [8]. 패치 기반 데이터셋은 일반적으로 패치 전 함수를 취약 샘플로, 패치 후 함수 또는 변경되지 않은 함수를 정상 또는 불확실 샘플로 구성한다. 그러나 시간 순서에 따라 동일 함수가 서로 다른 CVE 패치 관계에 포함되면, 같은 코드가 취약 라벨과 정상 라벨로 다르게 수집될 수 있다. 이러한 라벨 불일치는 함수 단위 라벨이 취약 원인 코드의 의미를 충분히 표현하지 못할 가능성을 시사한다.

Tracer는 트레이스를 활용한다는 점에서 본 연구와 공통점을 가진다 [3]. 그러나 Tracer는 트레이스 내 연산자, 위험 API, 경계 조건의 출현 빈도를 사람이 정의한 특징 벡터로 변환한다. 반면 본 연구는 트레이스를 딥러닝 모델과 LLM 모델의 분류 대상 입력으로 사용한다.

```

01 void CWE195_Signed_to_Unsigned_fgets_memcpy_bad() {
02     int data;
03     data = -1;
04     char inputBuffer[CHAR_ARRAY_SIZE] = "";
05     if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL) {
06         data = atoi(inputBuffer);
07     } else { printf("fgets() failed."); }
08     char source[100]; char dest[100] = "";
09     memset(source, 'A', 100 - 1); source[99] = '\0';
10     if (data < 100) {
11         memcpy(dest, source, data);
12         dest[data] = '\0';
13     }
14     printf(dest);
15 }

```

(a) Function-level Vulnerable Sample

Trace Extraction

	Source	Sink
Step1	if (fgets (inputBuffer , CHAR_ARRAY_SIZE, stdin) != NULL) {	
Step2	data = atoi (inputBuffer);	
	if (data < 100) {	
		memcpy(dest, source, data);

(b) Trace-level Vulnerable Sample

그림 5. 함수 단위 샘플과 트레이스 단위 샘플 비교

III. 트레이스 단위 데이터셋 구성 방법

함수 단위 라벨링 문제를 줄이기 위해 본 논문은 트레이스 단위 데이터셋을 제안한다. 트레이스는 입력 유입 지점(Source)에서 시작해 중간 전달 문장(Step)을 거쳐 보안상 위험한 연산 지점(Sink)에 도달하는 문장 리스트이다. 본 연구의 트레이스는 정적 오염 분석으로 추출한 Source-Sink 데이터 흐름을 나타내며, 함수 전체가 아니라 취약 판단에 필요한 원인과 결과의 흐름을 표현한다.

그림 5는 함수 단위 샘플과 Source, Step, Sink로 구성한 트레이스 단위 샘플의 차이를 보인다. 트레이스 단위 데이터셋은 함수 단위 라벨링의 세 가지 문제에 대응한다. 첫째, 패치 전후의 작은 코드 차이만으로 취약 여부를 판단하기 어려운 경우에도 Source에서 Sink까지 값이 전달되는 원인 - 결과 흐름을 함께 제시하여 취약/정상 구분 근거를 명확히 한다. 둘째, 함수 전체 대신 취약 관련 트레이스만 입력으로 사용하므로 주변 정상 코드가 취약 패턴으로 함께 학습되는 포괄적 라벨 범위 문제를 줄일 수 있다. 셋째, 라벨 판단 기준을 Source-Sink 흐름에 두어 시간 흐름에 따라 동일 코드가 서로 다른 함수 라벨로 수집되는 충돌 가능성을 낮출 수 있다.

본 연구는 함수 간 호출을 포함한 트레이스를 추출하기 위해 Infer 1.2.0의 Pulse 오염 분석 기능을 사용하였다 [11]. 오염 분석에는 분석 대상 소스 코드와 Source/Sink 후보가 필요하다. SARD-Juliet

은 테스트케이스 내 주석을 바탕으로 Source/Sink를 정하였으며, 실제 CVE는 취약점 설명, 패치 전후 코드와 함수 호출 관계를 직접 분석하여 Source/Sink를 선정하였다.

```

01 void CWE195_Signed_to_Unsigned_fgets_memcpy_bad() {
02     int data;
03     data = -1;
04     char inputBuffer[CHAR_ARRAY_SIZE] = "";
05     if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL) {
06         data = atoi(inputBuffer);
07     } else { printf("fgets() failed."); }
08     char source[100]; char dest[100] = "";
09     memset(source, 'A', 100 - 1); source[99] = '\0';
10     if (data < 100) {
11         memcpy(dest, source, data);
12         dest[data] = '\0';
13     }
14     printf(dest);
15 }

```

그림 6. 정적 오염 분석 기반 트레이스 추출 절차

Pulse taint 설정에서는 함수명을 procedure로 지정하여 Source와 Sink를 모델링한다. SARD-Juliet은 Tree-sitter로 선정한 Source/Sink 행의 구문을 분석하여 함수 호출을 후보로 추출하였다. 실제 CVE는 Source/Sink 행의 함수 호출을 직접 확인하여 Pulse 설정에 수동으로 입력하였다. Source는 함수의 반환값과 모든 인자를, Sink는 모든 인자를 분석하도록 설정했으며, JSON 설정 파일은 '--pulse-taint-config' 옵션으로 Infer에 전달하였다. 그림 6과 같이 Infer가 보고한 'bug_trace'에서 Source, 중간 Step과 Sink의 파일 및 행 정보를 읽고 해당 원본 코드 문장을 수집하여 트레이스를 구성하였다. 하나의 'bug_trace' 필드에 복수 경로가

표 1. 연구 질문별 데이터셋, 평가 대상 및 모델

RQ	Sample Unit	Training and RAG Dataset	Description	Test Dataset	Models	
					Deep Learning	LLM
RQ1	Function	Extended RealVul	Function-level dataset extending CVE coverage of RealVul	Extended RealVul-FuncPair	- DeepWukong - LineVul - PDBERT	- grok-4.1-fast
RQ2a	Trace	SARD-Juliet	Trace dataset extracted from SARD-Juliet, a synthetic vulnerability detection benchmark	SARD-Juliet		- gpt-4o
RQ2b		SA Trace		SA TracePair TracePair for 10 CVEs		- claude-sonnet-4.5 - gemini-2.5-pro

반환된 경우에는 경로를 구성하는 노드 수가 가장 많은 경로를 선택하였다. 예시에서는 'fgets'로 입력된 값이 'atoi'와 'data < 100' 조건문을 거쳐 'memcpy'의 복사 길이 인자로 사용되는 경로가 트레이스로 구성된다.

트레이스가 추출되면 SARD-Juliet은 해당 테스트 케이스 함수명에 'bad'가 포함된 트레이스를 취약으로, 'good'이 포함된 트레이스를 정상으로 라벨링하였다. 실제 CVE는 패치 전 코드베이스에서 추출한 트레이스를 취약으로, 패치 후 코드베이스에서 추출한 트레이스를 정상으로 라벨링하였다. SARD-Juliet에서 트레이스가 추출되지 않은 테스트 케이스는 데이터셋 구성 대상에서 제외하였다. 실제 CVE에서 Infer가 Source부터 Sink까지 이어지는 전체 흐름을 찾지 못한 경우, 패치 전후 코드와 함수 호출 관계를 바탕으로 Source-Sink 경로를 수동으로 분석하였다. 이 경로를 구간별로 나누어 각 구간이 Infer에서 추출되는지 확인한 뒤, 확인된 구간을 연결하여 하나의 트레이스로 복원하였다. SARD-Juliet의 Infer/Pulse 설정과 Source/Sink 분류, 트레이스 추출 스크립트는 GitHub 저장소에 공개하였다 [12].

IV. 실험 및 결과

4.1. 연구 질문과 실험 데이터셋

실험은 세 가지 연구 질문을 중심으로 구성하였다. RQ1은 함수 단위 데이터셋이 취약 함수와 패치 후 정상 함수를 구분하기 어렵게 만드는지 확인한다. RQ2a는 트레이스 단위 데이터셋 구축으로

SARD-Juliet 대상으로 취약과 정상 구분을 잘하는지 평가한다. RQ2b는 트레이스 단위 데이터셋으로 학습한 효과가 실제 오픈소스 CVE에서도 적용되는지 확인한다.

연구 질문별 샘플 단위, 학습 또는 RAG 구축 데이터셋, 평가 데이터셋, 평가 모델은 표 1과 같다.

RQ1은 Extended RealVul을 학습 및 RAG 구축에 사용하고, Extended RealVul-FuncPair를 평가에 사용한다. FuncPair는 동일 CVE의 패치 전 취약 함수와 패치 후 정상 함수를 쌍으로 구성한 평가 단위이다. RQ2a와 RQ2b는 SARD-Juliet SA Trace를 학습 및 RAG 구축에 사용하며, 각각 SARD-Juliet SA TracePair와 10개 CVE 대상 TracePair를 평가한다. SARD-Juliet SA TracePair는 같은 테스트 케이스의 'bad' 트레이스와 'good' 트레이스로 구성하고, CVE TracePair는 같은 CVE의 패치 전 취약 트레이스와 패치 후 정상 트레이스를 쌍으로 구성한다.

Extended RealVul은 2002~2019년에 공개된 CVE를 수집한 RealVul을 2025년까지 확장한 함수 단위 데이터셋이다. 전체 220,207개 샘플은 취약 함수 5,701개(2.59%)와 정상 함수 214,506개(97.41%)로 구성된다. CVE 공개 시점을 기준으로 2002~2019년의 146,552개(66.55%)를 학습 데이터셋으로, 2020~2025년의 73,655개(33.45%)를 테스트 데이터셋으로 분할하였다. 정상 함수는 RealVul과 같이 각 프로젝트의 학습 구간과 테스트 구간에서 마지막 CVE 패치 커밋을 기준으로 C/C++ 함수 중 취약 함수를 제외하여 구성하였다 [8]. 공백을 제거한 코드의 MD5를 계산하고, 취약 함수와 해시가 같은 정상

샘플을 제외하였다. Extended RealVul-FuncPair는 모델이 학습한 취약 함수와 이를 수정한 패치 함수를 구분하는지를 평가하는 데이터셋이다. 취약 함수는 Extended RealVul 학습 데이터셋에서 가져오고, 패치 함수는 각 CVE의 패치 커밋에서 별도로 추출하였다. 따라서 Extended RealVul-FuncPair에 포함된 CVE와 프로젝트는 Extended RealVul 학습 데이터셋과 중복되며, 두 데이터셋은 독립적이지 않다. 취약 함수 4,365개와 패치 함수 4,365개로 총 8,730개(4,365쌍)이다.

SARD-Juliet SA Trace와 SARD-Juliet SA TracePair는 취약점 탐지 도구 평가용 합성 벤치마크인 SARD-Juliet에서 Infer로 추출한 트레이스 단위 데이터셋이다 [11]. 트레이스마다 사용자 정의 함수명과 변수명을 정규화하고 코드의 공백을 제거한 뒤 MD5 해시를 계산하였다. 해시가 같은 트레이스는 중복으로 간주하고 하나만 유지하였다. 트레이스 단위 데이터셋을 사용한 모델이 취약 트레이스와 정상 트레이스를 구분하는지 평가하기 위해 취약/정상 트레이스 쌍을 구성할 수 있는 테스트 케이스 40개를 분리하여, 취약 트레이스 40개와 정상 트레이스 40개, 총 80개(40쌍)로 구성된 SARD-Juliet SA TracePair를 만들고, 나머지 트레이스로 SARD-Juliet SA Trace를 구축하였다. SARD-Juliet SA Trace의 전체 875개는 취약 트레이스 486개(55.54%)와 정상 트레이스 389개(44.46%)로 구성된다. 이를 테스트 케이스 단위로 80:20 분할하여 학습 데이터셋 701개와 테스트 데이터셋 174개로 구성하였다. 10개 CVE 대상 TracePair는 소규모 탐색 평가셋으로 취약 트레이스 10개와 정상 트레이스 10개로 총 20개(10쌍)를 구성하였다.

4.2. 평가 절차

딥러닝 기반 평가는 그림 7과 같이 데이터셋을 함수 또는 트레이스 단위로 전처리한 뒤, 전처리된 샘플을 모델에 입력하여 취약 또는 정상으로 예측하도록 학습 및 평가하는 순서로 수행된다.

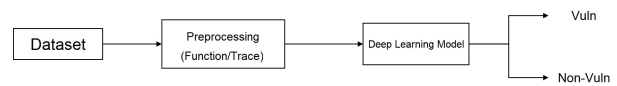


그림 7 딥러닝 기반 취약점 평가 방식

RAG 기반 LLM few-shot 평가는 그림 8과 같이 유사 예시 검색과 프롬프트 분류로 구성된다. RAG는 검사 대상과 유사한 예시를 검색한 뒤, 검색된 예시를 프롬프트에 함께 제공하는 절차이다. 본 연구는 CodeBERT 임베딩을 이용해 취약점 탐지 대상 샘플과 유사한 예시 6개를 검색한다 [13]. 검색된 예시와 탐지 대상 코드를 함께 프롬프트에 포함하고, LLM이 해당 코드를 취약 또는 정상으로 분류하도록 한다. temperature는 Claude를 제외한 모델에서 0.7, Claude에서 Anthropic Messages API 기본값인 1.0을 사용하였다. LLM 생성 seed는 별도로 설정하지 않았다. 각 입력을 모델별로 5회 반복 평가한 뒤 성능 지표의 평균을 보고하였다.

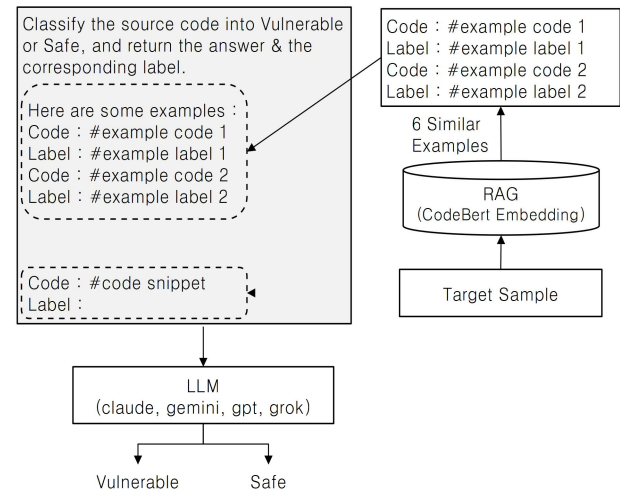


그림 8. RAG 기반 LLM fewshot 취약점 평가 방식

4.3. RQ1: 함수 단위 FuncPair 평가

Extended RealVul-FuncPair의 딥러닝 기반 RQ1 평가에서는 그림 9와 같이 PDBERT가 최고 F1-score 0.67을 보였다. 그러나 일부 모델은 대부분의 샘플을 취약으로 예측했고, 다른 모델은 취약 함수를 거의 탐지하지 못하였다. 이는 낮은 탐지 성능뿐 아니라 모델이 취약 함수와 패치 후 정상 함수를 충분히 구분하지 못하고 한쪽 라벨로 예측이 치우쳤음을 의미한다. Recall은 실제 취약 함수를 얻

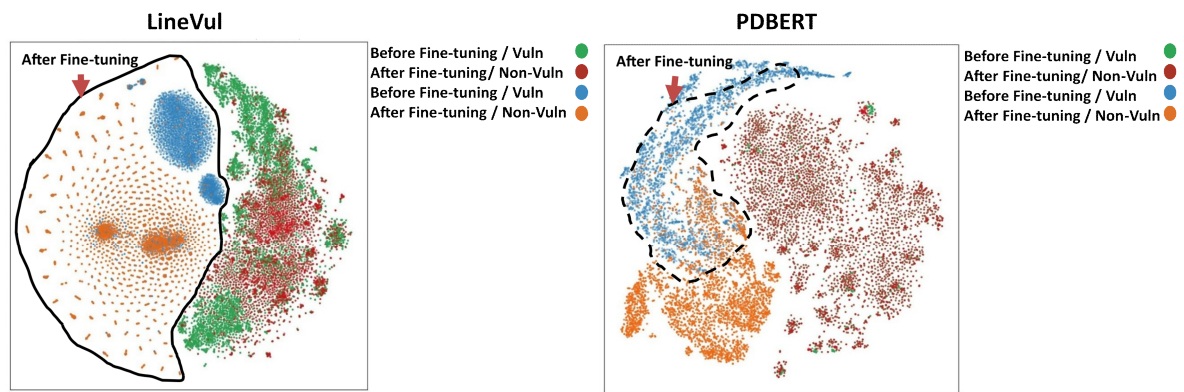


그림 11. RQ1 함수 단위 데이터셋의 t-SNE 분포

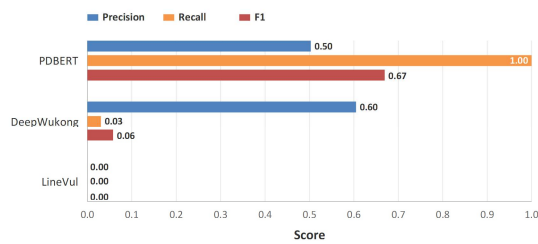


그림 9. Extended RealVul-FuncPair에서 딥러닝 성능

마나 놓치지 않았는지를 나타내고, Precision은 취약으로 예측한 함수가 실제로 취약한지를 나타낸다. 따라서 Recall이 높지만 Precision이 낮으면 취약 함수는 탐지했다라도 패치 후 정상 함수까지 취약으로 함께 분류한 것으로 해석된다. 반대로 Recall이 낮거나 성능이 0에 가까우면 실제 취약 함수를 정상으로 놓친 것으로 해석된다. 두 양상은 함수 단위 FuncPair 입력만으로 취약 함수와 패치 후 정상 함수를 안정적으로 구분하기 어렵다는 점을 시사한다.

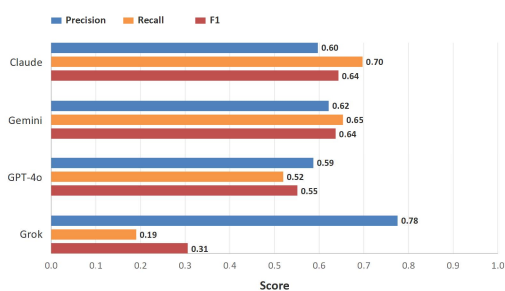


그림 10. Extended RealVul-FuncPair에서 LLM 모델 성능

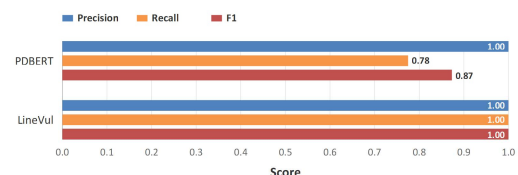
LLM 기반 RQ1 평가에서도 최고 F1-score는 0.64 수준에 머물렀으며, 그림 10에서 보듯이 Precision과 Recall의 불균형이 모델별로 나타났다. 이는 딥러닝 결과와 마찬가지로 함수 단위 입력만으로 취약 함수와 패치 후 정상 함수를 구분하는 데 한계가

있음을 시사한다.

t-분포 확률적 이웃 임베딩(t-distributed Stochastic Neighbor Embedding, t-SNE)는 고차원 임베딩을 2차원으로 축소해 라벨별 분포를 살펴보는 시각화 방법이다. 정량 지표에서 관찰된 경향이 임베딩 분포에서도 나타나는지 살펴보기 위해 Extended RealVul-FuncPair의 파인튜닝 전후 임베딩 벡터를 그림 11과 같이 시각화하였다. LineVul과 PDBERT 모두 파인튜닝 후에도 취약 샘플과 패치 후 정상 샘플의 임베딩이 명확히 분리되지 않고 중첩되어 나타났다. 이러한 중첩 양상은 앞선 Precision, Recall, F1 성능 지표에서 시사하는 바와 일관된다. 다만 t-SNE는 초기화 방식과 하이퍼파라미터에 민감하므로, 본 연구의 시각화 결과만으로 성능 차이를 해석하기는 어렵다.

4.4. RQ2a: SARD-Juliet TracePair 평가

SARD-Juliet TracePair의 딥러닝 기반 RQ2a 평가에서는 그림 12와 같이 트레이스 단위 입력의 최고 F1-score가 1.00으로 나타났다. 이 결과는 SARD-Juliet TracePair에서 트레이스 표현이 취약 사례와 정상 사례를 잘 구분함을 시사한다.

그림 12 SARD-Juliet TracePair에서 딥러닝 모델의 성능
LLM 기반 RQ2a 평가에서도 그림 14와 같이 최고

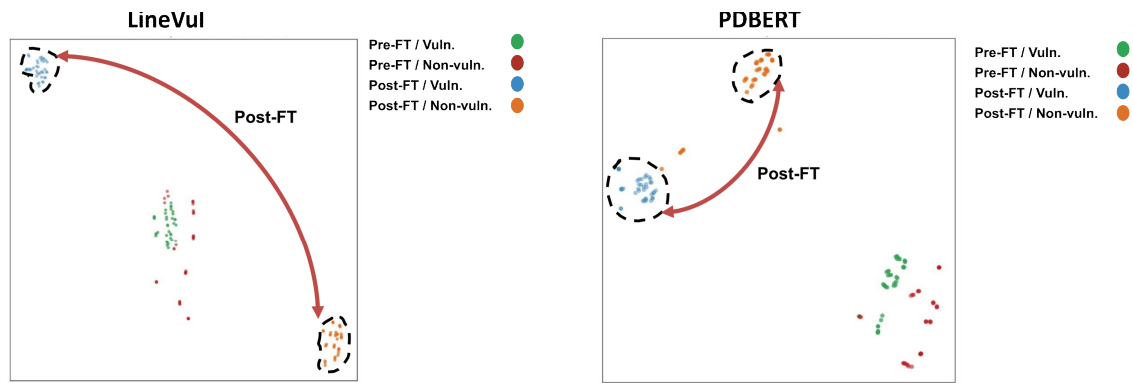


그림 13. RQ2a SARD-Juliet TracePair의 t-SNE 분포

F1-score는 0.96으로 나타났다. 이 결과는 SARD-Juliet 조건에서 LLM 기반 평가에서도 트레이스 단위 입력이 취약 사례와 패치 후 정상 사례를 잘 구분함을 시사한다.

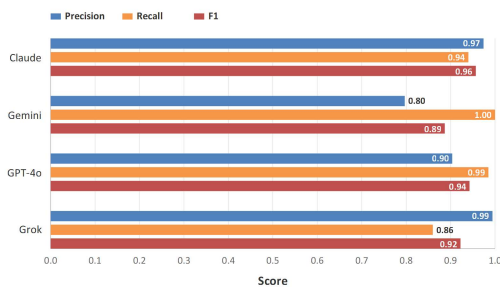


그림 14. SARD-Juliet TracePair에서 LLM 모델의 성능

SARD-Juliet TracePair의 t-SNE 결과에서는 그림 13과 같이 파인튜닝 전 취약 트레이스와 패치 후 정상 트레이스가 섞여 있었으나, 파인튜닝 후 두 라벨의 임베딩이 더 명확히 분리되었다. 이러한 분리 양상은 앞선 Precision, Recall, F1 성능 지표와 시각적으로 일관된다.

다만 RQ1은 실제 오픈소스 기반 Extended RealVul을, RQ2a는 합성 벤치마크 SARD-Juliet을 사용하였기 때문에 관찰된 성능 차이에는 샘플 단위뿐 아니라 벤치마크 난이도와 도메인 차이가 영향을 주었을 가능성이 있다. 따라서 이를 함수 단위와 트레이스 단위의 차이만으로 해석하기 어렵다는 한계가 있다.

4.5. RQ2b: 10개 CVE 대상 TracePair 평가

10개 CVE 대상 TracePair는 SARD-Juliet

TracePair에서 확인한 트레이스 단위 데이터셋의 효과가 실제 오픈소스 CVE 조건에서도 관찰되는지 검토하기 위해 구축하였다. CWE 유형은 2024 CWE Top 25를 우선 검토하고, C/C++ 패치 코드에서 source-sink 대응을 구성할 수 있는지를 함께 고려하였다 [14]. 이 기준에 따라 CWE-78과 CWE-400은 포함하였고, SQL Injection인 CWE-89는 SQL 질의 생성 지점과 DB 호출 지점을 함께 확인해야 하므로 제외하였다. 대신 C/C++ 코드에서 포맷 문자열 전달 흐름을 확인할 수 있는 CWE-134를 추가하였다. 최종 CVE는 공개 설명, 패치 전후 코드, 함수 호출 흐름으로 source, sink, 취약 결과를 확인할 수 있는 사례로 제한하였으며, 표 2와 같이 10개 CVE로 구성하였다. 각 CVE의 설명과 Source/Sink는 GitHub 저장소에 공개하였다 [15].

표 2. 10개 CVE 대상 TracePair 구성

CWE	CVE	Open-Source Software
CWE-134 (Format String Bug)	CVE-2015-8617	PHP (interpreter)
	CVE-2017-12588	rsyslog (log collection)
CWE-400 (Resource Exhaustion)	CVE-2017-11142	PHP (HTTP POST process)
	CVE-2019-12973	OpenJPEG (JPEG codec)
CWE-78 (OS Command Injection)	CVE-2017-15108	spice vdagent (VM guest agent)
	CVE-2017-15924	shadowsocks-libev (encrypted proxy)
	CVE-2018-6791	KDE Plasma Workspace (device manager)
	CVE-2018-16863	ghostscript (document viewer)
	CVE-2019-13638	patch (patch utility)
	CVE-2019-16718	radare2 (binary analysis tool)

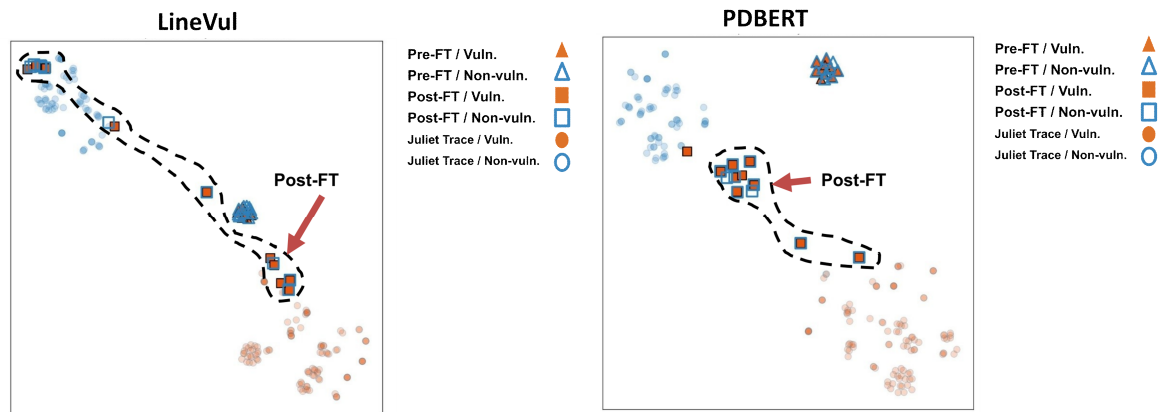


그림 17. RQ2b CVE TracePair의 t-SNE 분포

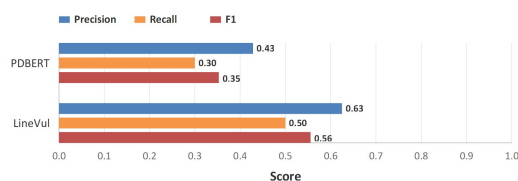


그림 15 10개 CVE 대상 TracePair에서 딥러닝 모델 성능
10개 CVE 대상 TracePair의 딥러닝 기반 RQ2b 평가에서는 SARD-Juliet TracePair에서 확인된 수준의 높은 성능을 보이지 않았다. 그림 15에서 최고 F1-score는 0.56에 머물렀고, Precision과 Recall의 불균형도 나타났다. 이 결과는 딥러닝 모델에서 트레이스 입력만으로는 취약과 패치 후 정상 구분이 제한적임을 시사한다.

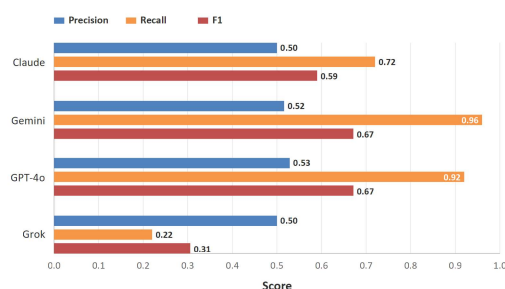


그림 16. 10개 CVE 대상 TracePair에서 LLM 모델의 성능

LLM 기반 RQ2b 평가에서도 최고 F1-score는 GPT-4o와 Gemini의 0.67에 머물렀으며, 그림 16에서 두 모델의 Precision은 0.52에서 0.53 수준이었다. 이는 실제 CVE 조건에서 LLM의 트레이스 기반 취약과 정상 구분도 제한적으로 나타날 수 있음을 시사한다.

그림 17은 10개 CVE 대상 TracePair의 파인튜닝

전후 임베딩 분포를 나타낸다. 파인튜닝 이후 일부 변화는 관찰되었으나, 취약 트레이스와 정상 트레이스의 임베딩은 명확히 분리되지 않고 중첩되었다. 이러한 중첩 양상은 실제 CVE TracePair에서 관찰된 Precision, Recall, F1 결과와 시각적으로 일관된다.

4.6. Infer 활용 트레이스 추출 방법의 한계

Infer로 트레이스를 추출하려면 먼저 분석 대상 프로젝트의 빌드 환경을 구성해야 한다. 실제 오픈소스 프로젝트마다 빌드 방식과 의존성이 달라 이 과정을 일괄적으로 자동화하기 어렵다. 또한 실제 CVE에는 SARD-Juliet의 테스트케이스 주석과 같은 Source/Sink 정보가 제공되지 않는다. 따라서 CVE 설명과 패치 전후 코드를 분석하여 Source/Sink를 수동으로 지정해야 한다. SARD-Juliet에서 확인된 Source/Sink 함수 후보를 데이터베이스로 구축하고 실제 CVE의 후보 선정에 활용하는 방법을 고려할 수 있다. 다만 실제 프로젝트의 다양한 API와 취약점 유형을 반영하려면 후보를 지속해서 확장하고 검증해야 한다.

빌드와 Source/Sink 설정을 완료해도 정적 분석 자체에는 한계가 있다. 전역 변수, 구조체 필드 및 함수 포인터를 경유하는 데이터 흐름은 Infer에서 누락될 수 있다. 전체 Source-Sink 경로가 한 번에 연결되지 않으면 구간별 추출 결과를 수동으로 확인하고 연결하여 복원해야 했다. 이처럼 빌드 환경 구성, Source/Sink 지정 및 분석 결과 복원에 모두

수작업이 필요했기 때문에 본 연구의 실제 CVE 평가 대상을 10개로 한정하였다.

V. 논의

CVE TracePair에서의 성능 제한은 단일 원인으로 설명하기 어렵다. 합성 SARD-Juliet과 실제 CVE 간 도메인 차이, Infer 기반 추출의 한계와 트레이스 길이 차이가 영향을 주었을 가능성이 있다. 표 3에서 SARD-Juliet 트레이스의 평균 길이는 5.92였으나, 10개 CVE 대상 트레이스는 12.50이었다. 다만 실제 CVE 트레이스 구축의 어려움으로 평가 대상을 10개 CVE로 한정하였으므로, 본 결과만으로 세 요인을 성능 차이의 원인으로 확정하기 어렵다.

표 3. SARD-Juliet와 CVE 트레이스의 평균 길이 비교

Dataset	Average Trace Length
SARD-Juliet Trace	5.92
TracePair for 10 CVEs	12.50

VI. 결론 및 향후 연구

함수 단위 라벨링의 한계를 보완하기 위해 트레이스 단위 취약점 탐지 데이터셋 형식을 제안하였다. 또한 트레이스 기반 LLM-RAG 취약점 재현 탐지 절차를 구현하고, SARD-Juliet TracePair와 10개 CVE 대상 TracePair를 구축하여 평가하였다. SARD-Juliet TracePair에서는 최고 F1이 딥러닝 1.00, LLM 0.96으로 높게 나타났다. 다만 RQ1과 RQ2a는 서로 다른 원본 데이터에 기반하므로, 두 실험의 성능 차이를 트레이스 표현의 효과관으로 해석하기 어렵다. 10개 CVE 대상 TracePair에서는 최고 F1-score가 딥러닝 0.56, LLM 0.67로 나타났다. 다만 실제 CVE 평가는 20개 트레이스를 사용한 소규모 사례이므로, 이 결과는 실제 CVE 조건에서의 탐색적 관찰로 한정된다. 합성 SARD-Juliet과 실제 CVE 간 도메인 차이, Infer 기반 추출의 한계 및 트레이스 길이가 관찰된 성능에 영향을 주었을 가능성이 있다.

향후 연구에서는 동일한 원본 데이터에서 함수 포

현과 트레이스 표현을 각각 구성한 통제 실험을 수행할 필요가 있다. 그리고 그림 18과 같이 트레이스 기반 LLM-RAG 프레임워크를 바탕으로 실제 CVE 중심의 TracePair를 확장하고, 소스코드에서 트레이스를 자동으로 추출하는 방법을 고도화해야 한다. 또한 C/C++ 소스코드뿐 아니라 다양한 프로그래밍 언어의 소스 프로그램과 바이너리를 대상으로 트레이스 기반 취약점 탐지 절차를 확장할 필요가 있다.

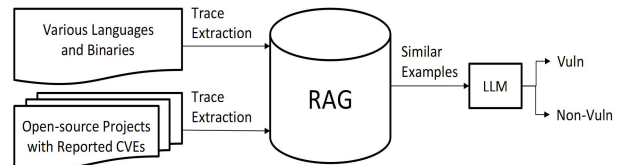


그림 18. 트레이스 기반 LLM-RAG 취약점 탐지 구조

REFERENCES

- [1] 전세옥, 장현지, 정민수, 이서준, 오다영, 최광훈, 김석희, 조성우, 김정미, “전기차 충전 인프라 오픈소스 소프트웨어에서의 취약한 코드 복제 탐지: VUDDY를 활용한 사례 연구”, *한국정보처리학회 학술대회논문집*, pp. 273-276, 전남대학교, 대한민국, 2024년 10월
- [2] S. Kim, S. Woo, H. Lee and H. Oh, “VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery”, in *Proc. of 2017 IEEE Symposium on Security and Privacy (SP)*, pp. 595-614, San Jose, CA, USA, May 2017.
- [3] W. Kang, B. Son and K. Heo, “TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities”, in *Proc. of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, pp. 1695-1708, Los Angeles, CA, USA, Nov. 2022.
- [4] X. Cheng et al., “DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network”, *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 3, pp. 1-33, 2021.
- [5] M. Fu and C. Tantithamthavorn, “LineVul: A Transformer-based Line-Level Vulnerability Prediction”, in *Proc. of 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pp. 608-620, Pittsburgh, PA, USA, May 2022.
- [6] Z. Liu, Z. Tang, J. Zhang, X. Xia and X. Yang, “Pre-training by Predicting Program Dependencies for Vulnerability Analysis Tasks”, in *Proc. of 2024 IEEE/ACM 46th International*

Conference on Software Engineering (ICSE), pp. 1863-1875, Lisbon, Portugal, Apr. 2024.

- [7] D. S. Ouchebara and S. Dupont, "Llama-Based Source Code Vulnerability Detection: Prompt Engineering vs Fine Tuning", in *Proc. of the 30th European Symposium on Research in Computer Security (ESORICS 2025)*, pp. 289-308, Toulouse, France, Sept. 2025.
- [8] P. Chakraborty, K. K. Arumugam, M. Alfadel, M. Nagappan and S. McIntosh, "Revisiting the Performance of Deep Learning-Based Vulnerability Detection on Realistic Datasets", *IEEE Transactions on Software Engineering*, vol. 50, no. 8, pp. 2163-2177, Aug. 2024.
- [9] S. Das, S. T. Fabiha, S. Shafiq and N. Medvidović, "Are We Learning the Right Features? A Framework for Evaluating DL-Based Software Vulnerability Detection Solutions", in *Proc. of 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 2893-2904, Ottawa, ON, Canada, Apr.-May 2025.
- [10] N. Risse and M. Böhme, "Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection", in *Proc. of the 33rd USENIX Security Symposium*, pp. 4247-4264, Philadelphia, PA, USA, Aug. 2024.
- [11] Infer Static Analyzer(n.d.). <https://fbinfer.com/> (accessed May., 14, 2026).
- [12] juliet-playground(n.d.). <https://github.com/seokjeon/juliet-playgrounddd> (accessed Aug., 5, 2026).
- [13] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages", *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536-1547, Online, Nov. 2020.
- [14] CWE Top 25 Most Dangerous Software Weaknesses - 2024(2024). https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html (accessed Jun., 19, 2026).
- [15] SARD-vs-CVE(n.d.). <https://github.com/seokjeon/SARD-vs-CVE> (accessed Aug., 5, 2026).

저 자 소 개

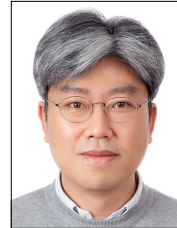


전세옥 (준회원)

2024년 전남대학교 소프트웨어공학과 학사 졸업.

2026년 전남대학교 정보보안융합학과 석사 과정.

<주관심분야 : 소프트웨어보안>



최광훈 (정회원)

2003년 KAIST 전산학과 공학박사.

2011년 연세대학교 미래캠퍼스 컴퓨터정보통신학부 조교수.

2016년 전남대학교 소프트웨어공학과와 인공지능학부 교수.

<주관심분야 : 프로그래밍언어, 컴파일러, 스마트홈, 소프트웨어보안, 신경망 검증, 침해사고대응>

망 검증, 침해사고대응>