이학 석사학위논문

SW 취약점 탐지에서 패치 기반 데이터셋의 함수 라벨링 한계와 트레이스 단위 데이터셋 구축 및 평가

전남대학교 대학원

정 보 보 안 융 합 학 과

전 세 옥

2026 년 8 월

SW 취약점 탐지에서 패치 기반 데이터셋의 함수 라벨링 한계와 트레이스 단위 데이터셋 구축 및 평가

이 논문을 이학 석사학위 논문으로 제출함

전남대학교 대학원

정 보 보 안 융 합 학 과

전 세 옥

지도교수 최 광 훈

전세옥의 이학 석사 학위논문을 인준함

심사위원장 엄익채 (인)

심 사 위 원 유동현 (인)

심 사 위 원 최광훈 (인)

2026 년 8 월

목 차

List of Figures

List of Tables

SW 취약점 탐지에서 패치 기반 데이터셋의 함수 라벨링 한계와 트레이스 단위 데이터셋 구축 및 평가

전 세 옥

전남대학교대학원 정보보안융합학과

(지도교수 : 최광훈)

최근 딥러닝 기반 취약점 탐지 연구는 각 논문에서 구성한 벤치마크에서 높은 취약점 탐지 성능을 보고해 왔다. 그러나 새로운 데이터셋에서는 F1 점수가 최대 91%까지 감소하거나 기존 논문에서 보고한 F1 점수가 재현되지 않는 한계가 확인되었다. 본 연구는 이 한계를 함수 단위 라벨링 문제에서 찾는다. 구체적으로, 짧은 패치 차이만으로는 취약/정상 샘플을 구분하기 어렵고, 같은 코드가 시간 흐름에 따라 서로 다른 라벨을 받을 수 있으며, 취약 원인과 무관한 정상 코드까지 취약 샘플에 포함될 수 있다. 이러한 조건에서는 인공지능 모델이 취약 코드 패턴과 취약/정상 의미 차이를 안정적으로 학습하기 어렵다.

본 연구는 함수 단위로 취약/정상 라벨을 부여하는 기존 데이터셋 라벨링의 한계를 보완하기 위해 Trace 단위 데이터셋을 제안한다. Trace 는 입력 지점인 소스(Source)에서 보안 민감 연산 지점인 싱크(Sink)까지 이어지는 데이터 흐름 경로상의 코드 문장 리스트이다. 본 연구는 RealVul 을 확장해 Extended RealVul-FuncPair 를 구성하고, SARD-Juliet SA TracePair 와 공개 취약점

식별자(Common Vulnerabilities and Exposures, CVE) 10 개에 대한 TracePair 를 구축하였다. 이를 바탕으로 딥러닝 기반 모델과 검색 예시를 함께 제공한 대규모 언어 모델의 취약/정상 구분 성능을 평가하였다.

평가 결과, Extended RealVul-FuncPair 에서는 함수 단위 데이터셋이 취약 함수와 정상 함수의 의미 차이를 안정적으로 드러내는 데 한계를 보였다. SARD-Juliet SA TracePair 에 대한 취약점 탐지 실험에서는 취약 Trace 를 정상 Trace 와 구분할 수 있어 Trace 기반 데이터셋의 장점을 보였다. 하지만 실제 10 개 CVE 에 대한 TracePair 에서는 그 구분이 안정적으로 이루어지지 않았다. 따라서 실제 CVE 적용을 위해서는 취약 원인 흐름을 반영한 Trace 데이터를 대규모로 확보해야 하며, 이를 위해 Infer 와 같은 정적 분석 도구의 Trace 추출 능력을 고도화하는 후속 연구가 필요하다.

1. 서 론

가. 연구 배경

오픈소스 소프트웨어는 여러 프로젝트에서 반복적으로 재사용된다. 이 과정에서 알려진 취약 코드가 다른 프로젝트로 다시 전파될 수 있다 [1]. 원본 프로젝트가 패치되더라도 복제된 코드까지 함께 수정된다고 보장할 수 없으므로, 패치되지 않은 전파 취약점의 탐지가 필요하다.

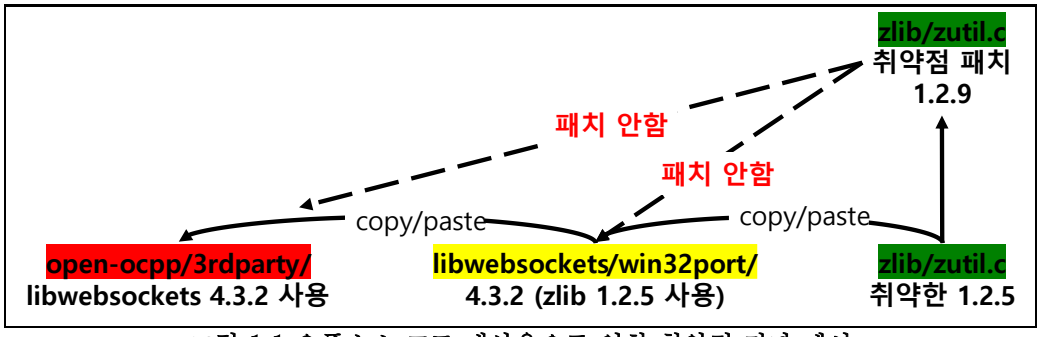


그림 1.1 오픈소스 코드 재사용으로 인한 취약점 전파 예시

그림 1.1 은 전기차 충전 인프라 오픈소스 소프트웨어인 open-ocpp 에 취약점이 전파된 사례를 보여준다 [7]. zlib 1.2.5 의 취약 코드가 libwebsockets 에 복제되었고, 이를 다시 사용한 open-ocpp 에도 포함되었다. 원본 zlib 는 1.2.9 에서 패치되었지만, 복제된 코드에는 패치가 적용되지 않았다.

공격자는 복제된 코드에 남아 있는 취약점을 악용할 수 있으므로, 원본 프로젝트의 패치 여부와 별개로 이를 선제적으로 탐지하고 조치해야 한다.

전파 취약점 탐지는 구문 기반 재출현 탐지에서 시작되었다. 이 접근은 취약 코드의 구문을 시그니처로 사용하므로 빠르지만, 코드 구조가 바뀐 변형에는 취약하다 [1]. 정적 분석 기반 재출현 탐지는 입력 Source 에서 결과 Sink 까지 이어지는 Trace 의 연산자, API 호출, 조건식 특징을 사용해 이러한 변형을 일부

다룬다 [6]. 그러나 사람이 설계한 특징에 의존하므로 새로운 취약 패턴이나 코드 표현 변형에는 한계가 있다.

AI 기반 취약점 탐지 연구는 이러한 수작업 특징 설계를 줄이기 위해 취약 코드 패턴을 데이터셋에서 학습한다. 딥러닝 기반 모델은 취약 코드를 토큰 시퀀스나 그래프로 표현해 취약 여부를 분류한다 [2], [3], [10], [16]. 대규모 언어 모델도 프롬프트와 검색 예시를 활용해 취약 여부를 판단하는 방식으로 사용된다 [20].

RealVul 데이터셋을 사용한 재평가 연구는 기존 벤치마크에서 보고된 딥러닝 기반 취약점 탐지 모델의 높은 성능이 CVE 기반 재평가에서도 유지되는지를 검토하였다 [5].

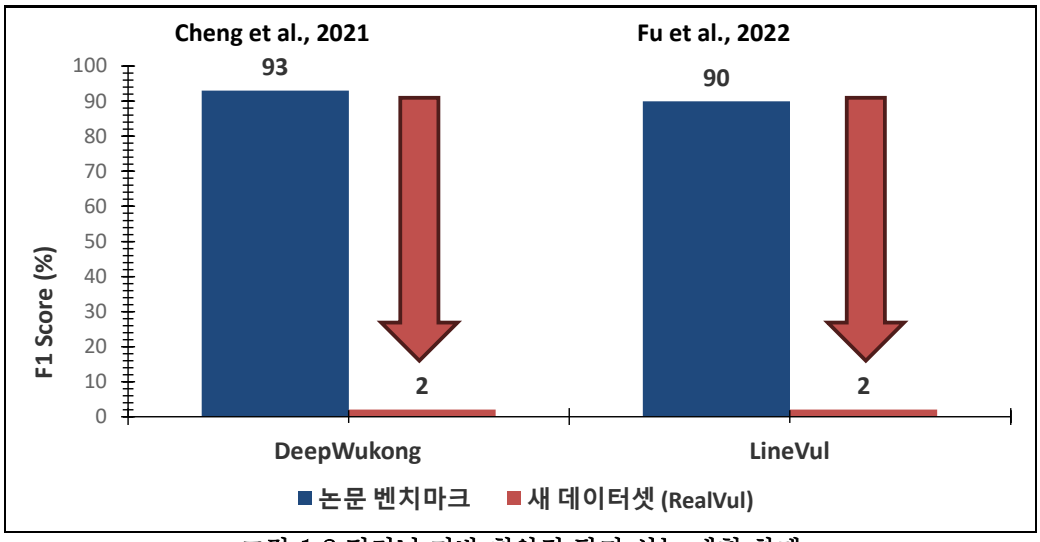


그림 1.2 딥러닝 기반 취약점 탐지 성능 재현 한계

DeepWukong 과 LineVul 은 각 논문에서 사용한 벤치마크에서는 높은 F1 점수를 획득하여 취약점 탐지 성능이 높다고 보고했지만, 그림 1.2 의 RealVul 재평가에서는 그 성능을 유지하지 못했다 [5]. 이는 취약점 탐지 모델의 오버피팅(overfitting) 가능성을 보여준다.

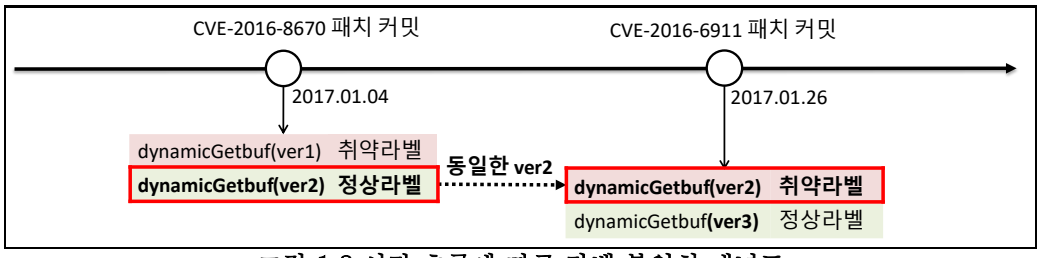


그림 1.3 시간 흐름에 따른 라벨 불일치 개념도

RealVul 의 결과는 성능 재현 한계가 모델 구조만의 문제가 아니라 데이터셋 라벨링 방식과도 연결된다. RealVul 데이터셋을 사용한 재평가 연구는 이러한 재현 한계의 원인 중 하나로 과거의 패치에서는 정상 라벨을 붙였으나 나중에 취약점이 발견되어 취약 라벨을 붙인 시간 흐름에 따른 라벨 불일치 문제를 지적하였다 [5]. 그림 1.3 처럼 한 CVE 에서 정상 코드로 수집된 동일 코드가 다른 CVE 분석에서는 취약 코드로 확인될 수 있다. 이런 라벨 충돌은 모델이 취약 코드 패턴을 일관되게 학습하기 어렵게 한다.

나. 연구 문제

본 연구는 기존 패치 기반 취약점 탐지 데이터셋의 함수 단위 라벨링 문제에 주목한다. 이 문제는 작은 패치 차이, 시간 흐름에 따른 라벨 충돌, 지나치게 넓은 취약 라벨 범위로 나타난다. 그 결과 모델은 취약 코드 패턴과 취약/정상 의미 차이를 안정적으로 학습하기 어렵다.

1) 패치 차이만으로는 취약/정상 구분이 어려움

패치 기반 데이터셋은 패치 전 함수를 취약 샘플, 패치 후 함수를 정상 샘플로 구성한다. 그러나 작은 조건식 변경만으로는 취약/정상 구분에 필요한 실행 문맥이 충분히 드러나지 않을 수 있다. 그림 1.4 는 이러한 경우를 보여준다.

```
1    irc_ctcp_dcc_filename_without_quotes (const char *filename)
2    {
3        int length;
4
5        length = strlen (filename);
6        if (length > 0)
7        {
8            if ((filename[0] == '₩"') && (filename[length - 1] == '₩"'))
9                return weechat_strndup (filename + 1, length - 2);
10       }
11       return strdup (filename);
12   }
```

(a) 취약 함수

```
1    irc_ctcp_dcc_filename_without_quotes (const char *filename)
2    {
3        int length;
4
5        length = strlen (filename);
6        if (length > 1)
7        {
8            if ((filename[0] == '₩"') && (filename[length - 1] == '₩"'))
9                return weechat_strndup (filename + 1, length - 2);
10       }
11       return strdup (filename);
12   }
```

(b) 정상 함수

그림 1.4 WeeChat CVE-2017-8073 패치 전후 코드 예시

그림 1.4 의 WeeChat CVE-2017-8073 사례에서 이 함수는 파일 이름 양끝의 큰따옴표를 제거하기 위해 filename + 1 부터 length - 2 만큼 문자열을 복사한다. 실제 패치 차이는 length > 0 을 length > 1 로 바꾼 조건식뿐이다. 그러나 길이가 1 인 문자열이 패치 전 조건을 통과하면 length - 2 가 음수가 되고, 이 값이 복사 크기로 사용되면서 잘못된 메모리 복사가 발생할 수 있다. 따라서 취약/정상 구분에는 조건식 자체뿐 아니라 length 가 복사 크기로 사용되는 제어 및 자료 흐름 문맥에 대한 이해가 필요하다.

2) 시간 흐름에 따른 라벨 불일치

패치 기반 데이터셋은 보통 패치 전 코드를 취약, 패치 후 코드를 정상으로 라벨링한다. 그러나 이 라벨은 시간 흐름에 따라 고정된 정답으로 남지 않을 수 있다. 그림 1.5 는 같은 dynamicGetbuf 함수 버전이 CVE-2016-8670 에서는 패치 후 정상 샘플로, CVE-2016-6911 에서는 패치 전 취약 샘플로 수집되는 경우를 보여준다 [5].

```
1   static int dynamicGetbuf(gdIOCtxPtr ctx, void *buf, int len)
2   {
3       int rlen, remain;
4       dpIOCtxPtr dctx;
5       dynamicPtr *dp;
6
7       dctx = (dpIOCtxPtr) ctx;
8       dp = dctx->dp;
9
10      remain = dp->logicalSize - dp->pos;
11      if(remain >= len) {
12          rlen = len;
13      } else {
14          if(remain == 0) {
15              return 0;
16          }
17
18          rlen = remain;
19      }
20
21      memcpy(buf, (void *) ((char *)dp->data +  dp->pos), rlen);
22      dp->pos + = rlen;
23
24      return rlen;
25  }
```

(a) CVE-2016-8670 취약 샘플

```
1   static int dynamicGetbuf(gdIOCtxPtr ctx, void *buf, int len)
2   {
3       int rlen, remain;
4       dpIOCtxPtr dctx;
5       dynamicPtr *dp;
6
7       dctx = (dpIOCtxPtr) ctx;
```

```
8       dp = dctx->dp;
9
10      remain = dp->logicalSize - dp->pos;
11      if(remain >= len) {
12          rlen = len;
13      } else {
14          if(remain <= 0) {
15              return 0;
16          }
17
18          rlen = remain;
19      }
20
21      memcpy(buf, (void *) ((char *)dp->data +  dp->pos), rlen);
22      dp->pos += rlen;
23
24      return rlen;
25  }
```

(b) CVE-2016-8670 정상 샘플 / CVE-2016-6911 취약 샘플

```
1   static int dynamicGetbuf(gdIOCtxPtr ctx, void *buf, int len)
2   {
3       int rlen, remain;
4       dpIOCtxPtr dctx;
5       dynamicPtr *dp;
6
7       dctx = (dpIOCtxPtr) ctx;
8       dp = dctx->dp;
9
10      if (dp->pos < 0 || dp->pos >= dp->realSize) {
11          return 0;
12      }
13
14      remain = dp->logicalSize - dp->pos;
15      if(remain >= len) {
16          rlen = len;
17      } else {
18          if(remain <= 0) {
19              return 0;
20          }
21
22          rlen = remain;
23      }
24
25      if (dp->pos +  rlen > dp->realSize) {
26          rlen = dp->realSize - dp->pos;
27      }
```

```
28
29      memcpy(buf, (void *) ((char *)dp->data +  dp->pos), rlen);
30      dp->pos += rlen;
31
32      return rlen;
33  }
```

(c) CVE-2016-6911 정상 샘플

그림 1.5 libgd dynamicGetbuf 에서 발생한 시간 흐름 라벨 불일치 사례

그림 1.5 에서 (b)는 CVE-2016-8670 기준으로는 remain <= 0 조건을 반영한 정상 샘플이다. 그러나 CVE-2016-6911 기준으로는 dp->pos 범위 검사가 부족한 취약 샘플이다. 이처럼 동일 코드가 정상과 취약 라벨을 모두 받으면 모델이 취약 코드 패턴을 일관되게 학습하기 어렵다.

3) 지나치게 포괄적인 취약 라벨 범위

함수 하나에 취약 라벨을 부여하면 실제 취약 코드와 주변 정상 코드가 함께 묶인다 [19], [22]. 이때 모델은 취약 원인 흐름뿐 아니라 같은 함수 안의 정상 코드까지 취약 패턴으로 학습할 수 있다 [21].

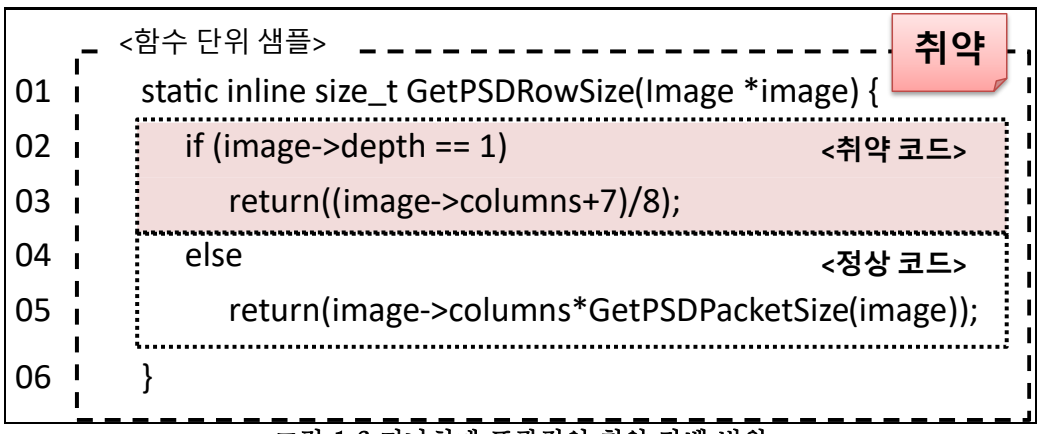


그림 1.6 지나치게 포괄적인 취약 라벨 범위

그림 1.6 의 ImageMagick CVE-2016-7525 사례에서 image->depth == 1 분기는 PSD 패킷 크기를 반영하지 못할 수 있다. 반면 else 분기는 다른 조건의 정상 크기 계산 코드이다. 두 흐름이 같은 취약 라벨 아래 묶이면 모델은 취약 원인 흐름과 주변 정상 코드를 구분하기 어렵다.

다. 해결방안과 기여

1) 해결 방안

본 연구는 함수 단위 라벨링 한계에 대응하기 위해 Trace 단위 데이터셋을 제안한다. Trace 는 입력 Source 에서 시작해 데이터 흐름을 따라 결과 Sink 까지 이어지는 코드 문장 리스트이다. Trace 단위 데이터셋은 취약 원인과 결과를 잇는 Source-Sink 흐름을 샘플로 제공하므로, 작은 패치 차이만으로 취약/정상 구분이 어려운 경우에도 판단에 필요한 문맥을 포함하고 있어 취약점 판단 근거를 보여준다.

또한 Trace 는 원인과 결과로 이어지는 문장들의 리스트이므로, 취약점과 무관한 주변 정상 코드를 포함하지 않고, 취약점과 관련된 코드만 포함한다. 동일한 이유로 Trace 단위 데이터는 취약점의 원인과 결과 관계를 명시적으로 제시하므로 나중에 번복하여 취약으로 판단하는 경우를 줄일 수 있다.

2) 기여

본 논문의 기여는 세 가지이다. 첫째, 함수 단위 라벨링 문제를 줄이기 위한 Trace 단위 취약점 탐지 데이터셋을 제안하였다.

둘째, 함수 단위 데이터셋과 Trace 단위 데이터셋을 비교할 수 있도록 Extended RealVul-FuncPair, SARD-Juliet SA TracePair, 10 개 CVE 에 대한 TracePair 를 구축하였다.

표 1.1 본 연구 주요 데이터셋 비교

데이터셋	입력 단위	비교 쌍 구성	본 연구에서 기여
RealVul[5]	함수	아니오	아니오
Extended RealVul	함수	아니오	예
Extended RealVul-FuncPair	함수 쌍	예	예
SARD-Juliet SA TracePair	Trace 쌍	예	예
10 개 CVE 에 대한 TracePair	Trace 쌍	예	예

표 1.1 에서 Extended RealVul-FuncPair 는 동일 CVE 의 취약 함수와 정상 함수를 묶은 함수 단위 평가 데이터셋이다. SARD-Juliet SA TracePair 와 10 개 CVE 에 대한 TracePair 는 취약 Trace 와 정상 Trace 를 묶은 Trace 단위 평가 데이터셋이다. 세 데이터셋 구성은 함수 단위 데이터셋과 Trace 단위 데이터셋의 취약/정상 분류 결과를 비교하여, 모델이 학습한 취약 코드 패턴에 근거해 취약/정상 차이를 구분하는지 평가하기 위한 구조이다.

셋째, 딥러닝 모델과 검색 예시를 함께 제공한 대규모 언어 모델을 사용해 함수 단위 데이터셋과 Trace 단위 데이터셋에서의 취약/정상 구분 성능을 비교하였다.

2. 관련 연구

취약점 탐지는 구문 기반 재출현 탐지에서 기계학습, 딥러닝, 대규모 언어 모델(large language model, LLM) 기반 탐지로 확장되었다. 관련 연구는 모델 구조뿐 아니라 데이터 중복, 라벨 품질, 시간 흐름에 따른 라벨 불일치가 취약 코드 패턴 학습을 어렵게 만든다는 점도 보고하였다 [5], [19], [21], [22]. 이 장은 기존 취약점 탐지 방법과 취약점 탐지 데이터셋 연구를 정리하고, 함수 단위 데이터셋과 Trace 단위 데이터셋을 비교해야 하는 필요성을 설명한다.

가. 구문 기반 취약점 재출현 탐지

오픈소스 소프트웨어에서는 이미 알려진 취약 코드가 재사용을 통해 다른 프로젝트나 버전으로 전파될 수 있다. 이 절에서 취약 코드 클론은 이미 알려진 취약 함수나 취약 코드 구간이 다른 파일, 버전, 프로젝트에 복제되어 남아 있는 경우를 뜻한다. 원본 취약점이 패치되더라도 복제본에 같은 수정이 반영되지 않으면 같은 취약점이 재출현할 수 있다. ReDeBug 는 보안 패치에서 추출한 취약 코드 조각을 정규화한 뒤, 운영체제 배포판 규모의 소스 패키지와 구문적으로 비교하여 패치되지 않은 취약 코드 클론을 탐지하였다 [15]. VUDDY 는 취약 함수의 정규화된 표현을 시그니처로 만들어 취약 코드 클론을 찾았다 [1]. 전기차 충전 인프라 사례에서도 VUDDY 로 라이브러리 복제를 통해 유입된 취약 코드 클론이 확인되었다 [7]. VulPecker 는 취약점 패치에서 얻은 코드 변경 정보를 코드 유사도 비교와 결합해 취약점별 유사 코드를 탐지하였다 [17].

구문 기반 접근은 빠르지만 변형된 코드를 안정적으로 찾기 어렵다. 문장이 삽입, 삭제, 수정되거나 같은 취약 동작이 다른 코드로 구현되면 정규화된 구문 시그니처만으로는 재출현을 놓칠 수 있다 [1].

VGRAPH 는 이 한계를 줄이기 위해 취약 함수와 패치 함수를 코드 구조와 제어/자료 흐름을 함께 나타내는 코드 속성 그래프(Code Property Graph, CPG)로 표현한다 [13]. 그런 뒤 CPG 에서 (출발 요소, 관계, 도착 요소) 형태의 삼중항 관계(triplet)를 추출한다. 이 관계로 취약 관계, 패치 관계, 공통 문맥 관계를 비교한다. 아래 그림은 VGRAPH 의 foo() 예시에서 취약 함수와 패치 후 함수를 보인다.

```
1  void foo() {
2    int x = input();
3    if (x > MIN) {
4      int y = x * 10;
5      output(y);
6    }
7  }
```

(a) 취약 함수

```
1  void foo() {
2    int x = input();
3    if (x > MIN) {
4      int y = x * 10;
5      if (y < MAX)
6        output(y);
7    }
8  }
```

(b) 패치 후 함수

그림 2.1 VGRAPH foo() 예시의 취약 함수와 패치 후 함수

표 2.1 은 위 두 함수에서 추출한 삼중항 관계를 보인다. CONTROLS, FLOWS_TO, DEF, REACHES 는 각각 제어 관계, 자료 흐름 관계, 정의 관계, 도달 관계를 뜻한다.

표 2.1 VGRAPH foo() 예시의 삼중항 관계 구분

구분	대표 삼중항 관계
취약 관계 삼중항	(x > MIN, CONTROLS, output(y)), (y = x * 10, FLOWS_TO, output(y))
패치 관계 삼중항	(y < MAX, CONTROLS, output(y)), (y = x * 10, FLOWS_TO, y < MAX)
공통 문맥 삼중항	(x = input(), DEF, x), (x = input(), REACHES, y = x * 10)

이 예시에서 취약 관계 삼중항은 x > MIN 조건과 y = x * 10 계산 결과가 별도 검사 없이 output(y)로 이어지는 흐름을 나타낸다. 패치 관계 삼중항은 패치 후 추가된 y < MAX 검사가 output(y)를 제어하고, 계산 결과가 먼저 이 검사로 전달되는 관계를 나타낸다. 공통 문맥 삼중항은 취약 함수와 패치 후 함수에 모두 남아 있는 코드 관계를 나타낸다. 취약 코드 클론은 변수명 변경이나 문장 추가처럼 원본과 일부 달라져도 취약점이 발생하는 주변 코드 구조를 유지할 수 있다. 따라서 VGRAPH 는 공통 문맥 삼중항을 이용해 검사 대상 함수가 원본 취약 함수와 취약점 주변의 코드 구조를 공유하는지를 먼저 확인한다. 이어서 그 공통 문맥 안에서 취약 관계와 패치 관계의 공유 정도를 비교한다. 검사 대상 함수가 공통 문맥과 취약 관계를 많이 공유하고 패치 관계를 적게 공유하면, VGRAPH 는 수정된 코드라도 패치되지 않은 취약 코드 클론으로 판단한다 [13].

VGRAPH 에서 공통 문맥 삼중항은 클론 후보가 원본 취약 함수와 필요한 코드 맥락을 공유하는지 확인하는 정보로 사용된다. 반면 본 연구는 함수 단위 샘플이 취약 원인 흐름과 직접 관련이 낮은 주변 코드까지 함께 포함해, 모델의 취약/정상 구분에 필요한 정보를 약하게 만들 수 있다는 문제를 다룬다.

나. 기계학습 기반 취약점 재출현 탐지

구문 기반 취약점 재출현 탐지는 코드가 삽입, 삭제, 재배열되거나 같은 취약 동작이 다른 형태로 구현되면 안정적으로 탐지하기 어렵다. Tracer 는 이 한계를 줄이기 위해 알려진 취약 코드에서 source 에서 sink 까지 이어지는 데이터 의존 Trace 를 추출한다 [6]. Tracer 는 각 Trace 를 정수 특징 벡터(feature vector)로 표현한다 [6]. 특징 벡터는 Trace 에 나타나는 기본 연산자와 공통 API 호출의 출현 횟수를 중심으로 구성된다. gimp CVE-2009-1570 사례에서는 fread 로 유입된 값이 비트 연산과 산술 연산을 거쳐 malloc 의 크기 인자로 전달된다 [6].

Tracer 는 이 Trace 를 ⟨fread: 1, |: 3, <<: 3, *: 2, +: 1, -: 1, /: 1, malloc: 1⟩처럼 source-sink 경로에 나타난 특징 항목별 출현 횟수로 표현한다. Tracer 는 새 프로그램의 검사 대상 Trace 도 같은 방식으로 벡터화하고, 알려진 취약 Trace 벡터와 유사도를 계산해 취약점을 탐지한다.

Tracer 의 특징 벡터는 데이터 의존 Trace 에서 만들어지지만, 특징 항목은 경험적 관찰과 수작업 설계에 기반한다. 따라서 새로운 취약 유형이나 기존 특징으로 표현하기 어려운 변형에는 대응이 제한될 수 있다. 본 연구는 사람이 정한 특징 벡터 대신 Trace 자체를 학습 및 평가 데이터 단위로 사용한다.

다. 딥러닝 기반 취약점 탐지

딥러닝 기반 취약점 탐지는 사람이 특징 벡터를 직접 설계하는 대신 모델이 취약 코드 패턴을 학습하게 한다. 이 연구들은 취약 코드를 토큰 시퀀스 혹은 그래프로 바꾸어 학습 및 평가에 사용한다.

토큰 시퀀스 계열은 취약점과 관련된 문장이나 함수를 토큰 리스트로 처리한다. VulDeePecker 는 API 호출 주변 데이터 흐름 문장을 사용하고 [16], SySeVR 은 제어 흐름과 데이터 흐름으로 확장한 취약 구문 후보를 사용한다 [10]. VulDeeLocator 는 C/C++ 컴파일러인 clang 을 활용해 소스 코드를 LLVM 중간 표현(LLVM Intermediate Representation, LLVM IR)으로 변환하고, 이 IR 기반 코드 표현을 사용한다 [18]. LineVul 은 Transformer 기반 모델로 함수 단위 취약 여부와 라인 수준 위치를 함께 예측하였다 [3]. VulBERTa 와 PDBERT 는 코드 문맥이나 제어·데이터 의존성을 사전학습에 활용하였다 [4], [24].

그래프 계열은 조건 분기와 변수 정의-사용 관계를 노드와 엣지로 명시한다. 그림 2.2 는 소스 코드를 그래프로 변환한 뒤 그래프와 라벨의 관계를 학습하는 일반적 흐름을 보인다.

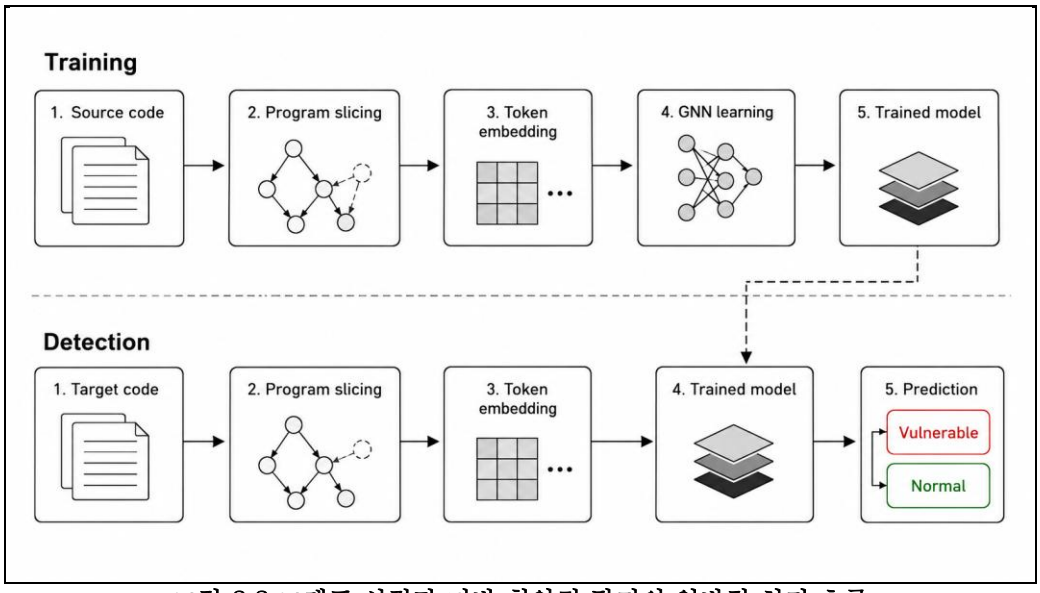


그림 2.2 그래프 신경망 기반 취약점 탐지의 일반적 처리 흐름

Devign 은 함수 내부 코드 구조와 의존 관계를 그래프로 표현하였다 [25]. DeepWukong 은 취약 후보 지점과 관련된 제어 흐름 및 데이터 흐름을 확장 흐름 그래프로 구성하였다 [2]. SnapVuln 은 취약점 유형별로 함수 간 관련 코드와 의존 관계를 추출하였다 [12]. 이처럼 딥러닝 기반 취약점 탐지 성능은 모델 구조뿐 아니라 샘플의 형식에도 영향을 받는다.

이 연구들은 모델 구조와 샘플 표현 방식을 개선해 취약 여부 분류 성능을 높이는 데 초점을 둔다. 반면 본 연구는 새 딥러닝 구조를 제안하지 않는다. 본 연구는 함수 단위 데이터셋과 Trace 단위 데이터셋에서 취약/정상 분류 결과가 어떻게 달라지는지 비교한다.

라. LLM 기반 취약점 탐지

LLM 은 코드 이해, 취약점 분석, 패치 생성 등 보안 작업에 활용된다. 퍼징(fuzzing)은 프로그램에 다양한 입력을 실행해 오류를 찾는 동적 분석 기법이다. AI Cyber Challenge(AIxCC)에 참여한 ATLANTIS 는 정적 분석도 함께 사용하지만, 퍼징과 실행 피드백으로 취약 동작을 재현하는 취약점 증명(proof-of-

vulnerability, PoV)을 생성하고 검증하므로 동적 분석 중심의 혼합형 접근이다 [8]. Branch Flipper 는 커버리지 기반 퍼징(coverage-guided fuzzing)에서 LLM 생성 입력을 실행하고 커버리지 피드백으로 평가하는 동적 분석 중심 접근이다 [9].

반면 Llama-based source code vulnerability detection 은 함수 소스 코드를 LLM 에 제시하고 취약 여부를 분류하는 정적 소스 코드 분류 연구이다 [20]. 이 연구는 여러 평가 설정이 취약 코드 분류 성능에 미치는 영향을 비교하였다 [20]. 비교 대상에는 프롬프트 설계(prompt engineering), 소수 예시 기반 프롬프팅(few-shot prompting), 검색 증강 생성(Retrieval-Augmented Generation, RAG), 파인튜닝(fine-tuning)이 포함된다. RAG 는 검색한 예시를 프롬프트에 함께 넣어 모델 답변에 활용하는 방법이다. 해당 연구는 단순 프롬프트만으로는 성능이 제한될 수 있으며, 파인튜닝이나 RAG 기반 검색 예시가 성능 향상에 중요할 수 있음을 보였다 [20].

따라서 LLM 기반 취약점 탐지 연구는 동적 분석 중심 자동 보안 분석과 정적 소스 코드 분류로 나뉜다. 본 연구도 LLM 을 취약점 분류기로 사용하지만, 평가 대상 샘플을 함수 단위 데이터셋과 Trace 단위 데이터셋으로 나누어 비교한다.

마. 취약점 탐지 데이터셋

딥러닝 기반 취약점 탐지 연구는 각 논문이 구성한 벤치마크에서 높은 성능을 보고하였다 [2], [3], [4], [10], [16], [21], [24], [25]. RealVul 연구는 기존 딥러닝 기반 취약점 탐지 모델을 별도의 데이터셋에서 재평가하였다 [5]. 그 결과 높은 벤치마크 성능이 안정적으로 재현되지 않음을 보였다.

이러한 재현 한계를 분석하려면 데이터셋 구성과 라벨 기준도 함께 검토해야 한다. 기존 취약점 탐지 데이터셋은 주로 CVE 와 연결된 취약점 패치 커밋을 활용한다. 일반적으로 패치 전 함수를 취약으로, 패치 후 함수를 정상으로 라벨링한다 [26].

이 방식은 대규모 데이터셋을 자동으로 구축할 수 있어 딥러닝 기반 탐지 연구의 실험 기반을 제공했다.

그러나 함수 단위 데이터셋으로 학습한 모델은 취약 코드 패턴을 안정적으로 학습하지 못했을 수 있다. VulnPatchPairs 는 같은 CVE 의 취약 함수와 정상 함수 쌍에서 취약/정상 구분 한계가 나타날 수 있음을 보였다 [19]. VIPer 는 취약점과 무관한 코드 문장 삽입만으로 일부 취약 함수 예측이 정상으로 바뀔 수 있음을 보고하였다 [14]. 이 결과는 모델이 취약 코드 패턴보다 데이터셋 특성에 과적합되었을 가능성을 보였다.

또한 PrimeVul 은 패치 커밋에서 변경된 함수를 모두 취약 샘플로 수집하면 취약점과 직접 관련 없는 함수도 취약 라벨에 포함될 수 있음을 지적하였다 [22]. REVEAL 은 데이터 중복과 데이터 누수가 성능 과대평가와 재현 약화로 이어질 수 있음을 분석하였다 [21]. RealVul 연구는 특정 시점에 정상 함수로 라벨링된 동일 코드가 다른 CVE 분석에서 취약 코드로 확인되는 시간 흐름에 따른 라벨 불일치를 보였다 [5].

표 2.2 는 관련 연구를 접근 유형과 세부 방식으로 구분한다.

표 2.2 취약점 탐지 관련 연구의 접근 유형과 세부 방식 비교

접근 유형	세부 방식	대표 연구	연구 초점
정적	구문 기반	ReDeBug [15], VUDDY [1], VulPecker [17], VGRAPH [13]	코드 형태 유사도로 알려진 취약 코드 재사용 탐지
	기계학습 기반	Tracer [6]	사람이 설계한 특징 벡터로 취약 유사 패턴을 탐지
	딥러닝 기반	VulDeePecker [16], SySeVR [10], LineVul [3], PDBERT [4]	모델이 취약 코드 패턴을 학습해 취약 여부를 자동으로 탐지
	LLM 기반 분류	Llama-based source code vulnerability detection [20]	LLM 에 함수 소스 코드를 제시해 취약 여부 판단

동적	LLM 기반 자동 보안 분석	AIxCC/ATLANTIS [8], Branch Flipper [9]	LLM 을 취약점 탐지, 경로 탐색, PoV 생성과 검증에 활용
데이터셋 분석	취약점 탐지 데이터셋	RealVul [5], VulnPatchPairs [19], VIPer [14], REVEAL [21], PrimeVul [22]	중복, 표면 특징 의존, 라벨 불일치를 데이터셋 한계로 분석

LLM 기반 연구는 정적 소스 코드 분류와 동적 분석으로 나뉜다. 데이터셋 연구는 샘플 중복과 라벨 불일치 같은 구성 문제가 딥러닝 기반 탐지 성능의 재현을 어렵게 만들 수 있음을 보였다.

이에 본 연구는 새로운 취약점 탐지 모델을 제안하기보다, 데이터셋의 샘플 단위가 취약/정상 구분에 필요한 정보를 충분히 제공하는지 평가한다. 기존 패치 기반 함수 단위 데이터셋에서는 패치 차이만으로 취약/정상 구분이 어려운 경우, 시간 흐름에 따른 라벨 불일치, 포괄적인 취약 라벨 범위가 문제가 될 수 있다. 본 연구는 Tracer 의 source-sink 경로 개념을 바탕으로 Trace 단위 데이터셋을 구축하고, 함수 단위 데이터셋과 Trace 단위 데이터셋에서 취약점 탐지 모델의 취약/정상 구분 성능이 어떻게 달라지는지 비교한다.

3. Trace 기반 데이터셋

패치 기반 함수 단위 라벨링은 작은 패치 차이, 시간 흐름에 따른 라벨 충돌, 지나치게 넓은 취약 라벨 범위 때문에 취약 함수와 정상 함수의 의미 차이를 흐릴 수 있다. 이 장은 판단 근거를 취약 원인과 결과가 연결되는 Source-Sink 흐름으로 좁혀 이러한 한계를 완화하기 위한 Trace 단위 데이터셋을 설명한다.

먼저 Trace 를 source-sink 데이터 의존 흐름의 코드 문장 리스트로 정의한다. 이어서 Trace 가 함수 단위 라벨링 한계를 완화하는 원리와 Infer 기반 추출 방법을 제시한다. 마지막으로 SARD-Juliet SA Trace, SARD-Juliet SA TracePair, 10 개 CVE 에 대한 TracePair 구축 절차를 설명하고, 이를 바탕으로 연구 질문(Research Question, RQ)을 정리한다.

가. Trace 정의

함수 단위 샘플에는 취약 여부 판단에 필요한 코드와 주변 코드가 함께 포함된다. 모델은 외부 데이터가 보안 민감 연산에 도달하는 흐름뿐 아니라 초기화, 출력, 주변 제어문처럼 취약점과 직접 관련 없는 코드도 함께 받는다. 따라서 함수 전체를 하나의 샘플로 두기보다, 취약 여부 판단에 직접 필요한 데이터 흐름만 분리한 샘플 단위가 필요하다.

본 연구에서 이 샘플 단위를 Trace 로 정의한다. Trace 는 Source 에서 시작해 데이터 흐름을 따라 Sink 까지 이어지는 코드 문장 리스트이다. Source 는 외부 데이터가 유입되는 지점이다. Step 은 해당 값이 변환, 대입, 조건 검사, 전달을 거치는 중간 지점이다. Sink 는 Source 에서 유래한 값이 보안 민감 연산에 사용되는 지점이다.

```
05  if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL) {
06      data = atoi(inputBuffer);
07  } else { printLine("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  printLine(dest);
15  }
```

(a) 함수 단위 샘플

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

(b) Trace 단위 샘플

그림 3.1 함수 단위 샘플에서 Trace 단위 샘플로의 변환 예시

그림 3.1 은 함수 단위 샘플에서 취약 판단 흐름만 Trace 로 남기는 예시이다. fgets 는 외부 입력을 읽는 Source 이고, atoi 와 조건문은 입력값을 변환하고 검사하는 Step 이다. memcpy 에서는 입력에서 유래한 data 가 복사 크기 인자로 사용되므로 Sink 이다. Trace 는 이처럼 취약 여부 판단에 필요한 입력 전달 흐름을 함수 안에서 분리한다.

나. Trace 기반 함수 라벨링 한계 완화

Trace 는 함수 단위 라벨링 한계에 대응하기 위해 취약 여부 판단에 필요한 Source-Sink 흐름을 샘플 단위로 구성한다. 함수 단위 라벨링의 세가지 문제는 Trace 단위 데이터셋에서 다음과 같이 완화된다.

첫째, 작은 패치 차이만으로 취약/정상 구분이 어려운 경우 함수 단위 데이터셋의 샘플에서는 판단에 필요한 코드가 주변 코드와 함께 포함되어 취약 원인 문맥을 파악하기 어려울 수 있다. 반면 Trace 는 Source 에서 Sink 까지 이어지는 코드로

샘플 범위를 좁혀 제시하므로, 입력값이 어떤 중간 지점을 거쳐 보안 민감 연산에 도달하는지 더 직접적으로 확인하게 한다. 그림 3.2 는 같은 Sink 호출이라도 Source 에서 시작한 크기 관계를 함께 보아야 취약/정상 차이를 확인할 수 있음을 보여준다.

단계	코드
Source	data = (int *)ALLOCA(10);
Step1	# define alloca(size) __builtin_alloca (size)
Step2	FUNC_1(data);
Sink	memcpy(data, source, 10*sizeof(int));

(a) 취약 Trace

단계	코드
Source	data = (int *)ALLOCA(10*sizeof(int));
Step1	# define alloca(size) __builtin_alloca (size)
Step2	FUNC_1(data);
Sink	memcpy(data, source, 10*sizeof(int));

(b) 정상 Trace

그림 3.2 SARD-Juliet SA Trace Stack Buffer Overflow Trace

그림 3.2 는 Trace 단위 샘플이 취약/정상 구분에 필요한 판단 근거를 함수 전체보다 선명하게 드러낸다는 점을 보여준다. 두 Trace 는 같은 memcpy Sink 를 갖지만, 취약 Trace 는 작은 할당 크기에서 큰 복사 크기로 이어지고 정상 Trace 는 할당 크기와 복사 크기가 대응된다. 즉, Trace 단위 데이터셋은 Source 에서 Sink 까지 이어지는 크기 관계만 남겨 취약 원인과 결과를 더 직접적으로 비교하게 한다.

둘째, 시간 흐름에 따른 정상/취약 라벨 충돌에 대해서도 Trace 는 판단 범위를 취약점의 원인과 결과 관계로 좁힌다. Trace 단위로 샘플을 구성하면 취약점의 원인과 결과 관계가 명확하기 때문에 나중에 번복하여 취약으로 판단하는 경우를 줄일 수 있다.

셋째, Trace 단위 데이터셋은 취약 라벨 범위를 함수 전체에서 취약 관련 흐름으로 좁힌다. 따라서 Trace 는 취약점과 무관한 주변 정상 코드를 포함하지

않고, 취약점과 관련된 코드만 포함한다. 그림 3.3 은 함수 전체가 아니라 메모리 해제 또는 재할당에서 Sink 사용까지 이어지는 흐름만 샘플에 포함하는 예시이다.

단계	코드
Source	free(data);
Step1	FUNC_1(data);
Sink	printf("%s₩n", line);

(a) 취약 Trace

단계	코드
Source	data = (char *)malloc(100*sizeof(char));
Step1	FUNC_1(data);
Sink	printf("%s₩n", line);

(b) 정상 Trace

그림 3.3 SARD-Juliet SA Trace Use after Free Trace

그림 3.3 에서 취약 Trace 는 free(data) 이후 같은 값이 사용되는 흐름을 담는다. 정상 Trace 는 사용 전에 malloc 으로 다시 확보되는 흐름을 담는다. 이 예시는 Trace 단위 샘플이 함수 전체가 아니라 취약 여부 판단에 필요한 Source, 중간 Step, Sink 를 중심으로 구성됨을 보여준다.

다. Trace 추출 방법

1) Taint 정적 분석과 Trace

본 연구의 Trace 는 입력 Source 에서 시작해 데이터 흐름을 따라 결과 Sink 까지 이어지는 코드 문장 리스트이다. 정적 분석 과정에서는 입력 Source 에서 시작한 값에 오염 표식을 붙이고, 그 값이 대입, 연산, 함수 호출을 거쳐 결과 Sink 까지 전달되는지를 추적한다. 본 연구는 이 결과에서 모델 입력에 필요한 source, 중간 전달 지점, sink 의 코드 라인만 수집한다. 따라서 Trace 는 실제 실행 경로가 아니라 정적 분석 기반의 잠재적 데이터 흐름이다.

본 연구는 Trace 수집 단계에서 Tracer [6]가 제시한 source-sink 오염 분석 방법과 데이터 의존 경로 기반 Trace 정의를 따른다. Tracer 는 오염 분석(taint

analysis)을 추상 상태, 추상 메모리, 오염 정보의 관계로 형식화한다. 오염 분석은 source 에서 시작한 값이 sink 까지 전달되는지를 추적하는 정적 분석 방식이다. 여기서 제어 지점(control point)은 분석기가 관찰하는 프로그램 위치이다. 대입문, 조건문, 함수 호출, sink 호출 위치가 모두 제어 지점이 될 수 있다. 소스 지점(source point)은 오염 표식이 처음 붙는 위치이다. 싱크 지점(sink point)은 그 표식이 붙은 값이 보안 민감 연산에 사용되는 위치이다. 수식에서 $\mathbf{C}$ 는 전체 제어 지점 집합이고, $\mathbf{X}$ 는 변수 집합이다. $\mathbf{M}$ 은 각 변수의 추상 값을 저장한 추상 메모리이며, $\mathbf{T}$ 는 오염 정보, $\mathbf{V}$ 는 값 정보를 의미한다.

$$\text{(Abstract state) } S = C \to M$$
$$\text{(Abstract memory) } M = X \to T \times V$$
$$\text{(Taint) } T = \wp(C)$$

(a) Abstract domains

$$[\![E]\!] : M \to T \times V$$
$$[\![n]\!](m) = \langle \emptyset, V(n)(m) \rangle$$
$$[\![x]\!](m) = m(x)$$
$$[\![E_1 + E_2]\!](m) = \langle T_1 \cup T_2, V(E_1)(m) +_v V(E_2)(m) \rangle$$
$$[\![E_1 - E_2]\!](m) = \langle T_1 \cup T_2, V(E_1)(m) -_v V(E_2)(m) \rangle$$
$$[\![\text{source}_l()]\!](m) = \langle \{l\}, V(\text{source})(m) \rangle$$
$$[\![C]\!] : M \to M$$
$$[\![x := E]\!](m) = m\{x \mapsto [\![E]\!](m)\}$$
$$[\![\text{assume}(x < n)]\!](m) = m$$
$$[\![\text{sink}(E)]\!](m) = m$$

(b) Abstract semantics

그림 3.4 일반 오염 분석(Generic Taint Analysis)의 추상 영역과 의미론

그림 3.4 의 $S = C \to M$ 은 각 control point 마다 하나의 추상 메모리 상태를 대응시킨다는 뜻이다. $M = X \to T \times V$ 는 추상 메모리가 각 변수에 오염 정보 T 와 값 정보 V 를 함께 저장한다는 뜻이다. $T = \wp(C)$는 오염 정보가 source point 들의 집합이라는 뜻이다. 예를 들어 $\emptyset$는 아직 어떤 source 의 영향도 받지 않은 값이고, $\{l\}$은 label l 을 가진 source 에서 온 값이다. 여러 source 의 영향이 합쳐지면 오염 정보는 $\{l_1, l_2\}$처럼 여러 label 을 포함할 수 있다.

그림 3.4(b)의 식도 같은 원리로 읽을 수 있다. 상수 n 은 source 에서 온 값이 아니므로 오염 정보가 비어 있다. 변수 x 는 메모리 m 에 저장된 현재 오염 정보를 그대로 읽는다. 덧셈과 뺄셈은 두 피연산자의 오염 정보를 합친다. 즉, 두 값 중 하나라도 source 의 영향을 받으면 계산 결과도 그 source 의 영향을 받은 값으로 본다. $\text{source}_l()$은 label l 의 오염 정보를 새로 만들고, 대입문 x := E 는 식 E 의 오염 정보를 변수 x 에 저장한다. 반면 sink(E)는 메모리를 바꾸는 명령이 아니라, sink 인자 E 가 어떤 source 의 영향을 받았는지 확인하는 지점이다.

Tracer 는 sink point c_2에서 추상 메모리 m 을 확인하고, 그 sink 에 도달한 source point 집합 $Q(c_2)(m)$을 이용해 경고(alarm)를 다음과 같이 정의한다 [6]. 여기서 C_s는 전체 sink point 집합이고, Ω 는 분석 결과로 보고되는 source-sink 쌍의 집합이다.

$$\Omega = \{ \langle c_1, c_2 \rangle \mid c_2 \in C_s, c_1 \in Q(c_2)(m) \}$$

이 경고에서 Trace 는 데이터 의존 그래프 위의 경로로 정의된다 [6]. 제어 흐름 관계 $c_1 \rightarrow c_2$는 프로그램 실행 순서상 c_1 다음에 c_2로 이동할 수 있음을 뜻한다. 그러나 Trace 의 간선(edge)은 단순한 실행 순서가 아니라 변수의 정의-사용(definition-use) 관계를 따른다. 즉, 어떤 지점에서 정의된 변수 값이 이후 지점에서 사용되고, 그 사이에 같은 변수가 다시 정의되지 않으면 두 지점은 데이터 의존 관계로 연결된다.

$$\begin{aligned} c_1 ; c_2 \Leftrightarrow \exists\, x \in X.\ & c_1 \rightarrow^{+} c_2 \\ & \wedge\ x \text{ is defined at } c_1 \\ & \wedge\ x \text{ is used at } c_2 \\ & \wedge\ x \text{ is not re-defined between } c_1 \text{ and } c_2 \end{aligned}$$

위 수식에서 $c_1 \rightarrow^{+} c_2$는 제어 흐름상 c_1에서 하나 이상의 step 을 거쳐 c_2에 도달할 수 있음을 의미한다. $c_1 ; c_2$는 어떤 변수 $x \in X$ 가 c_1에서 정의되고 c_2에서 사용되는 데이터 의존 관계를 뜻한다. 경고 $\omega = \langle c_0, c_n \rangle$에 대한 오염 Trace(tainted trace)

집합도 이 관계로 정의된다. 즉 source point c_0에서 sink point c_n까지 인접한 모든 step 이 데이터 의존 관계로 연결된 경로이다.

$$\mathcal{T}\_\omega = \{ \langle c_0, \ldots, c_n \rangle \mid \forall i \in [0, n\text{-}1].\ c_i \,;\, c_{i+1} \}$$

여기서 $\mathcal{T}\_\omega$ 는 경고 ω 에 대해 가능한 오염 Trace 들의 집합이다. 각 trace 는 $\langle c_0, \ldots, c_n \rangle$처럼 제어 지점이 순서대로 나열된 경로이다. 이 수식은 본 연구에서 Trace 를 임의의 코드 라인 목록이 아니라 source–sink 데이터 의존 경로로 보는 근거가 된다. 본 연구는 Tracer 의 특징 벡터(feature vector) 생성과 유사도 순위화(ranking)를 사용하지 않는다. 대신 Infer 가 출력한 bug_trace 의 source, sink, 중간 step 위치를 원본 코드 라인으로 수집하여 모델 입력용 Trace 표현을 구성한다.

2) Infer 기반 Trace 추출 방법

본 연구는 Joern 기반 CPG 도구 [11]와 Infer 정적 오염 분석기 [28]를 Trace 추출 후보로 검토하였다. Joern 은 빌드 환경이 완전하지 않아도 CPG 질의를 적용하기 쉽다. 그러나 실제 CVE 코드에서는 CPG edge 가 누락되거나 source 와 sink 를 명확히 지정하기 어려운 사례가 보고되었다 [27].

이 한계를 고려하여 본 연구는 더 정교한 Trace 를 정적 분석할 수 있는 Infer 를 사용하였다. Infer 는 프로그램 빌드 과정에 삽입되어 정적 분석을 수행하며 [28], Tracer 사례에서도 취약점 관련 정적 분석에 활용되었다 [6].

Infer 기반 Trace 추출 절차는 네 단계이다.

1. **Source/Sink 정의**: 취약점 유형별 source 와 sink 를 Infer 오염 설정에 반영한다.
2. **Infer 오염 분석**: Infer 에 소스 코드를 입력해 source 에서 sink 까지의 전달 여부를 분석하고 bug_trace 가 포함된 JSON 을 출력한다.

3. **Trace 위치 식별**: bug_trace 에서 파일명, 줄 번호, source, 중간 step, sink 위치를 확인한다.

4. **Trace 구성**: 확인한 줄 번호의 원본 코드를 수집해 모델 입력용 Trace 표현으로 만든다.

```
24    #define SNPRINTF snprintf
      void
29    CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_s
      nprintf_01_bad()
30    {
31    char * data;
32    char dataBuffer[100];
33    data = dataBuffer;
34
35    memset(data, 'A', 100-1);
36    data[100-1] = '₩0';
37    {
38    char dest[50] = "";
39
40    SNPRINTF(dest, strlen(data), "%s", data);
41    printLine(data);
42    }
43    }
```

(a) 소스코드 샘플

```
 "bug_trace": [
   {
    "column_number": 5,
    "description": "source of the taint here: value passed as argument
`#0` to `memset` with kind `Simple`",
    "filename":
"CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_snprint
f_01.c",
    "level": 0,
    "line_number": 35
   },
   {
    "column_number": 9,
    "description": "macro expanded here",
    "filename":
"CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_snprint
f_01.c",
    "level": 0,
```

```
      "line_number": 40
    },
    {
      "column_number": -1,
      "description": "flows to this sink: value passed as argument `#3` to
`snprintf` with kind `Simple`",
      "filename":
"CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_snprint
f_01.c",
      "level": 0,
      "line_number": 24
    }
  ],
  "bug_trace_length": 2,
  "bug_trace_max_depth": 0,
  "bug_type": "TAINT_ERROR",
  "bug_type_hum": "Taint Error",
  "category": "Sensitive data flow",
  "column": 9,
  "extras": {
    "taint_extra": {
      "taint_sink": "snprintf",
      "taint_source": "memset",
      "tainted_expression": "data"
    }
  },
  "file":
"CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_snprint
f_01.c",
  "hash": "0370c09bc0def5943c29d2211310a454",
  "line": 40,
  "procedure":
"CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_snprint
f_01_bad",
  "procedure_start_line": 29,
  "qualifier": "Built-in Simple taint kind, matching any Simple source with
any Simple sink except if any Simple sanitizer is in the way. `data` is
tainted by value passed as argument `#0` to `memset` with kind
`Simple` and flows to value passed as argument `#3` to `snprintf` with
kind `Simple`.",
  "severity": "ERROR"
}
```


(b) Infer 가 추출한 bug_trace

그림 3.5 Infer 기반 Source-Sink Trace 추출 예시

그림 3.5 는 Infer 의 bug_trace 를 원본 코드 라인으로 변환하는 예시이다. 35 번 줄의 memset(data, 'A', 100-1)은 추적 대상 값이 생성되는 source 이다. 40 번 줄의 SNPRINTF(dest, strlen(data), "%s", data)는 data 가 문자열 출력 연산에 사용되는 sink 이다. 24 번 줄의 #define SNPRINTF snprintf 는 sink 호출 해석에 필요한 참조 step 으로 처리한다.

Infer 기반 Trace 추출 방법의 한계는 다음과 같다. Infer 기반 정적 분석은 source 에서 sink 까지 이어지는 데이터 흐름을 정교하게 추적할 수 있으나, Infer 는 LLVM 기반 중간 표현을 분석하므로 분석 대상 소스코드를 LLVM 으로 컴파일하는 과정이 필요하다. 또한 taint analysis 를 수행하려면 오염 값이 생성되는 source 와 보안 민감 연산에 해당하는 sink 를 사전에 정의해야 한다. 실제 CVE 코드만으로는 빌드 환경 재현과 source/sink 후보 정의가 불안정해 대규모 Trace 수집이 어렵다. 따라서 본 연구는 Trace 단위 데이터셋 구축 절차를 안정적으로 구성하기 위해 SARD-Juliet 을 활용하였다.

라. SARD-Juliet SA Trace 와 SARD-Juliet SA TracePair

1) SARD-Juliet SA Trace 데이터셋

SARD-Juliet 은 미국 NIST 의 Software Assurance Reference Dataset(SARD)에 포함된 Juliet Test Suite 이다. 이 데이터셋은 정적 분석(Static Analysis, SA) 도구와 취약점 탐지 기법 평가에 널리 사용된다.

본 연구는 Infer 기반 Trace 수집 절차를 규칙 기반으로 자동화하기 위해 SARD-Juliet 을 사용한다. SARD-Juliet 은 실제 CVE 코드보다 빌드가 안정적이며, Juliet 주석과 함수명 규칙으로 Source/Sink 후보를 정할 수 있다. 또한 동일 테스트 케이스(testcase) 안의 bad/good 구조가 고정되어 있다. 여기서 bad/good 은

Juliet 이 취약 예제와 정상 예제를 구분하기 위해 쓰는 함수명 표기이다. 이 구조를 이용하면 취약 Trace 와 정상 Trace 후보를 안정적으로 추출할 수 있다.

SARD-Juliet SA Trace 구축은 세 단계로 진행하였다. 첫째, Juliet 주석과 함수명 패턴으로 Source/Sink 후보를 선정한다. 둘째, 확정한 Source/Sink 와 소스코드를 Infer 에 입력해 bug_trace 를 추출한다. 셋째, 사용자 정의 함수와 변수 식별자를 정규화하고, 중복을 제거한 뒤 테스트 케이스 기준으로 데이터를 분리한다.

후보 유형	줄번호	코드
	23	void CWE121_Stack_Based_Buffer_Overflow__CWE806_char_declare_memcpy_01_bad()
	24	{
	25	char * data;
	26	char dataBuffer[100];
	27	data = dataBuffer;
	28	/* FLAW: Initialize data as a large buffer that is larger than the small buffer used in the sink */
Bad 후보	29	memset(data, 'A', 100-1); /* fill with 'A's */
	30	data[100-1] = '₩0'; /* null terminate */
	31	{
	32	char dest[50] = "";
	33	/* POTENTIAL FLAW: Possible buffer overflow if data is larger than dest */
Bad 후보	34	memcpy(dest, data, strlen(data)*sizeof(char));
	35	dest[50-1] = '₩0'; /* Ensure the destination buffer is null terminated */
	36	printLine(data);
	37	}
	38	}

(a) bad 함수 예시

후보 유형	줄번호	코드
	45	static void goodG2B()
	46	{
	47	char * data;
	48	char dataBuffer[100];
	49	data = dataBuffer;
	50	/* FIX: Initialize data as a small buffer that as small or

```
                smaller than the small buffer used in the sink */
Good 후보  51        memset(data, 'A', 50-1); /* fill with 'A's */
           52        data[50-1] = '₩0'; /* null terminate */
           53        {
           54            char dest[50] = "";
           55            /* POTENTIAL FLAW: Possible buffer overflow if data is
                larger than dest */
Bad 후보   56            memcpy(dest, data, strlen(data)*sizeof(char));
           57            dest[50-1] = '₩0'; /* Ensure the destination buffer is
                null terminated */
           58            printLine(data);
           59        }
           60    }
```

(b) goodG2B 함수 예시

그림 3.6 SARD-Juliet 주석 기반 Source/Sink 후보 지점 식별 예시

그림 3.6 은 Juliet 주석으로 bad/good 후보 라인을 찾는 예시이다. FLAW 와 POTENTIAL FLAW 주석 아래 코드는 bad 후보로, FIX 주석 아래 코드는 good 후보로 본다. 이 후보 라인은 Source/Sink 가 될 수 있지만, 최종 역할은 함수명 패턴까지 확인해 결정한다.

```
역할      줄번호  코드
          121     static void goodB2G2()
          122     {
          123         char * data;
          124         char dataBuffer[100] = "";
          125         data = dataBuffer;
          126         if(staticTrue)
          127         {
          128             {
          129                 /* Read input from a file */
          130                 size_t dataLen = strlen(data);
          131                 FILE * pFile;
          132                 /* if there is room in data, attempt to read the input
                    from a file */
          133                 if (100-dataLen > 1)
          134                 {
          135                     pFile = fopen(FILENAME, "r");
          136                     if (pFile != NULL)
          137                     {
          138                         /* POTENTIAL FLAW: Read data from a file */
```

Bad source	139	if (fgets(data+ dataLen, (int)(100-dataLen), pFile) == NULL)
	140	{
	141	printLine("fgets() failed");
	142	/* Restore NUL terminator if fgets fails */
	143	data[dataLen] = '₩0';
	144	}
	145	fclose(pFile);
	146	}
	147	}
	148	}
	149	}
	150	if(staticTrue)
	151	{
	152	/* FIX: Specify the format disallowing a format string vulnerability */
Good sink	153	printf("%s₩n", data);
	154	}
	155	}

(a) 대표 B2G 함수 goodB2G2 전체 예시

역할	줄번호	코드
	181	static void goodG2B2()
	182	{
	183	char * data;
	184	char dataBuffer[100] = "";
	185	data = dataBuffer;
	186	if(staticTrue)
	187	{
	188	/* FIX: Use a fixed string that does not contain a format specifier */
Good source	189	strcpy(data, "fixedstringtest");
	190	}
	191	if(staticTrue)
	192	{
	193	/* POTENTIAL FLAW: Do not specify the format allowing a possible format string vulnerability */
Bad sink	194	printf(data);
	195	}
	196	}

(b) 대표 G2B 함수 goodG2B2 전체 예시

그림 3.7 SARD-Juliet goodB2G2/goodG2B2 기반 Source/Sink 역할 판정 예시

그림 3.7 은 B2G/G2B 함수에서 Source/Sink 역할이 달라지는 예시이다. B2G/G2B 는 Juliet 함수명에서 bad 후보와 good 후보의 배치 방향을 나타내는 표기이다. B2G 함수에서는 bad 후보 라인이 Source, good 후보 라인이 Sink 가 된다. G2B 함수에서는 good 후보 라인이 Source, bad 후보 라인이 Sink 가 된다.

Infer 오염 설정은 라인 번호가 아니라 함수 호출 단위로 Source 와 Sink 를 정의한다. 따라서 본 연구는 구문 분석 도구인 tree-sitter 로 후보 라인의 함수 호출식(call expression)을 추출하였다. 추출 결과는 Infer 의 Pulse 정적 분석기가 사용하는 오염 설정으로 변환하였다.

두 번째 단계에서는 설정과 소스코드를 Infer 에 입력해 bug_trace 를 추출하였다.

세 번째 단계에서는 Trace 를 모델 입력 전에 정규화하였다. 정규화는 사용자 정의 변수명이나 함수명처럼 취약 여부와 직접 관련 없는 식별자 의존을 줄이기 위한 절차이다. 사용자 정의 변수와 함수는 일반 토큰으로 바꾸고, 취약 판단에 필요한 API 호출명과 상수는 유지하였다.

```
1  data = (char *)malloc(50*sizeof(char));
2  memcpy(dest, data, strlen(dest)*sizeof(char));
3  printLine(dest);
4  printf("%s₩n", line);
```

(a) 정규화 전 취약 Trace

```
1  VAR_1 = (char *)malloc(50*sizeof(char));
2  memcpy(VAR_2, VAR_1, strlen(VAR_2)*sizeof(char));
3  FUNC_1(VAR_2);
4  printf("%s₩n", VAR_3);
```

(b) 정규화 후 취약 Trace

```
1  data = (char *)malloc(100*sizeof(char));
2  memcpy(dest, data, strlen(dest)*sizeof(char));
3  printLine(dest);
4  printf("%s₩n", line);
```

(c) 정규화 전 정상 Trace

```
1  VAR_1 = (char *)malloc(100*sizeof(char));
2  memcpy(VAR_2, VAR_1, strlen(VAR_2)*sizeof(char));
3  FUNC_1(VAR_2);
4  printf("%s₩n", VAR_3);
```

(d) 정규화 후 정상 Trace

그림 3.8 SARD-Juliet SA Trace 정규화 예시

그림 3.8 에서 정규화 후 사용자 정의 식별자는 VAR_1, VAR_2, VAR_3, FUNC_1 로 바뀐다. 반면 malloc, memcpy, strlen, printf 와 할당 크기 차이는 유지된다. 따라서 식별자를 제거해도 취약/정상 Trace 를 구분하는 핵심 차이가 남는다.

정규화 후에는 동일 라벨의 중복 Trace 를 하나만 유지하였다. 또한 동일 테스트 케이스에서 생성된 Trace 가 학습 데이터셋과 테스트 데이터셋에 동시에 포함되지 않도록 테스트 케이스 기준 8:2 비율로 분리하였다.

2) SARD-Juliet SA TracePair 데이터셋

SARD-Juliet SA Trace 는 모델 학습에 사용할 개별 Trace 데이터셋이다. 이에 비해 SARD-Juliet SA TracePair 는 같은 테스트 케이스에서 대응되는 취약 Trace 와 정상 Trace 를 한 쌍으로 평가하여, 모델이 취약 코드 패턴을 학습했는지 확인하기 위해 구축한 평가 데이터셋이다.

이 데이터셋은 동일 테스트 케이스의 취약 Trace 와 정상 Trace 를 1:1 로 묶은 평가 세트이다. 정상 Trace 추출은 취약 Trace 와 같은 절차를 따르되, Source/Sink 후보 라인만 대응되는 good 계열 함수의 FIX 주석 아래 정상 코드에서 고른다. goodB2G/goodG2B 함수는 이름의 방향대로 Source 와 Sink 역할을 정하고, 단일 good 함수는 앞선 후보 라인을 Source, 뒤 후보 라인을 Sink 로 사용한다. 이후 bug_trace 추출, 정규화, 중복 제거는 취약 Trace 와 동일하게 적용하였다. 선택된 테스트 케이스의 모든 행은 학습용 SARD-Juliet SA

Trace 에서 제거해 정보 유출을 방지하였다. 그림 3.2 와 그림 3.3 의 예시처럼, 이 쌍 구성은 함수 단위 샘플이 아니라 Source 에서 Sink 까지 이어지는 Trace 를 사용한다.

마. 10 개 CVE 에 대한 TracePair

SARD-Juliet SA TracePair 는 source 에서 sink 까지 이어지는 데이터 흐름을 명확히 정의할 수 있다. 그러나 SARD-Juliet 은 합성 데이터셋이므로 전역 변수, 구조체 필드, 함수 포인터처럼 실제 CVE 코드의 복잡한 흐름을 충분히 반영하지 못할 수 있다.

본 연구는 실제 오픈소스 CVE 10 건에서 취약 Trace 1 개와 정상 Trace 1 개를 대응시켜 총 10 쌍의 TracePair 를 구축하였다. 목적은 SARD-Juliet SA Trace 로 학습한 모델이 실제 CVE 의 취약 Trace 와 정상 Trace 도 구분할 수 있는지 평가하는 것이다.

먼저 CWE Top 25 2024 에 포함된 항목 중 C/C++ 패치 코드로 source 와 sink 를 확인하고 TracePair 를 구성할 수 있는 후보를 검토하였다 [29]. 1 차 후보는 CWE-78, CWE-89, CWE-400 이었다. CWE-78 과 CWE-400 은 이 기준에 부합해 유지하였다. CWE-89 는 데이터베이스 질의 실행 문맥 의존성이 커 본 연구의 C/C++ 패치 코드 기반 source-sink 대응 구성에 적합하지 않았다 [30]. 대신 CWE-134 는 포맷 문자열 전달 흐름을 C/C++ 코드에서 확인할 수 있어 대체 유형으로 포함하였다 [31]. 최종 평가 유형은 CWE-78, CWE-134, CWE-400 이다.

CVE 후보는 Big-Vul 에서 해당 CWE 의 오픈소스 CVE 를 검토해 선정하였다 [26]. 공개 CVE 설명, 패치 전후 코드, 함수 호출 흐름만으로 source, sink, 취약 결과를 확인할 수 있는 사례로 제한했고, 선행 사례 분석의 11 개 CVE 중 Infer

기반 Trace 추출에 필요한 컴파일 환경을 재현하기 어려운 HTCondor CVE-2011-4930 은 제외하였다 [27]. 최종 10 개 CVE 는 표 3.1 에 제시하였다.

표 3.1 10 개 CVE 에 대한 TracePair 구축에 활용한 CVE 와 Source/Sink

CWE	CVE	오픈소스 SW	Source	Sink
CWE-134 (Format String Bug)	CVE-2015-8617	PHP (인터프리터)	zend_throw_or_error()	zend_vspprintf()
	CVE-2017-12588	Rsyslog (로그 수집)	actionPrepare()	zsocket_bind()
CWE-400 (Resource Exhaustion)	CVE-2017-11142	PHP (HTTP POST 처리)	php_stream_read()	memchr()
	CVE-2019-12973	OpenJPEG (JPEG 코덱)	bmptoimage()	opj_t1_encode_cblks()
CWE-78 (OS Command Injection)	CVE-2017-15108	Spice Vdagent (VM 게스트 에이전트)	udscs_do_read()	system()
	CVE-2017-15924	Shadowsocks-libev (암호화 프록시)	recvfrom()	system()
	CVE-2018-6791	KDE Plasma Workspace (장치 관리자)	showActionsDialog()	runCommand()
	CVE-2018-16863	Ghostscript (문서 뷰어)	param_read_string()	popen()
	CVE-2019-13638	Patch (패치 프로그램)	xstrdup()	execl()
	CVE-2019-16718	radare2 (바이너리 분석 도구)	r_bin_get_symbols()	system()

10 개 CVE 에 대한 TracePair 는 Infer 를 활용한 수작업 Trace 재구성 방식으로 만들었다. Infer 는 구간별 데이터 흐름 추출에 사용하였다. source/sink 후보 선정,

전체 경로 정의, 구간별 source/sink 갱신, 구간 재연결 검증은 코드 검토에 기반해 수행하였다.

절차는 세 단계이다. 첫째, 각 CVE 의 취약 코드를 분석해 전체 Trace 의 source/sink 후보를 정하였다. 둘째, 취약 입력이 최종 sink 까지 전달되는 전체 경로를 먼저 정의하였다. 셋째, Infer 를 반복 실행해 구간별 Trace 를 추출하고, 구간 사이의 데이터 의존성을 확인해 하나의 Trace 로 재연결하였다.

첫째, source/sink 후보 선정 단계에서는 CVE 설명, 패치 전후 코드, 취약 함수 호출 흐름을 함께 검토하였다. Source 는 외부 입력이나 공격자가 제어할 수 있는 값이 처음 유입되는 위치로 정하였다. Sink 는 그 값이 보안 민감 연산에 사용되는 위치로 정하였다. 정상 Trace 는 같은 입력 전달 흐름에서 취약 sink 가 안전한 호출이나 검증된 사용으로 바뀌는 위치를 기준으로 구성하였다. 표 3.1 의 Source 와 Sink 는 이 기준에 따라 각 CVE 에서 수작업으로 정리한 Infer 입력 시작점과 도착점이다.

둘째, 전체 경로 정의 단계에서는 source 에서 최종 sink 까지 값이 전달되는 순서를 수동으로 정의하였다. 전역 변수, 구조체 필드, 함수 포인터, 함수 간 전달이 포함된 사례에서는 Infer 가 전체 Trace 를 한 번에 출력하지 못할 수 있으므로, 구간별 추출 결과를 재연결할 수 있도록 전체 경로를 먼저 정하였다. 그림 3.9 와 그림 3.10 은 전체 경로 정의가 필요한 대표 예시이다.

단계	코드
Source	outfile = xstrdup (optarg);
Step1	outname = outfile;
Step2	outfd = make_tempfile (&TMPOUTNAME, 'o', outname, O_WRONLY | binary_transput, instat.st_mode & S_IRWXUGO);
Step3	int make_tempfile (char const **name, char letter, char const *real_name, int flags, mode_t mode)
Step4	dirname = dir_name (real_name);
Step5	sprintf (template, '%s/%s.%cXXXXXX', dirname, basename, letter);
Step6	*name = template;

Step7	do_ed_script (inname, TMPOUTNAME, &TMPOUTNAME_needs_removal, outstate.ofp);
Step8	void do_ed_script (char const *inname, char const *outname, bool *outname_needs_removal, FILE *ofp)
Step9	sprintf (buf, '%s %s%s', editor_program, verbosity == VERBOSE ? '' : '- ', outname);
Sink	execl ('/bin/sh', 'sh', '-c', buf, (char *) 0);

(a) 취약 Trace

단계	코드
Source	outfile = xstrdup (optarg);
Step1	outname = outfile;
Step2	outfd = make_tempfile (&TMPOUTNAME, 'o', outname, O_WRONLY | binary_transput, instat.st_mode & S_IRWXUGO);
Step3	int make_tempfile (char const **name, char letter, char const *real_name, int flags, mode_t mode)
Step4	dirname = dir_name (real_name);
Step5	sprintf (template, '%s/%s.%cXXXXXX', dirname, basename, letter);
Step6	*name = template;
Step7	do_ed_script (inname, TMPOUTNAME, &TMPOUTNAME_needs_removal, outstate.ofp);
Step8	void do_ed_script (char const *inname, char const *outname, bool *outname_needs_removal, FILE *ofp)
Sink	execlp (editor_program, editor_program, '-', outname, (char *) NULL);

(b) 정상 Trace

그림 3.9 Patch (CVE-2019-13638) Trace

그림 3.9 는 Patch (CVE-2019-13638)의 OS command injection Trace 예시이다. 취약 Trace 는 파일명이 TMPOUTNAME 을 거쳐 셸 명령 문자열 buf 로 구성된 뒤 /bin/sh -c 로 실행되는 흐름을 담는다. 정상 Trace 는 같은 파일명 전달 흐름을 공유하지만, 셸 명령 문자열을 만들지 않고 execlp 에 인자를 직접 전달한다. 이 사례는 전역 변수와 함수 간 전달을 포함하므로 Infer 의 전체 Trace 추출이 제한될 수 있어, 전체 경로를 먼저 정의하였다.

단계	코드
Source	udscs_do_read(&conn);
Step1	static void udscs_do_read(struct udscs_connection **conp)
Step2	udscs_read_complete(connp);
Step3	conn->read_callback(connp, &conn->header, conn->data.buf);
Step4	static void daemon_read_complete(struct udscs_connection **connp,

	struct udscs_message_header *header, uint8_t *data)
Step5	VDAgent *agent = udscs_get_user_data(*connp);
Step6	void *udscs_get_user_data(struct udscs_connection *conn)
Step7	return conn->user_data;
Step8	vdagent_file_xfers_data(agent->xfers, (VDAgentFileXferDataMessage *)data);
Step9	void vdagent_file_xfers_data(struct vdagent_file_xfers *xfers, VDAgentFileXferDataMessage *msg)
Step10	snprintf(buf, PATH_MAX, "xdg-open '%s'&", xfers->save_dir);
Sink	status = system(buf);

(a) 취약 Trace

단계	코드
Source	udscs_do_read(&conn);
Step1	static void udscs_do_read(struct udscs_connection **conp)
Step2	udscs_read_complete(connp);
Step3	conn->read_callback(connp, &conn->header, conn->data.buf);
Step4	static void daemon_read_complete(struct udscs_connection **connp, struct udscs_message_header *header, uint8_t *data)
Step5	VDAgent *agent = udscs_get_user_data(*connp);
Step6	void *udscs_get_user_data(struct udscs_connection *conn)
Step7	return conn->user_data;
Step8	vdagent_file_xfers_data(agent->xfers, (VDAgentFileXferDataMessage *)data);
Step9	void vdagent_file_xfers_data(struct vdagent_file_xfers *xfers, VDAgentFileXferDataMessage *msg)
Step10	gchar *argv[] = { "xdg-open", xfers->save_dir, NULL };
Sink	if (!g_spawn_async(NULL, argv, NULL, G_SPAWN_SEARCH_PATH, NULL, NULL, NULL, &error))

(b) 정상 Trace

그림 3.10 Spice Vdagent (CVE-2017-15108) Trace

그림 3.10 은 Spice Vdagent (CVE-2017-15108)의 OS command injection Trace 예시이다. 취약 Trace 는 수신 데이터가 콜백 함수와 구조체 필드를 거쳐 xfers->save_dir 에 도달하고, 이후 system 호출로 이어지는 흐름을 담는다. 정상 Trace 는 같은 입력 전달 흐름을 공유하지만 argv 배열을 구성해 g_spawn_async 에 전달한다. 이 사례는 함수 포인터 호출과 구조체 필드 접근을 포함하므로 Infer 의 전체 Trace 추출이 제한될 수 있어, 전체 경로를 먼저 정의하였다.

셋째, 구간 추출과 재연결 단계에서는 먼저 전체 경로의 시작 source 와 최종 sink 를 Infer 설정으로 입력하였다. 전체 Trace 가 출력되지 않으면 최종 sink 의 직전 step 을 sink 로 지정해 다시 실행하고, 결과가 나올 때까지 sink 를 한 step 씩 앞당겼다. 이 방식으로 현재 source 에서 도달 가능한 가장 긴 구간 Trace 를 추출한 뒤, 추출 구간의 sink 직후 step 을 다음 source 로 지정하였다. 이후 다시 최종 sink 를 목표로 같은 과정을 반복해 다음 구간을 추출하였다. 각 구간은 앞 구간의 마지막 값이 다음 구간의 입력 인자, 반환값, 전역 변수, 포인터 또는 필드 접근으로 이어지는지 확인해 연결하였다. 연결이 불명확한 구간은 Trace 에 포함하지 않았다.

표 3.2 부터 표 3.5 까지는 둘째 단계에서 정의한 그림 3.9 와 그림 3.10 의 Trace 를 셋째 절차로 분할 추출한 구간과 재연결 근거를 정리한다. 각 표의 구간 시작점과 종료점은 구간별 Infer 추출에 사용한 source/sink 경계이다.

표 3.2 CVE-2019-13638 취약 Trace 분할 추출 구간별 시작점/종료점 예시

구간	취약 Trace 구간	구간 시작점	구간 종료점	데이터 의존 연결 방식
1	[Source] → [Step2]	xstrdup	make_tempfile	xstrdup 의 반환값이 전역 변수 outfile 에 저장되고, outname = outfile;를 통해 지역 변수 outname 으로 별칭 연결된 뒤 make_tempfile 의 real_name 인자로 전달되는지를 확인한다.
2	[Step2] → [Step4]	make_tempfile	dir_name	make_tempfile 호출에서 전달된 outname 이 호출 대상 함수의 real_name 으로 바인딩되고, dir_name (real_name)의 인자로 사용되는지를 확인한다.
3	[Step4] → [Step5]	dir_name	sprintf	real_name 에서 계산된 디렉터리 정보가 dirname 에 저장되고, 이 값이

				sprintf 에서 임시 파일명 문자열 template 을 구성하는 인자로 사용되는지를 확인한다.
4	[Step5] → [Step7]	make_tempfile	do_ed_script	sprintf 가 구성한 template 이 *name = template;로 저장되고, 호출자가 &TMPOUTNAME 을 넘겼기 때문에 이 값이 전역 변수 TMPOUTNAME 으로 갱신된 뒤 do_ed_script 의 두 번째 인자로 전달되는지를 코드 해석으로 연결한다.
5	[Step7] → [Step9]	do_ed_script	sprintf	호출자의 TMPOUTNAME 이 do_ed_script 의 형식 매개변수 outname 으로 바인딩되고, 이후 sprintf 의 outname 인자로 사용되어 셸 명령 문자열 buf 를 구성하는지를 확인한다.
6	[Step9] → [Sink]	sprintf	execl	sprintf 가 셸 명령 문자열을 전역 버퍼 buf 에 기록하고, execl 이 같은 buf 를 /bin/sh -c 의 명령 인자로 소비하는지를 코드 해석으로 연결한다.

표 3.3 CVE-2019-13638 정상 Trace 분할 추출 구간별 시작점/종료점 예시

구간	정상 Trace 구간	구간 시작점	구간 종료점	데이터 의존 연결 방식
1	[Source] → [Step2]	xstrdup	make_tempfile	취약 Trace 와 동일하게 xstrdup 의 반환값이 전역 변수 outfile 에 저장되고, outname = outfile;를 통해 지역 변수 outname 으로 별칭 연결된 뒤 make_tempfile 의 real_name 인자로 전달되는지를 확인한다.
2	[Step2] → [Step4]	make_tempfile	dir_name	make_tempfile 호출에서 전달된 outname 이 호출 대상 함수의 real_name 으로 바인딩되고, dir_name

				(real_name)의 인자로 사용되는지를 확인한다.
3	[Step4] → [Step5]	dir_name	sprintf	real_name 에서 계산된 디렉터리 정보가 dirname 에 저장되고, 이 값이 sprintf 에서 임시 파일명 문자열 template 을 구성하는 인자로 사용되는지를 확인한다.
4	[Step5] → [Step7]	make_tempfile	do_ed_script	sprintf 가 구성한 template 이 *name = template;로 저장되고, 전역 변수 TMPOUTNAME 으로 갱신된 뒤 do_ed_script 의 두 번째 인자로 전달되는지를 코드 해석으로 연결한다.
5	[Step7] → [Sink]	do_ed_script	execlp	호출자의 TMPOUTNAME 이 do_ed_script 의 형식 매개변수 outname 으로 바인딩되고, 정상 Trace 에서는 이 값이 셸 명령 문자열을 거치지 않고 execlp 의 outname 인자로 직접 전달되는지를 확인한다.

표 3.4 와 표 3.5 는 같은 기준으로 그림 3.10 의 Spice Vdagent CVE-2017-15108 TracePair 를 정리한다.

표 3.4 CVE-2017-15108 취약 Trace 분할 추출 구간별 시작점/종료점 예시

구간	취약 Trace 구간	구간 시작점	구간 종료점	데이터 의존 연결 방식
1	[Source] → [Step2]	udscs_do_read	udscs_read_complete	udscs_do_read(&conn);에서 전달된 connection 포인터가 udscs_read_complete(connp); 호출로 이어지는지를 확인한다.
2	[Step2] → [Step4]	udscs_read_complete	daemon_read_complete	udscs_read_complete 내부의 conn->read_callback(connp, &conn->header, conn->data.buf);가 구조체 필드에 저장된

				콜백을 통해 daemon_read_complete 를 호출하는지를 코드 해석으로 연결한다.
3	[Step4] → [Step5]	daemon_read_complete	udscs_get_user_data	daemon_read_complete 에 전달된 connp 가 udscs_get_user_data(*connp); 호출의 입력으로 사용되는지를 확인한다.
4	[Step5] → [Step8]	udscs_get_user_data	vdagent_file_xfers_data	udscs_get_user_data 가 반환한 conn->user_data 가 agent 로 해석되고, agent->xfers 가 vdagent_file_xfers_data 의 첫 번째 인자로 전달되는지를 코드 해석으로 연결한다.
5	[Step8] → [Step10]	vdagent_file_xfers_data	snprintf	vdagent_file_xfers_data 내부에서 xfers->save_dir 가 snprintf(buf, PATH_MAX, "xdg-open '%s'&", xfers->save_dir);의 인자로 사용되는지를 확인한다.
6	[Step10] → [Sink]	snprintf	system	snprintf 가 셸 명령 문자열을 buf 에 기록하고, system(buf);가 같은 buf 를 셸 명령으로 실행하는지를 확인한다.

표 3.5 CVE-2017-15108 정상 Trace 분할 추출 구간별 시작점/종료점 예시

구간	정상 Trace 구간	구간 시작점	구간 종료점	데이터 의존 연결 방식
1	[Source] → [Step2]	udscs_do_read	udscs_read_complete	취약 Trace 와 동일하게 udscs_do_read(&conn);에서 전달된 connection 포인터가 udscs_read_complete(connp); 호출로 이어지는지를 확인한다.
2	[Step2	udscs_read_compl	daemon_read_co	conn-

	] → [Step4]	ete	mplete	>read_callback(connp, &conn->header, conn->data.buf);가 콜백 필드를 통해 daemon_read_complete 를 호출하는지를 코드 해석으로 연결한다.
3	[Step4] → [Step5]	daemon_read_complete	udscs_get_user_data	daemon_read_complete 에 전달된 connp 가 udscs_get_user_data(*connp); 호출의 입력으로 사용되는지를 확인한다.
4	[Step5] → [Step8]	udscs_get_user_data	vdagent_file_xfers_data	udscs_get_user_data 가 반환한 conn->user_data 가 agent 로 해석되고, agent->xfers 가 vdagent_file_xfers_data 의 첫 번째 인자로 전달되는지를 코드 해석으로 연결한다.
5	[Step8] → [Sink]	vdagent_file_xfers_data	g_spawn_async	정상 Trace 에서는 xfers->save_dir 가 gchar *argv[] = { "xdg-open", xfers->save_dir, NULL };의 argv[1]로 들어가고, 같은 argv 배열이 g_spawn_async 에 전달되는지를 코드 해석으로 연결한다.

10 개 CVE 에 대한 TracePair 는 실제 오픈소스 CVE 에서 Trace 표현의 적용 가능성을 확인하기 위한 소규모 평가 데이터셋이다. 각 쌍은 source/sink 와 Trace 경로를 수작업으로 정의하고, Infer 에서 전체 경로가 출력되지 않을 때 sink 를 단계적으로 앞당기는 반복 절차로 가장 긴 구간 Trace 를 추출한 뒤, 각 구간을 데이터 의존성 기준으로 재연결해 구성하였다. 또한 취약 Trace 와 정상 Trace 를 직접 검토하여 시간 흐름에 따른 라벨 충돌이 평가 쌍에 포함되지 않도록 확인하였다.

바. 연구 질문

본 연구는 두 RQ 를 설정한다. RQ1 은 패치 기반 함수 단위 라벨링이 취약 함수와 정상 함수 구분을 어떻게 제약하는지 확인한다. 이 질문은 작은 패치 차이, 시간 흐름에 따른 라벨 충돌, 넓은 취약 라벨 범위가 모델 판단에 남기는 영향을 다룬다.

RQ2 는 함수 단위 데이터셋과 Trace 단위 데이터셋에서 취약/정상 구분 양상이 어떻게 달라지는지 확인한다. Trace 단위 데이터셋은 Source 에서 Sink 까지 이어지는 핵심 코드 흐름으로 샘플을 구성한다. 본 연구는 딥러닝 모델과 LLM 이 두 데이터셋 구성 단위에서 보이는 취약/정상 구분 성능을 비교한다.

RQ1. 패치 기반 함수 단위 라벨링은 취약/정상 함수 구분에 어떤 한계를 만드는가?

RQ2. 함수 단위 데이터셋 평가와 Trace 단위 데이터셋 평가에서 딥러닝 및 LLM 기반 취약/정상 구분 성능은 어떻게 달라지는가?

4. 실험 및 분석

4 장은 3 장에서 정리한 RQ 에 대한 실험 결과와 분석을 제시한다. 함수 단위 데이터셋에서는 높은 함수 단위 평가 점수가 동일 CVE 의 취약/정상 함수 쌍 구분으로 이어지지 않았다. SARD-Juliet SA TracePair 에서는 높은 구분 성능을 보였다. 그러나 10 개 CVE 에 대한 TracePair 에서는 같은 수준으로 재현되지 않았다.

가. 실험 설정

1) 데이터셋

표 4.1 은 RQ 별 데이터셋 역할을 정리한다. RQ1 은 Extended RealVul 로 학습과 예시 검색을 수행하고, Extended RealVul-FuncPair 로 평가한다. RQ2 는 SARD-Juliet SA Trace 로 학습과 예시 검색을 수행하고, 두 TracePair 로 평가한다.

표 4.1 RQ 별 데이터셋 및 평가 구성

RQ	학습 및 예시 검색	평가 데이터셋	결과 초점
RQ1	Extended RealVul	Extended RealVul-FuncPair	패치 기반 함수 라벨의 영향과 동일 CVE 함수 쌍 구분
RQ2	SARD-Juliet SA Trace	SARD-Juliet SA TracePair, 10 개 CVE 에 대한 TracePair	Trace 단위 데이터셋의 취약/정상 구분과 실제 CVE 적용 양상

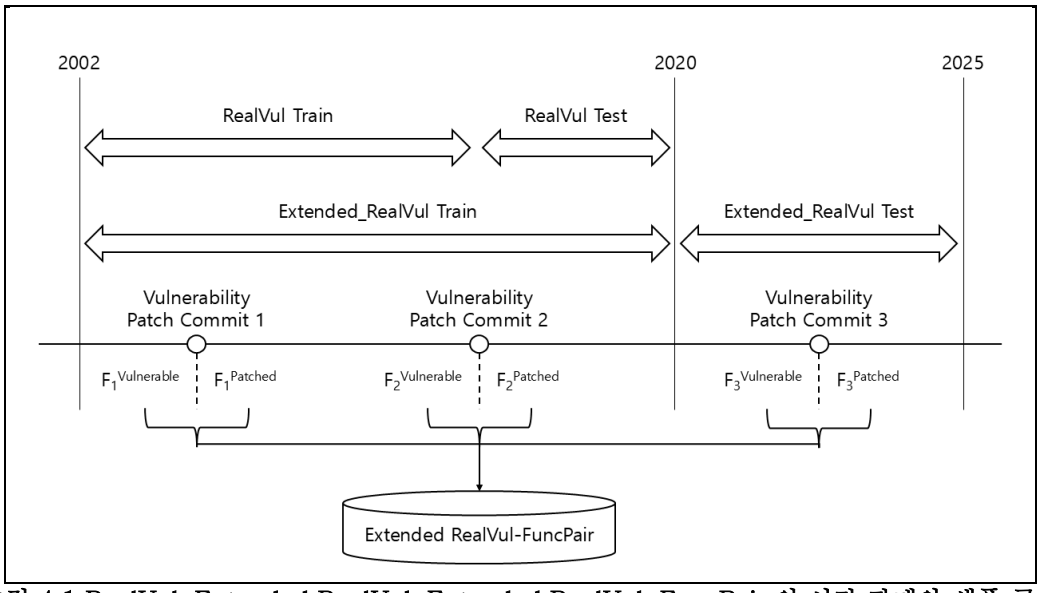


그림 4.1 RealVul, Extended RealVul, Extended RealVul-FuncPair 의 시간 관계와 샘플 구성

그림 4.1 은 RealVul, Extended RealVul, Extended RealVul-FuncPair 의 관계를 보여준다. Extended RealVul 의 학습 데이터셋은 RQ1 딥러닝 학습에 사용하고, 테스트 데이터셋은 LLM 예시 검색에 사용한다. Extended RealVul-FuncPair 는 동일 CVE 의 취약 함수와 정상 함수를 묶은 RQ1 평가 데이터셋이다.

SARD-Juliet SA Trace 는 Trace 기반 모델 학습과 LLM 예시 검색에 사용한다. SARD-Juliet SA TracePair 는 SARD-Juliet 조건의 RQ2 평가 데이터셋이고, 10 개 CVE 에 대한 TracePair 는 실제 오픈소스 CVE 조건의 RQ2 평가 데이터셋이다.

2) 딥러닝 기반 모델 평가 설정

딥러닝 기반 평가는 각 데이터셋을 함수 또는 Trace 샘플로 전처리한 뒤, 모델이 각 샘플을 취약/정상으로 분류하도록 구성하였다. 그림 4.2 는 데이터셋, 전처리, 모델, 예측으로 이어지는 공통 흐름을 나타낸다.

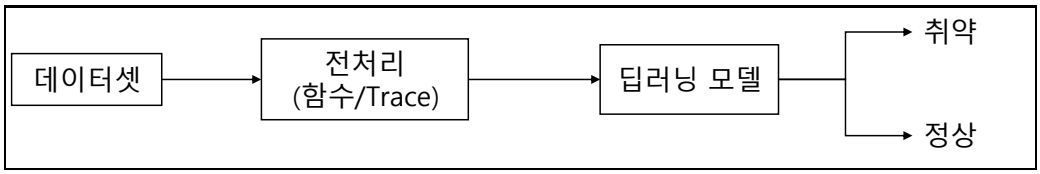


그림 4.2 딥러닝 기반 모델 평가 절차

Extended RealVul-FuncPair 에는 DeepWukong [2], LineVul [3], PDBERT [4]를 적용하였다. 이 데이터셋은 동일 CVE 의 취약 함수와 정상 함수를 포함하므로 함수 단위 구분 성능을 평가하는 데 사용하였다.

SARD-Juliet SA TracePair 와 10 개 CVE 에 대한 TracePair 에는 LineVul 과 PDBERT 를 적용하였다. 두 모델은 코드 토큰 시퀀스를 처리할 수 있어 Trace 형태의 부분 코드에도 적용할 수 있다. DeepWukong 은 그래프 기반 전처리를 전제로 하므로 Trace 단위 평가에는 사용하지 않았다.

3) LLM 기반 모델 평가 설정

LLM 평가는 딥러닝 모델에서 관찰한 취약/정상 구분 차이를 다른 모델 계열에서도 비교해 해석을 보강하기 위한 실험이다. 구체적으로 그 차이가 특정 딥러닝 모델 구조에 한정되는지, 데이터셋 단위 차이와 함께 나타나는지 확인한다. 본 연구는 별도 파인튜닝 없이 상용 LLM 에 같은 평가 샘플을 제시하고 취약/정상 분류 결과를 비교하였다.

평가는 예측 대상과 유사한 입력-라벨 예시를 검색해 분류 프롬프트에 넣는 방식으로 수행하였다. 이 방식은 RAG 와 소수 예시 기반 프롬프팅(Few-shot prompting)을 결합한다. 제로샷 프롬프팅(zero-shot prompting)은 관련 연구에서 한계가 보고되었으므로 [20], 본 연구는 RAG 기반 Few-shot 만 평가하였다.

LLM 평가의 주요 설정은 아래 요약표와 같다.

표 4.2 LLM 평가 설정 요약

구분	설정	내용
LLM	평가 모델	gpt-4o, claude-sonnet-4-5-20250929, gemini-2.5-pro, grok-4-1-fast-non-reasoning
	temperature	Claude 를 제외한 나머지 모델은 0.7 을 사용하고, Claude 는 Anthropic Messages API 의 기본값인 1.0 을 사용
	반복 평가	각 입력을 5 회 평가하고 평균 지표를 제시
	출력 라벨 및 해석	Safe/Vulnerable 만 추출해 각각 정상/취약으로 해석
RAG	검색 예시 수	예측 대상마다 유사한 입력-라벨 예시 6 개 검색
	검색 임베딩	코드 표현 사전학습 모델인 CodeBERT [23]로 코드와 Trace 를 벡터 표현으로 변환하고, 검색 저장소와 예측 대상을 같은 벡터 공간에 배치
	검색 임베딩 입력 길이	CodeBERT 검색 질의 기준 하위 단어 토큰(subword token) 앞 512 개

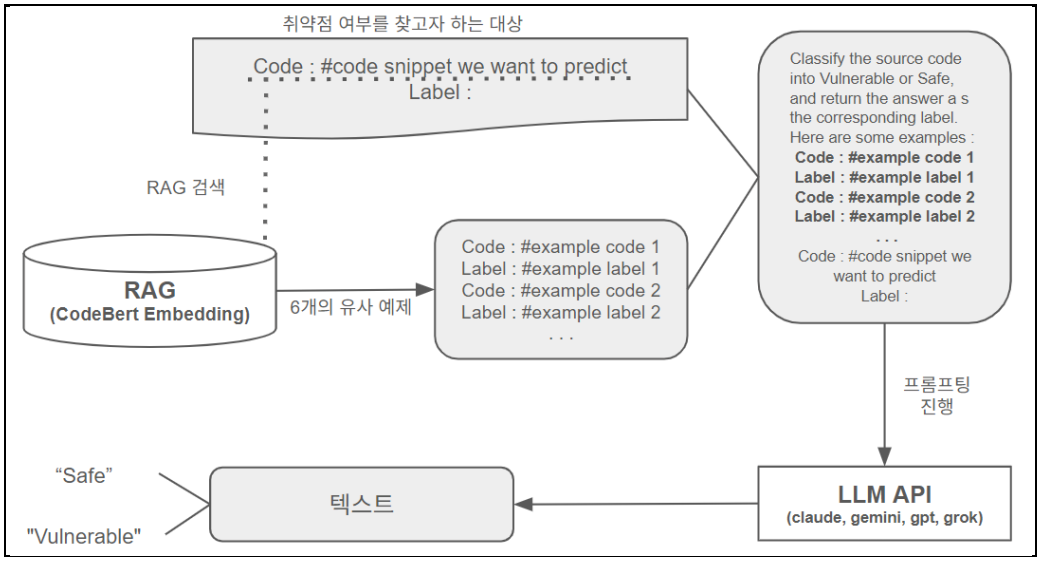


그림 4.3 RAG 기반 Few-shot LLM 취약점 분류 절차

그림 4.3 은 위 설정에 따른 RAG 기반 Few-shot 분류 절차를 나타낸다. 예측 대상과 가장 유사한 입력-라벨 예시를 선택해 분류 지시문과 함께 LLM 에 제공하였다. 프롬프트 템플릿과 예시 삽입 형식은 부록 B 에 제시한다.

각 실험은 RAG 검색 DB 를 다르게 두었다. Extended RealVul-FuncPair 평가는 Extended RealVul 테스트 데이터셋을 검색 저장소로 사용하였다. SARD-Juliet

SA TracePair 와 10 개 CVE TracePair 평가는 SARD-Juliet SA Trace 데이터셋을 검색 저장소로 사용하였다.

나. RQ1. 패치 기반 함수 단위 라벨링은 취약/정상 함수 구분에 어떤 한계를 만드는가?

RQ1 은 함수 단위 라벨이 취약/정상 함수 구분을 어떻게 제한하는지 확인한다. 먼저 Extended RealVul 에서 기존 함수 단위 분류 결과를 확인한다. 이어서 동일 CVE 의 취약 함수와 정상 함수를 묶은 Extended RealVul-FuncPair 에서 높은 함수 단위 점수가 실제 쌍 구분으로 이어지는지 검토한다. 본 절에서 정밀도(Precision)는 취약으로 예측한 샘플 중 실제 취약 샘플의 비율, 재현율(Recall)은 실제 취약 샘플 중 취약으로 올바르게 예측한 비율, F1 은 두 지표의 조화평균을 뜻한다.

표 4.3 Extended RealVul 기반 함수 단위 분류 결과

Model	TN	FP	FN	TP	Precision	Recall	F1
DeepWukong	2948824	241	2780	4	0.016	0.0014	0.0026
PDBERT	73289	0	3	363	1.00	0.992	0.996
LineVul	399803	0	437	0	0.00	0.00	0.00

표 4.3 에서 DeepWukong 은 정상 함수를 대부분 정상으로 분류했지만, 취약 함수를 올바르게 찾은 사례가 4 개에 그쳐 F1 0.0026 에 머물렀다. PDBERT 는 정상 함수를 취약으로 잘못 분류한 사례가 없고 놓친 취약 함수가 3 개뿐이어서 F1 0.996 의 높은 함수 단위 점수를 보였다. LineVul 은 정상 함수를 취약으로 잘못 분류하지는 않았지만 취약 함수를 올바르게 찾은 사례도 없어 F1 0.00 에 머물렀다.

표 4.4 Extended RealVul-FuncPair 에서의 딥러닝 기반 취약 함수와 정상 함수 구분 결과

모델	취약 예측	정상 예측	TP	TN	FP	FN	총
DeepWukong	124 (2%)	5822 (98%)	75	3432	49	2390	5946
LineVul	0 (0%)	8730 (100%)	0	4365	0	4365	8730
PDBERT	8679 (99%)	51 (1%)	4365	51	4314	0	8730

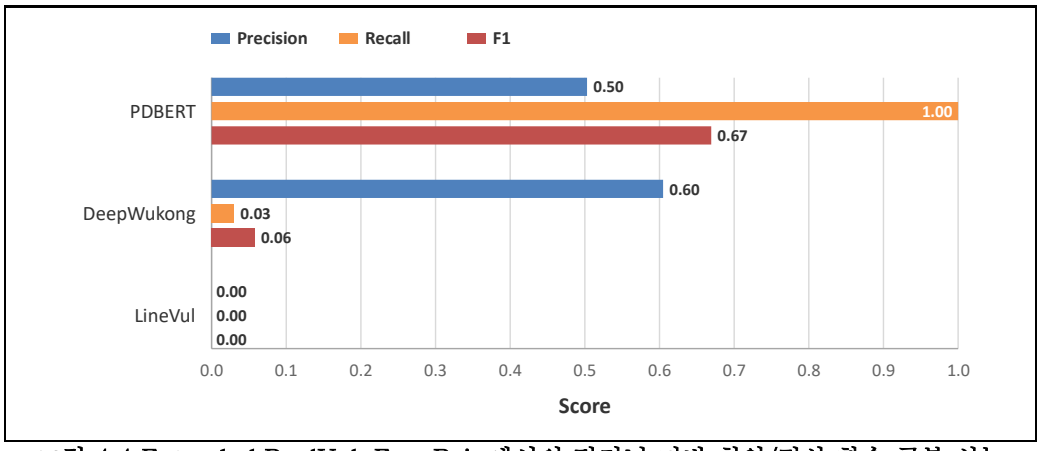


그림 4.4 Extended RealVul-FuncPair 에서의 딥러닝 기반 취약/정상 함수 구분 성능

표 4.4 와 그림 4.4 는 취약 함수와 정상 함수 쌍에 대한 딥러닝 모델의 구분 결과를 보인다. 세 모델 모두 구분이 불안정했다. PDBERT 는 재현율 1.00 과 F1 0.67 을 보였지만, 정밀도는 0.50 에 머물렀다. 이는 8730 개 중 8679 개를 취약으로 예측해 정상 함수 4314 개를 오분류했기 때문이다. DeepWukong 은 정밀도 0.60 이었으나 재현율이 0.03 으로 낮았고, LineVul 은 모든 샘플을 정상으로 예측해 정밀도, 재현율, F1 이 모두 0.00 이었다. 따라서 표 4.3 의 높은 함수 단위 점수는 취약/정상 함수 쌍 구분 능력을 보장하지 않았다.

t-분포 확률적 이웃 임베딩(t-distributed stochastic neighbor embedding, t-SNE)은 고차원 임베딩을 2 차원 공간에 투영해 샘플 분포를 확인하는 방법이다. 본 연구에서는 앞서 표 4.4 와 그림 4.4 에서 정밀도, 재현율, F1 로 확인한 실험 결과를 해석하기 위해 t-SNE 를 사용했다. 구체적으로 표 4.4 와 그림 4.4 에서 나타난 취약/정상 함수 구분 한계가 모델 임베딩 분포에서도 관찰되는지 확인했다. 그림 4.5 와 그림 4.6 에서는 라벨 분리가 뚜렷하지 않다.

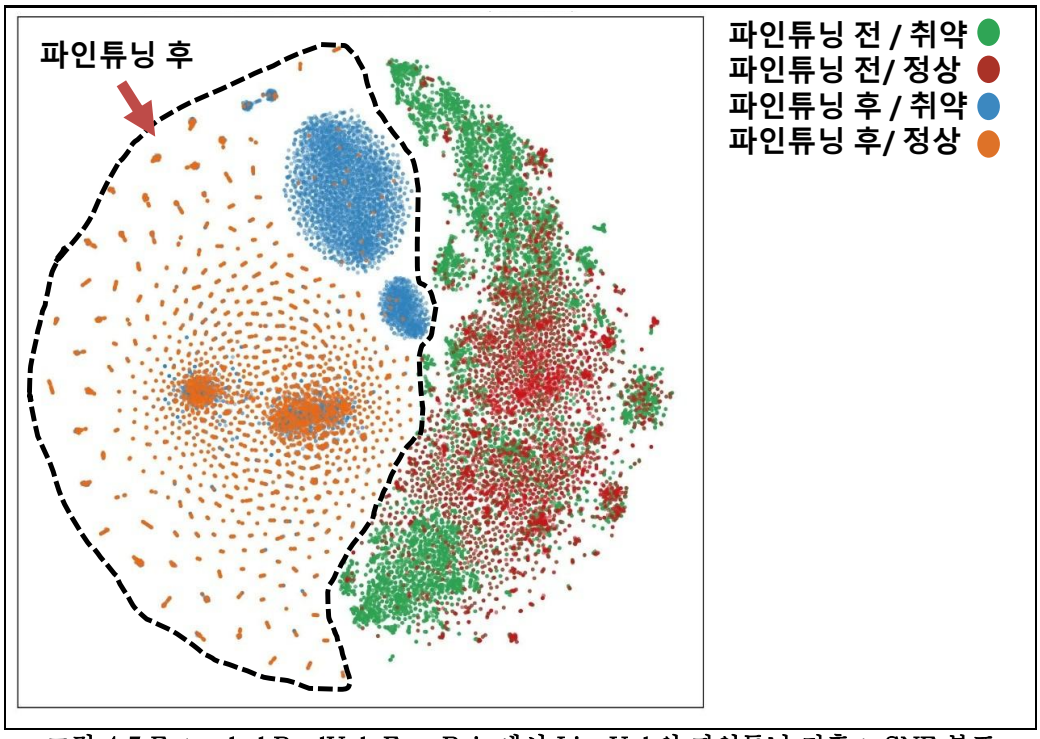


그림 4.5 Extended RealVul-FuncPair 에서 LineVul 의 파인튜닝 전후 t-SNE 분포

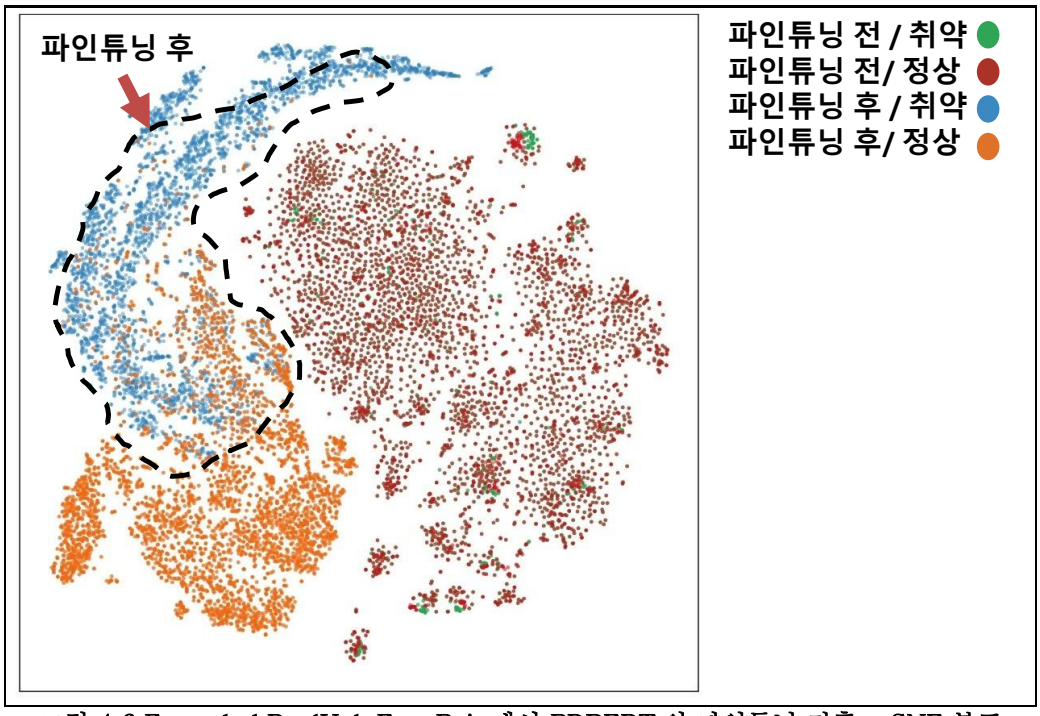


그림 4.6 Extended RealVul-FuncPair 에서 PDBERT 의 파인튜닝 전후 t-SNE 분포

LineVul 은 파인튜닝 후에도 취약 샘플과 정상 샘플이 명확한 군집으로 나뉘지 않는다. PDBERT 도 일부 군집 형태만 보일 뿐 파인튜닝 후 두 라벨이 안정적으로 분리되지 않는다.

표 4.5 Extended RealVul-FuncPair 에서 RAG Few-shot LLM 기반 취약/정상 함수 구분 결과

모델	정답 레이블	예측 레이블 “Safe”	예측 레이블 “Vulnerable”	F1 점수
Claude Sonnet 4.5	Safe	2258.8	2030.6	0.64
	Vulnerable	1306.6	3008	
Gemini 2.5 Pro	Safe	2588.4	1715.4	0.64
	Vulnerable	1493.6	2814.4	
GPT-4o	Safe	2769	1595.8	0.55
	Vulnerable	2097.2	2267.8	
Grok 4.1 Fast	Safe	4124.4	240.6	0.31
	Vulnerable	3534	831	

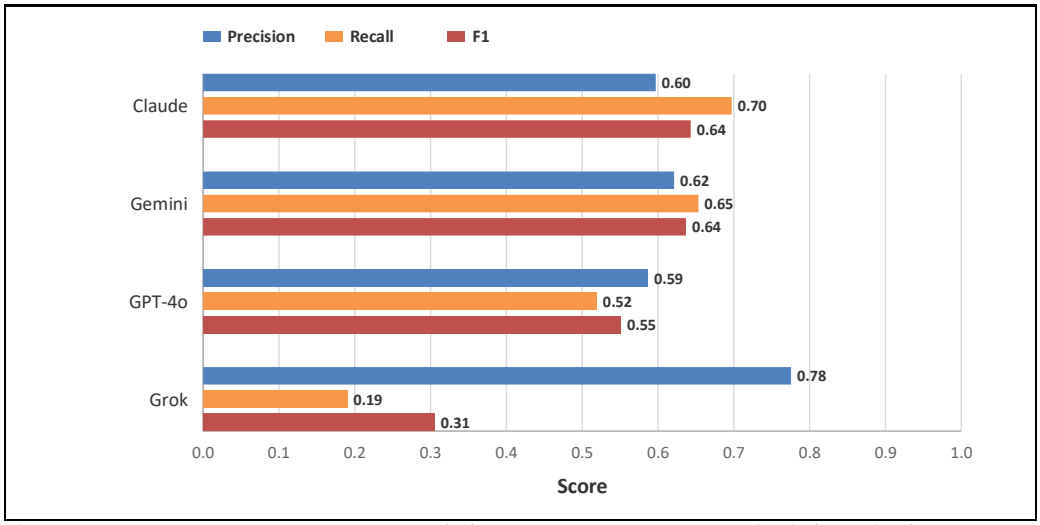


그림 4.7 Extended RealVul-FuncPair 에서 RAG Few-shot LLM 의 취약/정상 함수 구분 성능

표 4.5 와 그림 4.7 은 같은 FuncPair 샘플을 RAG 기반 Few-shot LLM 에 평가한 결과이다. 가장 높은 F1 도 0.64 였다. Claude 와 Gemini 는 정밀도와 재현율이 0.60 에서 0.70 사이였고, GPT-4o 는 재현율이 0.52 로 더 낮았다. Grok 은 정밀도 0.78 로 취약 예측 자체는 비교적 맞았지만 재현율 0.19 에 그쳐 많은 취약 함수를 정상으로 놓쳤다. 함수 단위 샘플과 유사 함수 예시만으로는 취약 함수와 정상 함수 차이를 안정적으로 판단하기 어려웠다.

RQ1 의 답은 명확하다. 높은 함수 단위 평가 점수는 같은 CVE 안의 취약 함수와 정상 함수 구분을 보장하지 않는다. PDBERT 는 표 4.3 에서 F1 0.996 을 보였지만 FuncPair 에서는 취약 예측으로 치우쳤고, LineVul 과 DeepWukong 은 정상

예측으로 치우쳤다. 이 결과는 함수 단위 데이터셋만으로는 모델이 취약 코드 패턴을 학습하기 어렵다는 점을 보여준다.

다. RQ2. 함수 단위 데이터셋 평가와 Trace 단위 데이터셋 평가에서 딥러닝 및 LLM 기반 취약/정상 구분 성능은 어떻게 달라지는가?

RQ2 는 RQ1 의 함수 단위 데이터셋 평가 결과와 Trace 단위 데이터셋 평가 결과를 비교한다. Trace 단위 평가는 SARD-Juliet SA TracePair 와 10 개 CVE 에 대한 TracePair 로 나누어 수행한다.

1) SARD-Juliet SA TracePair 취약/정상 구분 결과

첫 번째 Trace 단위 평가는 SARD-Juliet source-sink Trace 조건에서 수행하였다. 평가 데이터셋은 취약 Trace 40 개와 정상 Trace 40 개로 구성된다. 딥러닝 학습과 LLM 용 RAG 예시 검색에는 SARD-Juliet SA Trace 학습 데이터셋의 701 개 Trace 를 사용하였다.

표 4.6 SARD-Juliet SA TracePair 에서의 딥러닝 기반 취약/정상 구분 결과

모델	테스트 데이터셋	취약 예측	정상 예측	TP	TN	FP	FN
LineVul	SARD-Juliet SA TracePair	40 (50%)	40 (50%)	40	40	0	0
PDBERT	SARD-Juliet SA TracePair	31 (38.8%)	49 (61.3%)	31	40	0	9

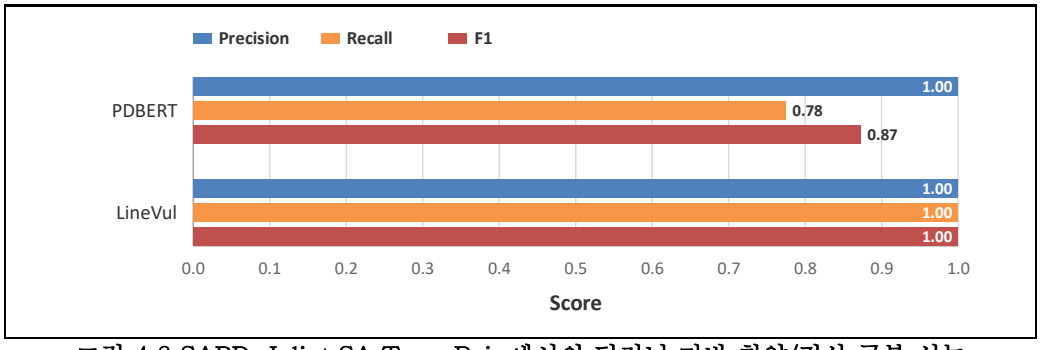
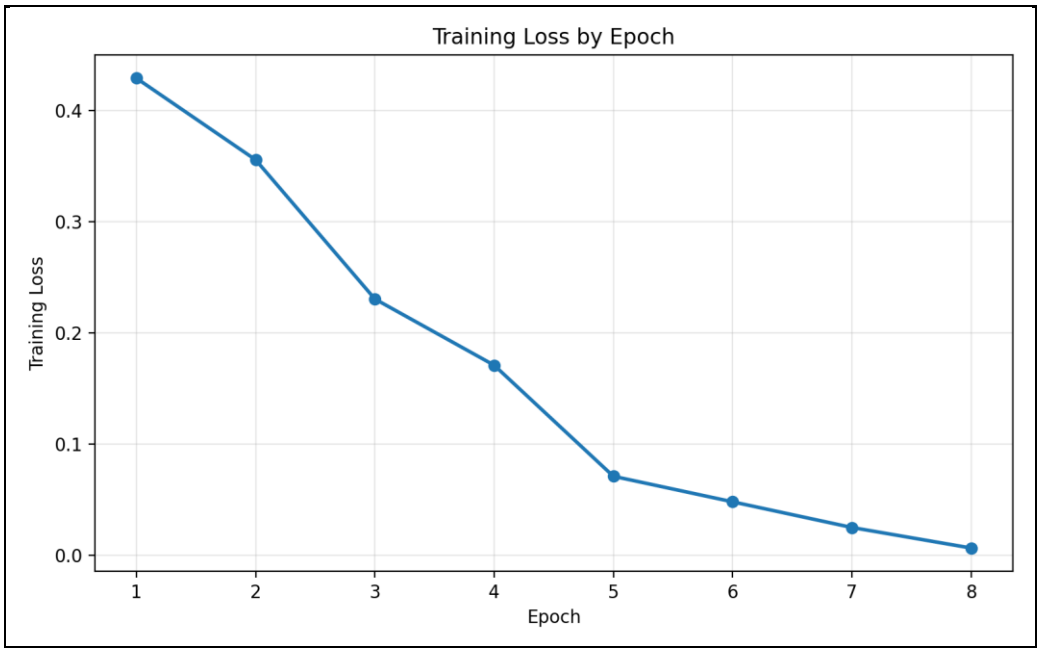
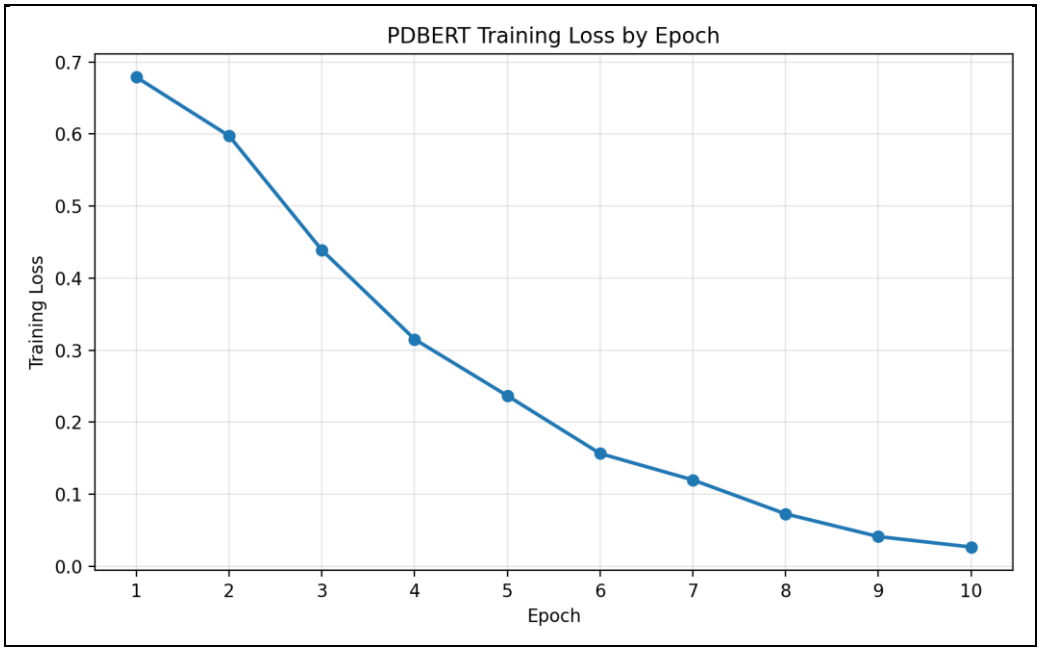


그림 4.8 SARD-Juliet SA TracePair 에서의 딥러닝 기반 취약/정상 구분 성능

표 4.6 과 그림 4.8 에서 LineVul 은 정밀도, 재현율, F1 이 모두 1.00 이었다. PDBERT 는 정상 Trace 를 취약으로 잘못 분류하지 않아 정밀도 1.00 을 보였지만, 취약 Trace 9 개를 놓쳐 재현율 0.78 과 F1 0.87 을 보였다. 이 결과는 RQ1 의 Extended RealVul-FuncPair 딥러닝 최고 F1 0.67 보다 높다.



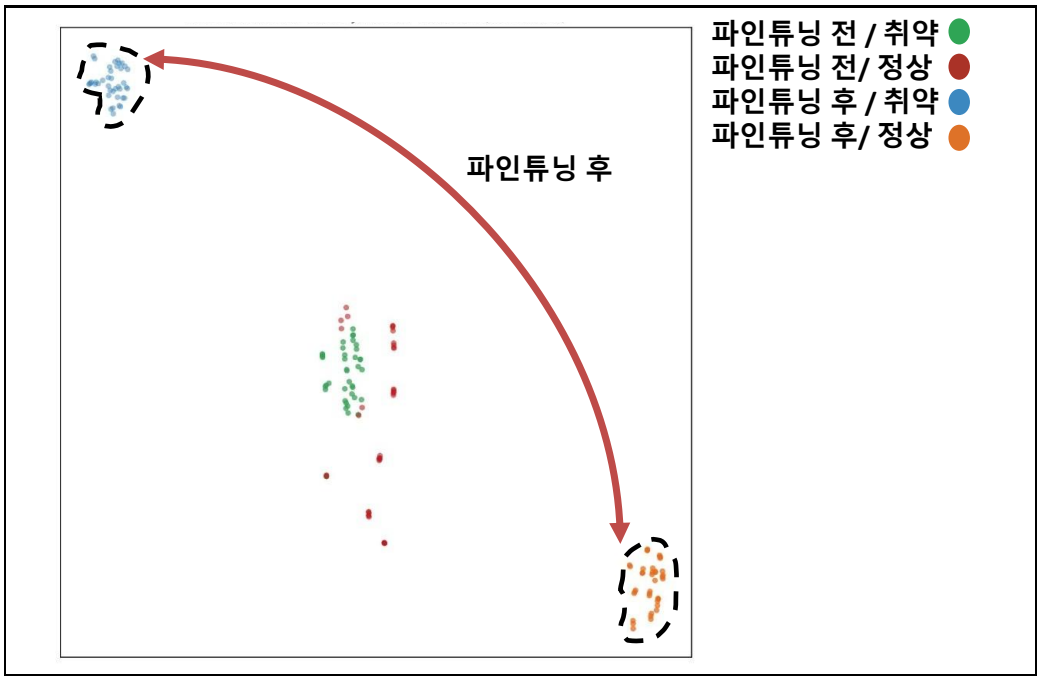
(a) LineVul



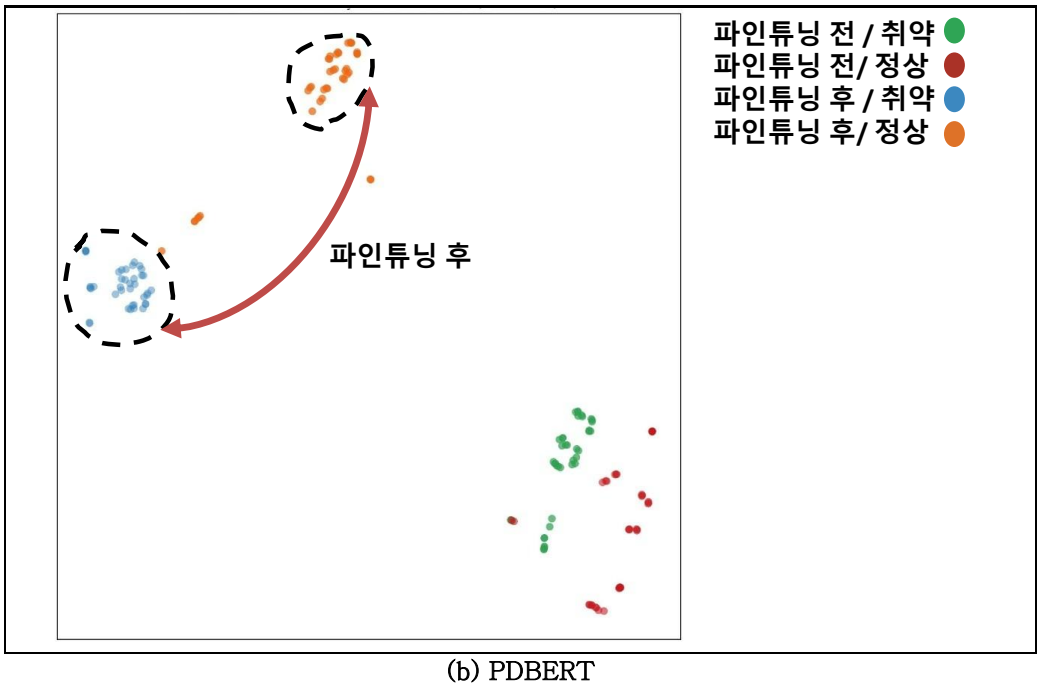
(b) PDBERT

그림 4.9 SARD-Juliet SA TracePair 에서 LineVul 과 PDBERT 의 학습 손실 변화

그림 4.9 는 LineVul 과 PDBERT 의 학습 손실(training loss)이 학습 과정에서 감소했음을 보여준다. 그림 4.9 의 손실 감소와 표 4.6 의 높은 구분 지표는 SARD-Juliet SA TracePair 조건에서 취약 Trace 와 정상 Trace 의 구분 가능성을 뒷받침한다.



(a) LineVul



(b) PDBERT

그림 4.10 SARD-Juliet SA TracePair 에서 LineVul 과 PDBERT 의 파인튜닝 전후 t-SNE 분포

표 4.6 과 그림 4.8 에서 나타난 높은 구분 지표는 모델 임베딩 분포에서도 관찰되었다. 그림 4.10 에서는 파인튜닝 전보다 두 라벨의 분포가 더 분리된다. 이 분리는 표 4.6 의 높은 구분 지표와 일치한다.

표 4.7 SARD-Juliet SA TracePair 에서의 RAG Few-shot LLM 기반 취약/정상 구분 결과

모델	정답 레이블	예측 레이블 "Safe"	예측 레이블 "Vulnerable"	F1 점수
Claude Sonnet 4.5	Safe	39	1	0.96
	Vulnerable	2.4	37.6	
Gemini 2.5 Pro	Safe	29.8	10.2	0.89
	Vulnerable	0	40	
GPT-4o	Safe	35.8	4.2	0.94
	Vulnerable	0.6	39.4	
Grok 4.1 Fast	Safe	39.8	0.2	0.92
	Vulnerable	5.6	34.4	

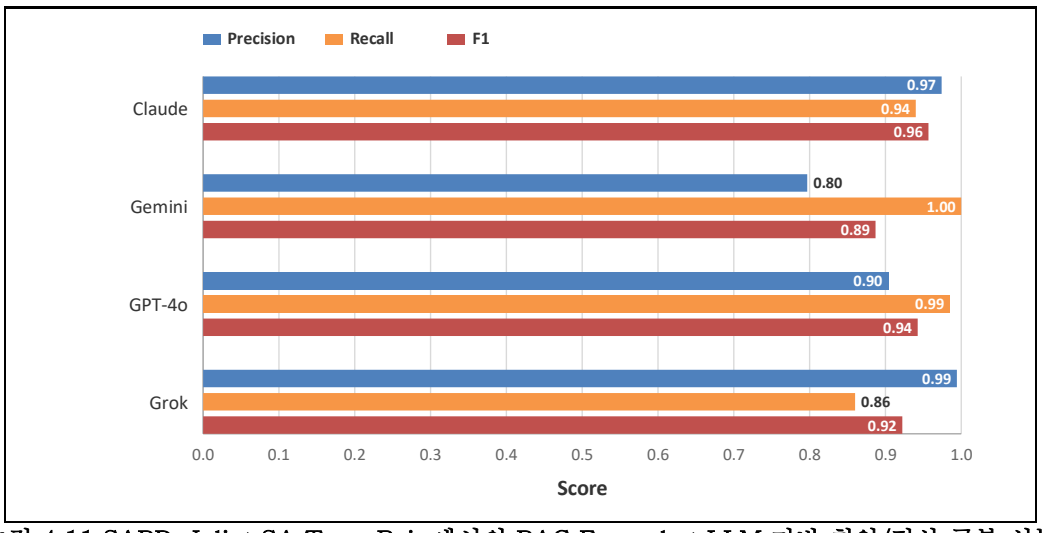


그림 4.11 SARD-Juliet SA TracePair 에서의 RAG Few-shot LLM 기반 취약/정상 구분 성능

표 4.7 과 그림 4.11 에서는 모든 LLM 의 F1 이 RQ1 의 함수 단위 데이터셋 평가 최고 F1 0.64 보다 높았다. Claude 와 GPT-4o 는 정밀도와 재현율이 모두 높았고, Gemini 는 재현율 1.00 을 보였지만 평균 정상 Trace 10.2 개를 취약으로 예측해 정밀도가 0.80 으로 낮아졌다. Grok 은 정밀도 0.99 와 재현율 0.86 을 보였다.

종합하면, SARD-Juliet SA TracePair 조건에서는 취약 Trace 와 정상 Trace 구분 성능이 함수 단위 데이터셋 평가보다 높게 나타났다. 이 결과는 SARD-Juliet

조건에서 Trace 단위 데이터셋이 함수 단위 데이터셋보다 취약 원인 흐름에 필요한 패턴을 더 직접적으로 드러냈음을 보여준다.

2) 10 개 CVE TracePair 취약/정상 구분 결과

10 개 CVE 에 대한 TracePair 에서는 SARD-Juliet SA TracePair 에서 나타난 높은 성능이 재현되지 않았다. 평가 데이터셋은 취약 Trace 10 개와 정상 Trace 10 개로 구성된다. 데이터셋 구성과 라벨 기준은 3 장 마절의 정의를 따른다.

표 4.8 과 표 4.9 에서 0 은 정상 예측, 1 은 취약 예측을 뜻한다. 취약 Trace 가 1, 정상 Trace 가 0 으로 예측될 때 해당 TracePair 를 올바르게 구분한 것으로 본다.

표 4.8 10 개 CVE 에 대한 TracePair 에서의 LineVul 파인튜닝 전후 예측 결과

프로젝트 및 CVE	파인튜닝 전 모델		파인튜닝 후 모델	
	취약 Trace	정상 Trace	취약 Trace	정상 Trace
patch (CVE-2019-13638)	0	0	1	1
php-src (CVE-2015-8617)	0	0	1	1
spice-vdagent (CVE-2017-15108)	0	0	0	0
php-src (CVE-2017-11142)	0	0	0	0
rsyslog (CVE-2017-12588)	0	0	0	0
plasma-workspace (CVE-2018-6791)	0	0	0	0
shadowsocks-libev (CVE-2017-15924)	0	0	1	1
openjpeg (CVE-2019-12973)	1	1	0	0
radare2 (CVE-2019-16718)	0	0	1	0
ghostpdl (CVE-2018-16863)	0	1	1	0

LineVul 은 파인튜닝 이후에도 여러 CVE 에서 취약 Trace 와 정상 Trace 를 같은 값으로 예측했다. 즉 SARD-Juliet SA Trace 기반 파인튜닝만으로는 실제 CVE TracePair 의 두 Trace 차이가 충분히 분리되지 않았다.

표 4.9 10 개 CVE 에 대한 TracePair 에서의 PDBERT 파인튜닝 전후 예측 결과

프로젝트 및 CVE	파인튜닝 전 모델		파인튜닝 후 모델	
	취약 Trace	정상 Trace	취약 Trace	정상 Trace
patch (CVE-2019-13638)	0	0	1	1
php-src (CVE-2015-8617)	0	0	1	1
spice-vdagent (CVE-2017-15108)	0	0	0	0
php-src (CVE-2017-11142)	0	0	0	0
rsyslog (CVE-2017-12588)	0	0	0	0
plasma-workspace (CVE-2018-6791)	0	0	0	0
shadowsocks-libev (CVE-2017-15924)	0	0	1	1
openjpeg (CVE-2019-12973)	0	0	0	0
radare2 (CVE-2019-16718)	0	0	0	1
ghostpdl (CVE-2018-16863)	0	0	0	0

표 4.9 는 PDBERT 가 파인튜닝 뒤에도 TracePair 안의 취약 Trace 와 정상 Trace 를 안정적으로 구분하지 못했음을 보여준다. 여러 CVE 에서 두 Trace 를 같은 라벨로 예측했고, radare2 에서는 정상 Trace 만 취약으로 예측했다. 따라서 PDBERT 의 취약 예측은 실제 취약 Trace 를 골라낸 결과로 보기 어렵다.

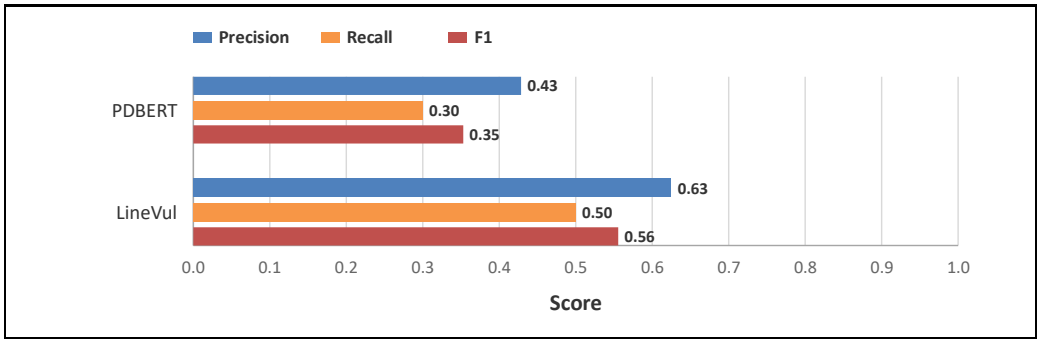
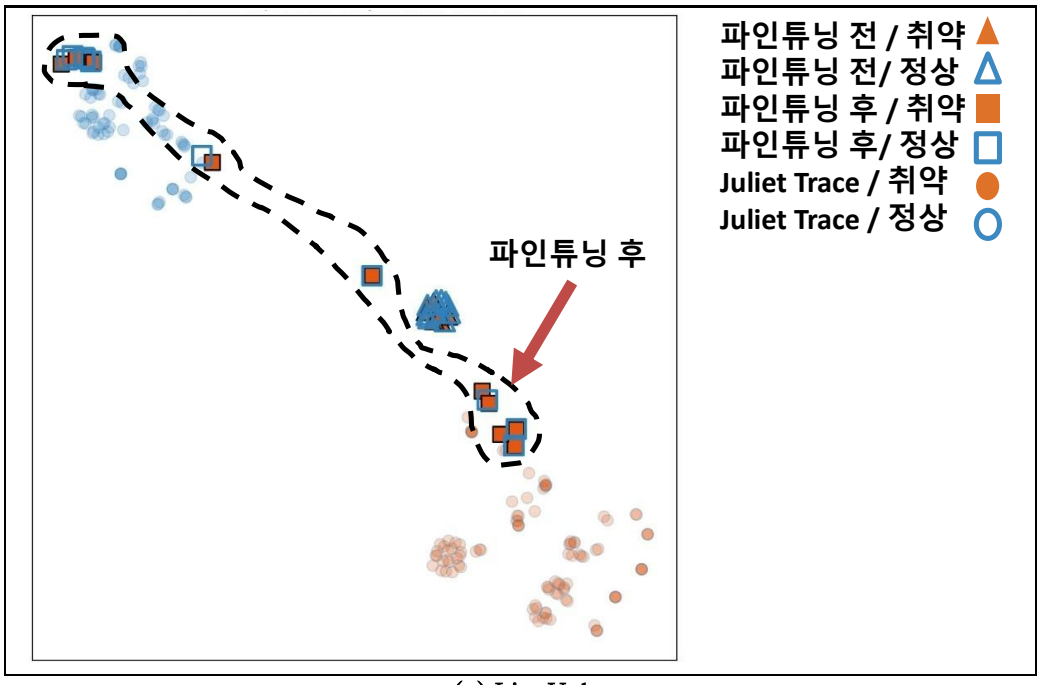


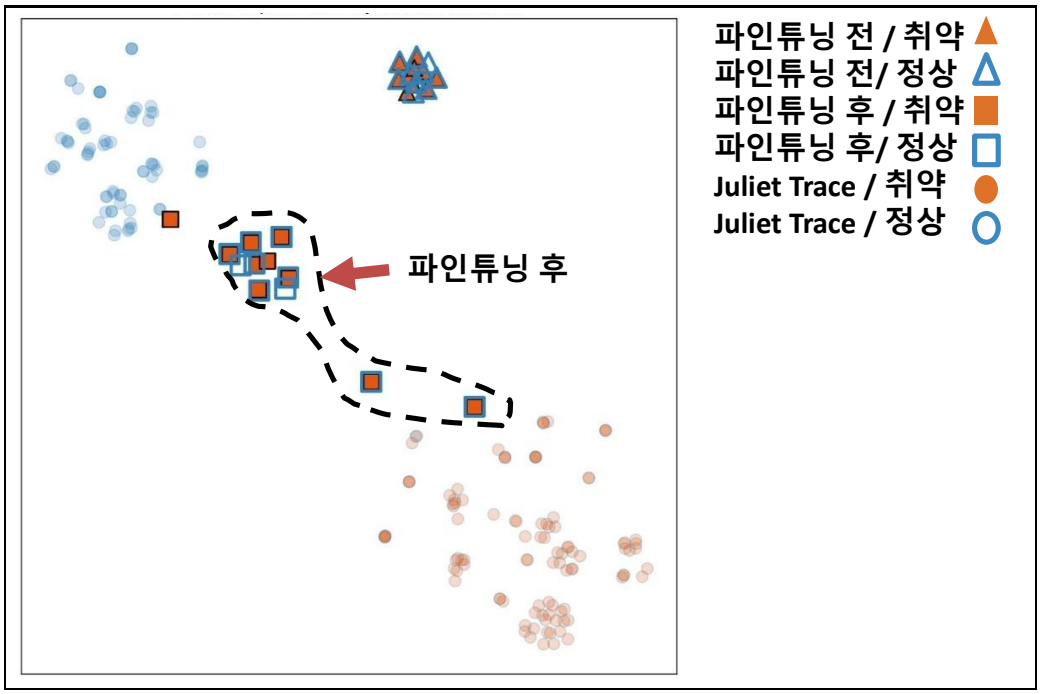
그림 4.12 10 개 CVE TracePair 에서의 딥러닝 기반 취약/정상 Trace 구분 성능

표 4.8, 표 4.9, 그림 4.12 를 종합하면 파인튜닝 후 LineVul 은 정밀도 0.63, 재현율 0.50, F1 0.56 이었고, PDBERT 는 정밀도 0.43, 재현율 0.30, F1 0.35 였다.

두 모델 모두 취약/정상 Trace 를 안정적으로 구분하지 못했다. 실제 CVE TracePair 에서는 SARD-Juliet SA TracePair 와 같은 수준의 구분 성능이 나타나지 않았다.



(a) LineVul



(b) PDBERT

그림 4.13 10 개 CVE 에 대한 TracePair 에서 LineVul/PDBERT 의 파인튜닝 전후 t-SNE 분포

표 4.8 과 표 4.9, 그림 4.12 에서 나타난 10 개 CVE TracePair 의 낮은 정밀도, 재현율, F1 과 구분 실패 양상은 모델 임베딩 분포에서도 관찰되었다. 그림 4.13 에서는 파인튜닝 후에도 취약 Trace 와 정상 Trace 의 라벨 분리가 뚜렷하지 않다. 이 분포는 그림 4.12 의 낮은 구분 지표와 앞선 표 4.8, 표 4.9 의 TracePair 구분 실패 양상과 일관된다.

표 4.10 10 개 CVE TracePair 에서 RAG Few-shot LLM 의 취약/정상 Trace 구분 결과

모델	정답 레이블	예측 레이블 "Safe"	예측 레이블 "Vulnerable"	F1 점수
Claude Sonnet 4.5	Safe	2.8	7.2	0.59
	Vulnerable	2.8	7.2	
Gemini 2.5 Pro	Safe	1	9	0.67
	Vulnerable	0.4	9.6	
GPT-4o	Safe	1.8	8.2	0.67
	Vulnerable	0.8	9.2	
Grok 4.1 Fast	Safe	7.8	2.2	0.30
	Vulnerable	7.8	2.2	

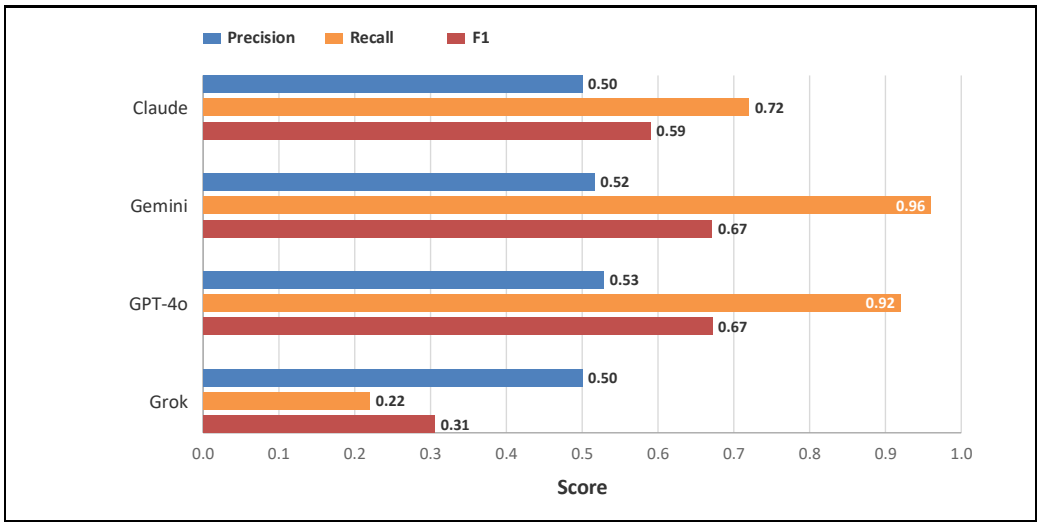


그림 4.14 10 개 CVE TracePair 에서 RAG Few-shot LLM 의 취약/정상 Trace 구분 성능

표 4.10 과 그림 4.14 에서도 LLM 평가는 SARD-Juliet SA TracePair 결과보다 낮았다. Gemini 와 GPT-4o 는 모두 F1 0.67 을 보였고 재현율도 각각 0.96, 0.92 였지만, 정밀도는 0.52 와 0.53 에 머물렀다. 이는 두 모델이 취약 Trace 를 비교적 많이 찾아냈더라도 정상 Trace 까지 취약으로 예측하는 경향이 컸음을

의미한다. Claude 와 Grok 도 정밀도 0.50 에 머물러 취약/정상 Trace 를 안정적으로 구분하지 못했다.

종합하면, SARD-Juliet SA TracePair 에서 확인한 Trace 단위 구분 성능은 10 개 CVE 에 대한 TracePair 에서 재현되지 않았다. 딥러닝 최고 F1 은 0.56, LLM 최고 F1 은 0.67 에 그쳤다.

라. 실제 CVE TracePair 적용 한계 분석 결과

10 개 CVE 에 대한 TracePair 의 성능 한계는 Juliet 예시와 CVE Trace 의 코드 흐름 차이, CVE Trace 구축 과정의 수동 복원과 source/sink 판단 의존성에서 나타났다. SARD-Juliet SA TracePair 에서 나타난 취약 Trace 와 정상 Trace 의 뚜렷한 구분은 10 개 CVE 에 대한 TracePair 에서 같은 수준으로 재현되지 않았다.

CVE Trace 와 Juliet Trace 의 차이는 먼저 Trace 길이에서 확인된다. SARD-Juliet SA Trace 의 평균 Step 수는 5.92 였지만, 10 개 CVE 대상 Trace 의 평균 Step 수는 12.50 이었다. 이는 평가 대상 CVE Trace 가 학습과 RAG 예시 검색에 사용한 Juliet Trace 보다 긴 흐름을 포함했음을 뜻한다.

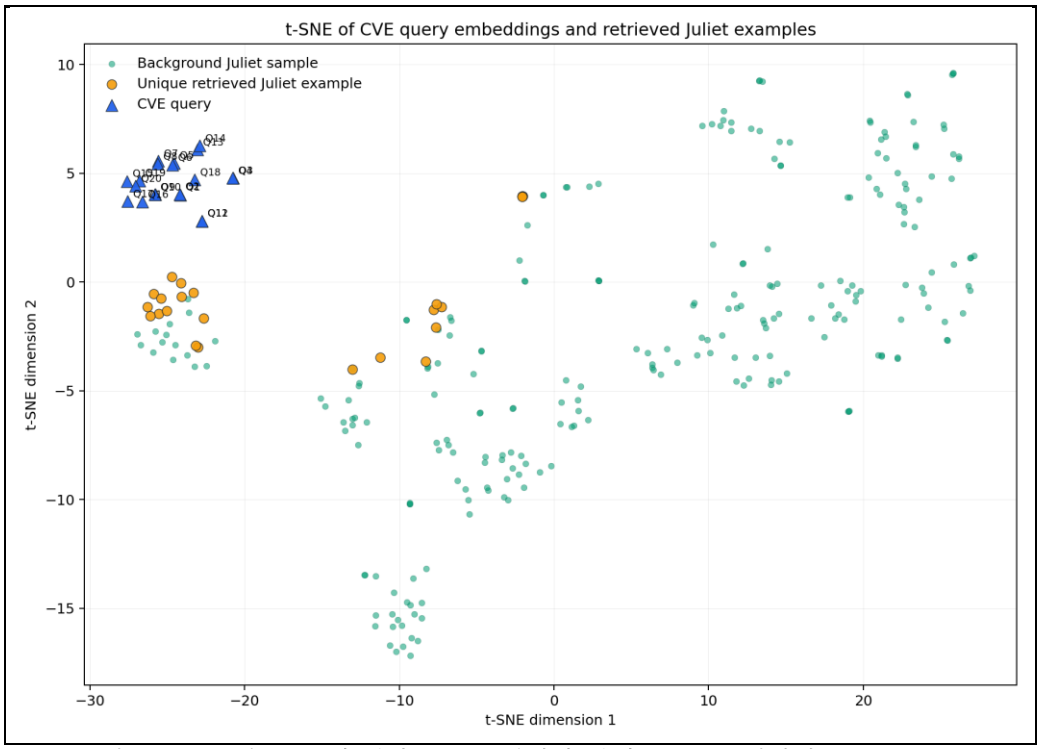


그림 4.15 10 개 CVE 에 대한 Trace 질의와 검색된 Juliet 예시의 t-SNE 분포

그림 4.15 는 검색된 Juliet 예시가 임베딩 기반 t-SNE 공간에서 CVE Trace 질의 근처에 놓였음을 보여준다. 그러나 이 근접성은 실제 코드 흐름, source-sink 문맥, 취약 원인, 프로젝트별 API 사용의 유사성을 보장하지 않는다.

표 4.11 10 개 CVE 에 대한 TracePair 의 RAG 검색 예시 수작업 검토 결과

검토 사항	결과
구문 유사성	부분 유사 5/120, 강한 유사 0/120
의미 유사성	부분 유사 5/120, 강한 유사 0/120
source/sink 일치	부분 일치 5/120, 전체 일치 0/120
CWE 일치	10/120

표 4.11 은 20 개 CVE Trace 질의의 top-6 Juliet 결과 120 개를 원문 기준으로 검토한 결과이다. 강한 구문 유사성, 강한 의미 유사성, 전체 source/sink 일치는 모두 0 건이었다. 따라서 임베딩 공간의 근접성은 검색된 Juliet 예시가 CVE Trace 의 코드 흐름과 취약 원인 문맥을 충분히 대표한다는 근거가 되지 못한다.

CVE Trace 구축 불확실성도 제한 요인이다. Infer 는 전역 변수, 구조체 필드, 함수 포인터를 통한 흐름을 누락할 수 있었다. Infer 가 source 에서 sink 까지

이어지는 전체 흐름을 하나의 Trace 로 연결하지 못한 경우에는, 여러 구간으로 나뉜 Trace 를 수동으로 이어 전체 흐름을 복원했다. 또한 source 와 sink 경계도 CVE 별 취약 코드 맥락을 해석해 지정했다. 이 과정은 Trace 품질과 재현성을 source/sink 정의와 복원 판단에 의존하게 만든다.

검색 질의의 최대 입력 길이 제한 때문에, 이 제한을 넘는 Trace 3 개에서는 후반부 source/sink 문맥이 검색에 덜 반영되었을 수 있다. 그러나 최종 LLM 프롬프트에는 평가 대상 CVE Trace 가 포함되었고, 해당 조건도 전체 20 개 중 3 개에 그쳤다.

종합하면, Juliet 예시가 CVE Trace 의 코드 흐름과 취약 원인 문맥을 충분히 대표하지 못한 점, 그리고 CVE Trace 구축 과정이 수동 복원과 source/sink 판단에 의존한 점이 10 개 CVE 에 대한 TracePair 의 주요 제한 요인으로 관찰되었다.

5. 결론 및 향후계획

본 논문은 패치 기반 함수 단위 데이터셋에서 나타나는 세 가지 라벨링 한계를 분석하고, 취약 원인 흐름 중심의 Trace 단위 데이터셋을 제안하였다. 함수 단위 라벨링은 작은 패치 차이, 시간 흐름에 따른 라벨 충돌, 지나치게 넓은 취약 라벨 범위 때문에 취약 함수와 정상 함수의 의미 차이를 흐릴 수 있다. 이 조건에서는 모델이 취약 코드 패턴과 취약/정상 의미 차이를 안정적으로 학습하기 어렵다.

평가 결과, 기존 연구[14], [21], [22] 에서 주장한 바와 일관되게 Extended RealVul-FuncPair 의 함수 단위 데이터셋에서는 취약 함수와 정상 함수의 의미 차이가 안정적으로 구분되지 않았다. SARD-Juliet SA TracePair 에서는 취약 Trace 와 정상 Trace 의 구분 가능성이 확인되었다. 그러나 10 개 CVE 에 대한 TracePair 에서는 같은 구분이 안정적으로 이루어지지 않았다.

Trace 는 함수 단위 샘플보다 취약점 판단에 필요한 source-sink 흐름을 더 직접적으로 보여준다. 다만 10 개 CVE 에 대한 TracePair 평가에서는 검색 예시의 대표성 부족과 실제 CVE Trace 구성상의 제한이 중요한 요인으로 관찰되었다.

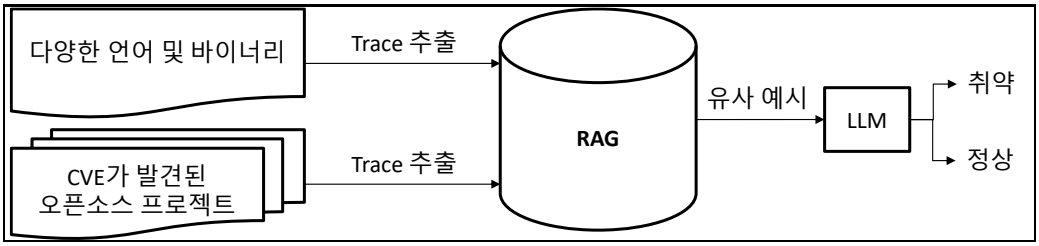


그림 5.1 RAG 기반 LLM Few-shot 취약점 탐지 확장 방향

그림 5.1 은 향후 연구 방향을 요약한다. 10 개 CVE TracePair 평가에서 SARD-Juliet SA TracePair 의 구분 결과가 같은 수준으로 재현되지 않은 주된 원인은 실제 CVE Trace 구축상의 제한과 RAG 검색 예시의 낮은 대표성이다. 따라서 후속 연구는 대규모 실제 CVE Trace 수집, Trace 구축 절차의 재현성, 검색 저장소의 대표성 검증을 우선해야 한다.

이를 위해 먼저 CVE 설명, 패치 전후 코드, 취약 함수 호출 흐름을 함께 검토해 source/sink 함수 후보를 결정하는 방안을 정해야 한다. Infer 가 Trace 를 여러 구간으로 나누어 출력한 경우에는 데이터 의존성이 확인되는 구간만 연결하고, 연결 근거와 복원 과정을 기록하는 자동화 또는 반자동 검토 도구가 필요하다. 또한 정적 분석 도구는 전역 변수, 구조체 필드, 함수 포인터 호출의 오염 전파 경로를 source-sink 흐름으로 안정적으로 추출해야 한다. RAG 검색 저장소는 CWE 와 source/sink 유형을 기준으로 구성해 평가 대상 Trace 의 취약 원인 문맥을 대표하도록 해야 한다. Trace 단위 취약점 탐지는 다양한 프로그래밍 언어와 바이너리 대상 소프트웨어로 확장할 필요가 있다.

참고문헌

[1] S. Kim, S. Woo, H. Lee and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 2017, pp. 595-614.

[2] X. Cheng et al., “Deepwukong: Statically detecting software vulnerabilities using deep graph neural network,” ACM Trans. Softw. Eng. Methodol., vol. 30, no. 3, pp. 1–33, 2021.

[3] M. Fu and C. Tantithamthavorn, "LineVul: A Transformer-based Line-Level Vulnerability Prediction," 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), Pittsburgh, PA, USA, 2022, pp. 608-620.

[4] Z. Liu, Z. Tang, J. Zhang, X. Xia and X. Yang, "Pre-training by Predicting Program Dependencies for Vulnerability Analysis Tasks," in 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 2024, pp. 1863-1875.

[5] P. Chakraborty, K. K. Arumugam, M. Alfadel, M. Nagappan and S. McIntosh, "Revisiting the Performance of Deep Learning-Based Vulnerability Detection on Realistic Datasets," in IEEE Transactions on Software Engineering, vol. 50, no. 8, pp. 2163-2177, Aug. 2024.

[6] W. Kang, B. Son and K. Heo, "Tracer: Signature-based Static Analysis for Detecting Recurring Vulnerabilities," in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), Los Angeles, CA, USA, 2022, 14 pages.

[7] 전세옥, 장현지, 정민수, 이서준, 오다영, 최광훈, 김석휘, 조성우, 김정미, "전기차 충전 인프라 오픈소스 소프트웨어에서의 취약한 코드 복제 탐지: VUDDY 를 활용한 사례 연구," ACK 2024, 2024.

[8] Team Atlanta, "ATLANTIS: AI-driven Threat Localization, Analysis, and Triage Intelligence System," ver. 1.0, Sep. 18, 2025. [Online]. Available: https://team-atlanta.github.io/papers/TR-Team-Atlanta.pdf (accessed Mar. 22, 2026).

[9] Y. Kim, "Branch Flipper: Unlocking Fuzz Blockers with Coverage-Grounded LLMs," 2025. [Online]. Available: https://theori-io.github.io/aixcc-public/afc/Branch%20Flipper.pdf (accessed Mar. 22, 2026).

[10] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," in IEEE Transactions on Dependable and Secure Computing, vol. 19, no. 4, pp. 2244-2258, 1 July-Aug. 2022

[11] F. Yamaguchi, N. Golde, D. Arp and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 2014, pp. 590-604

[12] Bozhi Wu, Shangqing Liu, Yang Xiao, Zhiming Li, Jun Sun, and Shang-Wei Lin. 2023. Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-procedural Slicing. In Proceedings of the 31st ACM Joint European Software Engineering

Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023). Association for Computing Machinery, New York, NY, USA, 1371–1383.

[13] B. Bowman and H. H. Huang, "VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets," 2020 IEEE European Symposium on Security and Privacy (EuroS&P), Genoa, Italy, 2020, pp. 53-69

[14] 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," 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), Ottawa, ON, Canada, 2025, pp. 2893-2904.

[15] J. Jang, A. Agrawal and D. Brumley, "ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions," 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 2012, pp. 48-62

[16] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS 2018), San Diego, CA, USA, Feb. 18-21, 2018. The Internet Society, 2018

[17] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: an automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual

Conference on Computer Security Applications (ACSAC '16). Association for Computing Machinery, New York, NY, USA, 201–213.

[18] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu and H. Jin, "VulDeeLocator: A Deep Learning-Based Fine-Grained Vulnerability Detector," in IEEE Transactions on Dependable and Secure Computing, vol. 19, no. 4, pp. 2821-2837, 1 July-Aug. 2022

[19] Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In Proceedings of the 33rd USENIX Conference on Security Symposium (SEC '24). USENIX Association, USA, Article 238, 4247–4264.

[20] Dyna Soumhane Ouchebara and Stéphane Dupont. 2025. Llama-Based Source Code Vulnerability Detection: Prompt Engineering vs Fine Tuning. In Computer Security – ESORICS 2025: 30th European Symposium on Research in Computer Security, Toulouse, France, September 22–24, 2025, Proceedings, Part I. Springer-Verlag, Berlin, Heidelberg, 289–308.

[21] S. Chakraborty, R. Krishna, Y. Ding and B. Ray, "Deep Learning Based Vulnerability Detection: Are We There Yet?" in IEEE Transactions on Software Engineering, vol. 48, no. 09, pp. 3280-3296, Sept. 2022

[22] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability Detection with Code Language

Models: How Far Are We? In Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25). IEEE Press, 1729–1741.

[23] 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 the Association for Computational Linguistics: EMNLP 2020, Online, Nov. 2020, pp. 1536–1547.

[24] H. Hanif and S. Maffeis, "VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection," 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy, 2022, pp. 1–8

[25] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in Advances in Neural Information Processing Systems 32 (NeurIPS 2019), Vancouver, BC, Canada, 2019, pp. 10197–10207.

[26] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20). Association for Computing Machinery, New York, NY, USA, 508–512.

[27] Se-Ok Jeon, Seokhwi Kim and Kwanghoon Choi, "Analyzing the Limitations of CVE-Based Source-Code Vulnerability Slice

Extraction," Poster, The 26th World Conference on Information Security Applications (WISA), Jeju, Republic of Korea, Aug. 20-22, 2025.

[28] Meta Platforms, Inc., "Infer Static Analyzer." [Online]. Available: https://fbinfer.com/ (accessed May 14, 2026).

[29] The MITRE Corporation, "CWE Top 25 Most Dangerous Software Weaknesses - 2024." [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html (accessed Jun. 19, 2026).

[30] The MITRE Corporation, "CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')." [Online]. Available: https://cwe.mitre.org/data/definitions/89.html (accessed Jun. 19, 2026).

[31] The MITRE Corporation, "CWE-134: Use of Externally-Controlled Format String." [Online]. Available: https://cwe.mitre.org/data/definitions/134.html (accessed Jun. 19, 2026).

Function-Labeling Limitations of Patch-Based Datasets in Software Vulnerability Detection and Trace-Level Dataset Construction and Evaluation

SeOk JEON

Department of Information Security Convergence

Graduate School Chonnam National University

(Supervised by Professor Kwanghoon Choi)

Recent deep-learning-based vulnerability detection studies have reported high vulnerability-detection performance on the benchmarks constructed in their respective papers. However, on new datasets, F1 scores can drop by up to 91%, and previously reported F1 scores may fail to be reproduced. This study traces these limitations to function-level labeling. Short patch differences can make vulnerable and normal samples difficult to distinguish. The same code can receive different labels over time. Normal code unrelated to the vulnerability cause can also be included in vulnerable samples. Under these conditions, artificial intelligence models have difficulty learning vulnerable code patterns reliably. They also have difficulty learning vulnerable/normal semantic differences.

This study proposes Trace-level datasets to complement the limitations of function-level vulnerable/normal labeling. A Trace is a list of code statements along the data-flow path from a source, where external input enters, to a sink, where the value reaches a security-sensitive operation. This study extends RealVul to construct Extended RealVul-FuncPair and constructs SARD-Juliet SA TracePair and TracePair for 10 Common Vulnerabilities and Exposures (CVEs). Using these datasets, it evaluates vulnerable/normal classification performance in deep-learning-based models and large language models with retrieved examples.

The evaluation showed three results. First, Extended RealVul-FuncPair showed that function-level datasets have limitations in reliably revealing semantic differences between vulnerable and normal functions. Second, vulnerability-detection experiments on SARD-Juliet SA TracePair distinguished vulnerable Traces from normal Traces and showed the advantage of Trace-level datasets. Third, TracePair for 10 CVEs did not show the same distinction reliably. Applying this approach to real CVEs therefore requires large-scale Trace data that reflect vulnerability-cause flows. Future work should improve the Trace extraction capability of static analysis tools such as Infer.

부록 A. 데이터셋 구축 소프트웨어 가이드

이 부록은 본 논문에서 사용한 데이터셋의 구축 소프트웨어 가이드이다. 대상 데이터셋은 Extended RealVul, Extended RealVul-FuncPair, SARD-Juliet SA Trace, SARD-Juliet SA TracePair 이다.

FuncPair 는 취약 함수와 해당 취약점 수정 커밋에서 확인한 패치 후 정상 함수 후보를 짝지은 함수 단위 데이터셋이다. TracePair 는 취약 Trace 와 대응 정상 Trace 를 짝지은 Trace 단위 데이터셋이다.

A.0 재현 범위와 산출물 요약

재현 범위는 네 개 데이터셋과 아래 최종 산출물로 한정한다.

데이터셋	최종 산출물
Extended RealVul	dataset_pipeline/output/extended_realvul/Real_Vul_data.csv
Extended RealVul-FuncPair	dataset_pipeline/output/extended_realvul/all_projects_vul_patch_dataset.csv
SARD-Juliet SA Trace	$LATEST_RUN/07_dataset_export/Real_Vul_data.csv
SARD-Juliet SA TracePair	$LATEST_RUN/07_dataset_export/vuln_patch/Real_Vul_data.csv

A.1 실행 환경 준비

저자 실행 환경은 Ubuntu 24.04 LTS, Linux 6.8.0-51-generic, Intel Core i7-13700KF, 62 GiB 메모리, NVIDIA GeForce RTX 4070 SUPER, 1 TB NVMe SSD, 2 TB HDD 이다. 이 사양은 참조 환경이며 필수 하드웨어 요구사항이 아니다.

A.1.1 공통 의존성 설치

소프트웨어 실행에 필요한 운영체제 패키지를 설치하고 Git LFS 를 활성화한다.

```
sudo apt update
sudo apt install -y ₩
  python3 python3-venv python3-pip git git-lfs wget curl xz-utils ₩
  clang libunwind8 universal-ctags jq p7zip-full unzip

git lfs install
```

A.1.2 작업 디렉터리와 저장소 준비

아래 명령은 재현 작업 루트를 만들고, VP-Bench 와 juliet-playground 저장소를 준비한 뒤 clone 상태를 확인한다. 이전 실행 산출물이 남아 있으면 새 작업 디렉터리를 사용한다.

```
export REPRO_ROOT="$HOME/dataset-guide-repro"

mkdir -p "$REPRO_ROOT"
cd "$REPRO_ROOT"

git clone https://github.com/seokjeon/VP-Bench
cd "$REPRO_ROOT/VP-Bench"
git submodule update --init --recursive
git status --short

cd "$REPRO_ROOT"
git clone https://github.com/seokjeon/juliet-playground.git
cd "$REPRO_ROOT/juliet-playground"
git status --short
```

A.1.3 VP-Bench 작업 공간 준비

VP-Bench 는 Python 스크립트로 Extended RealVul 과 Extended RealVul-FuncPair 를 구축하므로, 저장소의 Python 의존성을 먼저 설치한다.

```
cd "$REPRO_ROOT/VP-Bench"
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

CVE commit 파일 목록, 변경 파일 본문, 패치 후 함수 후보를 GitHub API 로 조회하므로 token 을 GITHUB_TOKEN 으로 지정한다.

```
export GITHUB_TOKEN="<GitHub API token>"
test -n "$GITHUB_TOKEN"
```

A.1.4 juliet-playground 작업 공간 준비

juliet-playground 는 Python pipeline 과 Infer 정적 분석을 함께 사용하므로, 먼저 Infer v1.2.0 을 설치하고 실행 버전을 확인한다.

```
INFER_VERSION="1.2.0"
INFER_ARCHIVE="infer-linux-x86_64-v${INFER_VERSION}.tar.xz"
INFER_URL="https://github.com/facebook/infer/releases/download/v${IN
FER_VERSION}/${INFER_ARCHIVE}"

cd /tmp
curl -fL -o "$INFER_ARCHIVE" "$INFER_URL"
tar -xf "$INFER_ARCHIVE"

sudo install -d /opt
if [ ! -d "/opt/infer-linux-x86_64-v${INFER_VERSION}" ]; then
  sudo mv "infer-linux-x86_64-v${INFER_VERSION}" /opt/
fi

export PATH="/opt/infer-linux-x86_64-
v${INFER_VERSION}/bin:$PATH"
infer --version
```

juliet-playground 의 Python 의존성을 설치한다.

```
cd "$REPRO_ROOT/juliet-playground"
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

A.2 Extended RealVul 및 Extended RealVul-FuncPair 구축

이 절은 원본 RealVul 확보 → VP-Bench 입력 배치 → NVD CVE JSON 파일 확보 → realvul/vpbench 실행 → CSV 병합 → FuncPair 생성 → 산출물 확인 순서로 진행한다. 목표는 RealVul 원본에서 학습 데이터셋을 만들고, NVD 기반 20 년~25 년 CVE 를 추가해 Extended RealVul 테스트 데이터셋을 만든 뒤, 취약

함수마다 패치 후 정상 함수 후보를 찾아 Extended RealVul-FuncPair 를 만드는 것이다.

A.2.1 입력 확보와 경로 설정

VP-Bench 작업 공간을 활성화하고 RealVul 입력 및 출력 경로를 준비한다.

```
cd "$REPRO_ROOT/VP-Bench"
source .venv/bin/activate

REALVUL_INPUT_DIR="$PWD/dataset_pipeline/input/realvul_zenodo"
REALVUL_ZIP="$REALVUL_INPUT_DIR/Replication_Package.zip"
REALVUL_EXTRACT_DIR="$REALVUL_INPUT_DIR/extracted"

mkdir -p ₩
  "$REALVUL_INPUT_DIR" ₩
  "$REALVUL_EXTRACT_DIR" ₩
  dataset_pipeline/archive ₩
  dataset_pipeline/input ₩
  dataset_pipeline/output/realvul
```

Zenodo metadata 에서 RealVul replication package 를 내려받고 원본 Dataset 디렉터리를 찾는다.

```
curl -fsSL ₩
  "https://zenodo.org/api/records/12707476" ₩
  -o "$REALVUL_INPUT_DIR/zenodo_record.json"

REALVUL_ZIP_URL=$(jq -r '.files[] | select(.key == "Replication
Package.zip") | .links.self' "$REALVUL_INPUT_DIR/zenodo_record.json")
test -n "$REALVUL_ZIP_URL"

curl -fL -C - -o "$REALVUL_ZIP" "$REALVUL_ZIP_URL"
unzip -q -n "$REALVUL_ZIP" -d "$REALVUL_EXTRACT_DIR"

REALVUL_DATASET_DIR=$(find "$REALVUL_EXTRACT_DIR" -type d -
name Dataset | head -n 1)
test -n "$REALVUL_DATASET_DIR"
```

A.2.2 VP-Bench 입력 배치

RealVul 프로젝트별 CSV 와 소스 코드 archive 를 VP-Bench 입력 위치의 .old 호환 파일명으로 복사한다. .old 는 별도 데이터셋이나 낡은 버전이 아니라, 현재 run.sh --mode realvul 이 원본 RealVul 파일을 식별하는 파일명 규칙이다. ${project}_dataset.csv 와 ${project}_source_code.tar.gz 는 직접 만들지 않고, run.sh --mode realvul 이 생성하게 둔다.

```
projects=(FFmpeg ImageMagick jasper krb5 openssl php-src qemu
tcpdump linux Chrome)

for project in "${projects[@]}"; do
  project_out="dataset_pipeline/output/realvul/${project}"
  mkdir -p "$project_out"

  dataset_csv="$REALVUL_DATASET_DIR/${project}_dataset.csv"

source_archive="$REALVUL_DATASET_DIR/${project}_source_code.tar.
gz"

  test -f "$dataset_csv"
  test -f "$source_archive"

  if [ -e "$project_out/${project}_dataset.csv" ] || ₩
     [ -e "$project_out/${project}_source_code.tar.gz" ] || ₩
     [ -d "$project_out/source_code" ]; then
    echo "기존 realvul 산출물이 남아 있다. 새 REPRO_ROOT 에서 다시
실행한다: $project_out" >&2
    exit 1
  fi

  cp "$dataset_csv" "$project_out/${project}_dataset.csv.old"
  cp "$source_archive" "$project_out/${project}_source_code.tar.gz.old"
done
```

A.2.3 NVD CVE JSON 파일 확보

NVD 기반 VP-Bench test CVE 재구성에 필요한 연도별 NVD CVE JSON 2.0 압축 파일을 내려받고 JSON 파일을 준비한다. 대상 프로젝트 git 저장소는 run.sh 실행 중 dataset_pipeline/cache/repositories/로 clone 된다.

```
for year in 2020 2021 2022 2023 2024 2025; do
  zip_file="nvdcve-2.0-${year}.json.zip"
  json_file="nvdcve-2.0-${year}.json"
  if [ ! -f "dataset_pipeline/input/${json_file}" ]; then
    curl -fL -A "Mozilla/5.0" ₩
      -o "dataset_pipeline/archive/${zip_file}" ₩
      "https://nvd.nist.gov/feeds/json/cve/2.0/${zip_file}"
    unzip -q -o "dataset_pipeline/archive/${zip_file}" -d
dataset_pipeline/input/
  fi
  test -f "dataset_pipeline/input/${json_file}"
done
```

A.2.4 Extended RealVul 실행과 CSV 병합

RealVul 원본 기반 학습 데이터셋과 NVD CVE JSON 파일에서 재구성한 테스트 데이터셋을 만든다.

```
bash ./dataset_pipeline/run.sh --projects all --mode realvul
bash ./dataset_pipeline/run.sh --projects all --mode vpbench
```

두 CSV 의 존재와 첫 행의 열 구조를 확인한 뒤 Extended RealVul CSV 로 병합한다. 병합 CSV 는 기존 열 구조를 유지하고 unique_id 를 1 부터 다시 부여한다.

```
export REALVUL_TRAIN_OUT=dataset_pipeline/output/realvul
export REALVUL_TEST_OUT=dataset_pipeline/output/vpbench
export
REALVUL_MERGED_OUT=dataset_pipeline/output/extended_realvul

test -f "$REALVUL_TRAIN_OUT/Real_Vul_data.csv"
test -f "$REALVUL_TEST_OUT/Real_Vul_data.csv"

mkdir -p "$REALVUL_MERGED_OUT"
```

```
python3 - <<'PY'
import csv
import os
from pathlib import Path

input_csvs = [
    Path(os.environ["REALVUL_TRAIN_OUT"]) / "Real_Vul_data.csv",
    Path(os.environ["REALVUL_TEST_OUT"]) / "Real_Vul_data.csv",
]
out_csv = Path(os.environ["REALVUL_MERGED_OUT"]) /
"Real_Vul_data.csv"

with input_csvs[0].open(newline="", encoding="utf-8") as f:
    header = next(csv.reader(f))
if "unique_id" not in header:
    raise SystemExit("unique_id column is required")

out_csv.parent.mkdir(parents=True, exist_ok=True)
with out_csv.open("w", newline="", encoding="utf-8") as out_f:
    writer = csv.DictWriter(out_f, fieldnames=header)
    writer.writeheader()

    next_uid = 1
    for path in input_csvs:
        with path.open(newline="", encoding="utf-8") as in_f:
            reader = csv.DictReader(in_f)
            if reader.fieldnames != header:
                raise SystemExit(f"CSV header mismatch: {path}")
            for row in reader:
                row["unique_id"] = str(next_uid)
                writer.writerow(row)
                next_uid += 1
PY

test -s "$REALVUL_MERGED_OUT/Real_Vul_data.csv"
```

A.2.5 Extended RealVul-FuncPair 생성

이 단계의 목적은 병합 CSV 에서 취약 함수와 패치 후 정상 함수 후보를 매칭하는 것이다. build_jasper_vul_patch_dataset.py 는 이름에 jasper 가 남아 있지만, 현재 CLI 는 전체 프로젝트를 처리한다. 매칭은 project, commit_hash, 함수명, local git cache, GitHub API 조회 결과를 사용한다.

- 입력: --realvul-csv 에 병합한 Extended RealVul CSV 를 지정한다.
- 출력: --output-csv 에 최종 FuncPair CSV 경로를 지정한다.
- 로그: --match-log-csv 에 매칭 성공 및 실패 과정의 로그 CSV 경로를 지정한다.

GitHub API token 을 확인하고 Extended RealVul-FuncPair 를 만든다.

```
test -n "$GITHUB_TOKEN"

python dataset_pipeline/scripts/build_jasper_vul_patch_dataset.py ₩
  --realvul-csv "$REALVUL_MERGED_OUT/Real_Vul_data.csv" ₩
  --output-csv
"$REALVUL_MERGED_OUT/all_projects_vul_patch_dataset.csv" ₩
  --match-log-csv
"$REALVUL_MERGED_OUT/all_projects_vul_patch_match_log.csv" ₩
  --github-token "$GITHUB_TOKEN"
```

A.2.6 출력 확인

다음 파일을 확인한다.

- dataset_pipeline/output/realvul/Real_Vul_data.csv: RealVul 원본 기반 학습 및 검증용 분할 파일이며, run.sh --mode realvul 이 생성한 파일이어야 한다.
- dataset_pipeline/output/vpbench/Real_Vul_data.csv: NVD CVE JSON 파일에서 재구성한 테스트용 분할 파일이다.
- dataset_pipeline/output/extended_realvul/Real_Vul_data.csv: 두 분할을 합친 Extended RealVul 전체 CSV 이며, unique_id 가 병합 후 다시 부여되어 있어야 한다.
- dataset_pipeline/output/extended_realvul/all_projects_vul_patch_dataset.csv: Extended RealVul-FuncPair 파일이며, 취약 함수 행과 패치 후 정상 함수 후보 행이 모두 있어야 한다.

A.3 SARD-Juliet SA Trace 및 SARD-Juliet SA TracePair 구축

시작점(Source)은 취약 흐름이 시작되는 후보 위치이다. 도달점(Sink)은 보안상 문제가 되는 값 사용 후보 위치이다. 오염 분석 설정(taint configuration)은 Source 에서 Sink 까지의 값 흐름을 Infer 에 알려주는 설정이다. Infer 는 Juliet C 테스트 케이스를 정적 분석해 Trace 후보를 산출하는 도구이다.

tools/run_pipeline.py full --all 은 Source/Sink 후보 산출, Infer 실행, Trace 정규화, CSV export 를 한 번에 수행한다.

A.3.1 입력 확인

이 절에서 사용하는 입력은 다음과 같다.

- Juliet C 테스트 케이스: juliet-test-suite-v1.3/C 에 manifest.xml 과 testcases/가 있어야 한다. 누락된 경우 NIST SARD 의 Juliet C/C++ 1.3 test suite #112(https://samate.nist.gov/SARD/test-suites/112)를 받아 같은 구조로 압축을 푼다.
- Source/Sink 후보: manifest.xml, 테스트 케이스 주석, 파일명과 함수명 규칙에서 읽는다.
- Infer taint 설정 파일: pipeline 은 run 디렉터리의 02a_taint/pulse-taint-config.json 을 생성하고 Infer 는 이 생성본을 사용한다. 생성본이 없을 때만 config/pulse-taint-config.json 을 fallback 으로 사용한다.
- 분할 파라미터: 아래 예시는 저자 실행에서 사용한 --pair-split-seed 1234, --pair-train-ratio 0.8 이다.

A.3.2 실행

juliet-playground 작업 공간을 활성화하고 Source/Sink 후보 산출, Infer 실행, Trace 정규화, 데이터셋 export 를 수행한다.

```
cd "$REPRO_ROOT/juliet-playground"
source .venv/bin/activate

python tools/run_pipeline.py full --all ₩
  --pair-split-seed 1234 ₩
  --pair-train-ratio 0.8
```

A.3.3 출력 확인

가장 최근 생성된 run 디렉터리를 찾고 Trace 및 TracePair CSV 가 생성됐는지 확인한다.

```
LATEST_RUN=$(ls -td artifacts/pipeline-runs/run-* | head -n 1)
echo "$LATEST_RUN"

test -s "$LATEST_RUN/07_dataset_export/Real_Vul_data.csv"
test -s
"$LATEST_RUN/07_dataset_export/vuln_patch/Real_Vul_data.csv"
```

- $LATEST_RUN/07_dataset_export/Real_Vul_data.csv: Trace 행이 있는지 확인한다.
- $LATEST_RUN/07_dataset_export/vuln_patch/Real_Vul_data.csv: 취약 Trace 와 정상 Trace 행이 모두 있는지 확인한다.

부록 B. RAG 기반 LLM Few-shot 취약점 탐지 가이드

이 부록은 RAG 기반 LLM Few-shot 평가 절차를 정리한다. 대상은 Extended RealVul-FuncPair, SARD-Juliet SA TracePair, 10 개 CVE TracePair 이다. 구현 코드는 seokjeon/CVD 저장소(https://github.com/seokjeon/CVD)의 llm_cvd 에 있다. 실행 흐름은 환경 준비, 데이터셋 다운로드, RAG 검색 DB 생성, LLM 평가, 결과 집계이다.

실행에는 다음 스크립트를 사용한다. llm_cvd/retrieval/build_index.py 는 검색용 CSV 의 코드 또는 Trace 를 CodeBERT 로 임베딩하고, FAISS 벡터 인덱스와 예시 메타데이터로 구성된 RAG 검색 DB 를 만든다. llm_cvd/evaluation/rag_api.py 는 top-6 예시를 검색해 LLM Few-shot 평가를 실행한다. llm_cvd/analysis/analyze_results.py 는 반복별 성능, 평균 성능, 지연 시간, 토큰 사용량, 비용을 집계한다.

B.1 실행 전제

모든 명령은 seokjeon/CVD 저장소 루트에서 실행한다. 먼저 Python 의존성을 설치한다.

```
python3 -m pip install -r requirements.txt
```

LLM API 키는 llm_cvd/.env 파일 또는 셸 환경변수에 설정한다. API 키가 들어간 .env 파일은 저장소에 커밋하지 않는다. 네 개 LLM 제공자를 모두 실행하려면 OPENAI_API_KEY, ANTHROPIC_API_KEY, GOOGLE_API_KEY, XAI_API_KEY 가 필요하다. .env 에는 필요에 따라 OPENAI_MODEL, CLAUDE_MODEL, GEMINI_MODEL, GROK_MODEL 도 지정할 수 있다.

평가 모델, 반복 횟수, 검색 예시 수, 출력 라벨은 본문 표 4.2 의 설정을 따른다.

본문 표 4.2 와 같이 CodeBERT 검색은 하위 단어 토큰 앞 512 개를 사용한다. 이를 위해 B.3 에서 RAG 검색 DB 를 만들기 전에 다음 명령을 seokjeon/CVD 저장소 루트에서 한 번 실행한다. right 로 설정하면 긴 입력의 뒤쪽을 자르고 코드 또는 Trace 의 앞부분을 기준으로 검색한다.

```
python3 -c 'from pathlib import Path;
p=Path("llm_cvd/retrieval/rag_retriever.py");
p.write_text(p.read_text().replace("embedding_truncation_side: str =
₩"left₩"", "embedding_truncation_side: str = ₩"right₩""))'
```

이 설정을 적용한 상태에서 B.3 의 RAG 검색 DB 생성 명령을 실행한다. 기존 인덱스가 있으면 --rebuild-index 옵션으로 이전 설정의 인덱스를 재사용하지 않는다.

B.2 실험용 데이터셋 준비

다음 명령은 실험 CSV 를 dataset/ 아래에 내려받는다.

```
mkdir -p dataset

base="https://github.com/seokjeon/CVD/releases/download/dataset"

for file in ₩
  juliet_Real_Vul_data.csv ₩
  juliet_Real_Vul_train_val.csv ₩
  juliet_Pair_Real_Vul_data.csv ₩
  cve_Real_Vul_data.csv ₩
  Extended_Real_Vul_data_test.csv ₩
  VulnPatchDS_Vul_data.csv
do
  curl -L --fail -o "dataset/${file}" "${base}/${file}"
done
```

내려받은 CSV 는 다음 세 실험에서 사용한다.

실험	RAG 검색 DB	평가 데이터
SARD-Juliet SA TracePair	dataset/juliet_Real_Vul_train_val.csv	dataset/juliet_Pair_Real_Vul_data.csv
10 개 CVE TracePair	dataset/juliet_Real_Vul_data.csv	dataset/cve_Real_Vul_data.csv

Extended RealVul-FuncPair	dataset/Extended_Real_Vul_data_test.csv	dataset/VulnPatchDS_Vul_data.csv

B.3 RAG 검색 DB 생성

다음 명령은 세 RAG 검색 DB 를 만든다. SARD-Juliet SA TracePair 는 평가 쌍을 제외한 juliet_Real_Vul_train_val.csv, 10 개 CVE TracePair 는 Juliet 전체 CSV, Extended RealVul-FuncPair 는 Extended_Real_Vul_data_test.csv 를 사용한다.

```
build_rag_db() {
  python3 -m llm_cvd.retrieval.build_index ₩
    --rag-dataset-root "$1" ₩
    --cache-dir llm_cvd/cache ₩
    --index-dir llm_cvd/indexes ₩
    --index-name "$2" ₩
    --index-batch-size 16 ₩
    --rebuild-index
}

build_rag_db dataset/juliet_Real_Vul_train_val.csv
juliet_Real_Vul_train_val_codebert
build_rag_db dataset/juliet_Real_Vul_data.csv
juliet_Real_Vul_data_codebert
build_rag_db dataset/Extended_Real_Vul_data_test.csv
extended_realvul_test_codebert
```

산출물은 llm_cvd/indexes/ 아래의 .index 와 _metadata.pkl 파일이다.

B.4 LLM 평가와 결과 집계

다음 함수로 세 실험을 실행한다.

```
run_rag_eval() {
  python3 -m llm_cvd.evaluation.rag_api ₩
    --db-name "$1" ₩
    --rag-dataset-root "$2" ₩
    --target-dataset-csv "$3" ₩
    --cache-dir llm_cvd/cache ₩
    --index-dir llm_cvd/indexes ₩
    --index-name "$4" ₩
```

```
    --providers chatgpt,claude,gemini,grok ₩
    --models "chatgpt=gpt-4o,claude=claude-sonnet-4-5-
20250929,gemini=gemini-2.5-pro,grok=grok-4-1-fast-non-reasoning"
₩
    --k 6 ₩
    --repeats 5 ₩
    --max-workers 4 ₩
    --output-dir llm_cvd/results ₩
    --run-name "$5" ₩
    --env-file llm_cvd/.env
}

run_rag_eval ₩
  sard-juliet ₩
  dataset/juliet_Real_Vul_train_val.csv ₩
  dataset/juliet_Pair_Real_Vul_data.csv ₩
  juliet_Real_Vul_train_val_codebert ₩
  sard_juliet_k6_r5

run_rag_eval ₩
  cve10 ₩
  dataset/juliet_Real_Vul_data.csv ₩
  dataset/cve_Real_Vul_data.csv ₩
  juliet_Real_Vul_data_codebert ₩
  cve10_k6_r5

run_rag_eval ₩
  extended-realvul-funcpair ₩
  dataset/Extended_Real_Vul_data_test.csv ₩
  dataset/VulnPatchDS_Vul_data.csv ₩
  extended_realvul_test_codebert ₩
  extended_realvul_funcpair_k6_r5
```

rag_api.py 는 같은 --run-name 의 기존 성공 결과를 건너뛴다. 오류가 있으면 원인을 처리한 뒤 같은 --run-name 으로 다시 실행한다. Extended RealVul-FuncPair 전체 평가는 8,730 개 샘플, 4 개 LLM, 5 회 반복으로 구성된다. 따라서 174,600 회 호출이 발생한다. 실행 전에 비용과 시간을 확인한다.

다음 명령은 JSONL 결과를 반복별 지표와 요약 지표로 변환한다.

```
for RUN_NAME in ₩
  sard_juliet_k6_r5 ₩
  cve10_k6_r5 ₩
  extended_realvul_funcpair_k6_r5
```

```
do
  python3 llm_cvd/analysis/analyze_results.py
"llm_cvd/results/${RUN_NAME}.jsonl" ₩
    --pricing llm_cvd/pricing_config.json ₩
    --per-repeat-output "llm_cvd/results/${RUN_NAME}_by_repeat.csv"
₩
    --output "llm_cvd/results/${RUN_NAME}_summary.csv"
done
```

B.5 프롬프트 템플릿

RAG Few-shot 평가는 다음 system prompt 와 user prompt 를 사용한다. 아래 템플릿은 실제 평가 프롬프트를 그대로 옮긴 것이다. 프롬프트 문구는 source code 를 대상으로 하지만, Trace 실험에서는 Trace 문자열을 같은 Code 필드에 넣었다.

B.5.1 System prompt

```
You are a binary vulnerability classifier. You must output exactly one of
these two labels: Vulnerable or Safe. Do not explain your reasoning. Do
not include punctuation, markdown, or extra words.
```

B.5.2 User prompt 템플릿

```
Classify the source code into Vulnerable or Safe, and return the answer
as the corresponding label. Here are some examples:

Code:
{example_code_or_trace_1}
Label:
{example_label_1}

Code:
{example_code_or_trace_2}
Label:
{example_label_2}

...

Code:
{example_code_or_trace_6}
```

Label:
{example_label_6}

Code:
{target_code_or_trace}
Answer with exactly one label, either Vulnerable or Safe.
Label:

부록 C. t-SNE 재현 가이드

이 부록은 본문 4 장의 t-분포 확률적 이웃 임베딩(t-distributed stochastic neighbor embedding, t-SNE) 그림을 재현하기 위한 절차를 정리한다. t-SNE 는 고차원 임베딩을 2 차원으로 줄여 샘플 분포를 확인하는 시각화 방법이다.

C.1 재현 범위

재현 그림	데이터셋 또는 입력	대상
그림 4.5, 그림 4.6	Extended RealVul-FuncPair	LineVul, PDBERT
그림 4.10(a), 그림 4.10(b)	SARD-Juliet SA TracePair	LineVul, PDBERT
그림 4.13(a), 그림 4.13(b)	10 개 CVE TracePair	LineVul, PDBERT
그림 4.15	10 개 CVE Trace 질의와 검색된 Juliet 예시	CodeBERT 기반 검색 임베딩

C.2 딥러닝 모델 t-SNE 재현

이 절은 LineVul 과 PDBERT 의 t-SNE 그림을 재현하는 절차이다.

C.2.1 실험 환경 설정

공통적으로 부록 A 에서 데이터셋 구축을 수행한 환경을 사용한다. Extended RealVul-FuncPair 는 Python 스크립트가 --extended-realvul 옵션으로 VP-Bench release 데이터셋과 파인튜닝 후 모델을 내려받아 준비하므로 별도 준비가 필요하지 않다. SARD-Juliet SA TracePair 와 10 개 CVE TracePair 는 부록 A.3.3 에서 생성한 최신 pipeline run($LATEST_RUN)과 juliet-playground/cases/Real_Vul_data.csv 를 사용한다.

모델 실행의 경우 Docker 컨테이너를 배포해야 한다. 컨테이너는 파인튜닝과 t-SNE 추출 과정에서 임베딩 벡터 값을 계산하므로 GPU 를 사용할 수 있어야 한다.

재현에 활용된 GPU 사양은 부록 A.1 에 제시되어 있다. 더불어 컨테이너 배포에는 Docker 와 Docker Compose 가 필요하다.

```
cd "$VP_BENCH_ROOT"
bash ./experiment/run.sh -m linevul,pdbert
```

C.2.2 LineVul

LineVul 재현은 juliet-playground 저장소의 tools/run_linevul.py 를 사용한다.

C.2.2.1 Extended RealVul-FuncPair

이 명령은 그림 4.5 에 대응하는 LineVul t-SNE 를 만든다.

```
cd "$JULIET_PLAYGROUND_ROOT"
python3 tools/run_linevul.py ₩
  --vpbench-root "$VP_BENCH_ROOT" ₩
  --container-name linevul ₩
  --extended-realvul
```

C.2.2.2 SARD-Juliet SA TracePair

이 명령은 그림 4.10(a)에 대응하는 LineVul t-SNE 를 만든다.

```
cd "$JULIET_PLAYGROUND_ROOT"
python3 tools/run_linevul.py ₩
  --vpbench-root "$VP_BENCH_ROOT" ₩
  --container-name linevul ₩
  --run-dir "$LATEST_RUN"
```

C.2.2.3 10 개 CVE TracePair

그림 4.13(a)처럼 SARD-Juliet SA TracePair 와 10 개 CVE TracePair 를 함께 배치하려면, 먼저 C.2.2.2 를 실행해 같은 실행 디렉터리의 SARD-Juliet SA TracePair 산출물을 준비한다.

이 명령은 그림 4.13(a)에 대응하는 LineVul t-SNE 를 만든다.

```
cd "$JULIET_PLAYGROUND_ROOT"
python3 tools/run_linevul.py ₩
```

```
  --vpbench-root "$VP_BENCH_ROOT" ₩
  --container-name linevul ₩
  --run-dir "$LATEST_RUN" ₩
  --cve-trace-pair ₩
  --triplet-tsne
```

C.2.3 PDBERT

PDBERT 재현은 juliet-playground 저장소의 tools/run_pdbert.py 를 사용한다.

C.2.3.1 Extended RealVul-FuncPair

이 명령은 그림 4.6 에 대응하는 PDBERT t-SNE 를 만든다.

```
cd "$JULIET_PLAYGROUND_ROOT"
python3 tools/run_pdbert.py ₩
  --vpbench-root "$VP_BENCH_ROOT" ₩
  --container-name pdbert ₩
  --extended-realvul
```

C.2.3.2 SARD-Juliet SA TracePair

이 명령은 그림 4.10(b)에 대응하는 PDBERT t-SNE 를 만든다.

```
cd "$JULIET_PLAYGROUND_ROOT"
python3 tools/run_pdbert.py ₩
  --vpbench-root "$VP_BENCH_ROOT" ₩
  --container-name pdbert ₩
  --run-dir "$LATEST_RUN"
```

C.2.3.3 10 개 CVE TracePair

그림 4.13(b)처럼 SARD-Juliet SA TracePair 와 10 개 CVE TracePair 를 함께 배치하려면, 먼저 C.2.3.2 를 실행한다.

이 명령은 그림 4.13(b)에 대응하는 PDBERT t-SNE 를 만든다.

```
cd "$JULIET_PLAYGROUND_ROOT"
python3 tools/run_pdbert.py ₩
  --vpbench-root "$VP_BENCH_ROOT" ₩
  --container-name pdbert ₩
  --code-truncation-side tail ₩
```

```
  --run-dir "$LATEST_RUN" ₩
  --cve-trace-pair ₩
  --triplet-tsne
```

C.3 검색 임베딩 t-SNE 재현

이 절은 그림 4.15 의 CodeBERT 기반 RAG 검색 임베딩 시각화를 재현하는 절차를 다룬다. 그림 4.15 는 CodeBERT 검색 임베딩 기준으로 CVE Trace 질의 주변에 어떤 Juliet 예시가 가까이 놓이는지 확인하는 그림이다.

C.3.1 실행 환경과 입력 준비

실행 환경은 부록 B 를 따라 준비한다. seokjeon/CVD 저장소 루트에서 부록 B.1 의 Python 의존성 설치, 부록 B.2 의 데이터셋 다운로드, 부록 B.3 의 juliet_Real_Vul_data_codebert RAG 검색 DB 생성을 먼저 수행한다.

실행 환경은 부록 B 를 따라 준비한다. seokjeon/CVD 저장소 루트에서 부록 B.1 의 Python 의존성 설치, 부록 B.2 의 데이터셋 다운로드, 부록 B.3 의 juliet_Real_Vul_data_codebert RAG 검색 DB 생성을 먼저 수행한다. 부록 B 를 수행하면 10 개 CVE Trace 질의 CSV, Juliet 검색 대상 CSV, CodeBERT 임베딩 검색 index, 검색 index 의 Juliet 샘플 정보가 준비된다.

검색 임베딩 t-SNE 재현에는 CVE Trace 질의별로 RAG 가 선택한 Juliet 예시 목록도 필요하다. 시각화 스크립트는 이 목록을 읽어 어떤 Juliet 샘플을 검색 결과 점으로 표시할지 결정한다. 이 파일은 다음 명령으로 만든다.

```
export CVD_ROOT="$HOME/CVD"
cd "$CVD_ROOT"

OUT_DIR=llm_cvd/resultStar/juliet_Real_Vul_data__cve_Real_Vul_data
mkdir -p "$OUT_DIR"

python3 llm_cvd/analysis/inspect_rag_retrieval.py ₩
  --rag-dataset-root dataset/juliet_Real_Vul_data.csv ₩
```

```
  --target-dataset-csv dataset/cve_Real_Vul_data.csv ₩
  --cache-dir llm_cvd/cache ₩
  --index-dir llm_cvd/indexes ₩
  --index-name juliet_Real_Vul_data_codebert ₩
  --k 6 ₩
  --output-csv "$OUT_DIR/rag_retrieval_inspection_cve_k6.csv"
```

C.3.2 실행

이 명령은 그림 4.15 에 대응하는 CodeBERT 검색 임베딩 t-SNE 를 만든다.

```
cd "$CVD_ROOT"

OUT_DIR=llm_cvd/resultStar/juliet_Real_Vul_data__cve_Real_Vul_data
python3 llm_cvd/analysis/visualize_rag_tsne.py ₩
  --inspection-csv "$OUT_DIR/rag_retrieval_inspection_cve_k6.csv" ₩
  --target-dataset-csv dataset/cve_Real_Vul_data.csv ₩
  --rag-dataset-root dataset/juliet_Real_Vul_data.csv ₩
  --index-dir llm_cvd/indexes ₩
  --index-name juliet_Real_Vul_data_codebert ₩
  --output-csv "$OUT_DIR/rag_tsne_cve_k6_points.csv" ₩
  --output-html "$OUT_DIR/rag_tsne_cve_k6.html" ₩
  --output-png "$OUT_DIR/rag_tsne_cve_k6.png"
```

감사의 글

먼저 연구 방향을 잡아 주시고, 연구가 더디게 진행될 때에도 인내심 있게 지도해 주신 최광훈 지도교수님께 깊이 감사드립니다. 교수님께서는 연구에 전념할 수 있는 환경을 마련해 주셨고, 논문의 완성도를 높일 수 있도록 세심하게 이끌어 주셨습니다.

논문 심사 과정에서 원고를 면밀히 검토해 주시고 보완할 부분을 짚어 주신 엄익채 교수님과 유동현 교수님께 감사드립니다. 두 분의 조언 덕분에 연구의 논리와 서술을 더 분명하게 정리할 수 있었습니다.

연구 과정에서 RAG 기반 LLM 평가에 도움을 준 연구실 동료 오다영과, 벤치마크를 위한 딥러닝 모델 재현 과정에서 여러 도움을 준 정민수에게도 감사의 마음을 전합니다.

학업과 연구 수행을 지원해 주신 융합보안대학원 사업과 KSign 에도 감사드립니다. 연구 지원이 원활히 이루어지도록 힘써 주신 KSign 의 김정미 이사님께도 깊이 감사드립니다.