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

전남대학교 정보보안융합학과
석사학위 청구논문 종심

2026년 6월 10일

전 세 옥

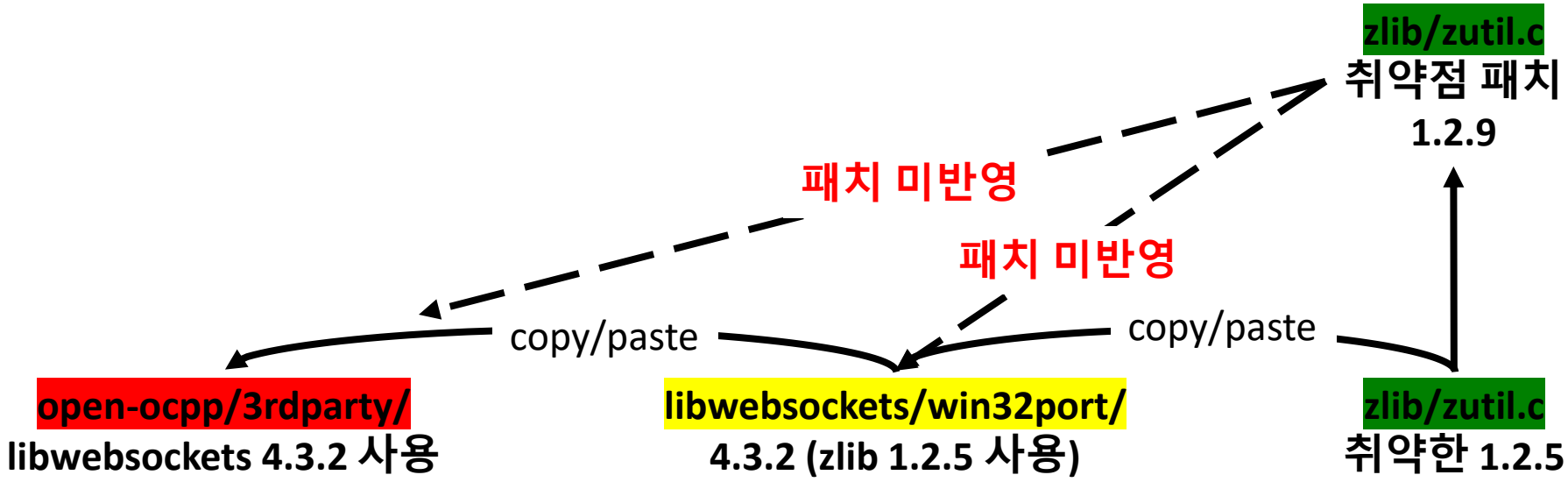
목차

배경 및 문제

전파되는 취약점, 따라오지 않는 패치 [1]

1. zlib 1.2.5 취약 코드가 libwebsockets에 복사
2. libwebsocket가 다시 open-ocpp에 복사

원본 zlib은 1.2.9에서 패치됐지만, 파생 프로젝트에는 미반영



→ 패치 누락 취약점 탐지 필요

배경 및 문제

유사 취약점 재출현 탐지 기법

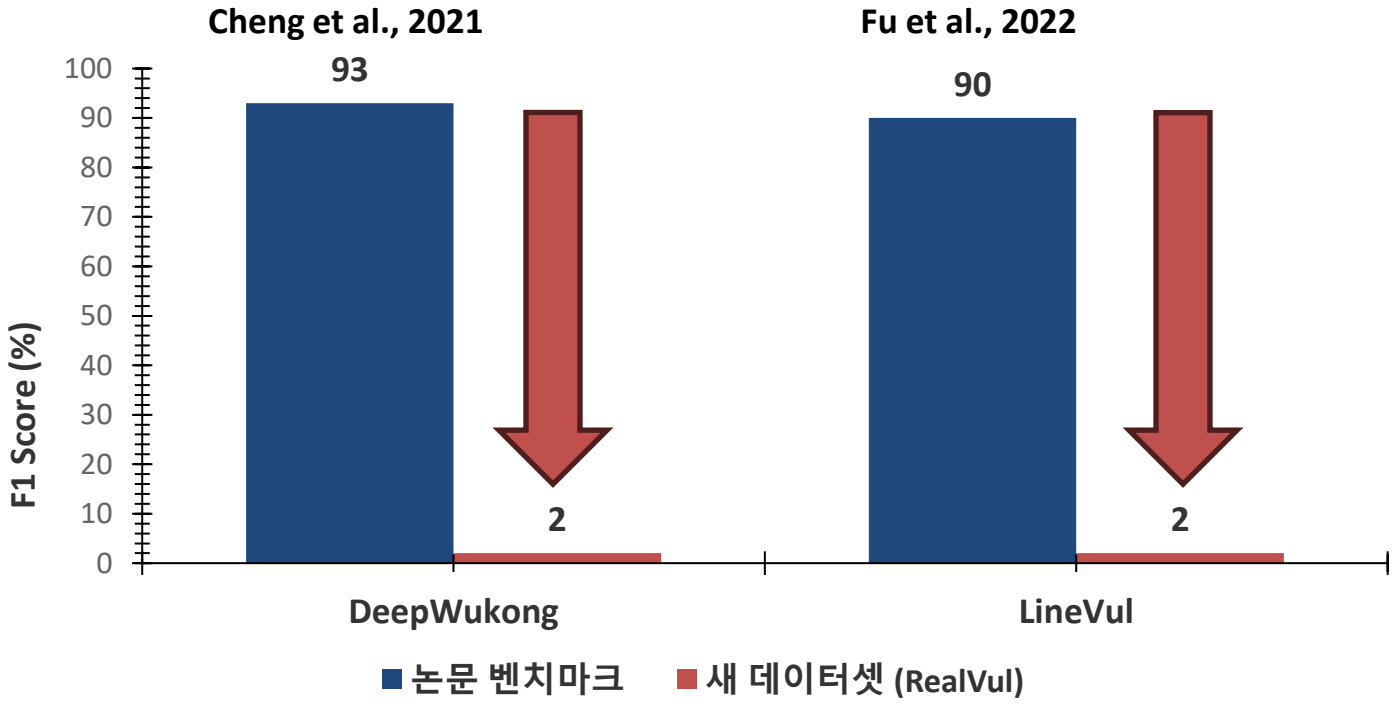
기법	대표 연구	설명
구문 기반 비교	VUDDY	함수 코드의 토큰 리스트(lexer의 출력) 유사도 기준 탐지
머신러닝	Tracer	연산자,위험 API, 경계값 조건의 출현 빈도를 원소로 하는 특징 벡터
딥러닝 모델	DeepWukong, LineVul, PDBERT	취약 코드 기반 데이터셋, GNN 또는 CodeBERT
Large Language Model	Llama-based CVD	프롬프트/RAG/파인튜닝, Claude, GPT, Gemini, LLama

연구 범위

배경 및 문제

새로운 데이터셋에서 논문에서 리포트했던 딥러닝 성능 재현 실패 [2, 3, 4]

DeepWukong(2021)·LineVul(2022)은 Redis 등을 포함한 기존 벤치마크에서 높은 탐지 성능(F1 **0.9**)을 보고
2024년 이후 연구는 RealVul과 같이 QEMU 등을 포함한 새로운 데이터셋에서 성능 재현 실패(F1 **0.02**)를 보고

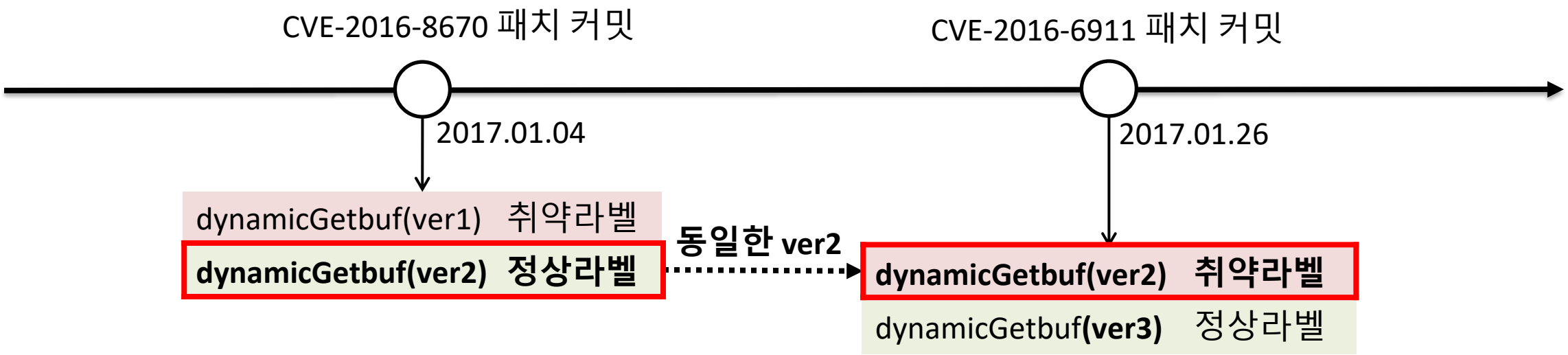


→ 성능 재현 실패의 원인으로 모델 구조뿐 아니라 데이터셋 측면의 검토가 필요

관련 연구

성능 재현 실패 원인: 시간 흐름에 따른 라벨 불일치 [2]

- “그때는 맞고 지금은 틀리다”



→ 동일 코드에 정상/취약 라벨이 함께 부여되어 취약 코드 패턴을 일관되게 학습하기 어려움

연구 방법

성능 재현 실패의 가능한 원인: 함수 단위 라벨링 문제

1. 패치 차이로는 취약/정상 구분 어려움

   주변 문맥 해석해야 취약/정상 구분 가능

2. 지나치게 포괄적인 취약 라벨 범위(granularity)

   취약 샘플에 취약 코드와 정상 코드가 함께 포함

3. 시간 흐름에 따른 정상/취약 라벨 충돌

   과거 취약점 패치 후 정상 라벨 코드가 이후 다른 CVE에서 취약 코드로 확인

연구 방법

함수 단위 라벨링: 1. 패치 차이로는 취약/정상 구분 어려움

작은 경계값 수정 패치만으로는 취약/정상 구분이 어렵고, 주변 문맥 해석이 필요

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

(a) 패치 전 취약 함수

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

(b) 패치 후 정상 함수

<libgd 취약점(CVE-2016-8670)의 패치 전후 코드 예시>

연구 방법

함수 단위 라벨링: 2. 지나치게 포괄적인 취약 라벨 범위

함수 단위 취약 샘플에 취약 코드와 정상 코드가 함께 포함

<함수 단위 샘플>

취약

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

< libgd CVE-2016-8670: 취약 함수 안에 정상 코드가 함께 포함된 사례 >

→ 정상 코드까지 취약 패턴으로 학습될 수 있는 함수 단위 라벨링 한계

연구 방법

함수 단위 라벨링: 3. 시간 흐름에 따른 정상/취약 라벨 충돌

과거 정상 샘플이 이후 취약 샘플로 재수집

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

CVE-2016-8670 패치 전: 취약

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

CVE-2016-8670 패치 후: 정상

CVE-2016-6911 패치 전: 취약

```
01 static int dynamicGetbuf(gdIOCtxPtr ctx, void *buf, int len) {
   ...
06  dp = dcts->dp;
07  if (dp->pos < 0 || dp->pos >= dp->realSize) {
08    return 0;
09  }
10  remain = dp->logicalSize - dp->pos;
11  if(remain >= len) {
12    rlen = len;
13  } else {
14    if(remain <= 0) {
15      return 0;
16    }
17    rlen = remain;
18  }
19  if (dp->pos + rlen > dp->realSize) {
20    rlen = dp->realSize – dp->pos;
21  }
22  memcpy(buf, (void *) ((char *)dp->data + dp->pos), rlen);
    ...
25 }
```

CVE-2016-6911 패치 후: 정상

<동일 코드가 정상 라벨로 학습되었으나, 이후 취약 라벨로 학습되는 사례>

연구 방법

함수 단위 라벨링 해결 방안: Trace 단위 데이터셋

Trace: 입력 Source에서 시작해, 실제 실행 경로를 거쳐, 결과 Sink까지의 문장 리스트

```
01  void CWE195_Signed_to_Unsigned_fgets_memcpy_bad() {
02      int data;
03      data = -1;
04      char inputBuffer[CHAR_ARRAY_SIZE] = "";
05      if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL) {
06          data = atoi(inputBuffer);
07      } else { 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 취약 샘플

Trace 추출

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 취약 샘플

< 음수 길이 값이 memcpy 크기 인자로 전달되는 Trace 사례 >

연구 방법

함수 단위 라벨링 해결 방안: Trace 단위 데이터셋

Trace: 입력 Source에서 시작해, 실제 실행 경로를 거쳐, 결과 Sink까지의 문장 리스트

Trace 단위가 함수 단위 라벨링 한계를 줄이는 이유

1. 패치 차이로는 취약/정상 구분 어려움
   → 취약 원인–결과 흐름을 명시한 Trace로 취약/정상 구분 용이

2. 지나치게 포괄적인 취약 라벨 범위
   → 함수 전체 대신 취약 관련 Trace만 입력으로 사용

3. 시간 흐름에 따른 정상/취약 라벨 충돌
   → 원인–결과 흐름 중심 Trace로 정상/취약 판단 근거가 명확하여 판단을 번복할 가능성이 낮음

연구 방법

정적분석(Taint analysis, Infer[5]) 기반 Trace 추출 방법

오염 분석: 입력(Source) 값이 특정 위험 지점(Sink)의 값에 영향을 주는 코드 경로를 자동으로 추출하는 분석

Source: fgets ~ Sink: memcpy

```
01  void CWE195_Signed_to_Unsigned_fgets_memcpy_bad() {
02      int data;
03      data = -1;
04      char inputBuffer[CHAR_ARRAY_SIZE] = "";
05      if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL) {
06          data = atoi(inputBuffer);
07      } else { 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  }
```

```
01  void CWE195_Signed_to_Unsigned_fgets_memcpy_bad() {
02      int data;
03      data = -1;
04      char inputBuffer[CHAR_ARRAY_SIZE] = "";
05      if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL) {
06          data = atoi(inputBuffer);
07      } else { 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  }
```

실험 및 결과

연구 질문

함수 단위 라벨링의 한계 검증
→ RQ1. 함수 단위 데이터셋은 취약/정상 구분을 어렵게 만드는가?

Trace 단위 데이터셋의 취약/정상 구분 성능 검증
→ RQ2a. Trace 단위 데이터셋은 취약/정상 구분 성능을 개선하는가?
→ RQ2b. Trace 단위 데이터셋의 효과는 오픈소스 CVE에서도 유지되는가?

실험 및 결과

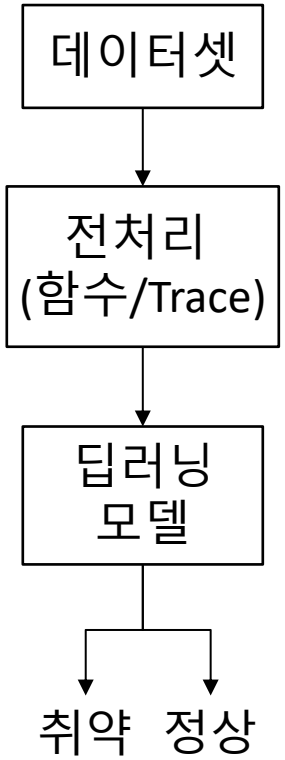
실험 데이터셋과 AI 모델

<table>
<tr><th rowspan="2">연구 질문</th><th rowspan="2">샘플 단위</th><th rowspan="2">학습 / RAG 구축</th><th rowspan="2">설명</th><th rowspan="2">평가</th><th colspan="2">모델</th></tr>
<tr><th>딥러닝</th><th>LLM</th></tr>
<tr><td>RQ1</td><td>함수</td><td>Extended RealVul</td><td>RealVul의 CVE 수집 범위를 확장한 함수 단위 데이터셋</td><td>Extended RealVul-FuncPair</td><td rowspan="3">• Deepwukong
• Linevul
• PDBERT</td><td rowspan="3">• grok-4-1-fast
• gpt-4o
• claude-sonnet-4-5
• gemini-2.5-pro</td></tr>
<tr><td>RQ2a</td><td rowspan="2">Trace</td><td rowspan="2">SARD-Juliet SA Trace</td><td rowspan="2">취약점 탐지 도구 평가용 합성 벤치마크 SARD-Juliet 에서 추출한 Trace 데이터셋</td><td>SARD-Juliet SA TracePair</td></tr>
<tr><td>RQ2b</td><td>10개 CVE 대상 TracePair</td></tr>
</table>

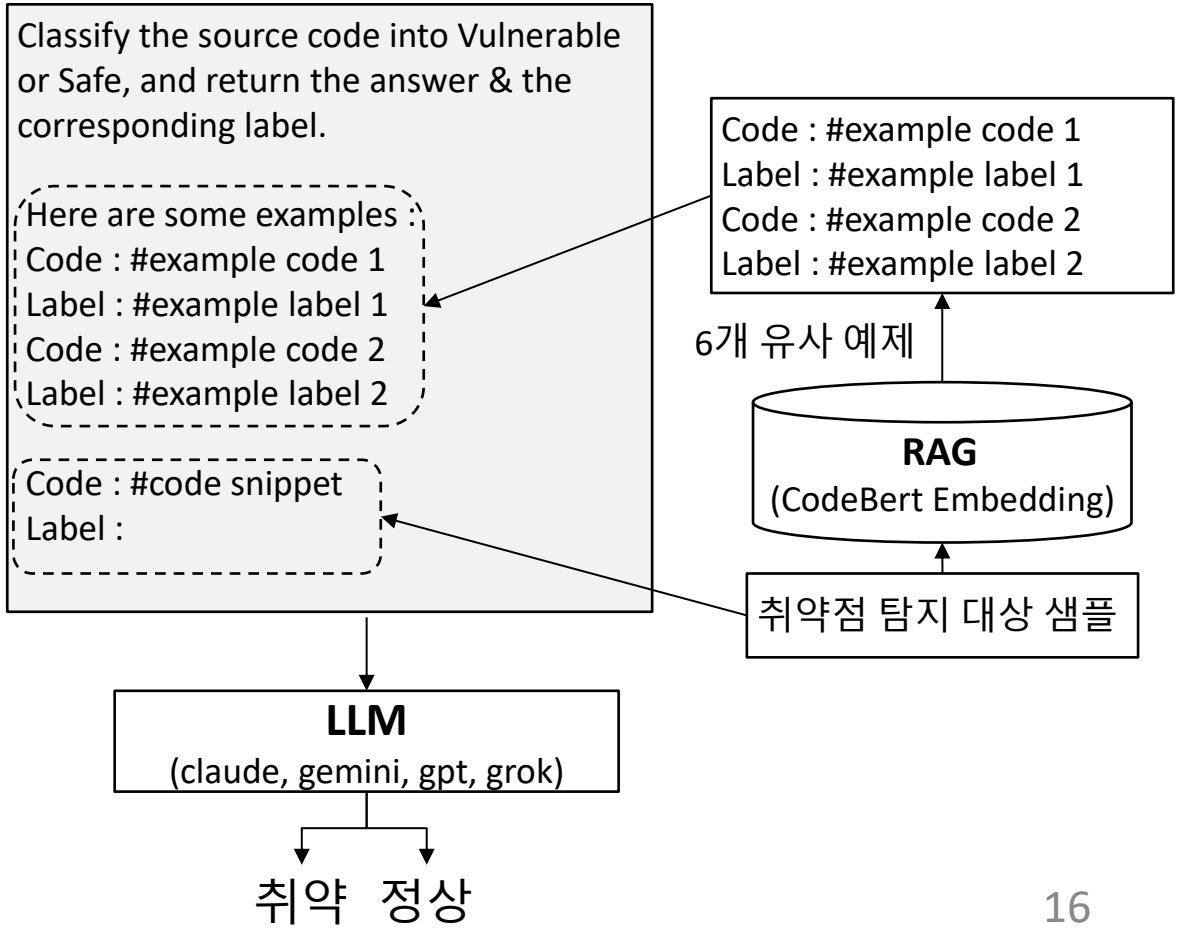
실험 및 결과

평가 방식

- 딥러닝 기반 평가 방식



- RAG 기반 LLM fewshot 평가 방식 [6]

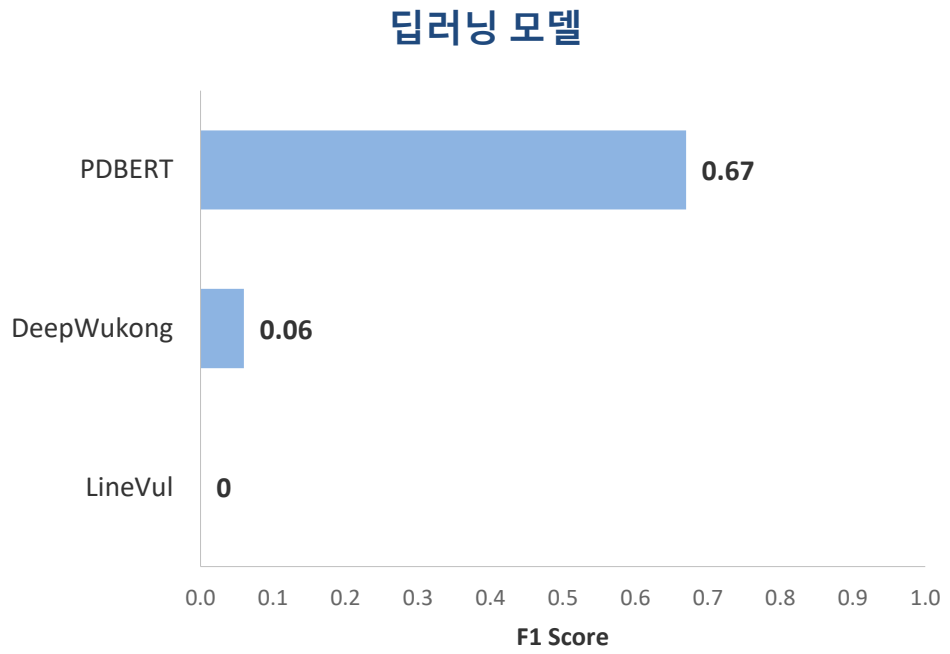


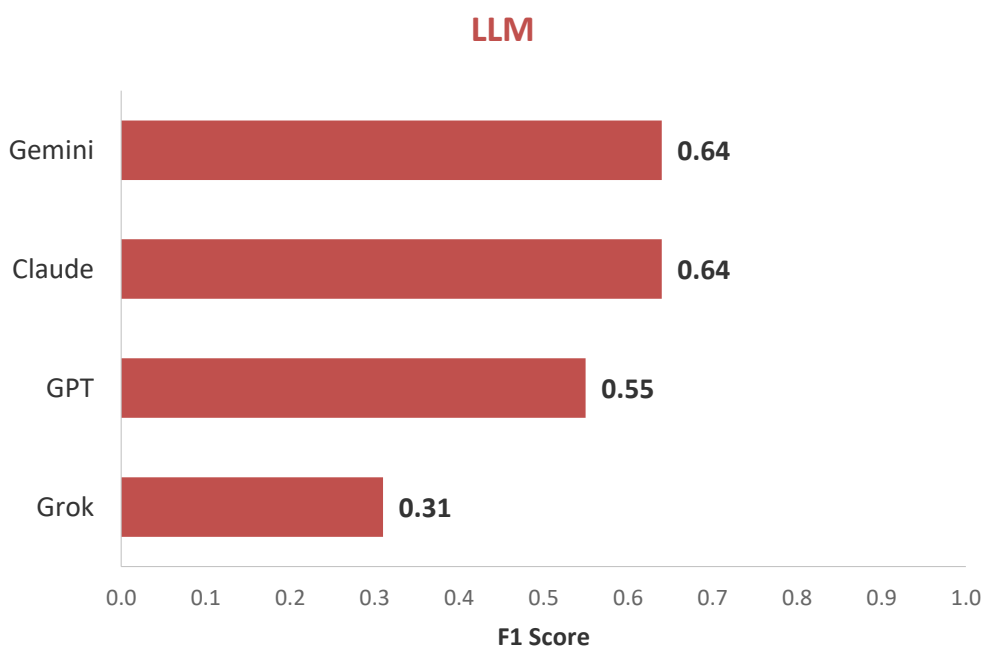
실험 및 결과

RQ1. 함수 단위 데이터셋은 취약/정상 구분을 어렵게 만드는가?

Extended RealVul-FuncPair에서
취약/패치 구분 f1 스코어 성능이 딥러닝과 LLM 모델 일관적으로 낮음 (최고 F1: 딥러닝 0.67, LLM 0.64)

평가: Extended RealVul-FuncPair 8,730개(취약·패치 각 4,365개)
딥러닝 학습: Train 146,552개 / LLM RAG: Test 73,655개



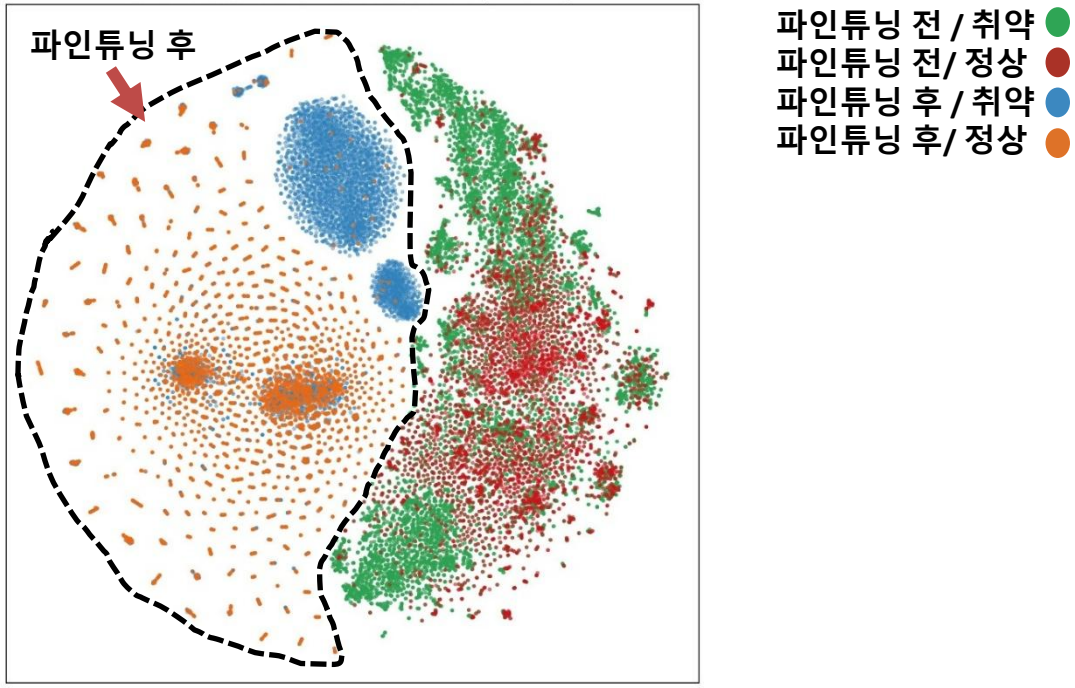


실험 및 결과

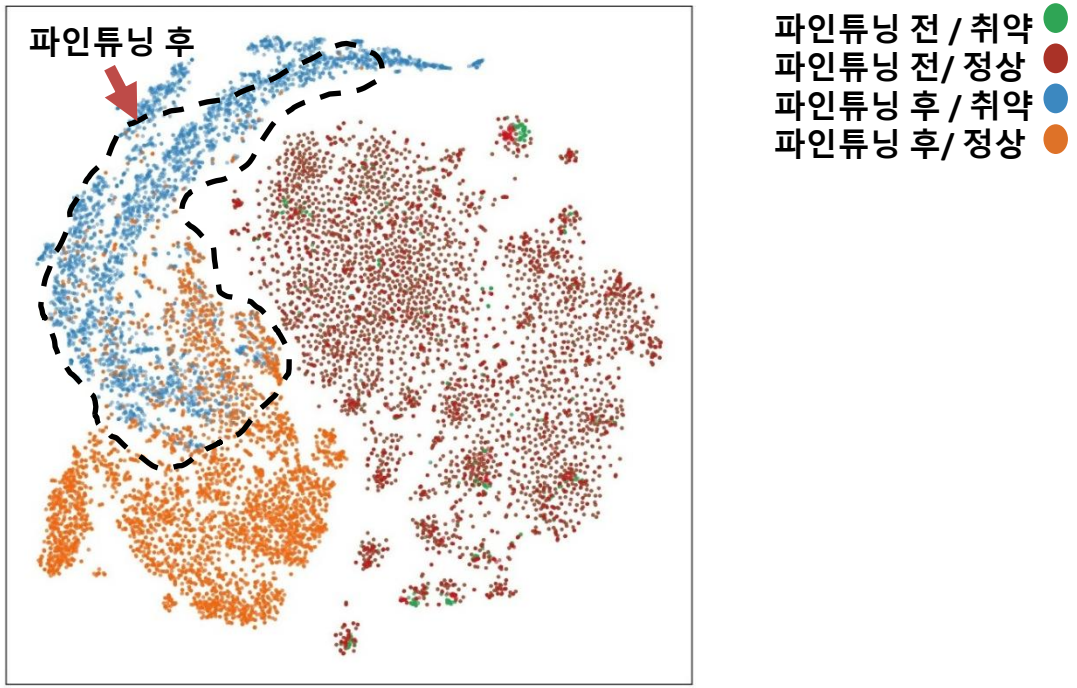
RQ1. 함수 단위 데이터셋은 취약/정상 구분을 어렵게 만드는가?

파인튜닝 전/후 Extended RealVul-FuncPair 데이터셋의 임베딩 벡터 t-SNE 분포 비교 결과,
파인튜닝 후 취약/정상 임베딩이 중첩되어, 함수 단위 데이터셋의 낮은 F1 성능과 일관됨

LineVul



PDBERT



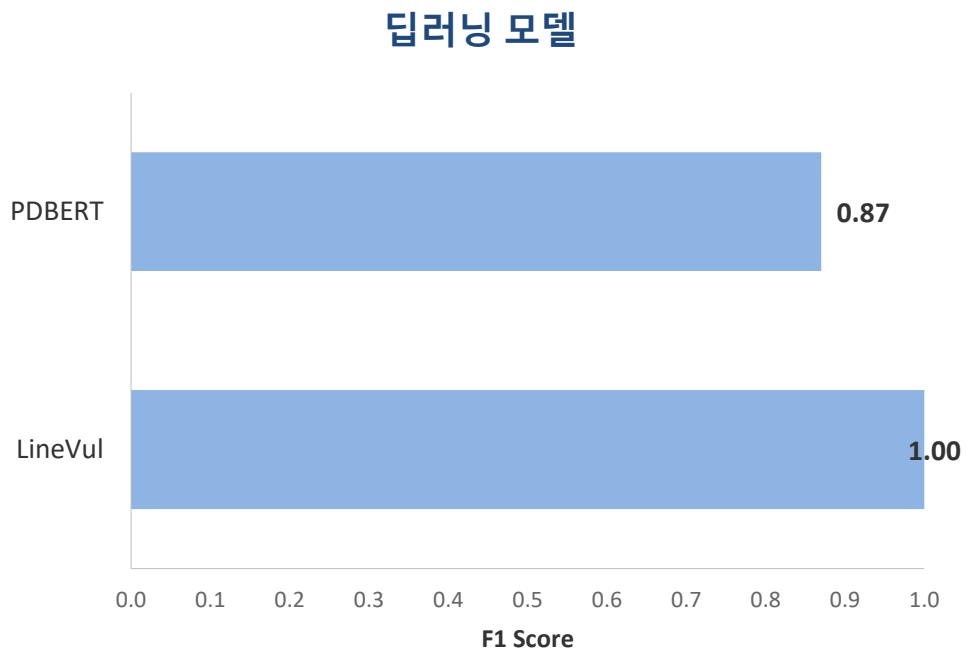
RQ 1 결과. 함수 단위 데이터셋은 취약/정상 구분 한계를 보임

실험 및 결과

RQ2a. Trace 단위 데이터셋은 취약/정상 구분 성능을 개선하는가?

SARD-Juliet TracePair실험에서
FuncPair 취약/패치 학습 및 탐지 성능보다 (최고 F1: 딥러닝 0.67, LLM 0.64)
TracePair 취약/패치 학습 및 탐지 성능이 크게 향상됨(최고 F1: 딥러닝 1.0, LLM 0.96)

평가: SARD-Juliet TracePair 80개(취약·패치 각 40개) / 딥러닝 학습 및 LLM RAG: SARD-Juliet Trace Train 701개

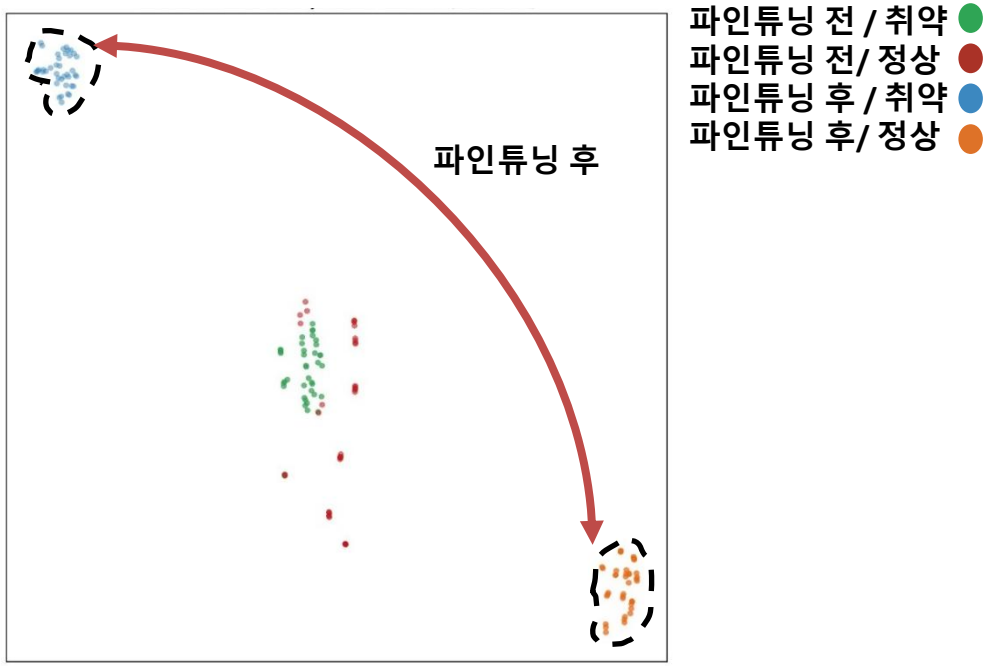


실험 및 결과

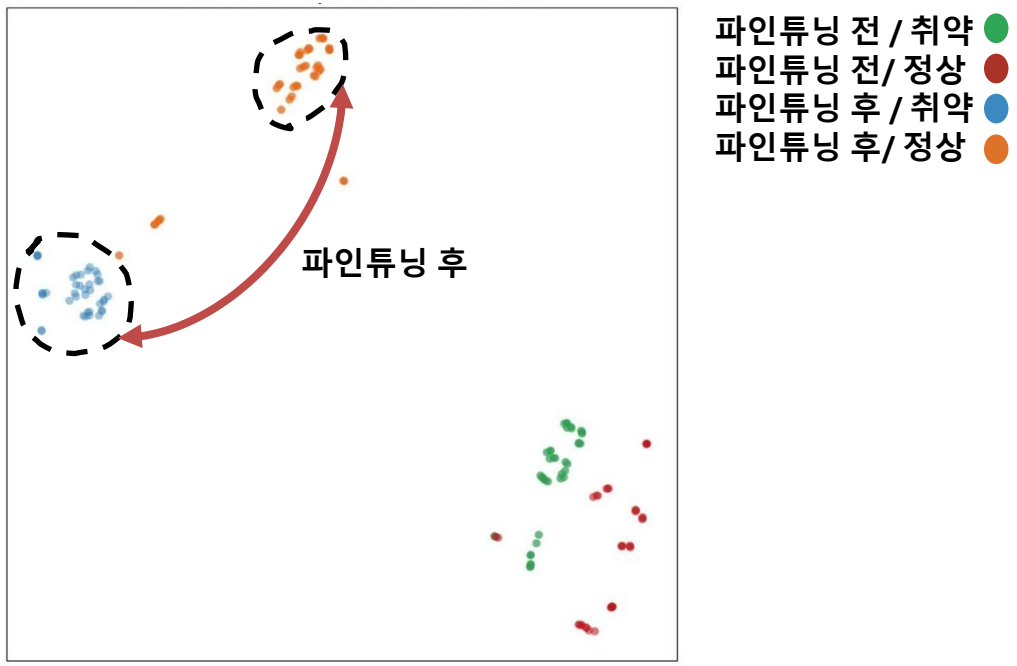
RQ2a. Trace 단위 데이터셋은 취약/정상 구분 성능을 개선하는가?

파인튜닝 전/후 SARD-Juliet TracePair 데이터셋의 임베딩 벡터 t-SNE 분포 비교 결과,
파인튜닝 후 취약/정상 임베딩이 명확히 구분 가능해, SARD-Juliet SA Trace의 높은 F1 성능을 설명

LineVul



PDBERT



RQ 2a 결과. Trace 단위 데이터셋은 취약/정상 구분 성능 개선 가능성을 보임

실험 및 결과

RQ2b. Trace 단위 데이터셋의 효과는 오픈소스 CVE에서도 유지되는가?

- SARD-Juliet TracePair에서는 Trace 단위 데이터셋의 취약/정상 구분 가능성을 확인
- 한계: SARD-Juliet TracePair는 합성 벤치마크로 실제 오픈소스 CVE 환경에서 추가 검증 필요
- 검증: 10개 CVE 대상 TracePair 데이터셋을 구축하여 적용 가능성 평가(github)

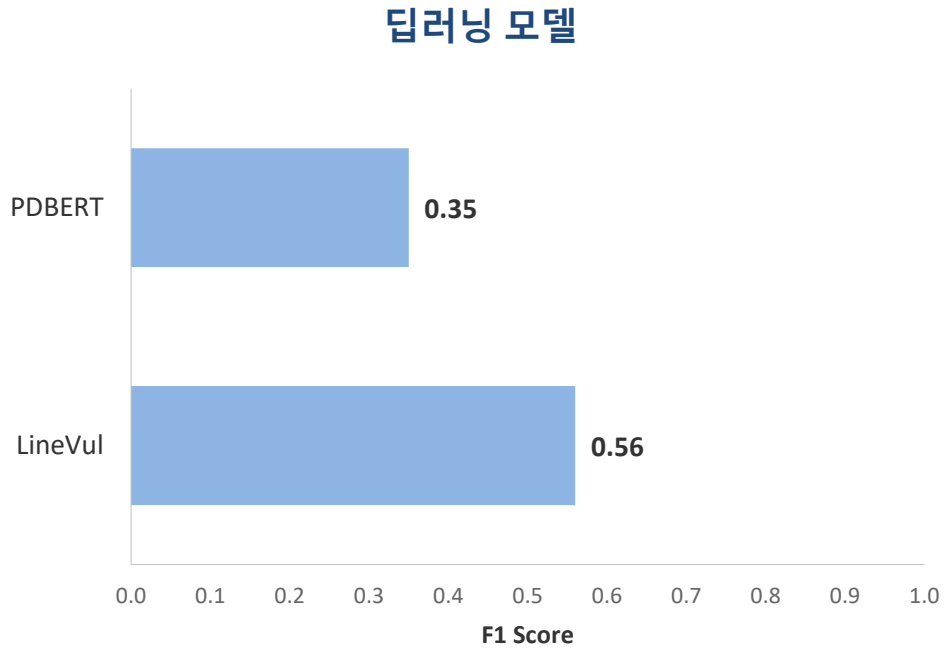
CWE	CVE	오픈소스 SW
CWE-134 (Format String Bug)	CVE-2015-8617	PHP (인터프리터)
	CVE-2017-12588	Rsyslog (로그 수집)
CWE-400 (Resource Exhaustion)	CVE-2017-11142	PHP (HTTP POST 처리)
	CVE-2019-12973	OpenJPEG (JPEG 코덱)
CWE-78 (OS Command Injection)	CVE-2017-15108	Spice Vdagent (VM 게스트 에이전트)
	CVE-2017-15924	Shadowsocks-libev (암호화 프록시)
	CVE-2018-6791	KDE Plasma Workspace (장치 관리자)
	CVE-2018-16863	Ghostscript (문서 뷰어)
	CVE-2019-13638	Patch (패치 프로그램)
	CVE-2019-16718	Radare2 (바이너리 분석 도구)

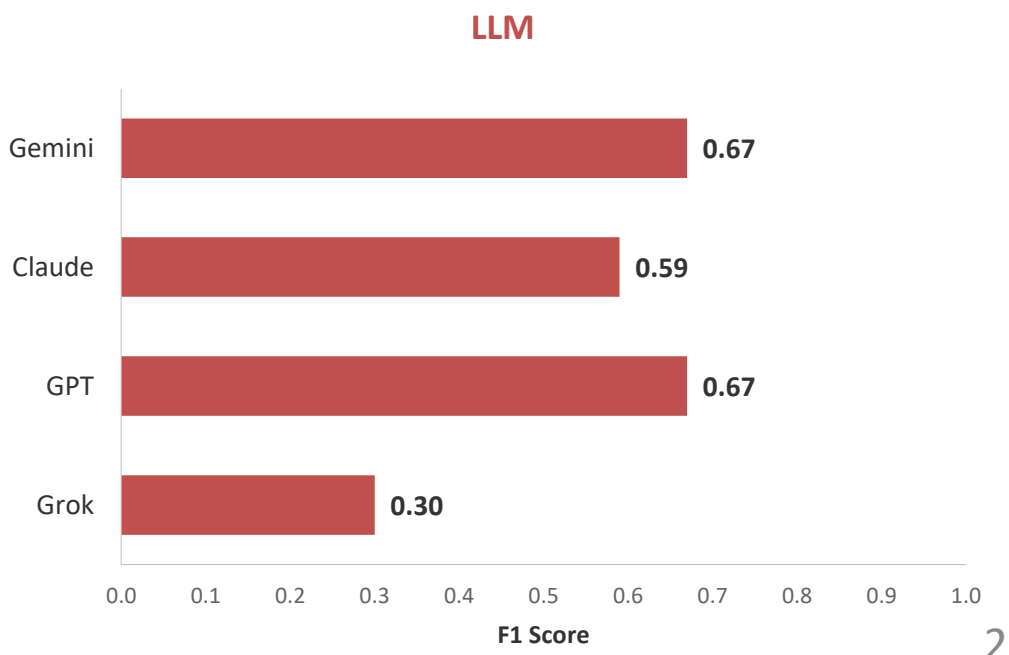
실험 및 결과

RQ2b. Trace 단위 데이터셋의 효과는 오픈소스 CVE에서도 유지되는가?

10개 CVE 대상 CVE TracePair실험에서
TracePair 취약/패치 학습 및 탐지 성능이 제한적임 (최고 F1: 딥러닝 0.56, LLM 0.67)

평가: 10개 CVE 대상 CVE TracePair 20개(취약·패치 각 10개)
딥러닝 학습: SARD-Juliet Trace Train 701개 / LLM RAG: SARD-Juliet Trace 875개



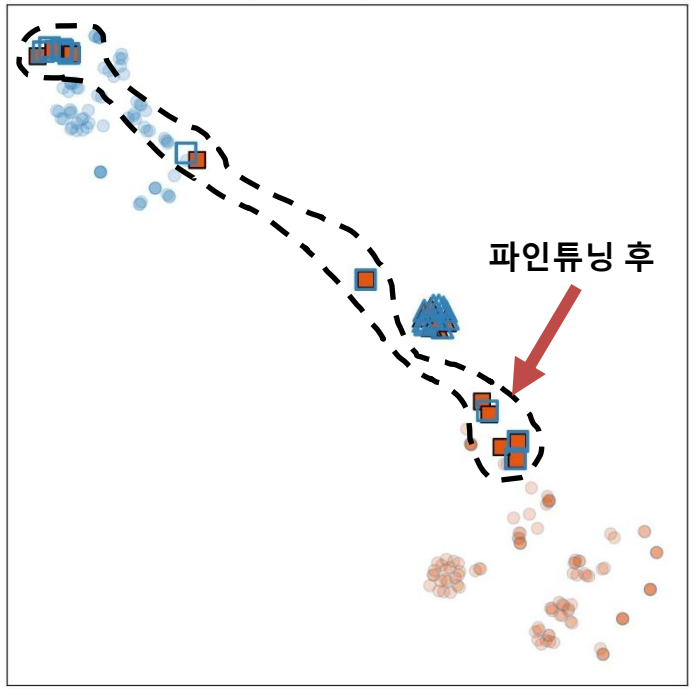


실험 및 결과

RQ2b. Trace 단위 데이터셋의 효과는 오픈소스 CVE에서도 확인되는가?

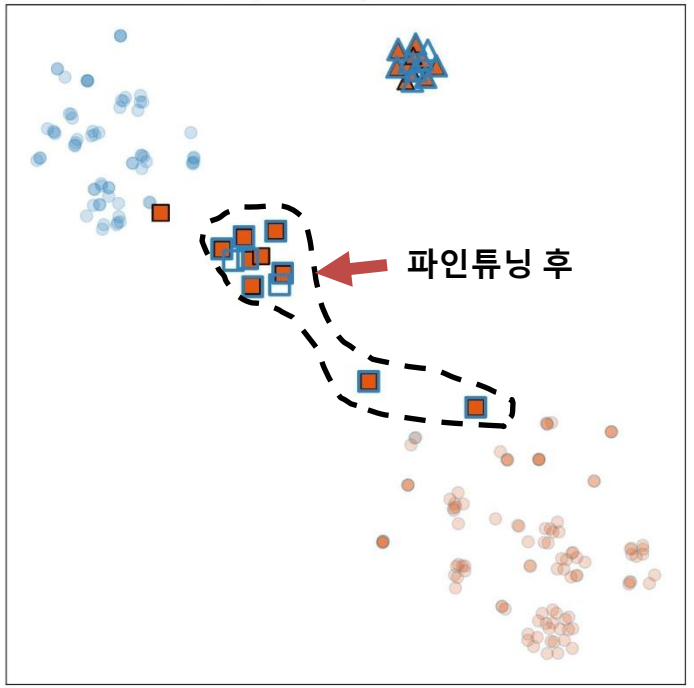
파인튜닝 전/후 CVE TracePair 데이터셋의 임베딩 벡터 t-SNE 분포 비교 결과,
파인튜닝 후 취약/정상 임베딩이 중첩되어, 10개 CVE 대상 CVE TracePair의 낮은 F1 성능과 일관됨

LineVul



파인튜닝 전 / 취약
파인튜닝 전/ 정상
파인튜닝 후 / 취약
파인튜닝 후/ 정상
Juliet Trace / 취약
Juliet Trace / 정상

PDBERT



파인튜닝 전 / 취약
파인튜닝 전/ 정상
파인튜닝 후 / 취약
파인튜닝 후/ 정상
Juliet Trace / 취약
Juliet Trace / 정상

RQ 2b 결과. 실제 CVE에서는 Trace 단위 개선 효과가 제한적임

논의 사항

10개 CVE 대상 TracePair 성능 제한 원인

1. SARD-Juliet Trace는 CVE Trace를 대표하지 못함

구분	평균 Trace 길이
SARD-Juliet Trace	5.92
10개 CVE 대상 Trace	12.50

2. 실제 CVE Trace는 자동 추출이 어려움

- State of the art 정적 분석 도구 Infer[5]의 한계로
  - 전역 변수, 구조체 필드, 함수 포인터 흐름은 정적 분석에서 누락 가능
  - 전체 Trace가 한 번에 연결되지 않아 구간별 추출 후 수동 복원이 필요
- CVE 별 Source/Sink 수동 정의로 Trace 품질이 CVE 별로 달라질 수 있음

논의사항

Tracer와 본 연구 비교

공통점: Trace 활용

차이점: 벡터 변환 방식

- **Tracer:** Trace 내 연산자,위험 API, 경계 조건 출현 빈도 기반 특징 벡터 변환
- **본 연구:** Trace 자체를 AI 모델에 넣어서 임베딩 벡터로 변환

결론 및 향후 연구

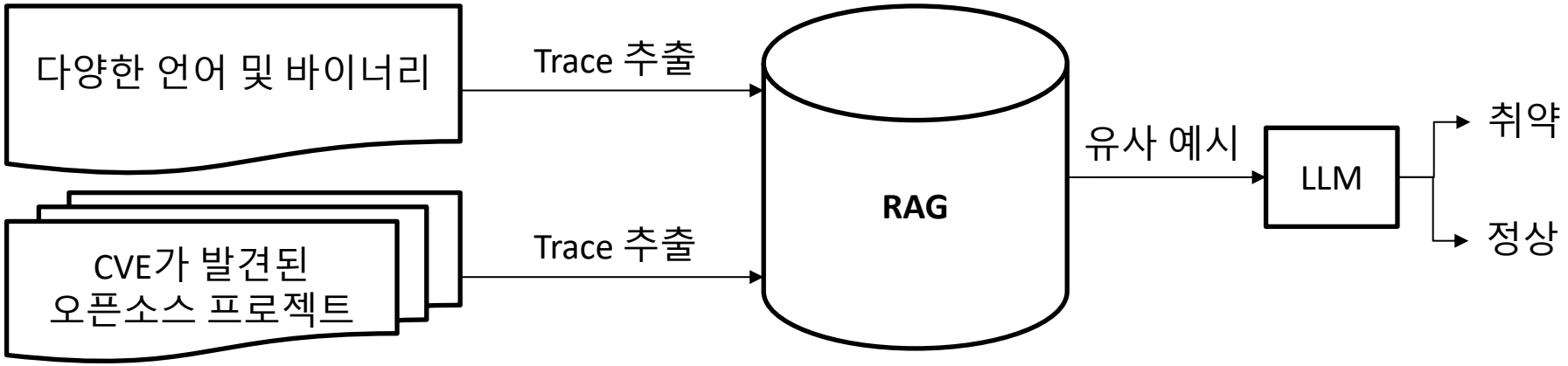
이 연구의 기여점

- 함수 단위 라벨링 한계를 보완하는 Trace 단위 취약점 탐지 데이터셋 제안
- Trace 기반 LLM-RAG 취약점 재현 탐지 절차 구현 및 실험
- SARD Juliet과 10개 CVE 대상 TracePair 구축
  (Extended RealVul FuncPair 데이터셋 구축)

결론 및 향후 연구

향후 연구

1. LLM RAG + Trace 형식의 취약점 탐지 프레임워크
2. 소스코드에서 Trace를 자동으로 추출하는 방법의 고도화
3. C/C++ 외 다양한 프로그래밍 언어의 소스 프로그램과 바이너리 대상 SW 취약점 탐지



< RAG 기반 LLM fewshot 취약점 탐지>

석사 과정 동안 발표한 논문

- 전세옥, 장현지, 정민수, 이서준, 오다영, 최광훈, 김석휘, 조성우, 김정미, "전기차 충전 인프라 오픈소스 소프트웨어에서의 취약한 코드 복제 탐지: VUDDY를 활용한 사례 연구," ACK 2024, 2024.

- Se-Ok Jeon, Seokhwi Kim and Kwanghoon Choi, "Analyzing the Limitations of CVE-Based Source-CodeVulnerability Slice Extraction," Poster, The 26th World Conference on Information Security Applications (WISA), Jeju, Republic of Korea, Aug. 20-22, 2025.

- 전세옥, 윤상권, 최광훈, 김석휘, 김건오, 파생 프로젝트 취약점 탐지를 위한 다중 시드 풀 퍼저, 2025년 한국정보보호학회 동계학술대회(CISC-W 2025), 11월 27-28일, 곤지암 리조트 EW 빌리지, 2025.

감사합니다.

Reference

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

[2] 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.

[3] 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.

[4] 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.

[5] Meta Platforms, Inc., "Infer Static Analyzer." [Online]. Available: https://fbinfer.com/.

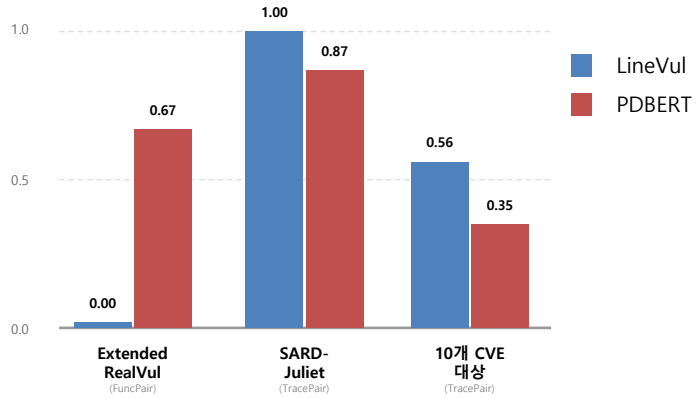
[6] Dyna Soumhane Ouchebara and Stéphane Dupont. 2025. Llama-Based Source Code Vulnerability Detection: Prompt Engineering vs Fine Tuning. In Computer Security – ESORICS 2025: 30th European Symposium on Research in Computer Security, Toulouse, France, September 22–24, 2025, Proceedings, Part I. Springer-Verlag,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

부록

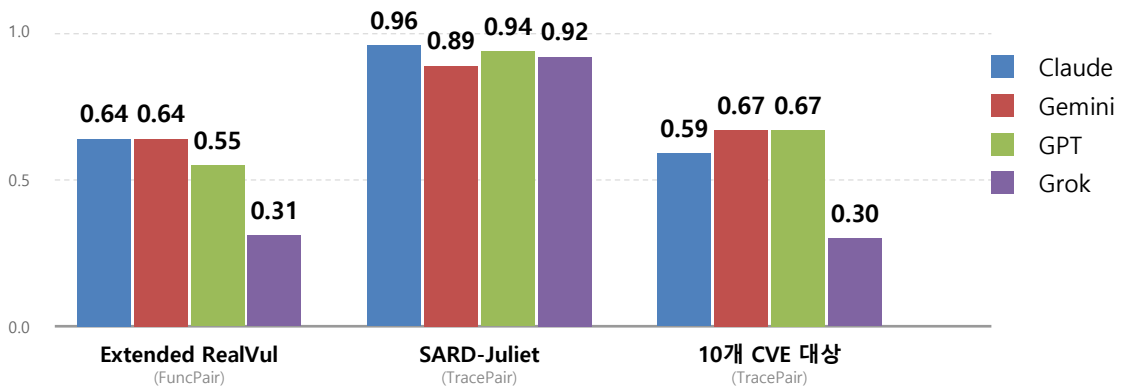
함수 단위 데이터셋의 한계와 Trace 단위 데이터셋의 효과

함수 단위 데이터셋
- Extended RealVul-FuncPair에서 취약/정상 구분 제한

Trace 단위 데이터셋
- SARD-Juliet TracePair에서는 구분 성능 향상
- 10개 CVE 대상 TracePair에서는 성능 개선 제한



<딥러닝 기반 취약/정상 구분 F1 성능>



<LLM 기반 취약/정상 구분 F1 성능>