

# Runtime Information -Augmented LLM for Precise MISRA-C Rule Check

카이스트 SWTV 연구실  
발표자 이아청




2025 여름 STAAR 워크샵

# MISRA C - C language development guideline

200여개의 Rule로 이루어진 C 프로그래밍 가이드라인

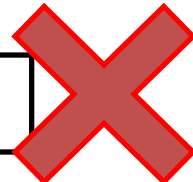
Rule 5.5 Identifiers shall be distinct from macro names

```
#define Sum(x, y) ( ( x ) + ( y ) )  
int16_t Sum;
```



Rule 9.3 Arrays shall not be partially initialized

```
int16_t arr[6] = {1, 2, 3};
```



# MISRA (Motor Industry Software Reliability Association)

자동차 산업 등 **임베디드** 소프트웨어 개발 과정에서  
undefined/undefined behavior를 피하기 위한 가이드라인

- C는 매우 강력한 언어로 임베디드 분야에서 널리 쓰이지만, 잘못 이해하고 사용되기 매우 쉬운 언어.
- ISO 26262 등의 안전 국제 표준을 지키기 위한 실질적인 방법론으로 사용

# MISRA (Motor Industry Software Reliability Association)

총 200개 Rule (2023 버전)

- 149개 decidable rule
- 51개 undecidable rule

# MISRA (Motor Industry Software Reliability Association)

## 총 200개 Rule (2023 버전)

- 149개 decidable rule
  - 일반적인 정적 분석 도구로 모두 탐지 가능.
  - e.g. The lowercase character “l” shall not be used in a literal suffix.
    - `int16_t a = 10l;`
- 51개 undecidable rule

# MISRA (Motor Industry Software Reliability Association)

## 총 200개 Rule (2023 버전)

- 149개 decidable rule

- 일반적인 정적 분석 도구로 모두 탐지 가능.
- e.g. The lowercase character “l” shall not be used in a literal suffix.
  - `int16_t a = 101;`

- 51개 undecidable rule

- e.g. All resources obtained dynamically shall be explicitly released
- 높은 오탐 비율!

# MISRA C 분석을 지원하며, 많이 사용되는 3개의 정적 분석 도구 비교 (Codesonar, Cppcheck, CodeQL)

## Codesonar - 상용 도구

- Interprocedure control/data flow, taint 분석 수행
- 사용자가 이해하기 쉽게 GUI 레벨로 분석 결과를 상세하게 보고

### Null Pointer Dereference<sup>2</sup> at osapi-module.c:107

Jump to warning location ↓

No properties have been set. | edit properties  
warning details...

Show Events | Options

OS\_SymbolLookup\_Static() /home/yeongbin/osal/srcs/equuleus-rc1/src/os/shared/src/osapi-module.c

```

91 int32 OS_SymbolLookup_Static(cpuaddr *SymbolAddress, const char *SymbolName, const char *Mod
92 {
93     int32 return_code = OS_ERR_NOT_IMPLEMENTED;
94     OS_static_symbol_record_t *StaticSym = OS_STATIC_SYMTABLE_SOURCE;
95
96     while (StaticSym != NULL)
97     {
98         if (StaticSym->Name == NULL)
99         {
100             /* end of list --
101              * Return "OS_ERROR" to indicate that an actual search was done
102              * with a not-found result, vs. not searching at all. */
103             return_code = OS_ERROR;
104             break;
105         }
106         if (strcmp(StaticSym->Name, SymbolName) == 0 &&
107             (ModuleName == NULL || strcmp(StaticSym->Module, ModuleName) == 0))

```

Event 5: StaticSym->Module, which evaluates to NULL, is passed to strcmp() as the first argument.  
• Dereferenced later, causing the null pointer dereference.

See related event 1. ▲ ▼ hide

#### Null Pointer Dereference<sup>2</sup>

The body of strcmp() dereferences StaticSym->Module, but it is NULL.

The issue can occur if the highlighted code executes.

See related event 5.

Show: All events | Only primary events

MISRA C 분석을 지원하며, 많이 사용되는 3개의 정적 분석 도구 비교  
(Codesonar, Cppcheck, CodeQL)

## CppCheck, CodeQL - 오픈 소스 도구

- 누구나 쉽고 빠르게 사용 가능
- 하지만 Codesonar에 비해서는 Shallow한 분석 기능, 간단하고 탐지하기 쉬운 경우만 대응 가능.



## 검증 대상 프로그램

	Description	LoC
<b>libmetal</b>	<b>libmetal</b> is a lightweight C library that provides a <b>hardware abstraction layer (HAL)</b> for <b>embedded systems</b> , primarily designed to support asymmetric multiprocessing (AMP) environments and <b>RTOS/Linux co-existence</b> .	4,214
<b>libmicrohttpd</b>	<b>GNU libmicrohttpd</b> is a small, fast, and <b>embeddable</b> C library that provides an <b>HTTP 1.1 server-side implementation</b> . It is designed to make it easy to add web server capabilities to C applications, especially in <b>embedded and lightweight environments</b> .	39,436
<b>littlefs</b>	<b>LittleFS</b> (short for <b>Little File System</b> ) is a lightweight, <b>fail-safe filesystem</b> designed specifically for <b>embedded systems</b> that use NOR or NAND flash memory.	5,472
<b>osal</b>	The <b>Operating System Abstraction Layer (OSAL)</b> is a key component developed by <b>NASA</b> to provide a portable interface between <b>flight software</b> and underlying operating systems or hardware.	14,986

# 기존 정적 분석 도구 - 여전히 높은 오탐 비율

## Codesonar 분석 결과 - (오탐 비율: 평균 58.8%)

	# of violation reports	# of true positives	# of false positives
libmetal	261	160 (61.7%)	100 (38.3%)
libmicrohttpd	384	121 (31.5%)	263 (68.5%)
littlefs	297	77 (25.9%)	220 (74.1%)
OSAL	141	88 (62.4%)	53 (37.6%)

## Cppcheck 분석 결과 - (오탐 비율: 평균 43.5%)

	# of violation reports	# of true positives	# of false positives
libmetal	54	43 (79.6%)	11 (20.4%)
libmicrohttpd	459	236 (51.4%)	223 (48.6%)
littlefs	171	92 (53.8%)	79 (46.2%)
OSAL	103	74 (71.8%)	29 (28.2%)

## CodeQL 분석 결과 - (오탐 비율: 평균 23.2%)

	# of violation reports	# of true positives	# of false positives
libmetal	136	95 (69.9%)	41 (30.1%)
libmicrohttpd	341	252 (73.9%)	89 (26.1%)
littlefs	84	78 (92.9%)	6 (7.1%)
OSAL	99	82 (82.8%)	17 (17.2%)

## LLM은? (Direct prompting approach)

### Direct prompting approach

- 함수 코드와 Rule Description을 제공하고,  
이 함수가 해당 가이드라인을 위반하였는가?를 질문

	# of violation reports	% of true positives	% of false positives
libmetal	275	38.5%	61.5%

Codesonar 분석 결과 - (오탐 비율: 평균 58.8%)

	# of violation reports	# of true positives	# of false positives
libmetal	261	160 (61.7%)	100 (38.3%)

Cppcheck 분석 결과 - (오탐 비율: 평균 43.5%)

	# of violation reports	# of true positives	# of false positives
libmetal	54	43 (79.6%)	11 (20.4%)

## Direct prompting 오류 예시 1. 잘못된 가정을 바탕으로 결정

```
1 static void metal_linux_dev_dma_unmap(  
2     struct metal_bus *bus, ...) {  
3     ...  
4     ldev->ldrv->dev_dma_unmap(lbus, ldev);  
5 }  
6
```

**ldev** 또는 **ldev->ldrv**가  
NULL값이라면  
undefined behavior 발생  
가능하다고 보고함.

하지만 실제 전체 시스템을 분석한 결과,  
NULL일 가능성이 없다. -> 오탐.

## Direct prompting 오류 예시 2. 없는 문제를 만들어낸다.

### 검증 대상 함수 코드

```
1 void OS_DeleteAllObjects(void) {  
2     uint32_t ObjectCount;  
3     ...  
4     OS_ForEachObject(0, &ObjectCount);  
5     ...  
}
```

`OS_ForEachObject`의 정의를  
같이 제공하지 않으면...

### LLM이 만들어낸 코드

```
1 uint32_t *g_ptr;  
2  
3 void OS_ForEachObject(osal_id_t id, int32_t * local_ptr) {  
4     ...  
5     g_ptr = local_ptr;  
6 }
```

`OS_ForEachObject`의 코드를 만들어내고,  
해당 코드가 질문한 Rule의 violation을 발생할 가능성이 있다고  
보고

## 개선 아이디어

1. 런타임 정보를 활용하여 LLM의 리포트를 다시 한번 검증

2. 새로 발견된 해당 프로젝트의 런타임 정보를 추적하여 차후 버전에 활용

## 런타임 정보 사용 예시

```
void OS_DeleteAllObjects(void) {  
    ...  
    while(true)  
    {  
        ObjectCount = 0;  
        ++TryCount;  
        OS_ForEachObject(0, OS_CleanupObject, &ObjectCount);  
        if (ObjectCount == 0 || TryCount > 4) { break; }  
        OS_TaskDelay(5);  
    }  
    while (ObjectCount > 0 && TryCount < 5);  
}
```

해당 조건을 만족하지 않으면  
루프를 벗어나지 못함.

해당 조건은 항상 False

## 런타임 정보 사용 예시

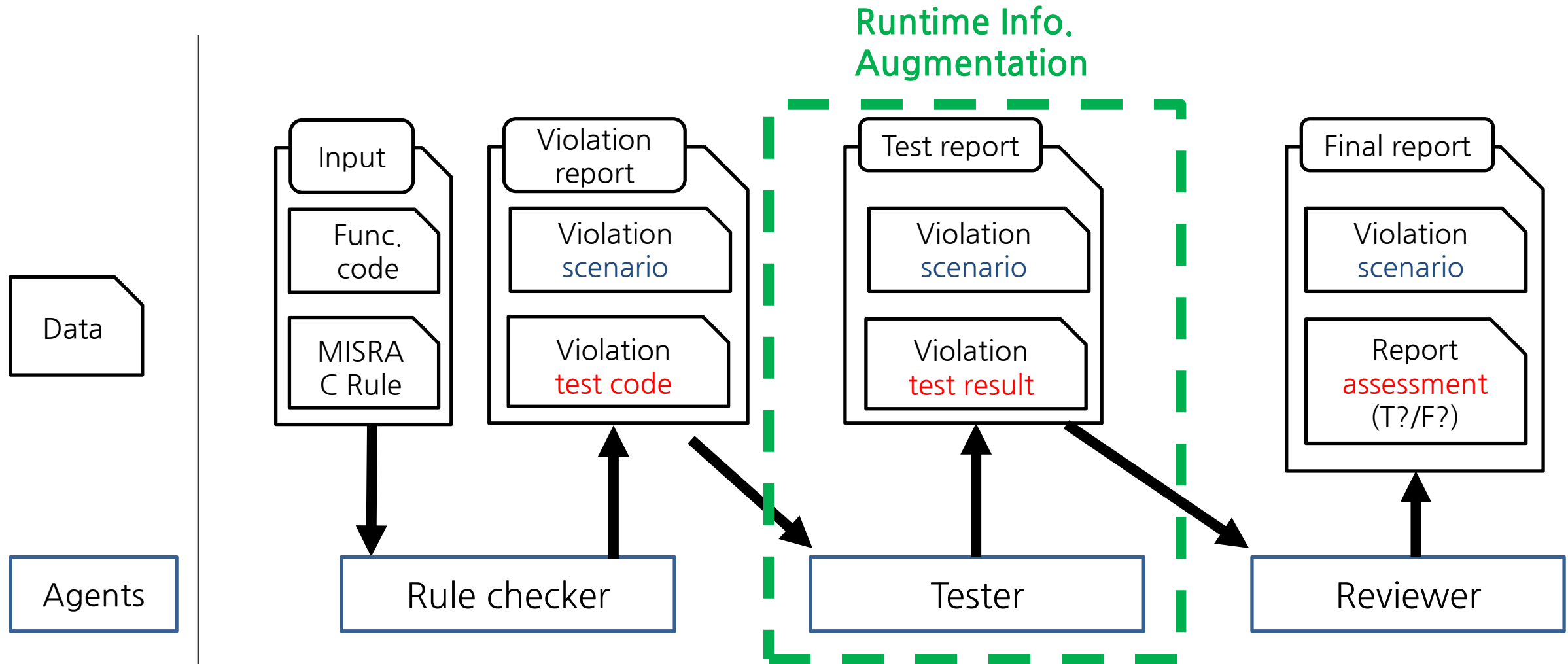
LLM에게 Violation 시나리오와 함께,  
해당 시나리오를 확인할 수 있는 로그 출력 함수를 삽입 해 달라고 요청

```
...
printf("Evaluating the controlling expression for the second while loop:\n");
printf("  Condition: (ObjectCount > 0 && TryCount < 5)\n");
printf("  (ObjectCount > 0) -> (%u > 0) is %d\n", ObjectCount, (ObjectCount > 0));
printf("  (TryCount < 5)      -> (%u < 5) is %d\n", TryCount, (TryCount < 5));
printf("Attempting to enter the second loop...\n");

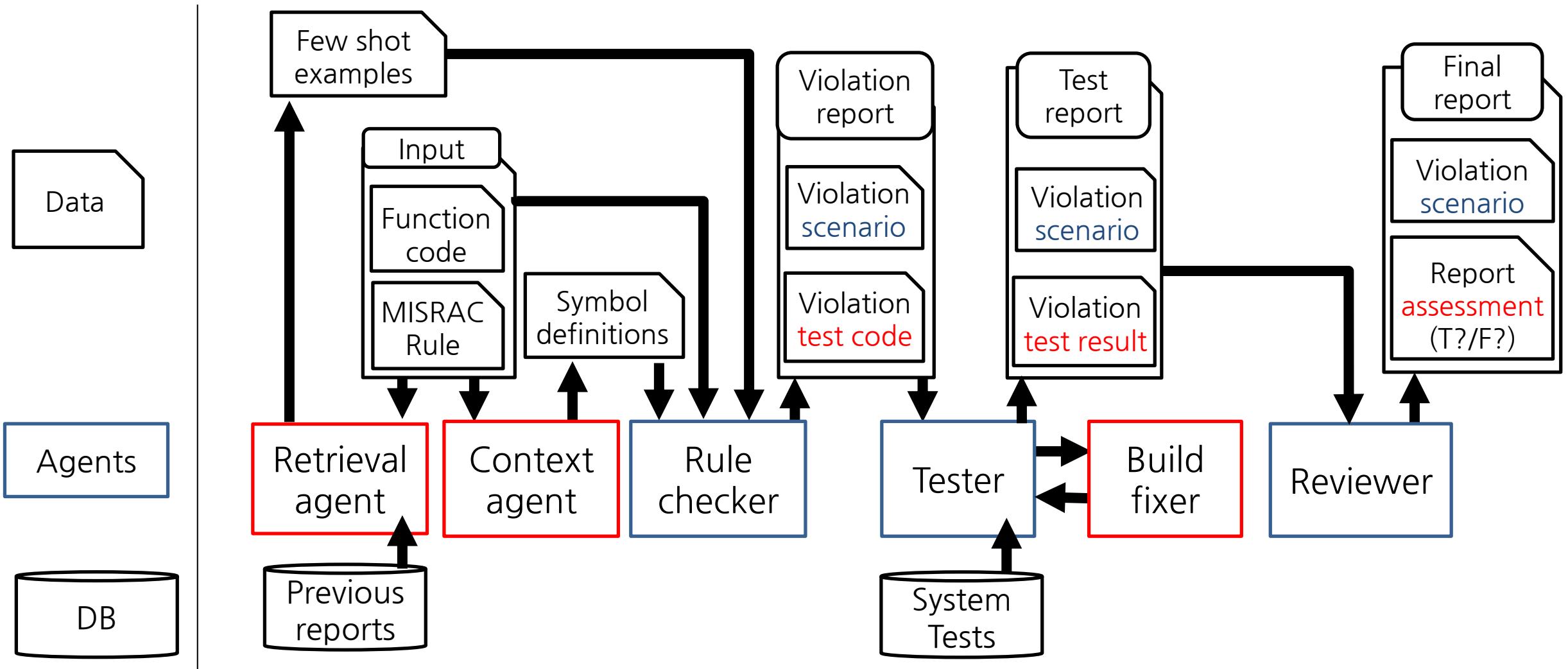
while (ObjectCount > 0 && TryCount < 5) {
    printf(
        "  ERROR: This message should NOT be printed, as the loop condition is "
        "always false.\n");
}
printf("\n--- OS_DeleteAllObjects finished ---\n");
```



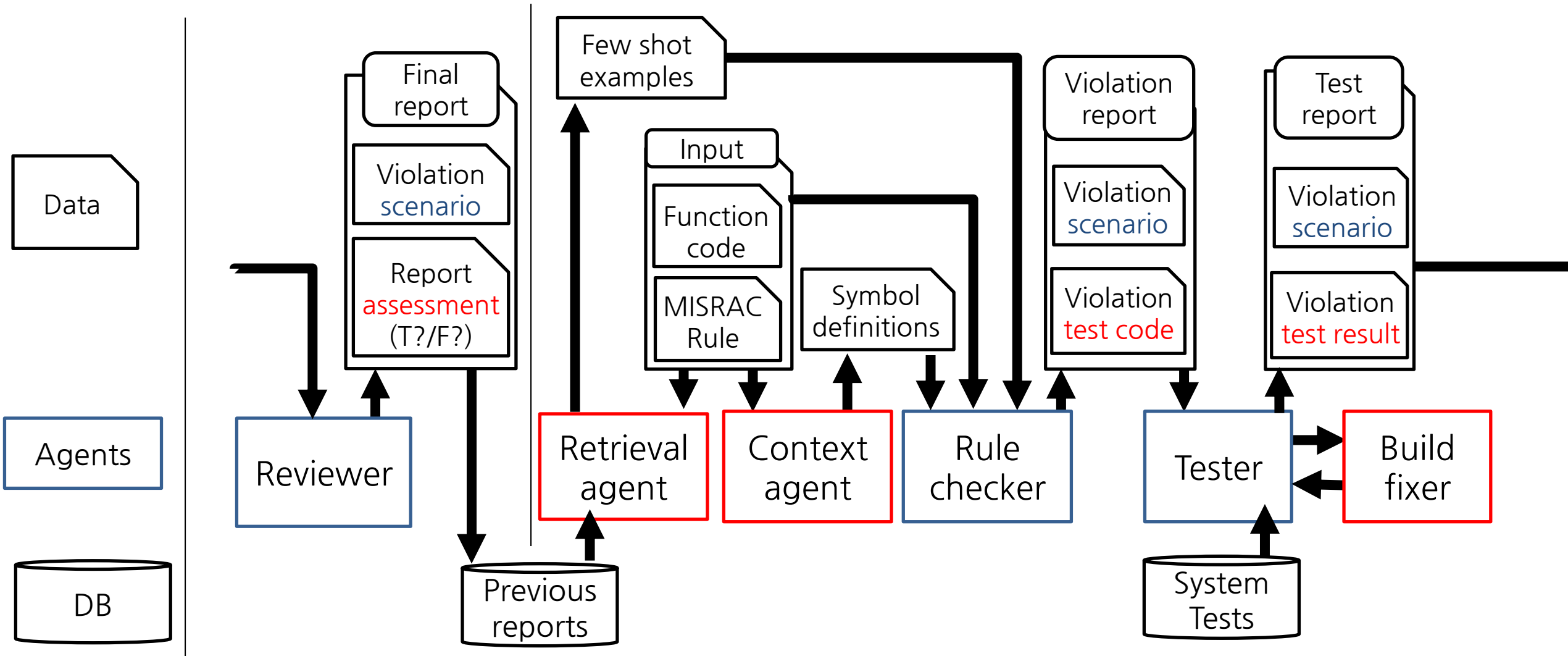
# Lasik - LLM-based MISRA C rule checker



## Lasik - 더 많은 에이전트...



# Lasik - 더 많은 에이전트...



## 수행 결과

### Direct prompting approach

	# of violation reports	% of true positives	% of false positives
libmetal	275	38.5%	61.5%

### Lasik

	# of violation reports	% of true positives	% of false positives
libmetal	190	51.5%	48.5%

### Codesonar 분석 결과 - (오탐 비율: 평균 58.8%)

	# of violation reports	% of true positives	% of false positives
libmetal	261	61.7%	38.3%

# 포스터 소개 1

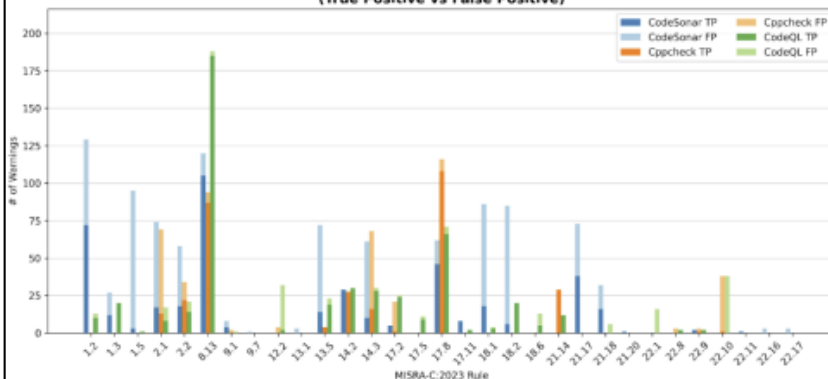
## MISRA C 규칙 위반 오탐 사례의 수집 및 LLM을 활용한 개선 방안

### 강영빈, 장세창

4개의 Safety-critical 오픈소스 프로젝트에 대해 3개의 정적 분석 도구 (CodeSonar, Cppcheck, CodeQL)를 적용하여 Undecidable MISRA C 규칙 위반으로 탐지되는 사례 수집  
프로젝트별 규칙마다 25% 이상, 최소 20개 이상의 경고 선택

→ 총 4개 프로젝트에 대해 2457개의 규칙 위반 알람 검토

MISRA C Rules Violation Comparison  
(True Positive vs False Positive)



정적분석 도구별 정탐/미탐율

	# of true positives	# of false positives	False positive rate (%)
CodeSonar	446	636	58.78%
Cppcheck	445	342	43.46%
CodeQL	462	136	22.74%

### 3. 정적 분석 도구의 MISRA C 규칙 위반 탐지 경향성

같은 분류의 MISRA C 규칙 위반 사례는 비슷한 패턴으로 발생함

#### 1. 시스템 라이브러리에 의존적인 코드에 관한 오탐

- MISRA C Rule 1.2: Language extensions should not be used.
- MISRA C Rule 2.1: A project shall not contain unreachable code.
- MISRA C Rule 2.2: A project shall not contain dead code.
- MISRA C Rule 14.3: Controlling expression should not be invariant.

#### 2. side effect와 object 수정에 관한 오탐

- MISRA C Rule 13.5: The right-hand operand of a logical && or || operator shall not contain persistent side effects.
- MISRA C Rule 8.13: A pointer should point to a const-qualified type whenever possible
- Rule 17.8: A function parameter should not be modified

#### 3. 복잡한 런타임 값 수정으로 인한 값 추적에 관한 오탐

- MISRA C Rule 1.3 There shall be no occurrence of undefined or critical unspecified behavior.

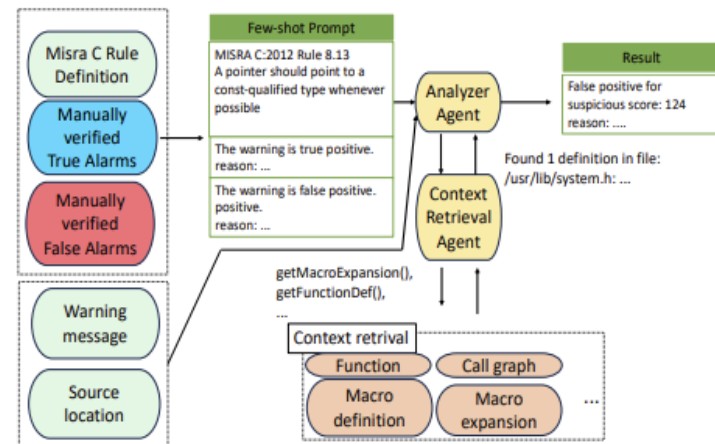
#### MISRA C 규칙 위반 미탐

- 실험에서 사용한 3개의 정적 분석기는 모두 MISRA C 2023을 지원함이 명시되어 있으나, 특정 규칙 위반은 탐지하지 못함

정적 분석 도구	미탐 MISRA C 규칙 위반
CodeSonar	12.2, 17.5, 18.6
Cppcheck	1.2, 1.3, 1.5, 9.1, 17.5, 17.11, 18.6, 21.18, 21.20, 22.11
CodeQL	9.1, 21.13, 21.20, 22.7, 22.11

### 4. 오탐 개선 방안 제시 - LLM을 활용한 오탐 의심 순위화

- 정적 분석 결과는 수 백~수 천 개의 알람이 발생
- 모든 알람을 사람이 분석하기에는 많은 시간과 노동력이 필요함
- 같은 분류의 MISRA C 규칙 위반 사례는 비슷한 패턴으로 발생함
- LLM Agent 구성으로 오탐 가능성에 의심도 점수 부여
- 오탐 의심도가 높은 알람을 사람이 우선적으로 검사
- 다중 정적 분석기 사용을 통해 증가하는 알람에 대한 개발자의 검토 부담 감소



# 포스터 소개 2

## 딥러닝 기반 결함 위치 추정 기법의 실험 설정에 따른 성능 평가 양희찬

### 1. Introduction

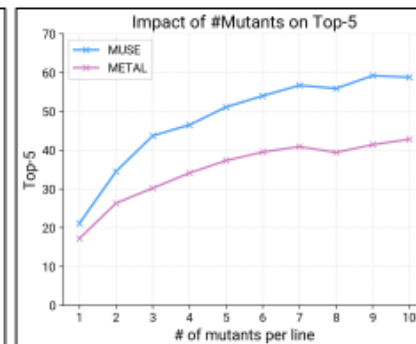
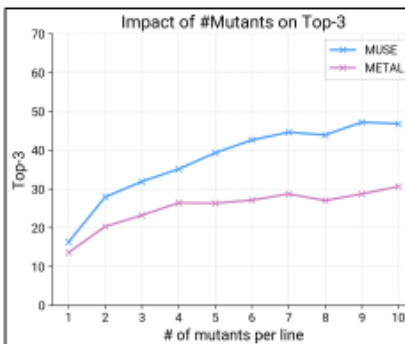
- **문제:**  
결함 위치 추정은 디버깅에서 시간이 많이 든다. 기존 자동 결함 위치 추정 기법인 SBFL과 MBFL은 정확도에 한계가 있다.
- **동기:**  
딥러닝 기반 접근인 DLFL이 주목받고 있지만, 실험 설정 (예: 라인당 변이 수, 특징 조합)에 따른 성능 차이는 충분히 연구되지 않았다.
- **목표:**  
본 연구는 Defects4J Lang, Mockito, Math 프로젝트들의 총 202개 결함을 대상으로 DLFL의 다양한 설정별 성능을 평가하고, 기존 기법들과 비교 분석한다.

### 2. Background

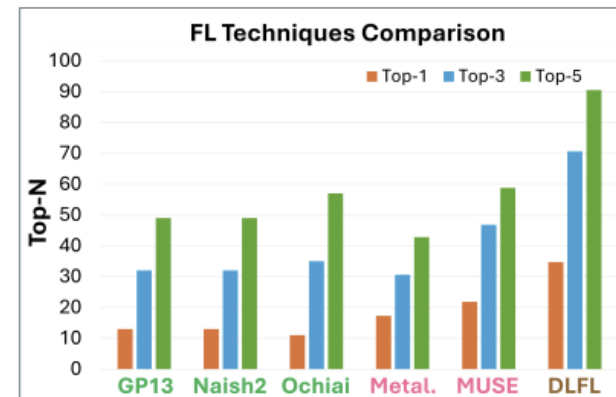
- **DLFL (Deep Learning-Based FL):** 코드 특징(feature)을 학습한 딥러닝 모델을 활용한 결함 예측 기법
- **DLFL 데이터셋 특징 구성:**
  - **SBFL (Spectrum-Based FL):** 테스트 커버리지 및 실행 결과에 기반한 통계적 기법
  - **MBFL (Mutation-Based FL):** 변이 실행 결과를 기반으로 결함 위치를 추정하는 접근 방식

### 5. 주요 결과

- **RQ1 결과:** MBFL 정확도는 라인당 변이 수 증가에 따라 향상된다
  - MUSE 기준 1 → 10 mutants/line일 때 Top-5 정확도는 21.1에서 42.8로, +102%p 상승했다 ... (높을수록 우수)
  - MUSE 기준, 1 → 10 mutants/line일 결함 구문의 평균 순위는:
    - MFR은 162.4 → 85.3 (↓, -47.5%p)
    - MAR은 195.5 → 130.4 (↓, -33.3%p) ... (낮을수록 우수)



- **RQ2 결과:** DLFL이 모든 지표에서 우수한 성능을 달성한다
  - Ochiai(SBFL) 대비 Top-5 정확도: 58.8 → 90.5 (+58.8%p)
  - MUSE(MBFL) 대비 Top-5 정확도: 57.0 → 90.5 (+58.9%p)



- **RQ3-1 결과:** 1 → 10 mutants/line일 때 모든 지표에서 성능 향상
  - 특히 MFR은 37.6 → 31.7 (↓, -15.6%p) 성능 향상이 보인다

변이 개수	Top-1	Top-3	Top-5	Top-10	MFR	MAR
1	33.3	63.8	82.7	107.8	37.6	59.2
10	34.7	70.6	90.5	113.3	31.8	53.9