AutoFiC: 취약점 탐지부터 PR 생성까지 자동화된 보안 패치 파이프라인

장인영 [O1], 오정민 [2], 김민채 [3], 김은솔 [4]
덕성여자대학교 [1], 가천대학교 [2], 국민대학교 [3], 명지대학교 [4]
726iy@duksung.ac.kr, ojmes5790@gmail.com, brianna0324@kookmin.ac.kr, kesol1530@gmail.com

AutoFiC: Automated Security Patch Pipeline from Vulnerability Detection to Pull Request Generation

Inyeong Jang[O1], JeongMin Oh[2], Minchae Kim[3], Eunsol Kim[4]
[1]Duksung Women's University, [2]Gachon University, [3]Kookmin University,
[4]Myongji University

요 약

정적 분석 도구(SAST)는 보안 취약점 탐지에 널리 활용되고 있으나, 탐지 이후의 패치 생성 및 적용 과정은 여전히 개발자의 수작업에 크게 의존하고 있다. 본 연구는 이러한 문제를 해결하기 위해 취약점 탐지부터 패치 생성 및 적용, Pull Request 생성에 이르는 전 과정을 자동화한 End-to-End 보안 패치 파이프라인 AutoFiC을 제안한다. AutoFiC은 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 결합하여, AST 기반의 복잡한 구문 분석을 직접 활용하지 않고도 취약점의 위치와 유형 정보를 LLM에 효과적으로 전달한다. 이를 통해 수정 범위를 취약 구간으로 명확히 한정함으로써, 패치 부작용을 최소화하는 경량 자동 패치 구조를 구현하였다. 91개의 실제 GitHub 오픈소스 저장소를 대상으로 한 실험 결과, AutoFiC은 94.5%의 파이프라인 성공률과 90.0%의 취약점 해결률을 기록하였다.

1. 서 론

최근 소프트웨어 시스템의 규모와 복잡도가 증가함에 따라, 소스 코드 수준에서의 보안 취약점 관리가 점점 더 중요한 과제로 부각되고 있다. 이러한 취약점을 사전에 식별하기 위해 정적 분석 도구(SAST)가 다양한 개발 환경에서 활용되고 있으나, 탐지 이후의 수정 과정은 여전히 개발자의 수작업에 크게 의존하고 있다. 이로 인해 취약점 탐지와 실제 수정 사이에는 상당한 시간적·운영적 간극이 존재하며, 이를 완화하기 위한 자동화된 보안 패치 기법에 대한 요구가 지속적으로 증가하고 있다[1].

이러한 배경에서 자동 프로그램 수리(Automated Program Repair, APR) 및 대규모 언어 모델(LLM)을 활용한 자동 패치 접근이 제안되었다. 기존 연구들은 LLM을 활용하여 취약 코드를 자동으로 수정할 수 있는 가능성을 보여주었으나, 수정 범위가 필요 이상으로 확대되거나 코드의 구조적 맥락을 충분히 반영하지 못하는 한계 또한 함께 논의되어 왔다. 이를 보완하기 위한 다양한 접근이 제안되었음에도 불구하고, 실제 개발 환경에서 활용 가능한 경량 자동 패치 파이프라인에 대해서는 여전히 명확한 설계가 정립되지 않은 실정이다[2].

본 연구는 이러한 공백을 해소하기 위해, SAST 결과로부터 생성한 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 결합한 LLM 기반 자동 패치 파이프라인을 제안한다. 제안 기법은 AST 기반 분석을 직접 활용하지 않으면서도 취약점의 위치와 유형을 구조적으로 전달함으로써 LLM의 이해도를 향상시키고, 수정 범위를 취약 구간 중심으로 제한하여 패치 부작용을 최소화한다. 또한 취약점 탐지부터 패치 생성 및 적용, Pull Request 생성까지 이어지는 End-to-End 자동화를 구현함으로써, 실제 개발 워크플로우에 적용 가능한 경량 자동 보안 패치 파이프라인의 설계 방향을 제시한다.

2. 관련 연구

자동 프로그램 수리(Automated Program Repair, APR) 연구는 규칙 기반 패치 생성 기법을 출발점으로 하여, 검색 기반 및 학습 기반 접근법으로 점차 확장되어 왔다. 기존 연구들은 코드 결함에 대해 자동으로 수정 코드를 생성할 수 있음을 실험적으로 입증하며, 다양한 결함 유형에 대한 적용 가능성을 제시하였다[3]. 그러나 대다수 연구는 제한된 코드 범위나 단일 결함을 대상으로 수행되었으며, 실험 환경과 평가 지표가 연구 목적에 따라 상이하게 설정되어 실제 소프트웨어 개발 환경으로의 일반화에는 한계가 존재한다.

최근에는 이러한 한계를 극복하기 위해 대규모 언어 모델(LLM)을 APR에 적용하려는 시도가 활발히

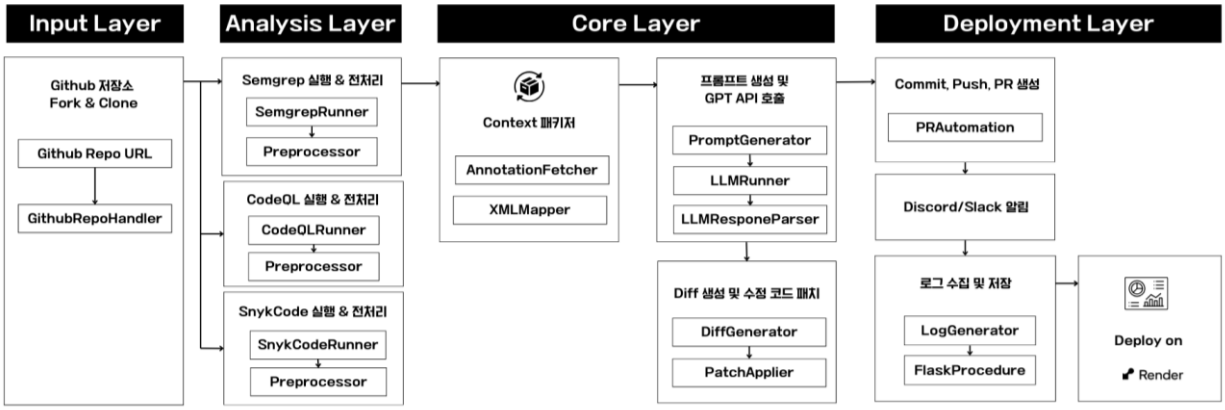

그림 1. 시스템 아키텍처

이루어지고 있다. LLM 기반 접근법은 코드와 자연어의 문맥을 동시에 학습한 모델을 활용함으로써, 기존 규칙 기반 기법 대비 보다 유연하고 다양한 수정 코드를 생성할 수 있음을 보였다[4]. 이를 통해 LLM이 자동 패치 생성에 활용될 수 있음이 실증적으로 확인되었으나, 프롬프트에 제공되는 코드의 단위, 수정 범위의 설정 방식, 그리고 생성된 패치에 대한 검증 전략은 연구마다 상이하게 설계되어 표준화된 방법론이 부재한 실정이다. 이러한 설계 차이는 패치의 안정성과 재현성 측면에서 추가적인 고려가 필요함을 시사한다.

한편, 자동 패치 생성의 전제 조건인 결함 식별 단계에서는 정적 분석 도구(SAST)가 취약점 위치와 유형을 제공하는 핵심적인 정보원으로 활용되어 왔다. 정적 분석 결과를 기반으로 경고를 분류하거나 우선순위를 결정하고, 수정 후보 위치를 식별하기 위한 다양한 기법이 제안되었으며[5], 일부 APR 연구에서는 이러한 분석 결과를 패치 생성 단계의 입력으로 직접 활용하기도 한다. 그러나 정적 분석 도구는 대량의 경고를 생성하는 특성으로 인해, 실제 수정으로 이어질 수 있는 정보를 효과적으로 선별·활용하는 데 한계를 보인다.

자동 프로그램 수리(APR) 전반을 대상으로 한 체계적 문헌 분석 연구에 따르면, 패치 생성 기법 자체에 대한 연구는 지속적으로 발전해온 반면, 생성된 패치를 실제 코드베이스에 적용하고 개발 워크플로우와 연계하는 과정은 연구마다 서로 다른 가정과 설계를 채택하고 있다[6]. 즉, 결함 탐지, 패치 생성 및 적용, 개발 프로세스 연계를 하나의 일관된 흐름으로 통합하려는 접근은 아직 정형화되지 않았다.

종합하면, 기존 연구들은 자동 패치 생성 알고리즘의 고도화, LLM의 적용 가능성, 정적 분석 결과의 활용 등 개별 요소 기술을 중심으로 발전해왔다. 그러나 취약점 탐지부터 수정 적용, 검증, 그리고 개발 워크플로우 연계까지를 포괄하는 체계적인 자동화 구조에 대해서는 연구마다 상이한 설계와 가정을 채택하고 있다. 본 연구는 이러한 선행 연구들을 바탕으로, 자동 보안 패치 파이프라인 설계에 대한 하나의 통합적 관점을 제시한다.

3. 제안 기법

3.1 시스템 아키텍처

그림 1은 제안하는 AutoFiC 시스템의 전체 아키텍처를 나타낸다. AutoFiC은 GitHub 저장소 Fork 및 Clone, 다중 SAST 기반 정적 분석, LLM 프롬프트 구성 및 패치 코드 생성, Pull Request 자동화, 대시보드 시각화로 이어지는 계층형 구조를 따른다. 전체 시스템은 Python 기반 CLI(Command Line Interface) 도구로 구현되어, 로컬 환경에서 일관되게 활용할 수 있다.

입력 계층(Input Layer)에서는 사용자로부터 분석 대상 GitHub 저장소의 URL과 SAST 도구 및 LLM 모델에 대한 설정을 CLI 인자(Argument)로 전달받는다. 시스템은 GitHub API를 활용하여 대상 저장소를 Fork하고 로컬 환경으로 Clone하여 분석을 위한 실행 환경을 구성한다.

분석 계층(Analysis Layer)에서는 Semgrep, CodeQL, SnykCode와 같은 정적 분석 도구 중 하나를 선택적으로 실행하여 소스 코드 내 취약점을 탐지한다. 각 도구의 분석 결과는 전처리 모듈을 거쳐 공통 스키마인 BaseSnippet 형태로 정규화된다. BaseSnippet은 파일 경로, 라인 범위, CWE 정보 등을 포함하며, 도구별 출력 형식의 차이를 추상화하여 후속 단계에 일관된 입력 데이터를 제공한다.

핵심 처리 계층(Core Layer)에서는 정규화된 스니펫을 기반으로 구조화된 XML 컨텍스트와 코드 주석(Annotation)을 생성한다. 생성된 컨텍스트는 프롬프트 엔지니어링 모듈로 전달되어 최적화된 프롬프트를 구성하며, 이를 LLM에 입력하여 취약점

패치 코드를 생성한다. LLM의 응답은 파싱 과정을 거쳐 Diff 형식으로 변환된 후, 자동 패치 모듈을 통해 코드베이스에 반영된다. 패치 적용 실패 시 재시도(Retry) 및 Fallback 메커니즘이 동작하여 전체 파이프라인의 가용성과 연속성을 보장한다.

배포 계층(Deployment Layer)에서는 패치가 적용된 Branch에 대해 Commit, 원격 저장소 Push, Pull Request 생성을 포함한 일련의 과정을 자동화한다. 최종 결과는 Flask 기반의 웹 대시보드를 통해 제공되며, Slack 및 Discord 연동을 통해 실시간 알림 서비스를 제공한다.

3.2 전체 워크플로우

AutoFiC은 정적 분석 도구(SAST)와 대규모 언어 모델(LLM)을 결합하여, 취약점 탐지부터 패치 생성 및 적용, Pull Request 생성에 이르는 전 과정을 자동화한 End-to-End 파이프라인을 제공한다. 전체 워크플로우는 (1) 저장소 초기화 및 환경 구성, (2) SAST 기반 취약점 탐지 및 정규화, (3) XML 및 Annotation 기반 컨텍스트 구성, (4) LLM 기반 패치 코드 생성, (5) Diff 생성 및 패치 적용, (6) Pull Request 생성 및 CI 연계의 6단계로 구성된다.

3.2.1 GitHub 저장소 Fork 및 Clone

사용자가 분석 대상 GitHub 저장소의 URL을 입력하면, AutoFiC은 GitHub API를 통해 해당 저장소를 Fork한 후 로컬 환경으로 Clone하여 분석 환경을 구축한다. 이 과정은 Branch 분기 및 Commit 이력 보존을 포함하는 표준 Git 워크플로우를 준수하도록 설계되었다. 이를 통해 원본 저장소의 무결성을 해치지 않으면서, 실제 개발 환경과 동일한 조건에서 패치를 검증할 수 있는 격리된 환경을 제공한다.

3.2.2 SAST 기반 취약점 분석

분석 단계에서는 Semgrep, CodeQL, SnykCode와 같은 SAST 도구 중 하나를 선택적으로 실행하여 취약점을 탐지한다. 각 도구는 상이한 탐지 규칙과 출력 형식을 가지므로, 분석 결과를 공통 스키마인 BaseSnippet 형태로 정규화한다. BaseSnippet은 파일 경로, 라인 범위, CWE 정보를 포함하며, 도구별 출력 형식의 차이를 추상화하여 후속 단계에서 일관된 입력 데이터를 보장한다.

3.2.3 컨텍스트 기반 프롬프트 구성 전략

SAST 결과만으로는 LLM이 코드 내부의 제어 흐름, 데이터 흐름, 취약 구간의 경계 등을 온전히 파악하기 어렵다는 한계가 있다. AutoFiC은 이를 보완하기 위해, 취약 코드 주변 컨텍스트를 구조화된 메타데이터, 위치 정보, 최소 단위 코드 스니펫으로 재구성한 뒤 프롬프트에 반영하는 전략을 사용한다. 이때 컨텍스트는 파일 경로와 라인 범위, CWE 유형, 경고 메시지, 관련 함수/블록 코드와 같은 정보로 구성되며, LLM이 어디를, 왜, 어떻게 수정해야 하는지를 명시적으로 이해하도록 돕는다.

컨텍스트 구성 단계에서는 SAST가 보고한 취약점 스니펫을 공통 스키마(BaseSnippet)로 정규화하고, 이를 기반으로 요약 메타데이터와 코드 조각을 생성한다. 메타데이터는 CUSTOM_CONTEXT.xml과 같은 구조화된 형식에 저장되며, 프롬프트에서는 대상 취약점과 직접 관련된 조각만 발췌하여 사용함으로써 입력 크기를 관리한다. 코드 측면에서는 취약 줄만 제공하는 대신, 동일 함수나 인접 블록을 함께 포함해 제어 흐름이 끊어지지 않도록 스니펫을 구성함으로써, LLM이 수정 시 주변 문맥을 고려할 수 있도록 한다. 주석 기반 마커(Annotation)는 이러한 컨텍스트를 보조하는 수단으로 활용되며, 프롬프트 설계의 핵심은 필요한 정보는 충분히 제공하되, 수정 범위를 안정적으로 제어하는 것에 맞추어져 있다.

3.2.4 LLM 기반 패치 생성

구축된 XML 컨텍스트와 주석이 포함된 코드 스니펫은 프롬프트 엔지니어링을 거쳐 LLM에 입력된다. 프롬프트는 취약점에 대한 상세 설명과 수정 제약 조건을 포함하도록 설계되었으며, LLM은 이를 바탕으로 수정된 코드를 생성한다. 생성된 결과물은 파싱 과정을 통해 자동 적용이 가능한 Unified Diff 형식으로 변환된다.

3.2.5 Diff 생성 및 패치 적용

LLM이 생성한 수정 코드와 원본 소스 코드 간의 차이는 Diff 파일로 생성되며, Git의 Patch 시스템을 통해 코드베이스에 반영된다. 이때 패치 적용 실패 등 예외 상황 발생 시, 사전에 정의된 Fallback 절차를 수행하여 파이프라인의 중단을 방지한다. 이러한 예외 처리 메커니즘은 자동화된 패치 프로세스의 안전성과 연속성을 보장한다.

3.2.6 Pull Request 생성 및 CI 연계

패치가 적용된 Branch는 자동으로 원격 저장소에 Push 되며, 이를 기반으로 Pull Request가 생성된다. AutoFiC은 Pull Request 생성 및 상태 변경 이벤트를 감지하는 GitHub Actions 워크플로우를 자동 생성하며, Discord 또는 Slack과 연동하여 실시간 알림을 제공한다. 또한, CI 환경에 필요한 민감 정보(Webhook URL 등)는 공개키 기반으로 암호화하여 GitHub Secrets에 안전하게 등록된다. 최종적으로 생성된 Pull Request에는 분석 결과 요약과 CI 상태 모니터링 정보가 포함되어 검토자의 효율적인 의사결정을 지원한다.

3.3 프롬프트 엔지니어링

AutoFiC의 프롬프트 엔지니어링은 취약 코드만을 LLM에 그대로 전달하는 방식에 그치지 않고, SAST 결과로부터 추출한 구조화 메타데이터(XML), 코드 스니펫, Annotation 기반 위치 힌트를 단계적으로 결합하여 LLM이 취약점의 의미적·구조적 맥락을 명확히 이해하도록 설계되었다. 모델은 취약점 설명과 정확한 위치 정보, 관련 코드 컨텍스트를 함께 입력 받아 수정 대상을 명확히 식별하고, 취약 구간 중심의 최소 수정 패치를 생성하도록 유도된다.

3.3.1 XML 기반 구조화 컨텍스트

AutoFiC은 SAST 도구별 분석 결과를 정규화한 BaseSnippet을 기반으로, 입력 이전 단계에서 CUSTOM_CONTEXT.xml 형태의 XML 기반 구조화 컨텍스트를 생성한다. 각 취약점 항목에는 파일 경로 및 라인 범위, 분석 도구 및 규칙 식별 정보, CWE ID, 원본 메시지와 요약 설명, 심각도 및 카테고리 정보가 포함된다.

```
<CUSTOM_CONTEXT version="1.1">
  <VULNERABILITY id="scripts/main.py:411-411">
    <FILE path="scripts/main.py"/>
    <RANGE start="411" end="411"/>
    <SEVERITY overall="ERROR" bit="ERROR"/>
    <MESSAGE>
      <ITEM>Unsafe subprocess.run() with shell=True</ITEM>
    </MESSAGE>
    <SNIPPET>
      subprocess.run(explorer_command, shell=True)
    </SNIPPET>
    <BIT>
      <TRIGGER>Command injection risk</TRIGGER>
      <STEPS>
        <STEP>Review line 411 in scripts/main.py</STEP>
      </STEPS>
    </BIT>
    <CLASSES>
      <CLASS>Command Injection</CLASS>
    </CLASSES>
    <WEAKNESSES>
      <CWE id="CWE-78"/>
    </WEAKNESSES>
  </VULNERABILITY>
</CUSTOM_CONTEXT>
```

그림 2. CUSTOM_CONTEXT.xml 구조 예시

프롬프트 구성 시 전체 XML을 그대로 LLM에 주입하지 않고, 대상 취약점과 직접 관련된 항목만 선별하여 Markdown 형식의 STUCTURED CONTEXT 블록으로 변환한다. 이를 통해 입력 토큰을 경량화하면서도, 단순 문자열 설명보다 풍부한 반정형 맥락 정보를 LLM에 제공할 수 있다.

3.3.2 Annotation 기반 위치 강조 기법

구조화 메타데이터가 취약점의 의미적 정보를 제공한다면, Annotation 기반 위치 강조는 코드 내에서 수정 범위를 명확히 한정하는 역할을 한다. AutoFiC은 취약점이 보고된 코드 스니펫과 해당 함수 또는 블록 전체를 프롬프트에 포함하고, 필요 시 취약 줄 또는 범위를 명시적 Annotation 마커(예: @BUG_HERE, @BUG_HERE_START, @BUG_HERE_END)로 표시한다. 단일 라인 취약점은 해당 줄 수준에서 Annotation을 적용하고, 범위 기반 취약점은 시작·종료 라인 기준으로 표현함으로써, 모델이 전체 파일을 과도하게 수정하는 것을 방지하고 취약 구간 중심의 패치를 생성하도록 유도한다. 패치 생성 이후에는 후처리 단계에서 Annotation 마커를 제거하여, 결과 코드의 가독성과 품질에 영향을 최소화한다.

3.3.3 프롬프트 설계 원칙

AutoFiC의 프롬프트는 다음 원칙에 따라 구성된다.

- 역할 및 제약 명시: 시스템 메시지에 보안 패치 어시스턴트 역할을 부여하고, 기존 기능 보존 및 신규 취약점 미도입과 같은 제약을 명시한다.
- 컨텍스트 최소화: 취약점과 직접 관련된 함수 또는 블록만 포함하되, 제어 흐름이 단절되지 않도록 스니펫 경계를 설정한다. XML 요약 블록과 코드 블록은 명확히 구분하여 제시한다.
- 수정 범위 제한: Annotaion을 통해 표시된 취약 범위만 수정하도록 요구하고, 나머지 영역은 최소 변경을 유지하도록 명시한다.
- 형식화된 출력 요구: 결과 출력 형식을 고정하여, 수정된 전체 함수 또는 변경 블록 전체를 포함하도록 요구함으로써 파싱 및 diff 기반 자동 패치 적용의 안정성을 확보한다.

```
The following is a Python source file that contains security vulnerabilities.
  ...
Detected vulnerabilities:
  ...
## STRUCTURED CONTEXT (Team-Atlanta Approach)
The following CUSTOM_CONTEXT.xml provides structured vulnerability information including:
- BIT (Bug Information Template) with TRIGGER, STEPS, REPRODUCTION
```

- Detailed CWE classifications and severity levels
- Environmental context and mitigation strategies ```xml… ```

Use the BIT information above to understand:
1. TRIGGER: What conditions activate this vulnerability
2. STEPS: How to locate and review the vulnerable code
3. REPRODUCTION: How to verify the issue

4. BIT_SEVERITY: The criticality level of this vulnerability

Please strictly follow the guidelines below when modifying the code:
- Modify **only the vulnerable parts** of the file with **minimal changes**.
- Preserve the **original line numbers, indentation, and code formatting** exactly.
- **Do not modify any part of the file that is unrelated to the vulnerabilities.**
- Output the **entire file**, not just the changed lines.
- This code will be used for diff-based automatic patching, so structural changes may cause the patch to fail.

Output format example:
1. Vulnerability Description: ...
2. Potential Risk: ...
3. Recommended Fix: ...
4. Final Modified Code:
5. Additional Notes: (optional)

그림 3. AutoFiC의 LLM 프롬프트 템플릿 예시

이와 같은 컨텍스트 설계는 LLM 기반 자동 패치에서 중요한 정확한 위치 지정과 구조적 힌트 제공을 동시에 달성하도록 하며, 4장에서 정량 실험을 통해 그 효과를 평가한다.

3.4 자동 패치 적용 및 Pull Request 자동화

AutoFiC은 LLM이 생성한 패치를 코드베이스에 안전하게 반영하고, Pull Request 생성과 CI 연계를 통해 이를 실제 개발 워크플로우에 통합하는 자동화 기능을 제공한다. 본 절에서는 LLM 응답 파싱, Diff 생성 및 패치 적용, Pull Request 생성 및 CI 연계의 세 단계로 구성된 자동화 절차를 설명한다.

3.4.1 LLM 응답 파싱

생성된 응답에서 전용 파서(Parser)를 통해 수정 코드 블록을 추출한다. 이때 응답이 지정된 형식을 위반하거나 구문 오류를 포함하는 경우, 해당 케이스를 패치 불가로 분류하거나 재생성 로직으로 전환한다. 또한 API 호출 중 발생하는 토큰 한도 초과 등의 외부 예외를 별도로 처리하여 시스템 안전성을 유지한다.

3.4.2 Diff 생성 및 패치 적용

파싱된 수정 코드는 원본 코드와의 비교를 통해 Unified Diff 형식으로 변환되며, 이후 Git Patch 메커니즘을 통해 자동 적용된다. AutoFiC은 자동 패치 과정에서 발생할 수 있는 충돌이나 문맥 불일치 문제에 대응하기 위해 이중 적용 전략을 사용한다.

우선 표준 Diff 기반 패치 적용을 1차적으로 시도하며, 이 과정이 실패할 경우 전체 파일 단위로 수정된 코드를 덮어쓴 뒤 다시 Diff를 생성하는 Fallback 절차를 수행한다. 이를 통해 Diff 적용 실패로 인한 파이프라인 중단을 방지하고 End-to-End 자동화 흐름을 지속적으로 유지하기 위한 안전장치를 마련한다.

실험 과정에서, LLM이 생성한 수정 내용이 논리적으로는 타당함에도 불구하고, 미세한 공백 차이, 주석 위치 변경, 출력 포맷 차이 등으로 인해 Diff 적용이 실패하는 사례가 다수 관찰되었다. 이러한 경우 Fallback 절차를 통해 패치 적용을 재시도함으로써, 자동화 파이프라인이 중단되지 않고 Pull Request 생성 단계까지 도달하는 비율이 증가하는 경향을 확인하였다. 이러한 결과는 Fallback 절차가 AutoFiC 파이프라인의 안정성과 연속성을 보장하는 핵심 구성 요소로 기능함을 시사한다.

3.4.3 Pull Request 생성 및 CI 연계

패치 적용이 완료된 Branch는 원격 저장소로 Push되며, 이를 기반으로 상세 정보를 포함한 Pull Request가 자동 생성된다. 시스템은 GitHub Actions 워크플로우를 자동 구성하여 Pull Request 이벤트를 감지하고, CI 테스트 결과를 모니터링한다. 모든 결과는 암호화된 채널을 통해 Slack 또는 Discord로 실시간 전달되며, 이를 통해 개발자는 보안 패치를 신속하게 검토하고 병합할 수 있도록 지원한다.

4. 실험 결과

4.1 실험 설계

본 연구에서 제안하는 AutoFiC 파이프라인의 실효성을 검증하기 위해, 실제 오픈소스 환경을 대상으로 파이프라인 동작 성공률과 취약점 패치 성능을 평가하였다. 또한 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법이 패치 성능에 미치는 효과를 분석하기 위해, 해당 기법 적용 전후의 결과를 비교 분석하였다.

4.1.1 데이터셋 및 실험 환경

실험 데이터셋은 GitHub에 공개된 Python 기반 저장소 중, 현실적인 개발 프로젝트 규모를 반영하기 위해 Star 수가 30 이상 100 이하인 중소 규모 프로젝트를 대상으로 구성하였다. GitHub API를 통해

수집된 저장소 중, Semgrep 정적 분석 결과 최소 1개 이상의 취약점이 탐지된 91개 저장소를 최종 실험 대상으로 선정하였다.

Star 수가 높은 대규모 프로젝트는 복잡한 CI 설정, 방대한 의존성, 장시간 테스트 실행 등을 포함하는 경우가 많아, 자동 패치 파이프라인의 실행 시간, 환경 재현성, 비교 가능성을 저하시킬 수 있다. 본 연구는 개별 프로젝트의 CI 환경 최적화나 복잡한 빌드/테스트 설정을 해결하는 것을 목표로 하지 않으며, 중소 규모 프로젝트를 대상으로 End-to-End 자동화 파이프라인의 일반적인 실효성과 안정성을 검증하는 데 초점을 두었다. 이에 따라 이러한 대규모 프로젝트는 실험 범위에서 제외하였다

실험은 Python 3.10 환경에서 수행되었으며, 패치 생성을 위한 LLM 모델로는 GPT-4o를 사용하였다. 취약점 탐지 및 해결 여부 판단의 일관성을 확보하기 위해 Semgrep을 기준 정적 분석 도구로 사용하였다. 패치 적용 전후에 동일한 규칙과 조건으로 Semgrep을 재실행하여 탐지 결과의 변화를 기반으로 취약점 해결 여부를 판정하였다.

4.1.2 평가 지표

평가 지표는 다음과 같이 설정하였다.

1. **파이프라인 성공률 (Pipeline Success Rate):** 전체 저장소 중 Fork부터 Pull Request 생성까지의 전 과정이 중단 없이 수행된 비율.
2. **취약점 해결률 (Vulnerability Fix Rate):** 파이프라인이 성공한 저장소를 대상으로, 패치 적용 후 Semgrep을 동일한 규칙과 조건으로 재실행하였을 때 기존에 탐지된 취약점이 보고되지 않는 비율.

본 연구에서의 취약점 해결률은 정적 분석 기준으로 동일 취약점의 재탐지 여부에 기반하며, 생성된 패치가 기존 소프트웨어의 동작을 완전히 보존하는지를 자동으로 증명하는 지표는 포함하지 않는다. 기능 보존성 검증은 저장소별 테스트 환경 구축과 안정적인 테스트 실행을 필요로 하므로 본 연구의 실험 범위를 벗어나며, 단위 테스트 생성 및 실행 기반의 자동 검증은 향후 연구 과제로 남긴다. 본 연구는 PR 기반 개발 프로세스를 전제로 하며, 생성된 패치의 기능적 동등성은 유지보수자의 코드 리뷰와 CI 테스트를 통해 최종적으로 검증되는 것을 가정한다.

4.2 전체 파이프라인 동작 성공 비율

AutoFiC 파이프라인을 91개 저장소에 적용한 결과, 86개 저장소에서 전 과정이 정상적으로 완료되어 94.5%의 파이프라인 성공률을 기록하였다. 실패한 5개 사례를 분석한 결과, 3건(3.30%)은 LLM의 토큰 제한 초과로 인해 발생하였으며, 나머지 2건(2.20%)은 GitHub API 통신 오류로 확인되었다. 이러한 결과는 제안 시스템의 설계가 별도의 개입 없이도 다수의 실제 오픈소스 프로젝트에 대해 안정적인 End-to-End 자동화를 가능하게 함을 시사한다.

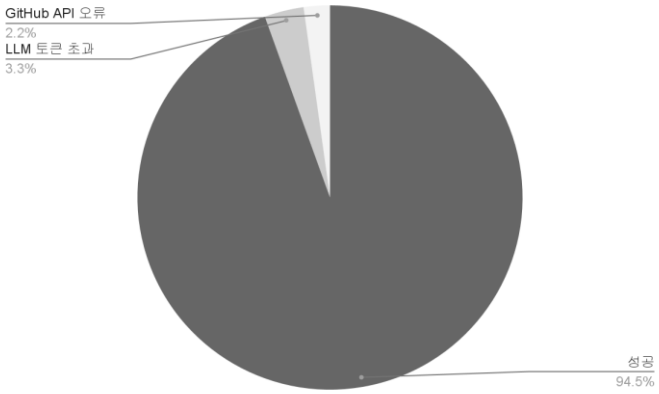

그림 4. 파이프라인 전체 동작 성공 비율

4.3 취약점 해결 여부

파이프라인이 정상적으로 완료된 86개 저장소를 대상으로, 패치 적용 전후에 동일한 조건으로 Semgrep을 재실행하여 취약점 해결 여부를 분석하였다. 그 결과, 총 441개의 취약점이 탐지되었으며 AutoFiC 적용 후 397개가 해결되어 90.0%의 취약점 해결률을 기록하였다.

표 1. CWE 유형별 취약점 해결률

취약점 유형	탐지 수	해결 수	해결률
CWE-116	2	2	100.0%
CWE-200	5	5	100.0%
CWE-276	9	8	88.9%
CWE-295	3	3	100.0%
CWE-319	75	72	96.0%
CWE-326	27	27	100.0%
CWE-327	65	59	90.8%
CWE-330	4	3	75.0%
CWE-489	9	6	66.7%
CWE-502	17	16	94.1%
CWE-532	17	15	88.2%
CWE-611	30	28	93.3%
CWE-668	7	7	100.0%
CWE-78	153	138	90.2%
CWE-79	17	7	41.2%

CWE-942	1	1	100.0%
합계	441	397	90.0%

표 1은 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 모두 적용한 AutoFiC 파이프라인에서의 CWE 유형별 취약점 해결 현황을 나타낸다. CWE-116, CWE-200, CWE-295, CWE-326, CWE-668, CWE-942는 100%의 해결률을 기록하였으며, CWE-319와 CWE-327 역시 각각 96.0%, 90.8%의 높은 해결률을 보였다. 가장 많은 빈도로 탐지된 CWE-78(Command Injection)의 경우, 153건 중 138건이 해결되어 90.2%의 해결률을 기록하였다. 반면 CWE-79(XSS)는 17건 중 7건만이 해결되어 41.2%로 상대적으로 낮은 해결률을 보였다.

표 2. XML 및 Annotation 도입 이전의 CWE 유형별 취약점 해결률

취약점 유형	탐지 수	해결 수	해결률
CWE-116	2	2	100.0%
CWE-200	5	5	100.0%
CWE-276	9	8	88.9%
CWE-295	3	3	100.0%
CWE-319	75	74	98.7%
CWE-326	27	24	88.9%
CWE-327	65	58	89.2%
CWE-330	3	3	100.0%
CWE-489	9	6	66.7%
CWE-502	17	16	94.1%
CWE-532	17	15	88.2%
CWE-611	30	28	93.3%
CWE-668	7	7	100.0%
CWE-78	153	135	88.2%
CWE-79	17	7	41.2%
CWE-942	1	1	100.0%
합계	440	392	89.1%

표 2는 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 적용하지 않은 경우의 CWE 유형별 취약점 해결 현황을 나타낸다. 전후 비교 결과, 전체 취약점 해결률은 89.1%에서 90.0%로 증가하였으며, 해결된 취약점 수는 392건에서 397건으로 총 5건 증가하였다. 특히 CWE-78과 CWE-326에서 해결된 취약점 수가 각각 3건 증가하여, 제안 기법 적용 이후 해당 유형의 패치 성능이 향상되었음을 확인할 수 있다.

이러한 결과는 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법이 취약점 수정 대상의 범위를 보다 명확히 한정함으로써, LLM이 불필요한 코드 영역을 탐색하지 않고 핵심 취약 지점에 집중하도록 유도했음을 보여준다. 특히 CWE-78, CWE-326과 같이 취약점 발생 위치와 수정 패턴이 비교적 명확한 유형에서 해결된 취약점 수가 증가한 점은, 위치 정보와 유형 정보를 명시적으로 제공하는 설계가 패치 정확도 향상에 기여했음을 수치적으로 뒷받침한다. 반면 CWE-79와 같이 출력 컨텍스트에 대한 의미적 해석이 요구되는 취약점 유형에서는 해결률 개선이 제한적으로 나타났으며, 이는 경량 컨텍스트 기반 접근 방식의 적용 범위와 한계를 동시에 시사한다.

기존의 LLM 기반 취약점 탐지 및 패치 생성 연구들은 개별 단계에서 높은 성능을 보이는 알고리즘을 제안해 왔다. 그러나 이러한 연구들을 실제 개발 환경에서 조합하여 사용하기 위해서는, 취약점 탐지 결과의 정규화, 패치 실패 처리, diff 적용 오류 대응, Pull Request 생성 및 CI 연계와 같은 추가적인 엔지니어링 작업이 요구된다. AutoFiC은 취약점 탐지–패치 생성–적용–Pull Request 생성까지의 전 과정을 하나의 자동화된 파이프라인으로 통합함으로써, 이러한 실무적 부담을 최소화하는 데 초점을 둔다. 실험 결과는 AutoFiC이 실제 오픈소스 프로젝트 환경에서도 높은 파이프라인 성공률을 유지하며 End-to-End 자동화를 안정적으로 수행할 수 있음을 보여준다.

5. 결론

본 연구는 정적 분석 도구(SAST)와 대규모 언어 모델(LLM)을 결합하여, 취약점 탐지부터 패치 생성 및 적용, Pull Request 생성까지 이어지는 DevSecOps 관점의 End-to-End 자동 보안 패치 파이프라인 AutoFiC을 제안하였다. 제안 시스템은 보안을 기존 개발 워크플로우에 자연스럽게 통합함으로써, 취약점 발견과 수정 사이의 시간적 간극을 줄이고 개발자의 수작업 부담을 완화하는 자동화된 보안 패치 환경을 제공한다.

실제 오픈소스 프로젝트를 대상으로 한 실험 결과, AutoFiC은 다양한 프로젝트 환경에서도 안정적으로 동작하며, 정적 분석 기준에서 탐지된 취약점을 자동으로 패치 생성 및 적용 단계까지 연결할 수 있음을 확인하였다. 특히 XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 도입함으로써, 취약점의 위치와 유형 정보가 LLM에 보다 명확히

전달되었고, 그 결과 패치 정확도와 일관성이 전반적으로 향상되는 경향을 보였다. 이는 복잡한 AST 기반 분석을 직접 활용하지 않더라도, 경량 구조 정보를 적절히 설계하여 제공하는 것만으로도 LLM 기반 자동 패치 파이프라인의 실용성을 높일 수 있음을 시사한다.

본 연구의 주요 기여는 다음과 같다. 첫째, DevSecOps 관점에서 SAST 기반 취약점 탐지부터 GitHub 워크플로우 연계까지의 전 과정을 자동화하여, 보안 패치가 개발 파이프라인에 자연스럽게 통합되는 실용적 시스템을 구현하였다. 둘째, XML 기반 구조화 컨텍스트와 Annotation 기반 위치 강조 기법을 활용하여 AST 기반 분석을 직접 활용하지 않으면서도 구조적 컨텍스트를 경량하게 제공하는 방법을 제시함으로써, 언어 의존성과 구현 복잡도를 낮추는 동시에 LLM의 취약점 이해도를 향상시켰다. 셋째, 실제 오픈소스 환경을 대상으로 한 대규모 실험을 통해 제안 기법의 실효성을 정량적으로 검증하고, 컨텍스트 도입 전후 비교를 통해 그 개선 효과를 명확히 확인하였다.

한편, 본 연구는 몇 가지 한계를 가진다. 본 실험에서는 정적 분석 도구가 보고한 결과를 실제 취약점으로 가정하여 자동 패치를 수행하였으며, 이로 인해 오탐으로 인한 불필요한 코드 수정 가능성을 명시적으로 배제하지 않았다. 이는 제안 파이프라인의 동작 특성과 자동화 흐름 자체를 평가하기 위한 실험적 선택이었으며, 오탐 여부에 따른 패치 적절성에 대한 정량적 평가는 본 연구 범위에 포함하지 않았다. 또한, CWE-79(XSS)와 같이 출력 컨텍스트에 따라 다양한 이스케이프 방식이 요구되는 취약점의 경우, 상대적으로 낮은 해결률을 보였다. 이는 경량 컨텍스트 설계가 템플릿 엔진 구조나 출력 지점의 의미적 맥락을 제한적으로만 전달하기 때문으로 해석된다. 또한 생성된 패치가 기존 기능을 완전히 보존하는지에 대한 자동화된 검증을 제한적으로 수행하였다. 취약점 제거 여부를 정적 분석 기준으로 확인하였으며, 패치 이후의 기능적 정합성이나 실행 의미 보존에 대한 체계적 검증은 향후 과제로 남아 있다. 더불어 복잡한 제어 흐름이나 프레임워크 특화 로직을 포함하는 취약점에 대해서도 추가적인 개선 여지가 존재한다.

향후 연구에서는 다중 정적 분석 도구를 교차 활용하여 공통으로 탐지된 취약점만을 선별하는 방식이나, 규칙 신뢰도 기반 필터링을 통해 오탐을 완화하는 전략을 탐색할 예정이다. 또한 경량 XML·Annotation 기반 접근법과 AST 기반 구조 정보를 선택적으로 결합하는 하이브리드 전략을 통해, 보다 복잡한 취약점 유형에 대한 대응 능력을 향상시키고자 한다. 나아가 자동 메커니즘과 단위 테스트 생성 기법을 결합하여 패치의 기능 보존성을 체계적으로 검증하고, Java, C/C++ 등 다양한 프로그래밍 언어로 확장함으로써 AutoFiC의 범용성과 실용성을 더욱 확대할 계획이다. 프레임워크별 특화 프롬프트와 도메인 지식을 반영한 패치 템플릿을 통해 난이도 높은 취약점의 해결률을 개선하고, 자동 검증 메커니즘과 단위 테스트 생성 기법을 결합하여 패치의 기능 보존성을 보다 체계적으로 검증하고자 한다. 마지막으로, Java, C/C++ 등 다양한 프로그래밍 언어로 확장하여 AutoFiC의 범용성과 실용성을 확대할 계획이다.

참고문헌

[1] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," Communications of the ACM, vol. 61, no. 4, pp. 58–66, 2018.

[2] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-Trained Language Models," Proceedings of the 45th International Conference on Software Engineering (ICSE), pp. 1482–1494, 2023.

[3] R. Macháček, A. Grishina, M. Hort, and L. Moonen, "The Impact of Fine-tuning Large Language Models on Automated Program Repair," Proceedings of the 41st International Conference on Software Maintenance and Evolution (ICSME), pp. 380–392, 2025.

[4] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," IEEE Transactions on Software Engineering, vol. 45, no. 1, pp. 34–67, 2019.

[5] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," Information and Software Technology, vol. 53, no. 4, pp. 363–387, 2011.

[6] A. T. M. F. Rabbi and M. A. Joarder, "Automatic program repair: A systematic literature review," Systematic Literature Review and Meta-Analysis Journal, vol. 4, no. 3, pp. 1–10, 2023.