프로그램 실행 중 발생한 버그의 실시간 자동 수정

노준영 ○ 김영재

울산과학기술원

nojon123@unist.ac.kr, kyj1411@unist.ac.kr

Axolotl: Automatically Fix Programs' Fault On-the-fly

Joonyeong Noh ○ YoungJae Kim

UNIST

요 약

기존 자동 프로그램 수정 (Automated Program Repair, APR) 연구들은 프로그램의 버그에 의해 실패하는 테스트 케이스가 있다고 전제하고, 해당 테스트를 성공하는 패치를 생성하도록 구성되어 있다. 따라서 프로그램의 테스트 케이스가 제공되고 프로그램을 매번 컴파일 후 실행할 수 있다고 가정하는데, 프로그램이 실제 서비스에 적용되면 테스트 케이스가 포함되어 있지 않은 경우가 많고 버그가 발생한 당시 이미 프로그램이 오랫동안 실행되어 처음부터 재실행하는 것이 힘든 경우가 많다. 본 연구에서는 프로그램 배포 후, 실제로 프로그램이 실행 중일 때 크래시 버그가 발생하여 예외를 일으킬 때, 프로그램을 중단하지 않고 일시정지한 후 런타임에 프로그램을 수정하고 난 다음 이어서 진행하는 방법을 연구하였다. 프로그램을 일시정지한 후 빠르게 좋은 품질의 패치를 얻기 위해 런타임에 얻을 수 있는 정보들과 ToT (Tree-of-Thought), 피드백 기법 등을 이용하여 대형 언어 모델(LLM)로 패치를 생성하였다. 그 후 생성된 패치를 검증하기 위해 black-box 퍼징 기법을 활용하였다. 이 연구를 통해 Python 프로그램에서 실행 도중 예외가 발생할 때 자동으로 수정한 후 프로그램을 이어서 실행하는 Axolotl 을 구현하였으며, 실험에서는 Python 을 사용하는 BugsInPy 벤치마크의 crash 버그들을 사용하여 평균적으로 11.3 분 이내에 92%의 패치 성공률을 보였다.

1. 서론

지난 2009 년에 출판된 Genprog [1] 이후, 최근까지 많은 자동 프로그램 수정 (Automated Program Repair, APR) 연구들이 진행되었다. 기존의 APR 연구들은 고품질의 패치를 생성하여 최대한 많은 버그들을 성공적으로 수정하는 데에 집중하고 있다. 이를 위해 많은 연구 [1-7]에서는 버그에 의해 실패하는 테스트들(failing test)과 버그에 상관없이 성공하는 테스트들(passing test)을 입력으로 받는다고 전제하여, 수많은 패치 후보들을 생성한 후 해당 테스트들을 실행하여 모든 테스트를 성공하면 성공적인 패치로 분류한다. 따라서 기존의 연구들은 이 과정에서 좋은 품질의 패치 후보들을 생성하여 패치 성공률을 높이는 데에 초점을 맞추고 있다.

기존 연구들에서는 프로그램들이 개발 과정 중이었다고 전제하여, 해당 프로그램들에게 실제로 패치 후보를 적용하여 반복적으로 컴파일하고 테스트를 실행하면서 패치를 생성하고 있다. 하지만 프로그램 배포 후에서는 테스트 케이스가 없고, 컴파일과 테스트 실행을 반복하며 패치를 검증하는 것은 매우 오랜 시간이 걸린다. 또 프로그램이 실제로 배포된 후에 버그가 발생하면 심각한 피해를 야기할 수 있다. 따라서 배포 이후에 발생하는 심각한 버그들은 개발자의 개입 이전에 프로그램의 중단이나 오작동을 방지하도록 매우 빠르면서도 심각한 부작용을 일으키지 않는 임시 패치를 진행할 필요가 있다.

NPEX [22]는 Java 프로그램에서 실행 도중 null pointer exception (NPE)이 발생했을 때, 주어진 테스트 없이 모델을 이용하여 생성한 패치를 검증한다. 이를 위해 기존에 개발자들이 NPE 를 패치했던 패턴을 이용해 모델을 학습한다. 그 후 프로그램을 이용해 심볼릭 실행 (symbolic execution)을 진행해 NPE 를 수정할 수 있는 명세를 추출한다. NPEX 는 Java 프로그램이 실행 중 NPE 가 발생한 상황에 특화되어 프로그램 실행 도중에 발생한 모든 예외에 대응할 수 없다.

Casino [23]과 Gresino [24]는 테스트가 주어진 상황에서 수많은 패치 후보들이 있을 때 그 후보들을 더 효율적으로 검증하기 위한 연구들이다. APR 에서 패치를 적용한 후 테스트를 실행하기 위해서는 프로그램을 매번 재컴파일해야 하는데, 이 과정이 오래 걸리고 일반적으로 오픈소스 프로젝트들은 모든 테스트를 실행하는데 시간이 오래 걸린다. Casino 와 Gresino 는 제한된 시간 내에 빠르게 패치를 찾아낼 수 있지만, 이 연구들 또한 프로그램을 중단한 후에 패치를 생성해야 하고 여전히 재컴파일과 테스트 실행이 필요하다.

위의 연구들은 프로그램의 실행 도중 흔히 발생하는 예외에 빠르게 대응할 수 있도록 하지만, 프로그램을 수정하기 위해 여전히 프로그램을 중단해야 한다는 한계가 있다. 또한 생성한 패치가 버그를 잘 고쳤는지 확인하기 위해 매번 프로그램을 재컴파일하고 수많은 테스트들 혹은 검증 과정을 실행해야 한다. 마지막으로 버그 수정이 완료된 후에는 프로그램을 처음부터 다시 실행해야 한다.

위의 한계들을 해결하기 위해 본 연구에서는 프로그램의 실행 도중 예외(exception)가 발생한 경우, 프로그램을 강제로 종료하지 않고 일시정지하여 실행중인 프로그램에 직접 패치를 적용한 후 이어서 실행하는 연구를 진행하였다. 이를 위해, 본 연구에서는 프로세스 체크포인트 기술을 활용하여 미리 실행중인 프로그램을 체크포인트하여 파일로 저장하고, 예외가 발생한 경우 저장된 프로세스를 불러와 패치를 생성 및 적용하고 프로그램을 이어서 실행할 수 있도록 하였다.

또한 최신 초대형 언어 모델 (LLM)과 Tree-of-Thought (ToT) [7], 그리고 피드백 기법을 활용하고 예외 메시지와 예외가 발생했을 때의 함수 인자 등 런타임 정보를 활용해 최대한 좋은 품질의 패치 후보를 생성해 프로그램이 오랜 시간 중단되지 않고 빠르게 패치를 생성하도록 구성하였다. 생성된 패치 후보가 올바른 패치인지 검증하기 위해 퍼징 기법을 활용해, 실행중인 프로그램에 직접 패치를 적용한 후 다양한 입력을 프로그램에 계속 적용하여 프로그램이 일어나서는 안 되는 부작용 (side-effect)를 일으키는지 확인하였다.

본 연구가 실제로 효과가 있는지 확인하기 위해 본 연구를 Python 을 이용해 Axolotl 로 명명한 도구를 구현하였으며, 실제 Python 오픈소스 프로젝트들의 버그들을 모아놓은 BugsInPy [25] 벤치마크에서 실제로 프로그램 내에서 예외가 발생하는 버그들만을 수집하여 실험을 진행하였다. LLM 으로 GPT-5.2 를 사용한 결과, 92%의 버그들을 성공적으로 수정할 수 있었으며 평균적으로 11.3 분이 걸렸다.

요약하면, 본 연구에서는 다음과 같은 내용들을 진행하였다.

- 실제 서비스중인 프로그램의 버그 수정: 실행중인 프로그램을 종료하지 않고 수정 후 이어서 실행
- 테스트들이 주어지지 않은 상황에서 패치 검증: 테스트가 없는 상황에서 퍼징 기법을 이용하여 패치가 버그를 수정하고 다른 부작용을 일으키지 않는지 검증
- 구현물 공개: 본 연구의 구현물(Axolotl)과 실험 데이터를 오픈소스로 공개
  - https://github.com/UNIST-LOFT/axolotl

2. 배경 지식

2.1. 자동 프로그램 수정 (APR)

기존의 APR 연구들은 일반적으로 얼마나 더 많은 버그들을 수정할 수 있는지에 집중하고 있으며, 이를 위해 더 좋은 품질의 패치를 생성하는 연구가 주를 이루고 있다.

따라서 기존의 연구들은 주로 패치를 더 잘 생성하는 방법을 중심으로 진행되고 있다. Genprog [1] 등은 버그가 있는 프로그램의 코드를 버그가 사라질 때까지 변형하여 패치를 생성하며, TBar [2] 등은 패치 템플릿을 사용하여 특정 패턴의 패치들을 생성한다. 최근에는 초대형 언어 모델 (LLM)을 사용하여 패치를 생성하는 연구 [5,6,7]가 활발히 진행되고 있다.

그림 1 은 기존의 일반적인 APR 연구들의 실행 흐름을 보여주고 있다. 입력으로 실패 테스트 (failing test)와 성공 테스트(passing test)를 받아, 패치 후보를 생성한 후 각 후보를 프로그램에 적용하여 실패 테스트와 성공 테스트를 실행한다. 해당 패치 후보가 모든 테스트를 성공하면 해당 패치는 그럴듯한 패치 (plausible patch)로 분류한다.

2.2. LLM 기반 패치 생성

최근 LLM 을 활용하여 패치를 생성하는 연구에서는 Chain-of-Thought (CoT) [11] 기법을 사용하여 패치를 생성한다. CoT 는 LLM 에 패치 생성을 요청할 때 패치를 한 번의 프롬프트만을 이용해 바로 생성하지 않고, 버그를 수정하기 위한 여러 과정을 LLM 을 이용해 진행하는 기법이다.

Nong et al. [10]은 보안 위협 (vulnerability)에 대응하기 위한 패치를 생성하기 위해 CoT 기법을 사용한다. 이 논문에서는 먼저 LLM 을 이용해 코드에 어떠한 보안 위협이 있는지 확인하고, 해당 보안 위협이 어디에 있는지 분석한 후 실제 패치를 생성하는 세 가지 과정을 위해 LLM 을 사용한다.

San2Patch [7]는 보안 위협에 대응하기 위해 CoT 를 확장한 Tree-of-Thought (ToT) 기법을 사용한다. ToT 는 CoT 의 각 과정에서 LLM 이 응답을 생성할 때 여러 개의 응답을 생성한 후, LLM 을 이용해 응답들에 점수를 매겨 높은 점수를 받은 응답에 우선순위를 주면서 샘플링하는 방법을 통해 좋은 응답을 더 많이 사용한다.

또 최근에는 패치 생성 후 테스트를 실행하여 실패한 경우, 해당 패치나 테스트 결과 등을 LLM 에 추가로 제공하여 더 좋은 품질의 패치를 얻는 피드백 기법도 연구가 이루어지고 있다. ThinkRepair [41]은 패치를 적용한 후 컴파일과 테스트 과정에서 나온 결과를 피드백으로 활용해 다음 패치 후보를 생성한다.

2.3. 프로세스 체크포인트

프로세스 체크포인트는 현재 시스템에서 실행중인 프로세스의 현재 상태를 파일로 저장하는 기술로, 추후에 원하는 경우 저장된 체크포인트를 불러와 이어서 실행할 수 있도록 한다.

CRIU [8]는 원하는 프로세스를 명령줄 (CLI) 환경에서 간단히 체크포인트하여 파일로 저장하는 기능을 제공하며, 추후에 해당 파일을 사용하여 프로세스를 복원한 후 이어서 실행할 수 있다. 또한 Docker [9]와 같이 사용하여 컨테이너도 체크포인트할 수 있는 기능을 제공한다. 추가적으로 체크포인트를 저장하고 불러오는 과정 중간에 셸 스크립트를 실행할 수 있다.

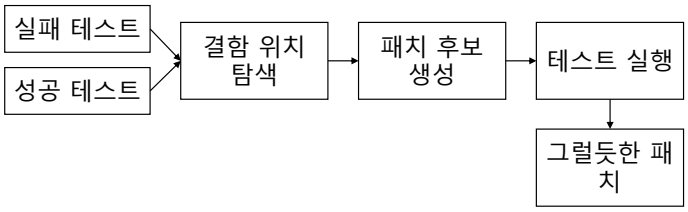

그림 1: APR 의 일반적인 흐름

3. 문제 제기

그림 1 은 기존 APR 연구들의 일반적인 흐름을 보여준다. 입력으로 실패 테스트 (failing test), 성공 테스트 (passing test), 버그가 있는 프로그램을 받고, 먼저 결함 위치 탐색 (fault localization)을 진행해 소스 코드에서 패치를 생성할 위치를 찾는다. 그 후, 다양한 방법들(2.1 절과 7.1 절 참조)을 통해 다량의 패치 후보들을 생성한다. 그 후에는 각 패치 후보들을 하나씩 프로그램에 적용해 실패 테스트를 실행하여 결과를 확인한다. 만약 모든 실패 테스트를 성공했다면 남은 성공 테스트들을 실행해 모두 성공하는지 확인한다. 모든 성공 테스트까지 모두 성공하면 해당 패치는 그럴듯한 패치 (plausible patch)로 분류한다.

이 흐름에서는 해당 프로그램의 테스트 케이스가 주어져야 하고, 각 패치마다 테스트를 실행해야 한다는 문제가 있다. 하지만 프로그램이 이미 서비스 중일 때에는 테스트들을 반복적으로 실행할 시간이 없고 심지어 테스트가 주어지지 않은 경우도 많다. 또한 패치가 완료된 후에 프로그램을 처음부터 다시 실행해야 한다는 문제가 있지만 서비스 중인 프로그램은 이미 많이 실행되어 프로그램을 처음부터 실행해 오류가 발생했던 시점으로 다시 돌아오는 것이 오래 걸리거나 불가능한 경우가 많다. 예를 들어, 본 연구의 실험에 사용한 BugsInPy [25] 벤치마크에 포함된 fastapi 와 httpie 프로그램은 웹 서버에 흔히 사용되어, 서버를 중단하여 버그를 수정한 후 재시작하는 것이 불가능하거나 시간이 오래 걸리는 경우가 많다. 따라서 기존의 APR 연구들은 프로그램의 배포 후가 아닌 개발 과정에서의 실행을 전제한다.

따라서 프로그램의 배포 후에는 예외가 발생한 경우 프로그램의 전체 테스트 케이스 없이 예외가 발생하는 시나리오 하나만으로 부작용이 없는 임시 패치를 빠르게 생성할 수 있어야 한다.

4. Axolotl

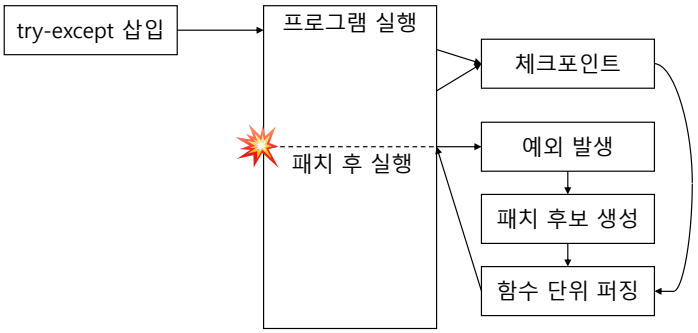

그림 2: Axolotl 의 개요

4.1. 개요

그림 2 는 본 연구를 구현한 Axolotl 의 전체 흐름을 보여준다. 프로그램을 실행하기 전에 자동으로 각 함수들의 몸체(body) 전체를 try-except 로 감싼 후, 프로그램을 실행한다. 프로그램 실행 중에는 패치 생성 후 다시 돌아갈 체크포인트를 지속적으로 저장한다. 예외가 발생하면 먼저 패치 후보들을 생성한 후, 저장해 두었던 체크포인트를 불러와 다양한 입력들을 생성하면서 함수 단위 퍼징을 진행해 각 패치들이 프로그램에서 부작용 (side-effect)을 유발하는지 확인한다. 패치가 부작용을 일으키지 않으면 다시 체크포인트를 불러와 패치를 적용하여 프로그램을 이어서 실행한다.

4.2. try-except 문 삽입

try-except 문은 프로그램 실행 전에 자동으로 각 함수의 몸체(body) 전체를 감싸는 형태로 삽입한다. 이를 통해 각 함수에서 예외가 발생하면 except 문으로 이동하여 Axolotl 의 패치 생성 단계로 넘어간다. try-except 문을 삽입할 때에는 Python 의 bytecode 단위에서 삽입하여 프로그램의 소스 코드 길이가 증가하지 않고 프로그램의 실행 시간 증가를 최소화하였다.

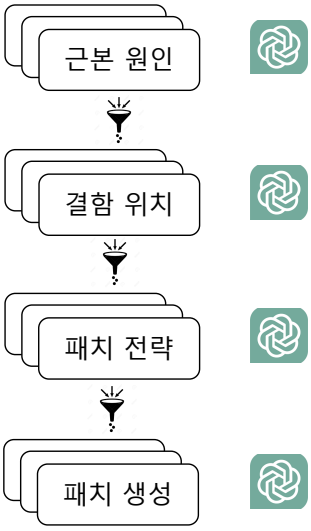

그림 3: 패치 후보 생성 과정

4.3. 프로그램의 체크포인트

예외 발생 시 패치 생성 후 실행 중인 프로그램으로 돌아오기 위해 Axolotl 은 주기적으로 실행 중인 프로그램의 체크포인트를 저장한다. 본 실험에서는 일정 시간마다 CRIU [8]를 이용하여 프로세스 전체를 저장하였다. 체크포인트는 별도의 프로세스를 통해 저장하여 원래 프로그램의 실행을 방해하지 않으며 오버헤드가 없다.

저장된 체크포인트는 패치 생성 및 적용 후 함수 단위 퍼징을 진행할 때 불러온다. 하지만 함수 단위 퍼징을 진행하려면 불러오려는 체크포인트가 패치된 함수에 진입하기 전이어야 패치가 적용할 수 있는데, 외부 프로세스에서는 프로그램 내부 동작을 알 수 없으므로 가장 최근에 저장된 체크포인트가 패치된 함수 이전에 저장되었는지 보장할 수 없다. 이를 위해 Axolotl 은 가장 최근에 저장된 체크포인트만 남기는 것이 아닌 기존에 저장했던 체크포인트를 계속 남겨 퍼징하려는 함수가 실행되기 직전의 체크포인트를 찾아 불러온다.

표 1: 패치 생성 시 사용한 프롬프트

원인 파악	<예외 로그> <stack trace> <Target Buggy Function> <버기 함수> Using the information above, create a detailed yet concise description of the exception. (후략)
결함 위치	<Goal> Your task is to map a provided <근본 원인> to the specific code snippet within the <버기 함수>. </Goal> <Instruction> 1. Read the <근본 원인> carefully. 2. Based on the <예외 종류> and <전략>, identify the **SINGLE most critical code snippet** that needs modification. (후략)
전략 생성	<Goal> (중략) </Goal> <Instruction> Please follow these steps to generate a robust fix strategy: 1. Analyze the exception details. (후략)
패치 생성	<Goal> Your task is to generate a corrected version of the provided <버기 함수> based on the <패치 전략>. </Goal> <버기 함수> <패치 전략> <예외 정보> Message: <예외 메시지> (후략)

4.4. 패치 후보 생성

프로그램 실행 도중 예외가 발생하면, 4.2 절의 try-except 문을 통해 Axolotl 의 패치 후보 생성 단계로 이동한다.

그림 3 에서와 같이 패치 후보를 생성할 때에는 San2Patch [7]의 ToT 를 사용한다. 먼저 예외의 근본적인 원인을 파악하고, 해당 함수에서 어디를 고칠지를 찾은 후 패치 전략을 얻은 다음 실제 패치 후보들을 생성한다. 각 과정에서는 ToT 기법을 적용하여 여러 응답을 받은 후 가장 좋은 응답을 선택한다. 표 1 은 실제 각 과정에서 사용된 프롬프트들을 보여주고 있다.

예외의 근본 원인 (root cause)을 파악할 때에는 먼저 예외 오류 메시지와 예외의 stack trace, 원래 함수의 소스 코드를 이용해 이 예외에 대한 설명과 근본 원인을 응답으로 받는다. 그 후 패치 위치를 찾을 때에는 앞에서 얻은 예외의 설명과 근본 원인, stack trace, 버그가 있는 함수의 소스 코드를 프롬프트로

주어 버기 함수 내에서 결함이 있는 위치를 찾는다. 그 다음 패치 전략을 얻을 때에는 앞에서 찾은 오류의 근본 원인과 버기 함수의 소스 코드, 결함 위치를 주고 패치 전략을 얻는다. 마지막으로 실제 패치를 생성할 때에는 위에서 얻은 모든 정보와 버기 함수를 주고 실제 패치 후보를 찾는다.

각 과정에서는 Tree-of-Thought(ToT) 전략을 사용하고 있다. 각 과정은 총 세 번씩 반복하여 세 개의 응답을 얻고, LLM 을 사용하여 각 응답마다 점수를 매겨 그 중 가장 높은 점수를 가진 응답만을 선택해 다음 단계로 진행한다.

패치 후보 생성 후에는 저장된 체크포인트를 불러와 실제로 적용해서 발생하던 예외가 발생하지 않는지 확인한다. 예외가 발생하지 않으면 그럴듯한 패치로 분류하여 수집한다.

예외가 발생하면 다시 전략 생성 단계로 돌아가는데, 이 때에는 피드백 기법을 사용한다. 기존에 실패했던 패치를 추가로 프롬프트에 제공하여 새로운 전략을 생성한다. 그 후 패치 후보를 새로 생성할 때에도 실패했던 패치를 추가로 제공하여 새로운 패치 후보를 얻는다. 이 과정을 그럴듯한(plausible) 패치를 생성할 때까지 최대 두 번 반복한다.

표 2: 퍼징에 사용된 변이 규칙

객체 타입	규칙
Enum	가능한 값 중 랜덤 선택
bool	부정(negate)
int	비트 뒤집기, 값 증가/감소, 특정 값으로 설정
float	비트 뒤집기
str	랜덤 문자 삽입/삭제, 비트 뒤집기
bytes	랜덤 바이트 삽입/삭제, 비트 뒤집기, 블록 복사/삭제
파일 경로	디렉토리 단위로 분할 후 str 규칙 적용
정규식	str 규칙 적용 후 패턴 유효성 확인
클래스 객체	필드를 따라가 재귀적으로 위 규칙 적용

4.5. 함수 단위 퍼징

그럴듯한 패치들을 생성한 후에는 각 패치들이 프로그램 내에서 부작용(side-effect)을 일으키지 않는지 확인하기 위해 함수 단위 black-box 퍼징 기법을 사용한다. 먼저 기존 프로그램 실행 중에 저장해 두었던 체크포인트를 불러온 후, 패치를 적용한다. 그 후에는 계속 해당 함수의 인자들을 다양하게 생성하여 버그가 있는 원래 함수와 패치된 함수를 각각 실행한다. 원래 함수에서 예외가 발생하지 않은 경우 패치된 함수에서의 실행 결과를 확인하는데, 패치된 함수에서 예외가 발생하면 잘못된 패치로 분류하여 제거한다. 각 패치마다 이 과정을 정해진 시간 동안 반복하여 잘못된 패치가 아니라면 올바른 패치로 분류하여 원래 프로그램에 적용한 후 프로그램을 이어서 실행한다.

LLM 으로부터 패치된 함수를 받으면 먼저 함수에 패치를 적용한다. 패치를 적용할 때에는 Python 내에 기본으로 정의된 compile()함수를 이용하여 bytecode 로 변환한 후 함수의 코드에 변환된 bytecode 를 적용한다.

퍼징을 진행할 때에는 원래 함수의 인자들을 변이(mutate)하여 실행하는데, AFL [30]에서 사용하는 변이 규칙들을 참고하였다. 표 2 가 각 Python 기본형들에 적용된 변이 규칙들을 보여주고 있는데, 인자가 기본형이 아닌 복잡한 클래스일 경우에는 클래스의 멤버 변수(field)를 따라가서 재귀적으로 변이한다. 입력을 하나 생성할 때마다 변이 규칙을 최대 10 번까지 랜덤하게 적용한다.

퍼징은 각 패치마다 10 분씩 진행하며, 매번 새로운 함수 인자를 생성한 후 함수를 실행한다. 그 결과 패치된 함수 안에서 원래 함수에서는 발생하지 않던 새로운 예외가 발생하면 잘못된 패치로 분류한다. 이 때에는 다시 패치 후보 생성 단계로 돌아가는데, 이 때 패치 후보를 생성할 때에는 LLM 에 잘못된 패치 후보를 피드백으로 제공하여 기존 패치와 다른 패치 후보를 생성한다.

5. 구현

Axolotl의 전체 구조는 4055 라인의 Python으로 구현하였으며, 실행중인 프로세스의 체크포인트를 저장하고 불러오기 위해 CRIU [8]를 사용하였다. Python 프로그램에서 예외를 탐지할 수 있도록 bytecode [26] Python 라이브러리를 사용하여 Python의 bytecode 수준에서 함수 전체에 try-except문을 삽입했다. 패치 후보를 생성하기 위해 OpenAI의 GPT-5.2 [31] 모델을 사용하였으나, 다양한 LLM 모델을 간단하게 추가할 수 있다. 패치 검증을 위한 퍼징은 자체적으로 구현하였으나 함수 인자들의 변이 규칙은 AFL [30]을 참고하였다 (4.5절 참고).

6. 실험

6.1. 환경 및 벤치마크

본 연구의 실험에서는 Python을 대상으로 하는 기존 연구 [27-29]에서 흔히 사용되는 BugsInPy [25] 벤치마크에서 프로그램이 직접 예외를 일으키는 버그들 중 재현이 불가능한 버그들을 제외하여 총 25개(표 4)를 선정하였다. 실험은 Ubuntu 22.04.5 및 Anaconda v24.11.3에서 진행하였다.

표 3: Axolotl의 실험 결과. 각 열에서 O는 패치성공을, X는 패치 실패를 의미한다.

	GPT-5.2				Qwen-3-Next			
	L1	L2	L3	실행 시간 (초)	L1	L2	L3	실행 시간 (초)
black-14	O	O	O	679.1	O	O	O	650.73
black-16	O	O	X	714.5	O	O	X	673.79
black-17	O	O	O	668.13	O	O	X	645.73
pandas-49	O	O	O	699.48	X	X	X	109.92
pandas-77	O	O	O	842.0	O	O	O	668.49
pandas-99	X	X	X	325.17	X	X	X	182.92
pandas-102	O	O	X	698.49	X	X	X	101.86
pandas-117	O	O	X	700.17	X	X	X	111.63
pandas-142	O	O	O	709.16	O	O	X	680.81
pandas-146	O	O	O	713.99	X	X	X	153.01
pandas-150	O	O	O	716.2	O	O	O	683.77
pandas-160	O	O	X	742.17	O	O	X	664.85
pandas-168	O	O	O	865.48	X	X	X	288.35
scrapy-15	O	O	O	698.94	O	O	O	676.81
scrapy-17	O	O	O	670.87	O	O	O	647.84
scrapy-29	O	O	O	1338.78	O	O	O	649.81
tornado-9	O	O	O	673.78	O	O	O	656.74
youtube-dl-5	O	O	X	730.1	X	X	X	88.97
youtube-dl-11	O	O	O	691.89	X	X	X	62.21
youtube-dl-16	X	X	X	276.28	X	X	X	157.07
youtube-dl-17	O	O	O	702.73	O	O	X	684.41
youtube-dl-22	O	O	X	730.73	X	X	X	161.4
youtube-dl-28	O	O	O	671.54	O	O	O	652.21
youtube-dl-33	O	O	O	695.02	O	O	X	656.13
youtube-dl-37	O	O	O	668.48	X	X	X	94.01
총합	23/25	23/25	17/25		14/25	14/25	8/25	
성공률	92	92	68	678.19	56	56	32	428.38

LLM 은 상용 모델로 GPT-5.2 [31]를 사용하였으며 오픈소스 모델로 Qwen-3-Next [40]를 사용하였다.

표 4: BugsInPy 벤치마크에서 실험에 사용한 버그

프로그램	버그 개수
black	3
pandas	10
scrapy	3
tornado	1
youtube-dl	8
총합	25

6.2. 연구 질문 (Research question, RQ)

본 연구에서는 Axolotl 을 이용하여 세 가지 연구 질문(RQ)에 답하기 위한 실험을 진행하였다.

- RQ 1: Axolotl 의 효과: Axolotl 이 얼마나 많은 예외들을 수정할 수 있는가?
- RQ 2: Axolotl 의 효율성: 패치를 생성하고 수정하는 데 얼마나 오래 걸리는가?
- RQ 3: Ablation Study: Axolotl 에서 패치 생성에 사용한 세 가지 기법 (런타임 정보, ToT, 피드백)이 패치의 품질에 각각 얼마나 영향을 주는가?

RQ 1 을 위해 Axolotl 을 BugsInPy 벤치마크에 적용하여 25 개의 버그 중 얼마나 많이 수정할 수 있는지 실험하였다. RQ 2 에서는 25 개의 버그들 중 패치에 성공한 버그들에서 평균적으로 어느 정도의 시간이 걸리는지를 측정하였다. RQ 3 에서는

Axolotl 에서 패치 생성에 사용한 런타임 정보와 ToT 기법, 피드백 기법이 각각 패치 생성에 얼마나 기여하는지를 측정하기 위해 각 기법을 사용하지 않고 패치를 생성하여 결과를 비교하였다.

ToT 의 각 단계에서는 3 개의 응답을 생성하여 그 중 가장 높은 점수를 가지는 응답을 선택하였다. 패치 검증에 실패하면 최대 2 번 패치 생성으로 돌아가 피드백을 제공하며, 그 후에도 실패하면 패치 실패로 간주하였다.

6.2. 결과

```
  if src_txt[-1] != "\n":
     src_txt += "\n"
```

그림 4-1: 버그 코드 블럭

```
- if src_txt[-1] != "\n":
+ if src_txt[-1:] != "\n":
     src_txt += "\n"
```

그림 4-2: 개발자 패치

```
- if src_txt[-1] != "\n":
+ if len(src_txt) > 0 and src_txt[-1] != "\n":
     src_txt += "\n"
```

그림 4-3: Qwen-3-Next 가 생성한 패치

```
- if src_txt[-1] != "\n":
+ if not src_txt.endswith("\n"):
     src_txt += "\n"
```

그림 4-4: GPT-5.2 가 생성한 패치

그림 4: black-17

6.2.1. RQ 1: Axolotl 의 효과

표 3 은 Axolotl 의 실험 결과를 보여준다. 표의 왼쪽은 GPT-5.2 의 결과이고 오른쪽은 Qwen-3-Next 의 결과이다. 표에는 3 개의 결과가 있는데, L1 은 목표 예외를 회피하는 데에 성공한 경우이고 L2 는 퍼징의 결과 패치가 함수에 부작용을 만들어내지 않는 경우, L3는 개발자 패치의 의도와 일치한 패치일 경우 결과를 보여준다. 예를 들어, GPT-5.2 에서 pandas-117 에서는 목표 예외를 성공적으로 회피하면서 해당 함수에 새로운 부작용을 만들어내지 않지만 개발자 패치와 비교했을 때 함수의 동작이 다르다. L3 의 경우, 본 연구의 저자들이 직접 생성된 패치를 확인하여 개발자의 패치와 의미적으로 동등한지 판별하였다.

표 3 의 L1 과 L2 에서 GPT-5.2 는 92%의 성공률을 보였고 Qwen-3-Next는 56%의 성공률을 보여 Qwen-3-Next 보다 GPT-5.2 가 더 좋은 성공률을 보였다. 또 L3 에서는 GPT-5.2 와 Qwen-3-Next 에서 각각 68%와 32%의 성공률을 보여 L1 과 L2 에 비해 낮은 성공률을 보였다.

L3에서 상대적으로 낮은 성공률을 보이는 주된 이유는 LLM 이 예외 회피를 최우선 목표로 패치를 생성하기 때문이다. 예를 들어, 그림 4 는 black-17 버그의 코드를 보여준다. 그림 4-1 의 버그 코드에서는 `src_txt` 의 길이가 0 일 경우 `src_txt[-1]`에서 `IndexError` 가 발생한다. 그림 4-2 의 개발자 패치는 `src_txt[-1]`을 `[-1:]`로 변경하여 문자열의 크기가 0 일 때에도 개행 문자(‘\n’)를 삽입한다. 하지만 그림 4-3 의 Qwen-3-Next 가 생성한 패치는 조건식에 `len > 0` 을 추가하여, 빈 문자열일 경우 개행 문자를 삽입하지 않으므로 개발자 패치와 동작이 달라지지만 `IndexError` 예외를 발생시키지는 않는다. 반면 그림 4-4 의 GPT-5.2 가 생성한 패치는 `endswith()` 메소드를 사용하여 빈 문자열일 경우에도 개행 문자열을 삽입한다. 따라서 black-17 버그에서는 Qwen-3-Next 의 경우 예외 회피에는 성공하였으나 프로그램의 동작이 개발자 패치와는 달랐다. 반면 GPT-5.2 는 개발자의 의도와 맞는 패치를 생성하였다.

Axolotl 은 런타임 에러를 회피하는 데에는(L1, L2) 좋은 성능을 보여주었으나, 개발자의 의도까지 파악하여 기능적 완전성을 갖춘 패치(L3)를 생성하는 데에는 한계를 보였다.

6.2.2. RQ 2: Axolotl 의 효율성

표 3 에서는 각 버그마다 실행 시간을 보여주고 있다. GPT-5.2 에서는 평균 678.19 초(약 11.3 분)가 소요되었고 Qwen-3-Next 에서는 평균 428.38 초(약 7.14 분)가 소요되어 Qwen-3-Next 가 더 빠르게 동작하는 것을 관찰하였다. GPT-5.2 는 상용 모델로 네트워크를 통해 프롬프트와 응답을 주고받아야 하므로 로컬에 설치하는 Qwen-3-Next 보다 네트워크 시간이 더 많이 필요하다.

Axolotl 의 실행 시간 대부분은 패치 검증을 위한 함수 단위 퍼징에 사용된다. 퍼징을 하기 전 패치를 생성하는 데에는 GPT-5.2 와 Qwen-3-Next 를 사용하였을 때 평균적으로 각각 126.19 초(약 2.1 분)와 92.38 초(약 1.5 분)가 걸려, 패치 생성은 매우 빠르게 이루어지는 것을 관찰하였다.

또 CRIU 를 이용하여 체크포인트를 생성하는 데에는 평균적으로 0.79 초의 오버헤드가 발생하여 원래 프로그램의 실행 속도를 크게 저해하지 않는다.

Axolotl 과 GPT-5.2 를 사용하였을 때 약 11.3 분이 소요되어, 프로그램 실행 도중 프로그램을 오랜 시간

동안 중지하지 않고 버그를 충분히 빠르게 수정할 수 있었다.

6.2.3. RQ 3: Ablation Study

Ablation Study 를 위해 앞에서 성공률이 더 높았던 GPT-5.2 를 사용해, Axolotl 에서 패치를 생성할 때 사용한 세 가지 요소들을 하나씩 제거하고 패치 생성을 시도하여 각 요소들이 Axolotl 의 성능에 미치는 영향을 관찰하였다. 6.2.1 절에서 L1 과 L2 는 성공률이 완전히 일치하여 Ablation Study 에서는 L2 결과를 제외하였다.

그림 5 는 각 요소를 제거했을 때의 성공률과 실행 시간을 보여준다. ToT 가 없는 경우에는 실행 시간이 가장 빨랐고 Axolotl 과 동일한 L1 성공률을 기록했지만, 버그의 원인과 패치 전략을 추론하는 과정이 없으므로 패치의 품질이 낮아져 L3 성공률은 낮아졌다. 피드백을 제공하지 않은 경우에는 LLM 이 기존에 실패했던 패치를 알지 못하므로, 패치 실패 후 다시 패치를 생성할 때에도 동일하거나 유사한 패치를 생성하였다. 동적 정보를 제공하지 않았을 때는 패치 위치를 식별하지 못해 가장 긴 실행 시간과 가장 낮은 패치 성공률을 보였다.

위 실험 결과는 예외를 효과적으로 회피하며 좋은 품질의 패치를 빠르게 생성하기 위해서 세 가지 요소가 모두 필요함을 보여주고 있다.

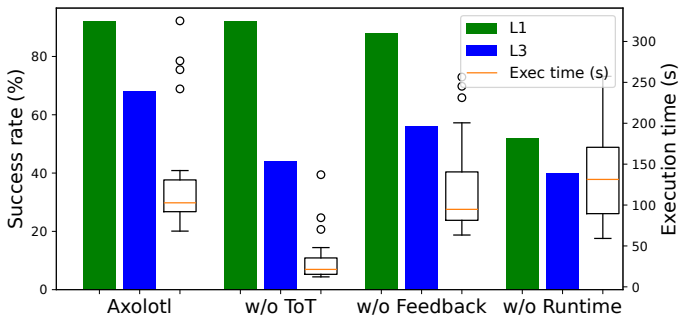

그림 5: Ablation Study 결과. w/o ToT, w/o Feedback, w/o Runtime은 각각 ToT, 피드백, 동적 정보를 사용하지 않았을 때의 결과이다. 초록색과 파란색 막대는 각각 L1과 L3 결과이고, 하얀색 상자는 총 실행 시간이다.

7. 토론 및 한계

본 연구에서 패치는 프로그램 실행 도중 발생한 예외를 회피하기 위한 목적으로 생성하였으며, 생성된 패치는 벤치마크에 포함된 개발자의 패치와 상이한 경우가 있다. 이는 위 실험의 RQ 1 의 결과 (6.2.1 절 참고)에서도 관찰할 수 있는데, L2 에서는 성공했으나 L3 에서는 실패한 경우 해당 함수에서는 부작용이 발생하지 않으나 실제 프로그램의 사용 시나리오에서는 다른 부작용이 발생할 수 있다. 이러한 경우에는 Axolotl 이 부작용을 새로운 예외로 감지하여 대응하는 또 다른 패치를 생성할 수 있다.

또 6.2.1 절의 실험 결과에서 L1 과 L2 의 패치 성공률은 동일한데, 이는 본 연구의 퍼징을 이용한 패치 검증 단계에서 함수의 유효한 인자를 충분히 만들지 못하기 때문이다. 패치가 새로운 예외를 일으키는지 확인하기 위해 먼저 새로 생성한 함수의 입력을 버기 함수에 실행하는데, 이 때 버기 함수가 예외를 일으키지 않아야 패치 검증에 사용할 수 있다. 그러나 예외를 일으키는 입력을 초기 시드로 제공해서 퍼징을 진행하면 예외를 일으키지 않는 입력을 생성하는 것이 어렵다.

또 본 연구에는 함수 전체를 try-except 문으로 감싼 후 프로그램을 실행할 때 해당 함수에서 발생한 예외가 이미 프로그램의 다른 부분에서 처리될 가능성이 있다는 한계가 있다. 즉, 발생한 예외가 의도한 것이고, 함수 밖에서 처리하도록 구현되어 있을 수 있다. 이 경우에는 간단한 정적 분석을 통해 이미 처리되어 있는 예외는 그대로 raise 하고 나머지 예외에만 Axolotl 를 적용할 수 있다.

마지막으로 본 연구에는 LLM 이 가지는 Data leakage 문제가 있다. BugsInPy 는 APR 연구에서 흔히 사용되는 벤치마크로 LLM 이 이미 버그들을 학습했을 가능성이 있으나, 이 문제는 LLM 기반의 많은 연구들 [6,7]도 공유하고 있다.

본 연구의 구현과 실험은 Python 으로 진행하였으나, 본 연구의 기법은 C, C++나 Java 등의 다른 언어에도 적용할 수 있다. 다만 생성된 패치를 실시간으로 프로그램에 적용하기 위해 복잡한 기술과 노력이 필요할 수 있다.

8. 관련 연구

8.1. 패치 생성

현재까지의 APR 연구에서는 여러 기법으로 많은 패치 후보들을 생성한 후, 각 패치들을 하나씩 적용하여 프로그램에 포함된 테스트 케이스들을 실행하여 모든 테스트들이 성공하는 패치들만을 출력하였다. Genprog [1], Arja [12], Arja-e [13]는 패치 후보를 생성하기 위해 유전 프로그래밍 (genetic programming) 기법을 사용한다. 버그가 있는 코드에서 시작해, 유전 프로그래밍을 이용해 프로그램을 조금씩 변형하여 패치를 생성한다. Avatar [14], Fixminer [15], TBar [2]는 템플릿을 이용해 특정 패턴의 패치 후보를 생성한다. Angelix [4], CPR [16], Prophet [17]도 템플릿을 사용하지만, 패치를 효과적으로 생성하기 위해 심볼릭 실행 등의 기법을 사용한다.

최근에는 AI 모델을 이용해 패치를 생성하는 기법들도 활발히 연구가 이루어지고 있다. Recoder [18], ARJANMT [19], SequenceR [20]은 전통적인 인공

신경망 (neural network) 모델을 사용하여 패치를 생성한다. AlphaRepair [5], AutoCodeRover [21], SRepair [6]은 LLM 모델을 사용하여 패치를 생성한다.

8.2. 패치 검증

기존의 일반적인 APR 연구들 [1-7,12-21,23,24]은 패치를 검증하기 위해 프로그램에 제공된 테스트 케이스들을 이용한다. NPEX [22]는 개발자들이 흔히 사용하는 패턴을 학습한 통계 모델을 이용해 패치를 검증한다. SPIDER [37]는 정적 분석을 활용해 AST 가 특정 패턴을 만족해야만 올바른 패치로 분류한다. UC-KLEE [38]는 일반적인 심볼릭 실행과 다른 Under-contrained 심볼릭 실행을 사용하여 초기 테스트 없이 검증을 진행할 수 있다. VulnFix [39]는 스냅샷 퍼징 (snapshot fuzzing)을 이용한다. 스냅샷 퍼징은 프로그램이나 함수의 입력을 변이하는 것이 아닌 프로그램의 상태 자체를 변이하며 퍼징을 진행한다.

9. 결론

본 연구에서는 프로그램 배포 후 실행 도중 크래쉬 버그가 발생한 경우에 프로그램을 중지하지 않고 일시정지한 후 프로그램을 수정하여 이어서 실행하는 기술을 연구하였고 Python 프로그램을 대상으로 하는 도구를 구현하여 Axolotl 로 명명하였다. 실험을 통해 GPT-5.2 를 활용한 경우 평균 11.3 분 안에 92%의 패치 성공률을 확인하였다.

[참고문헌]

[1] Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W., GenProg: A generic method for automatic software repair, IEEE Transactions on Software Engineering, 38, 1, 54-72, 2012.

[2] Liu, K., Koyuncu, A., Kim, D., & Bissyandé, T. F., TBar: Revisiting template-based automated program repair, Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 31-42, 2019.

[3] Yi Li, Shaohua Wang, and Tien N. Nguyen, DEAR: a novel deep learning based approach for automated program repair, Proceedings of the 44th International Conference on Software Engineering, 511-523, 2022.

[4] Mechtaev, S., Yi, J., & Roychoudhury, A., Angelix: Scalable multiline program patch synthesis via symbolic analysis, Proceedings of the 38th International Conference on Software Engineering, 691-701, 2016.

[5] Chunqiu Steven Xia and Lingming Zhang, Less Training, More Repairing Please: Revisiting Automated Program Repair Via Zero-Shot Learning, Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 959-971, 2022.

[6] Xiang, J., Xu, X., Kong, F., Wu, M., Zhang, Z., Zhang, H., & Zhang, Y., How far can we go with practical function-level program repair?, arXiv preprint arXiv:2404.12833, 2024.

[7] Kim, Y., Shin, S., Kim, H., & Yoon, J., Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis, 34th USENIX Security Symposium (USENIX Security 25), 4401-4419, 2025.

[8] CRIU Project, CRIU: Checkpoint/Restore In Userspace, https://criu.org, 2026.

[9] Merkel, D., Docker: lightweight linux containers for consistent development and deployment, Linux Journal, 239, 2, 2014.

[10] Nong, Y., Aldeen, M., Cheng, L., Hu, H., Chen, F., & Cai, H., Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities, arXiv preprint arXiv:2402.17230, 2024.

[11] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., ... & Zhou, D., Chain-of-thought prompting elicits reasoning in large language models, Advances in neural information processing systems, 35, 24824-24837, 2022.

[12] Yuan, Y., & Banzhaf, W., Arja: Automated repair of java programs via multi-objective genetic programming, IEEE Transactions on Software Engineering, 46, 10, 1040-1067, 2020.

[13] Yuan, Y., & Banzhaf, W., Toward better evolutionary program repair: an integrated approach, ACM Transactions on Software Engineering and Methodology, 29, 1, 5:1-5:53, 2020.

[14] Koyuncu, A., Liu, K., Bissyandé, T. F., Kim, D., Klein, J., Monperrus, M., & Le Traon, Y., FixMiner: Mining relevant fix patterns for automated program repair, Empirical Software Engineering, 25, 3, 1980-2024, 2020.

[15] Liu, K., Koyuncu, A., Kim, D., & Bissyandé, T. F., Avatar: Fixing semantic bugs with fix patterns of static analysis violations, Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, 456-467, 2019.

[16] Shariffdeen, R., Noller, Y., Grunske, L., & Roychoudhury, A., Concolic program repair, Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 390-405, 2021.

[17] Long, F., & Rinard, M., Automatic patch generation via learning from successful patches, Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 298-312, 2016.

[18] Zhu, Q., Sun, Z., Xiao, Y., Zhang, W., Yuan, K., Xiong, Y., & Zhang, L., A Syntax-Guided Edit Decoder for Neural

Program Repair, Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 341-353, 2021.
[19] Li, D., Wong, W. E., Jian, M., Geng, Y., & Chau, M., Improving search-based automatic program repair with Neural Machine Translation, IEEE Access, 10, 51167-51175, 2022.
[20] Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L. N., Poshyvanyk, D., & Monperrus, M., Sequencer: Sequence-to-sequence learning for end-to-end program repair, IEEE Transactions on Software Engineering, 47, 9, 1943-1959, 2021.
[21] Zhang, Y., Ruan, H., Fan, Z., & Roychoudhury, A., Autocoderover: Autonomous program improvement, Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 1592-1604, 2024.
[22] Lee, J., Hong, S., & Oh, H., Npex: Repairing java null pointer exceptions without tests, Proceedings of the 44th International Conference on Software Engineering, 1532-1544, 2022.
[23] Kim, Y., Han, S., Khamit, A. Y., & Yi, J., Automated Program Repair from Fuzzing Perspective, Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 854-866, 2023.
[24] Kim, Y., Park, Y., Han, S., & Yi, J., Enhancing the Efficiency of Automated Program Repair via Greybox Analysis, Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 1719-1731, 2024.
[25] Widyasari, R., Sim, S. Q., Lok, C., Qi, H., Phan, J., Tay, Q., ... & Ouh, E. L., BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies, Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 1556-1560, 2020.
[26] Dartiailh, M., bytecode: A Python module to generate and modify bytecode, https://github.com/MatthieuDartiailh/bytecode, 2026.
[27] Oh, W., & Oh, H., PyTER: effective program repair for Python type errors, Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 922-934, 2022.
[28] Cai, X., & Jiang, L., Adapting Knowledge Prompt Tuning for Enhanced Automated Program Repair, Proceedings of the 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering, 360-371, 2025.
[29] Fu, A., Xu, P., Li, J., Kuang, B., & Gao, Y., InstructRepair: Instruct Large Language Models with Rich Bug Information for Automated Program Repair, IEEE Transactions on Information Forensics and Security, 20, , 1113-1125, 2025.
[30] Zalewski, M., AFL, https://github.com/google/AFL, 2026.
[31] OpenAI, ChatGPT (GPT-5.2 version), https://chat.openai.com, 2026.
[32] Fioraldi, A., Maier, D., Eißfeldt, H., & Heuse, M., AFL++: Combining Incremental Steps of Fuzzing Research, Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT), 2020.
[33] Serebryany, K., LibFuzzer – a library for coverage-guided fuzz testing, https://llvm.org/docs/LibFuzzer.html, 2026.
[34] Kim, T. E., Choi, J., Heo, K., & Cha, S. K., DAFL: Directed Grey-box Fuzzing guided by Data Dependency, Proceedings of the 32nd USENIX Security Symposium, 4931-4948, 2023.
[35] Rong, H., You, W., Wang, X., & Mao, T., Toward Unbiased Multiple-Target Fuzzing with Path Diversity, Proceedings of the 33rd USENIX Security Symposium, 2475-2492, 2024.
[36] Bao, A., Zhao, W., Wang, Y., Cheng, Y., McCamant, S., & Yew, P. C., From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification, Proceedings of the 34th USENIX Security Symposium, 6977-6997, 2025.
[37] Machiry, A., Redini, N., Camellini, E., Kruegel, C., & Vigna, G., Spider: Enabling fast patch propagation in related software repositories, Proceedings of the 2020 IEEE Symposium on Security and Privacy, 1562-1579, 2020.
[38] Ramos, D. A., & Engler, D., Under-Constrained Symbolic Execution: Correctness Checking for Real Code, Proceedings of the 24th USENIX Security Symposium, 49-64, 2015.
[39] Zhang, Y., Gao, X., Duck, G. J., & Roychoudhury, A., Program vulnerability repair via inductive inference, Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 691-702, 2022.
[40] Qwen Team, Qwen3 Technical Report, arXiv, 2505.09388, 2025.
[41] Yin, X., Ni, C., Wang, S., Li, Z., Zeng, L., & Yang, X., Thinkrepair: Self-directed automated program repair, Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 1274-1286, 2024.