컴퓨터공학 종합 설계 과제 계획서

과제 계획서

□ 과제 목적 및 필요성

목표는 LLVM Pass를 활용하여 Fuzzing 도구인 AFL의 효율성을 향상시키는 것이다. 소스코드에서 조건 검사를 포함한 코드 블록에 대한 접근성을 높이는 방법에 중점을 둔다. 예를 들어, 다음과 basic block이 있을 때:

```
if (input == 0xcafebabe) {
    // vulnerable code
}
```

이 조건을 만족시키기 위해 x의 값을 무작위로 변이(mutation)하는 전통적인 Fuzzing 접근은 매우 비효율적이다. 이는 x가 32비트 정수일 때, 2^32 번의 시도가 필요할 수 있음을 의미한다. 이는 현실적으로 매우 많은 시간이 소요될 수 있으며, 특정 조건에 도달하지 못할 가능성도 크다.

이를 해결하기 위해, LLVM Pass를 사용하여 다음과 같이 코드를 변형할 수 있다:

```
if (((input >> 24) & 0xff) == 0xca) {
  // update coverage
  if (((input >> 16) & 0xff) == 0xfe) {
    // update coverage
    if (((input >> 8) & 0xff) == 0xba) {
      // update coverage
      if (((input >> 0) & 0xff) == 0xbe) {
        // vulnerable code
      }
    }
  }
}
```

이 변형을 통해, 각 바이트를 개별적으로 비교하게 된다. 이 접근은 한 바이트가 일치하면 퍼저에서 새로운 커버리지로 인식하여 해당 입력값을 기반으로 추가 변이를 시도함으로써, 전체 32비트 값을 한꺼번에 맞추려는 것보다 훨씬 더 효율적으로 조건을 만족시킬 수 있다. 이 경우, 최대 2^8 * 4 = 2^10 번의 시도만으로 조건을 충족시킬 수 있게 된다. 이외에도 다른 basic block들을 조사하여 최적화를 적용해보거나 배열 접근 시간을 단축해보는 등 다른 case들에 대해서도 추가적으로 적용해 볼 예정이다.

이 원리를 일반화하여, LLVM을 통한 컴파일 단계에서 이러한 변환을 자동으로 적용할 수 있는 Pass를 작성하는 것이 목표이다. 이를 통해 Fuzzing 과정에서 같은 코드 커버리지를 달성하는 시간을 크게 줄일 수 있으며, 이는 소프트웨어 보안 테스트의 효율성을 높이는 데 기여

할 것이다.

□ 과제 수행내용

설계 주제명

LLVM Pass를 활용한 Fuzzing 효율성 개선

팀명

CookIR

개발환경

1. 운영 체제: Ubuntu
2. 프로그래밍 언어: C++
3. 컴파일러 및 도구:
 - LLVM 11.0 이상
 - Clang: LLVM IR 생성을 위한 프론트엔드
 - CMake: LLVM project 빌드 자동화 툴
 - AFL: LLVM Pass를 적용한 바이너리 퍼징 툴
 - Git, Github: 버전 관리 및 협업

필수적인 사전 보유 역량

- 프로그래밍 역량: C++ 에 대한 숙련된 이해
- LLVM 구조 이해: LLVM IR, Pass 인프라스트럭처 등
- LLVM Pass 작성 방법: Function Pass, Module Pass 등 작성 및 적용 방법
- Fuzzing 기법 및 AFL 사용 경험
- 컴파일러 최적화 기법: 코드 최적화 원리와 기법 이해

결과물

 - LLVM Pass tutorial 문서
 - LLVM Pass Shared Library(.so)
 - 변환된 바이너리와 기존 바이너리 간의 Fuzzing 성능 비교 보고서

기능 요구사항

입력 데이터의 단계적 변형을 통한 탐색 효율성 향상

취약점 탐지 성능 개선

변환 후 바이너리에 대한 안정성(원본 바이너리와 동일한 기능 제공)

성능 요구사항

코드 경로 탐색 속도의 개선

데이터 요구사항

- 샘플 테스트 코드
- 입력 데이터 변형의 단계별 로그 생성 및 저장
- Fuzzing 과정에서 생성된 입력 데이터와 출력 결과 비교

성공 기준

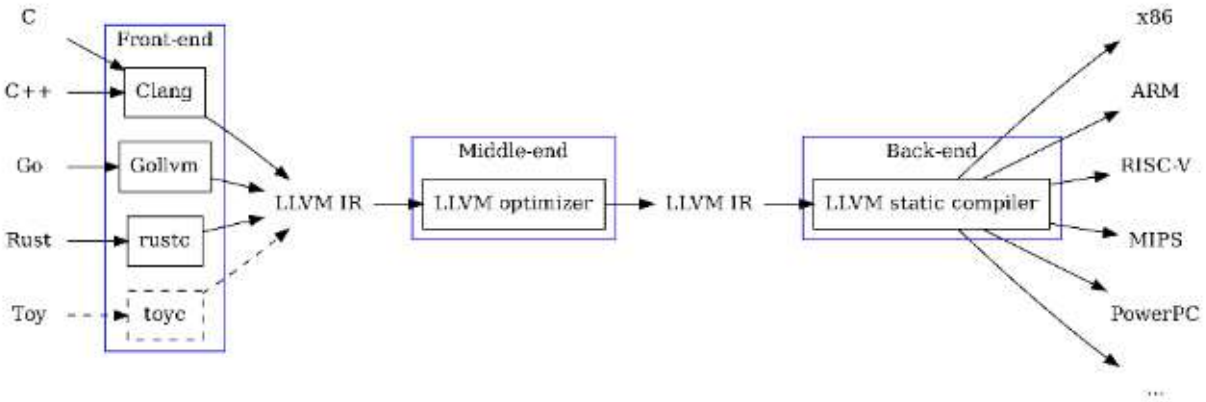
- 기술적 성과:
  - o LLVM Pass를 통해 자동화된 코드 변환이 성공적으로 이루어짐
  - o 변환된 코드로 퍼징 시 성능 향상이 객관적으로 입증됨
- 학습 및 성장:
  - o 팀원들의 LLVM 및 퍼징 기술에 대한 이해도 향상
  - o 컴파일러 최적화와 소프트웨어 보안 테스트에 대한 지식 습득
- 협업 및 문서화:
  - o 팀원 간 원활한 협업을 통해 프로젝트 진행
  - o 상세하고 이해하기 쉬운 문서 및 튜토리얼 작성

리스크 관리

- 기술적 어려움:
  - o LLVM Pass 작성의 난이도로 인한 개발 지연 가능성
  - o 퍼징 결과의 변동성으로 인한 성능 비교의 어려움
- 해결 방안:
  - o 초기 단계에서 LLVM Pass 개발에 충분한 시간 할애
  - o 다양한 테스트 케이스를 통해 결과의 신뢰성 확보
  - o 필요 시 전문가나 커뮤니티의 도움 요청

□ 기대효과 및 활용방안

기존의 무작위 변이 방식보다 더 빠르게 특정 조건을 만족하는 입력을 찾아 코드 커버리지를 빠르게 넓힐 수 있다. 따라서 전체적인 Fuzzing 성능이 향상되어 더 많은 보안 취약점을 발견할 수 있을 것으로 기대된다.



LLVM IR은 C/C++, Rust, Swift 등 다양한 프로그래밍 언어에서 생성될 수 있으며, 여러 아

키텍처로 변환될 수 있다. 따라서 작성된 LLVM Pass는 다양한 언어와 플랫폼에 적용 가능하여 폭넓은 활용성을 갖는는다.
소프트웨어 개발 주기의 테스트 과정에서 퍼징을 활용한다면, LLVM Pass를 적용하여 더욱 빠른 속도로 퍼징을 수행할 수 있다.
컴파일러 및 보안 분야의 학생과 연구자들에게 LLVM Pass 작성과 Fuzzing에 대한 인사이트를 제공할 수 있으며, 실습자료로 활용할 수 있다.
컴파일러 관련 기술을 통해 보안 분야의 문제를 해결함으로써 다양한 기술의 융합으로 문제를 해결하는 새로운 아이디어를 제공할 수 있다.