Concolic 테스팅과 Fuzzing을 결합한 유닛 테스팅 자동화 기술

김윤삼
브이플러스랩㈜
yunsam.kim@vpluslab.kr

김문주 ○
브이플러스랩㈜/KAIST
moonzoo.kim@vpluslab.kr

Automated Unit Testing Technique to Utilize Concolic Testing and Fuzzing Together

Yunsam Kim
VPlusLab Inc.
https://vpluslab.kr

Moonzoo Kim○
VPlusLab Inc./KAIST
https://vpluslab.kr

요 약

Concolic 테스팅 기술과 Fuzzing 기술은 높은 테스트 커버리지를 달성하는 테스트 입력값들을 자동으로 생성하기 때문에, 소프트웨어 오류 검출에 효과적이다. 두 기술은 서로 상이한 방식으로 동작하기 때문에 각각의 장단점이 다른 만큼, 두 기술을 함께 적용하여 테스팅 효과를 향상시킬 수 있다. 본 연구에서는 Concolic 테스팅과 Fuzzing을 조합한 새로운 기술을 개발하여, 실험을 통해 그 효과를 살펴보도록 한다.

1. 서 론

4차 산업 혁명의 중심인 소프트웨어의 복잡도가 날로 증가하고 있으며, 이로 인한 소프트웨어 오류로 인한 수많은 사건사고가 발생하고 있다 (예. 90명이 사망한 토요타 자동차 급발진 사건, 300여명이 사망한 보잉 737 MAX 추락 사건). 따라서, 소프트웨어 테스팅의 중요성이 날로 높아 지고 있으며, 소프트웨어 개발자가 수작업으로 진행하는 테스팅보다 테스트 커버리지가 높고, 예외상황 오류 검출 효과가 뛰어난 소프트웨어 자동 테스팅 기술에 대한 필요성이 부각되고 있다.

Concolic 테스팅과 Fuzzing 기술은 소프트웨어 자동 테스팅 기술 중 각광받고 있는 기술들이다.

- Concolic 테스팅은[1], 소스 코드 분석과 동적 테스팅, AI 기반의 SMT Solver를 동시에 이용하는 자동 테스팅 기법이다. 이 기술은, 소스 코드 분석 및 동적 테스팅에서 얻은 정보를 사용하여, 특정 코드 라인을 실행하기 위한 입력값의 조건을 구한 뒤, SMT solver로 조건을 만족하는 입력을 구하는 작업을, 모든 가능한 수행 경로를 다 실행할 때까지, 반복한다.
- Fuzzing 기술은[2], 타겟 소스 코드를 분석하지 않고, 테스트 입력값들을 랜덤하게 그리고 매우 빠르게 생성한다. 이 때, 해당 테스트 실행으로부터 얻어진 테스트 커버리지 정보를 바탕으로, 테스트 커버리지를 높일 수 있도록 휴리스틱을 적용하여 기존 테스트 입력값을 변이하여 새로운 테스트 입력값을 생성하는 과정을 주어진 시간동안 반복한다.

Concolic 테스팅 기술은 타겟 프로그램의 소스코드 로직을 정확하게 분석하기 때문에 높은 테스트 커버리지를 달성할 수 있는 장점이 있는 반면, 동적 실행에서 심볼릭 경로 조건식을 추출하고 추출된 복잡한 조건식을 SMT solver로 풀어내는 매우 복잡한 과정을 수없이 반복하기 때문에, 테스팅 속도가 느리다. Fuzzing은 타겟 프로그램의 소스코드 로직 분석을 하지 않고, 기본적으로 테스트 입력값을 byte 단위로 mutate해서 새로운 테스팅 입력값을 만들기 때문에, 테스팅 속도가 매우 빠른 장점이 있다. 하지만, 타겟 소스코드 로직 분석을 안하기 때문에, 높은 테스트 커버리지를 달성하지는 못하는 단점이 있다.

2. CONUZZ 프레임워크

본 논문에서는, 서로 다른 두 기술인 Concolic 테스팅과 Fuzzing의 장점을 합쳐 보다 높은 테스팅 효과를 얻을 수 있는 새로운 자동 유닛 테스팅 기술인 CONUZZ (CONolic + fUZZing)를 소개한다.

2.1 CONUZZ의 장점

CONUZZ는 Fuzzing의 장점인 빠르고 다양한 테스트 입력값 생성과 Concolic 테스팅의 장점인 정확한 SW 동적 분석 능력을 결합하여, 보다 효과적인 테스팅 효과 달성을 목표로 한다. 또한, Deep path를 탐색하는 능력이 제한적인 Fuzzing의 약점 및 Concolic Testing의 핵심이지만 속도가 느린 symbolic execution 엔진을 상호 보완하고자 한다.

2.2 CONUZZ 동작 방식

그림1에서 보듯이 CONUZZ는 먼저 concolic 테스팅의 depth first search (DFS) strategy를 적용하여 테스트 입력값들을 생성하고, 생성된 테스트 입력값들을 시드 테스트 풀로 사용하여 fuzzing을 적용하여 추가적으로 테스트 커버리지를 향상하는 방식으로 동작 한다.

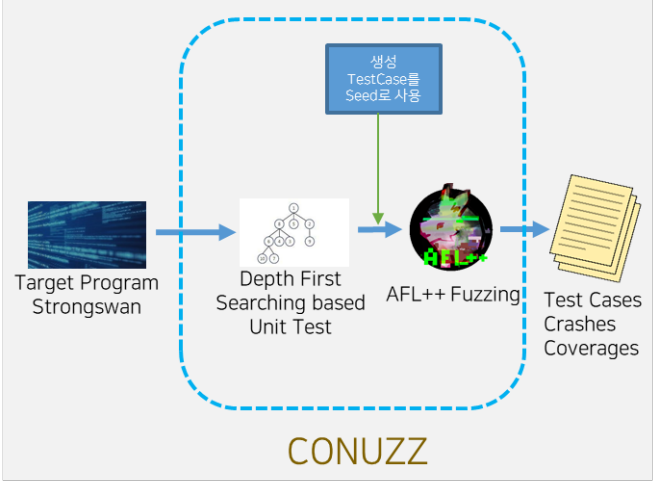

그림1. CONUZZ 프레임워크

이 때, concolic 테스팅이 생성한 첫번째 테스트 입력값이 crash인 경우, 해당 실행에 해당하는 심볼릭 경로 조건식 생성이 안되기 때문에 추가적인 테스트 입력값 생성이 불가능하다. 이런 경우, 그림 2에서 보이듯이, Fuzzing에서 처음 테스트 입력값들을 생성하고, 생성된 각각의 테스트 입력값을 concolic 테스팅의 초기 테스트 입력값으로 활용하여 다양한 테스트 입력값들을 생성한 후 함께 모아서 최종적으로 시드 테스트 풀로 활용하여 Fuzzing을 적용한다.

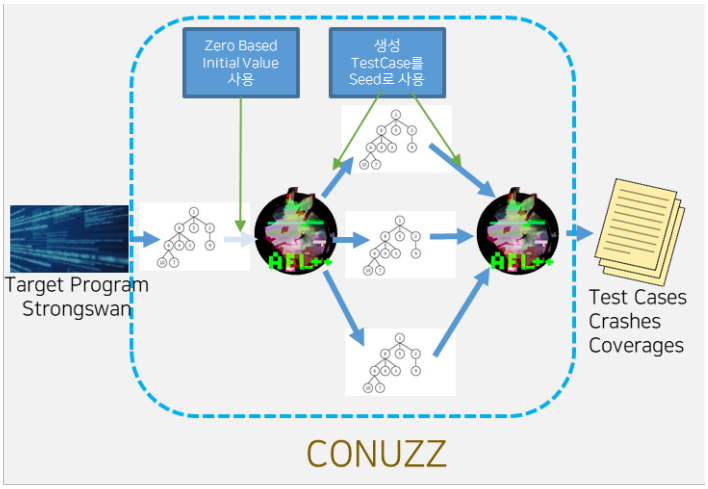

그림2. Concolic 테스팅의 첫번째 TC가 crash인 경우

CONUZZ는 concolic 테스팅 엔진 및 유닛 드라이버/스텁 생성 도구로 CROWN 2.0 [3]을 사용하며, AFL++[4]를 fuzzing 도구로 사용하도록 개발되었다.

3. CONUZZ 적용 실험 결과

본 연구에서는 함수 1427개/6만여줄로 구성된 보안관련 프로젝트인 S프로젝트에 CONUZZ를 적용한 결과를 분석하도록 한다. 본 실험은 AMD Ryzen 7 3800X (Ubuntu 18 64bit)에서 함수 하나당 Concolic 테스팅에 최대 60초, Fuzzing에 최대 10분을 적용하였다.

3.1 Crash 오류 검출 결과

CONUZZ 실험을 통해 인증서 관련 crash 오류 등 여러 신규 오류가 검출되었다. 예를 들어, 데이터를 포함한 자료구조 T가 해당 데이터의 크기를 별도의 field인 len로 관리하는 경우에, 악의적으로 len에 실제 데이터 크기 보다 큰 값을 저장하는 경우, buffer overflow 오류가 발생함을 발견하였다 (참고.CWE-131 Incorrect calculation of Buffer Size [5]).

3.2 CONUZZ vs. Concolic 테스팅

1427개의 타겟 함수 중 concolic 테스팅에서 첫번째 테스트 케이스가 crash하여 유의미한 테스트를 생성하지 못했으나, CONUZZ는 유의미한 테스트 케이스를 생성하는 경우가 57개 함수, concolic 테스팅에서는 crash가 발견되지 않았으나, CONUZZ는 crash 발생하는 테스트 입력값을 생성한 함수가 28개 함수, concolic 테스팅 보다 CONUZZ가 더 많은 테스트 입력값을 생성한 경우가 468개였다. 따라서, CONUZZ를 통한 테스팅 성능 향상을 확인할 수 있었다.

4. 결론 및 향후 연구

본 연구에서는 Concolic 테스팅과 Fuzzing을 결합한 CONUZZ를 개발하여 테스팅 효과를 향상시켰다. 실험을 통해, CONUZZ는 crash오류 검출에 효과적이며, concolic 테스팅 단독 사용에 비해 장점이 많은 것을 알 수 있었다. 향후에는 Concolic 테스팅과 Fuzzing을 좀 더 fine-granularity로 결합하여 테스팅 효과를 보다 향상시키는 연구를 수행할 계획이다.

참고 문헌

[1] Baldoni et al., A Survey of Symbolic Execution Techniques, ACM Computing Surveys, Volume 51, Issue 3, May 2019

[2] Manes et al., The art science and engineering of fuzzing : A survey. IEEE Transactions on Software Engineering, vol. 47, Nov. 2021,

[3] Automated Concolic Unit Testing Tool CROWN 2.0, https://www.vpluslab.kr/crown2

[4] AFL++ (American Fuzzy Lop plus plus), https://github.com/AFLplusplus/AFLplusplus

[5] CWE-131: Incorrect Calculation of Buffer Size, https://cwe.mitre.org/data/definitions/131.html