결함위치탐지 모델 학습을 위한 데이터셋 생성 방법: JsonCpp 사례연구

양희찬[O] 이아청 이인섭
KAIST KAIST LIGNex1
heechan.yang@kaist.ac.kr ahcheong.lee@kaist.ac.kr insub.lee@lignex1.com
남효주 정남훈 조규태 김문주
LIGNex1 LIGNex1 LIGNex1 KAIST/브이플러스랩
hyoju.nam@lignex1.com namhoon.jung@lignex1.com kyutae.cho2@lignex1.com moonzoo.kim@gmail.com

How to Create a Dataset to Train a ML Model for Fault Localization: A Case Study on JsonCpp

Heechan Yang[O] Ahcheong Lee Insup Lee
KAIST KAIST LIGNex1
Hyoju Nam Namhoon Jung Kyutae Cho Moonzoo Kim
LIGNex1 LIGNex1 LIGNex1 KAIST/V+Lab

요 약

SW 결함의 위치를 자동으로 검출하기 위해, 많은 연구자들이 대규모 데이터를 기반한 ML 모델을 학습하는 연구를 수행하고 있다. 따라서, 결함 위치 탐지 모델을 학습하기 위한 대규모 데이터의 중요성이 부각되고 있다. 본 연구는 JsonCpp를 대상으로, 165개의 결함 버전을 분석하여 고품질의 결함 위치 탐지 데이터셋을 구축했다. 이 과정에서 결함 버전의 부족, 테스트 케이스 부족, CCT(결함위치를 실행함에도 불구하고 성공한 테스트 케이스), 테스트 pass 및 fail 판별 오라클의 어려움, 그리고 MBFL 실험의 과도한 시간 소요와 같은 다양한 어려움들에 대한 해결책을 정리한다. 구축된 데이터셋은 SBFL과 MBFL 의심도 공식들에 적용시 매우 높은 정확도로 결함 위치를 탐지했으며 (Naish2 기준 acc@10 92.72%와 MUSE 기준 acc@10 100%), 결함 위치 탐지 모델 학습에 유용하게 사용되리라 기대한다.

1. 서론

SW 품질을 향상하기 위한 테스팅 및 디버깅 과정에서 가장 많은 시간이 소요되는 부분은, 결함 위치 탐지(FL: Fault Localization) 과정이다. 이를 자동화한 두 가지 주요 FL 방법으로 스펙트럼 기반 결함 위치탐지(SBFL: Spectrum-Based Fault Localization)와 변이 기반 결함 위치 탐지(MBFL: Mutation-Based Fault Localization)가 있으며, SBFL와 MBFL의 방법론을 결합하여 소프트웨어의 결함 위치를 효과적으로 탐지하는 연구가 활발히 진행되었다[1, 2, 3]. 이는 SBFL와 MBFL 기법으로부터 추출할 수 있는 동적 특징 데이터로부터 모델을 학습하여 결함의 위치를 탐지했다.

데이터 기반 FL 모델을 활용하여 FL 정확도를 높이기 위해서는 품질이 뛰어난 학습 데이터셋이 필수적이다. 본 논문에서는 JsonCpp 오픈소스 프로젝트의 165개 결함 버전을 분석하여 고품질의 FL 데이터셋을 구축하는 과정에서 발생한 다양한 어려움들과 해결책을 소개한다. 또한, 구축된 데이터셋은 SBFL의 의심도 부여 공식인 Naish2와 Ochiai에서 acc@10을 동일하게 92.72% 정확도를 보여주고 있으며 MBFL의 의심도 부여 공식인 Metallaxis 와 MUSE 기준 acc@10가 100%를 달성하여 데이터셋의 우수한 품질을 검증한다.

2. 결함위치탐지 기술 요약 설명

FL은 결함의 위치를 실행하는 테스트 케이스(TC)가 적어도 1개 이상 있다는 가정이 있으며, 해당 결함 구문을 실행하는 서로 다른 커버리지의 TC가 많을 수록, 해당 결함 위치를 추정하는 정확도가 높아진다. 이는, 각 구문을 실행한 TC의 실행 결과 정보가 FL에 유용한 정보로 사용되기 때문이다. SBFL과 MBFL 기법의 설명으로 이에 대해 더 자세히 알아볼 수 있다.

SBFL 방법론에서는, 소스코드 각 라인에 대한 TC들의 커버리지 정보를 활용하여 프로그램의 각 구문에 대한 의심도 점수를 계산한다. 각 구문에 대해, 구문을 실행하고 pass하는 TC의 개수(ep), 실행하고 fail하는 TC의 개수(ef), 실행하지 않고 pass하는 TC의 개수(np)와 실행하지 않고 fail하는 TC의 개수(nf)를 측정하고, 해당 정보로부터 Ochiai와 Naish2 등으로 잘 알려진 결함 위치 의심도 공식을 계산한다[4]. SBFL 공식이 부여하는 의심도 점수는 해당 구문이 fail한 TC에서 자주 등장하고 pass한 TC에서는 드물게 나타날 때 높게 책정된다. 이를 통해 결함이 의심되는 코드 위치를 식별할 수 있다.

MBFL 방법론은 의도적으로 프로그램 코드에 변형을 만들어, 이 변형된 코드 버전을 TC들과 함께 실행하여 각 코드 구문의 의심도를 평가한다[6]. 이 방식은 각 변형이 테스트 결과에 미치는 영향을 분석하여 결함이 있는 코드를 식별한다. 구체적으로, 어떤 코드 구문의 변형에서 기존에 fail한 TC중 pass로 바뀌는 TC의 개수(f2p)가 많으면, 해당 구문에 결함이 있을 가능성이 높다고 판단한다. 이는 결함이 있는 구문을 올바르게 수정했을 때 TC가 성공적으로 수행될 가능성이 높기 때문이다. 반면, 코드 구문의 변형에서 기존에 pass한 TC중 fail로 바뀌는 TC의 개수(p2f)가 많으면, 해당 구문이 결함일 가능성이 낮다고 판단한다. 이는 올바른 코드를 잘못 변경했을 때 추가적인 오류가 발생하여 TC가 fail할 가능성이 높기 때문입니다. 해당 기법으로 변이 기반 동적 특징으로부터 결함 구문의 의심도를 높이고 결함이 아닌 구문들의 의심도를 낮춰 더욱 정밀하게 결함 위치 탐지할 수 있다.

이 논문은 산학연주관 핵심 SW(응용연구) 연구개발 과제(계약번호 UC210018AD) 지원을 받아 수행된 연구임.

key	ep	ef	np	nf	fail_TCs	#max_mutant	m1:f2p	m1:p2f	m2:f2p	m2:p2f	…	…	m12:f2p	m12:p2f	bug
line1	4	0	0	6	6	12	-1	-1	0	2	…	…	0	2	0
line2	1	6	3	0	6	12	6	0	3	1	…	…	-1	-1	1
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
lineN	4	2	0	4	6	12	0	0	0	4	…	…	-1	-1	0

[표 1] 최종 FL 데이터셋 구성 요약 예시

3. 고품질 결함 위치 탐지 구축의 어려움

3.1 결함 위치 데이터셋의 구성 요소

FL 데이터셋의 구성으로는 각 구문 별 구문을 식별 값(key), 구문의 4가지의 스펙트럼 특징(ep, ef, np, nf)과 MBFL을 수행하기 위해 각 구문에 생성된 변이들의 변화된 TC 결과 (m1:p2f, m1:f2p, … mN:f2p, mN:p2f), 그리고 각 구문의 결함 여부를 식별해주는 값(bug)이 있다. 추가로 MBFL의 Metallaxis와 MUSE 의심도 부여 공식에 필요한 결함 버전의 총 fail하는 TC들의 개수(fail_TCs)가 있고, MBFL 기법에서는 각 구문 별 최대 변이 개수를 설정할 수 있기 때문에 해당 데이터셋에 설정된 구문 별 최대 변이 개수(#max_mutant)를 나타내는 열도 포함된다.

최종 FL 데이터셋의 구성 요소는 표 1에서 참고할 수 있다. 표 1의 FL 데이터셋 예시는 총 32개의 열을 포함한다. 각 구문 별 변이들의 변화된 TC 결과를 표현 할 때 <n>번째의 변이 경우 m<n>으로 표현되며 변이의 f2p와 p2f는 m<n>:f2p, m<n>:p2f로 표현한다. 또한, `#max_mutant`가 12개로 설정되어있기 때문에 각 구문의 변이 별 변화된 TC들의 결과는, 12개의 변이들의 f2p 12개와 와 p2f 12개로 한정된다. 이때 해당 구문의 생성된 변이 개수가 최대 변이 개수 보다 적게 생성될 때 나머지 변이의 f2p와 p2f 값은 `-1`을 주고, 컴파일 오류를 유발하는 변이는 TC들을 실행할 수 없기에 f2p와 p2f 값을 `-1` 값을 준다. 이와 같은 구성으로 SBFL와 MBFL 의심도 공식에 적용하여 각 구문 별 의심도 점수를 계산할 수 있다.

3.2 버그 버전의 부족으로 인한 어려움

결함의 위치가 명확히 알려진 결함 버전 (buggy version)을 수집하는 일이 FL 데이터셋 구축의 첫번째 어려움이다. 결함 버전 부족의 어려움의 해결 방안으로는 Oss-Fuzz[7]와 BugOSS[8]처럼 결함 버전들이 모인 benchmark으로부터 결함 버전을 수집할 수 있으나, 충분한 학습 데이터를 구축하기에는 부족할 수 있다.

결함 버전 부족으로 인해 모델 학습의 제한을 극복하기 위해, 변이 생성 도구 활용으로 인공적으로 결함을 생성하여 결함 버전의 수를 증가시키는 방법을 제안한다. 인공 결함이 실제 결함과 유사한 관계가 있기에[9] 인공적으로 생성한 결함들이 버그 버전 부족의 문제를 해결할 수 있다.

3.3 낮은 테스트 커버리지의 문제점

보유중인 TC들의 분기 커버리지가 낮은 경우 FL 성능은 저하된다. 이는, 결함위치를 실행하는 TC 개수가 부족하여, 실제 결함 구문과 다른 구문들과의 실행결과 분별력이 감소하기 때문이다.

이 문제로 인해 저하되는 데이터셋의 품질을 높이기 위해서는 높은 커버리지를 도달하는 TC들이 필요하다. 높은 커버리지를 도달하는 TC들을 확보하기 위해 fuzzing 기법 혹은 concolic 기법을 활용하여 더 많은 커버리지를 도달하는 입력 값을 생성하고, 이를 기반으로 TC를 추가하여 데이터셋의 품질을 향상시킬 수 있다.

3.4 Coincidentally Correct Test case (CCT)

CCT는 결함을 실행했음에도 불구하고, pass하는 테스트 케이스를 의미한다. CCT는 FL 데이터셋의 품질을 저하시키는 주요 요인 중 하나이다. 이는, SBFL에 경우 특정 구문을 실행하고 pass하는 TC가 많을수록 해당 구문에 낮은 의심도 점수를 부여하기 때문이다. 또한, MBFL에서는 p2f이 많을수록 의심도를 낮게 평가하기에 결함 구문을 실행하나 pass하는 TC가 많을 경우 p2f의 값이 높게 나올 확률이 높아지며 최종 FL 성능을 저하시킨다. 따라서 CCT를 제외하는 것은 FL 데이터셋의 품질을 향상시키는데 중요한 요소로 판단된다.

3.5 Pass 및 Fail 판별 오라클 정의의 어려움

결함 위치 탐지는 대상 프로그램을 실행한 테스트 케이스의 pass/fail 결과를 기반으로 수행된다. 따라서, TC의 pass 및 fail 여부를 판별하는 테스트 오라클이 필수적이나, 테스트 케이스의 결과를 판별하는 오라클을 정의하는 것은 어려운 과제이다. 따라서, 본 논문에서는 FL 데이터셋 구축 시 명확하게 pass 및 fail 판별하는 오라클을 보유하고 있는 프로그램을 선택하는 것이 중요한 요소로 판단한다.

3.6 MBFL의 실행 시간 관리

MBFL에서는 많은 양의 변이를 생성하여 각각의 TC를 생성된 각 변이에 실행하는 과정을 거친다. 이 과정은 프로그램의 컴파일 및 실행 시간으로 인해 시간이 많이 소요된다. 이러한 시간 소모를 줄이기 위해 본 논문에서는 결함 버전에서 fail하는 TC들이 실행한 구문에만 변이를 생성하는 것을 제안한다. 또한, 각 구문마다 생성되는 변이의 수를 N개로 제한하여 총 생성 변이 개수를 줄여 TC의 실행 시간을 대폭 줄일 수 있게 된다.

4. JsonCpp의 FL 데이터셋

4.1 JsonCpp의 버그 버전 관련

본 논문은 LIGNex1에서 공식적으로 사용하고 있는 오픈소스 프로젝트인 JsonCpp를 선택하여 FL 모델 학습을 위한 데이터셋을 구축한다. JsonCpp에 대해 총 165개의 결함 버전을 제작하여 FL 데이터셋을 구축했다. 그중 3개는 실제 결함이며 (1개 결함은 Oss-Fuzz[7] benchmark에서 assertion failure를 일으키는 결함 버전을 수집했으며, 2개는 JsonCpp의 github[10] issue에서 heap overflow와 integer overflow를 일으키는 결함 버전을 수집), C/C++ 프로그램을 대상으로 하는 변이 생성 도구, MUSIC[11]의 후속버전인 MUSIC++를 활용하여 약 20,000개의 변이를 생성하고 그 중 1개 이상의 fail하는 TC를 보유하는 변이들로 추려, 무작위로 162개를 선택했다.

4.2 JsonCpp의 TC 관련 (CCT와 오라클 포함)

JsonCpp는 pass와 fail을 명확히 판별하는 TC 127개를 보유하고 있다. 그림 1는 오라클 정의가 된 JsonCpp의 단일 TC 예를 보인다. 그림 1의 단일 TC에서는 단일 함수를 실행하여 명확한 assertion을 통해 pass와 fail을 판별한다.

127개의 TC들의 누적 라인 커버리지는 90.4%를 도달하고 분기 커버리지는 47.5%를 도달한다. 충분한 라인 커버리지는 각 구문의 분별력을 주며 낮은 분기 커버리지에도 좋은 FL 성능을 보인다.

또한, 165개의 결함 버전에서 fail하는 TC 개수는 평균 8.27개이고, CCT의 평균 개수는 4.29개이다. 데이터셋에서 동적 특징을 추출할 때, 각 결함 버전의 CCT는 제외하고 추출한다.

```
JSONTEST_FIXTURE_LOCAL(CharReaderTest, parseWithNoErrors) {
  Json::CharReaderBuilder b;
  CharReaderPtr reader(b.newCharReader());
  Json::String errs;
  Json::Value root;
  char const doc[] = R"({ "property" : "value" })";
  bool ok = reader->parse(doc, doc + std::strlen(doc), &root, &errs
    );
  JSONTEST_ASSERT(ok);
  JSONTEST_ASSERT(errs.empty());
}
```

[그림 1] JsonCpp 단일 TC 예시

4.3 MBFL 동적 특징 데이터셋 추출 실험

MBFL 동적 특징 추출에 소요되는 시간을 줄이고 고품질의 FL 데이터셋을 구축하기 위해서 gcovr[12]과 llvm-cov[13]을 활용해서 fail하는 TC들이 실행하는 구문을 먼저 파악한 후 해당 구문들에 대해서만 변이를 생성한다. 또한, 제한된 시간에 데이터셋을 구축하기 위해서 본 실험에서는 각 구문 별 최대 변이 생성 개수를 12개로 제한했다. 그 결과, 165개의 결함 버전에서 fail하는 TC들이 실행하는 구문의 개수는 평균 1,138.92개, 최대 2,822개로 측정되며 MUSIC++를 사용해서 생성된 결함 버전당 변이의 개수는 평균 4,109.21개, 최대 10,576개가 생성되었다. MBFL 동적 특징 데이터를 추출하기 위해서는 각 결함 버전에서 생성된 변이들에 대해 CCT를 제외한 TC들을 모두 실행하여 결과를 추출하는 과정을 거친다.

165개의 JsonCpp 결함 버전에서 생성된 변이들의 총 개수는 678,019개이며, 각 변이들을 빌드하고 TC들을 실행하는 과정의 시간 소요가 크다. 그로 인해, MBFL 동적 특징 추출 과정은 Ubuntu 18.04.6 OS 환경에 AMD Ryzen 7 3800XT 8-core cpu와 32GB 메모리를 장착한 머신 25대에서 수행했다. 그 결과, 각 165개의 결함 버전에 대한 MBFL 동적 특징 추출 실험을 병렬적으로 실행하여 약 6시간 소요되었다.

4.4 데이터셋 성능 평가

JsonCpp의 165개의 결함 버전으로부터 구축한 동적 특징 데이터셋의 최종 모습은 표 1와 같으며 데이터셋을 평가하기 위해 실제 FL 성능을 확인한다. 각 구문에 대한 동적 특징 정보를 SBFL의 GP13, Naish2, Ochiai 의심도 공식에 적용하고 MBFL의 의심도 공식인 Metallaxis와 MUSE에 적용하여 구문 별 의심도 점수를 계산했다. 각 함수에 속한 구문 중 가장 높은 의심도 점수를 함수의 의심도 점수로 선택하여 함수 단위 FL 성능을 평가했다. 총 363개 함수로 측정됐고 165개의 결함 버전 중 실제 결함이 위치한 함수를 상위 5위 안에 드는 결함 버전의 개수(acc@5)와 상위 10위 안에 드는 결함 버전의 개수(acc@10)로 정확도를 측정했다. 평가의 결과는 표 2에서 확인할 수 있다.

표 2에서 확인할 수 있듯이, 본 논문에서 5 가지의 어려움을 극복하여 결함 위치 탐지 데이터셋의 품질을 향상시킬 수 있음을 검증한다.

	FL 공식	acc@5	acc@5 백분율	acc@10	acc@10 백분율
SBFL	GP13	136	82.42%	153	92.72%
	Naish2	136	82.42%	153	92.72%
	Ochiai	136	82.42%	153	92.72%
MBFL	Metallaxis	160	96.96%	165	100%
	Muse	159	96.36%	165	100%

[표 2] JsonCpp FL 데이터셋 성능 평가

5. 결론

본 연구는 FL 데이터셋의 품질을 향상시키기 위해 다섯 가지 주요 고려 사항을 정리했다 (결함 버전의 부족, TC 부족, CCT, 테스트 pass 및 fail 판별 오라클의 어려움, 그리고 MBFL 실험의 과도한 시간 소요). 본 연구를 활용하여, 결함위치탐지 분야의 발전에 도움이 되리라 기대한다.

6. 참조 문헌

[1] Y. Kim, S. Mun, S. Yoo, et al. “Precise learn-to-rank fault localization using dynamic and static features of target programs”. ACM TOSEM 2019.
[2] X. Li, W. Li, Y. Zhang, et al. “DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization”. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019.
[3] Y. Song, X. Xie, and B. Xu. “When debugging encounters artificial intelligence: state of the art and open challenges”. Sci. China Inf. Sci 2023.
[4] S. Yoo, X. Xie, F. Kuo, T. Chen, and M. Harman. “Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis”. ACM TOSEM 2017.
[5] M. Papadakis, Y. Traon. “Metallaxis-FL: mutation-based fault localization”. Software Testing, Verification and Reliability, 2015.
[6] S. Moon, Y. Kim, M. Kim, and S. Yoo, “Ask the mutants: Mutating faulty programs for fault localization,”. ICST 2014.
[7] OSS-Fuzz. https://google.github.io/oss-fuzz/
[8] J. Kim, and S. Hong. “BugOSS: A Regression Bug Benchmark for Evaluating Fuzzing Techniques”. ICST 2023.
[9] M. Ojdanic, A. Garg, A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon. “Syntactic Vs. Semantic similarity of Artificial and Real Faults in Mutation Testing Studies”. 2021.
[10] JsonCpp Github. https://github.com/open-source-parsers/jsoncpp
[11] D. L. Phan, Y. Kim, and M. Kim, “Music: Mutation analysis tool with high configurability and extensibility,” ICST Workshops 2018.
[12] gcovr. https://gcovr.com/en/6.0_a/
[13] llvm-cov. https://llvm.org/docs/CommandGuide/llvm-cov.html