기호 실행에서의 인공 지능 적용에 대한 연구: 퍼징과 취약점 탐지에서의 활용

하회리*, 안선우*, 김현준*, 백윤흥*
*서울대학교 전기,정보공학부, 반도체공동연구소
wrha@sor.snu.ac.kr, swahn@sor.snu.ac.kr, hjkim@sor.snu.ac.kr, ypaek@snu.ac.kr

A Study on the Application of Artificial Intelligence in Symbolic Execution: Usage in fuzzing and vulnerability detection

Whoi Ree Ha*, Sunwoo Ahn*, Hyunjun Kim*, Yunheung Paek*
*Dept. of Electrical and Computer Engineering and Inter-University Semiconductor Research Center (ISRC),
Seoul National University

요 약

기호 실행 (symbolic execution)은 프로그램을 특정 상태로 구동하는 입력 값을 찾는 코드 분석 기법이다. 이를 사용하면 자동화 소프트웨어 테스트 기법인 퍼징 (fuzzing)을 훨씬 효율적으로 사용하여 더 많은 보안 취약점을 찾을 수 있지만, 기호 실행의 한계점으로 인하여 쉽게 적용할 수 없었다. 이를 해결하기 위해 인공 지능을 활용한 방법을 소개하겠다.

1.서론

기호 실행(symbolic execution)은 프로그램을 특정 상태로 구동하는 입력 값에 대한 이유를 나타내는 코드 분석 기법이다 [1]. 프로그램의 경로를 따라 연산을 수행하고 분석하여 특정 경로에 대한 제약 조건 (symbolic path constraints)을 수집한다. 특정 경로에 대한 제약 조건을 취득하면, 역으로 그 조건들을 풀어, 해당 경로를 위한 입력 값 생성도 가능하기 때문에 소프트웨어 테스팅 툴로 각광 받고 있다.

기호 실행으로 생성된 입력 값을 seed로 삼으면 특정 경로를 꾸준히 반복 실행하여 분석할 수 있다. 또한 제약 조건들을 분석하여 다양한 경로 및 상태에 대한 도달 가능성을 확인할 수 있으며, 실행 공간을 효율적으로 검색할 수 있다. 이런 특성들 때문에 자동화 소프트웨어 테스트 기법인 퍼징에 자주 사용되어 왔다 [2].

2. 퍼징 (Fuzzing)

퍼징은 소프트웨어 버그를 찾기 위해 자주 사용되는 기술이다. 퍼징에서는 무작위 테스트 입력을 생성하고 이 입력 값을 사용하여 프로그램을 실행한다. 이를 통해 해당 프로그램의 잠재적인 보안 취약점을 찾는다 [3].

이론적으로 퍼징은 주어진 시간 내에 발견된 취약점의 수를 최대화하는 프로그램 입력을 찾는 최적화 문제이다. 하지만 보안 취약점은 불연속적으로 분포되어 있기 때문에, 퍼징으로 최대한 많은 양의 코드를 실행하여 코드 커버리지(code coverage)를 최대화하는 것을 목적으로 한다. 따라서, 다양한 경로에 도달할 수 있는 제약 조건과 입력 값을 찾을 수 있는 기호 실행을 사용하면 퍼징을 훨씬 효율적으로 활용할 수 있다.[4]

3. 기호 실행의 문제점

기호 실행은 강력한 기술이지만 실제 적용 가능성에는 많은 제약이 있다. 첫번째로, 화이트 박스 형식을 취하기에 코드에 대한 엑세스가 필요하다. 코드를 완전히 분석하기 위해서 해당 코드의 언어도 이해해야 함으로 특정 언어만 인식할 수 있다. 따라서 다른 언어로 제작된 프로그램이나 코드가 주어지지 않은 경우에 적용할 수 없다.

두번째로, 기호 제약 조건(symbolic path constraints)을 효율적으로 풀 수 없는 경우가 빈번하다. 기호 실행 프로그램은 기호 제약 조건을 풀 때, SAT/SMT solver를 사용하는데, 이들은 명확한 한계점을 가지고 있다. 예를 들어, 비선형 조건에 대해서 기존 solver를 적용하면 굉장히 느리고 문제점이 많다 [5]. 또한 string에

대해서도 정확한 처리를 하지 못한다 [6].

세번째로, 기호 변수(symbolic variable)에 대한 가능 경로가 너무 많아지는 path explosion 의 경우, 정확한 기호 제약 조건을 추론하지 못한다. Path explosion 은 기호 실행할 때, 실행 공간이 기하급수적으로 커지는 경우를 말한다 [7].

아래 <그림 1>의 코드와 같은 경우, input 의 각 byte 가 'B'가 맞는지 아닌지에 대해서 경로가 존재하기 때문에 총 2^{100} 가능한 실행 경로가 생성된다. 따라서 이 모든 경로에 대한 제약 조건을 생성하는데에 메모리가 부족하거나 너무 오랜 시간이 걸린다.

<그림 1> path explosion 의 예제 코드[8]

```
1 int counter = 0, values = 0;
2 for ( i = 0 ; i < 100 ; i ++ ) {
3     if (input[i] == 'B') {
4         counter ++;
5         values += 2;
6     } }
7 if (counter == 75)    bug ();
```

4. 인공 지능의 적용 및 성능 지표

위와 같은 기호 실행의 문제점들을 해결하기 위해 인공 지능을 활용할 수 있다. 인공 지능은 기호 실행으로 추론하기 어려운 기호 제약 조건의 관계를 학습하여 상호 보완적인 기능을 할 수 있다 [9].

I. Preprocessing

기호 실행을 하기에 앞서 정적 분석을 통해 프로그램 코드에서 보안 취약점 (candidate vulnerability points, CVP)이 될 수 있는 지점들을 확인한다. 이 경우는 divide-by-zero 나 buffer overflow 를 일으킬 수 있는 모든 코드에 해당한다.

<그림 2> CVP 예제 [9]

```
max = psf_calc_signal_max (infile);
while (readcount > 0) {
 readcount = sf_readf_double(infile,data,frames);
 for (k = 0; k < readcount; k++)
  data[k] /= max; // potential divide-by-zero
```

II. Design

CVP 를 찾은 후, 기호 실행을 사용하다 위에서 언급된 문제점으로 인해 효율적이지 않은 상황이 오면 그 시점부터 인공 지능을 활용하여 제약 조건을 찾고 푸는 방식이다. 기호 실행이 효율적이지 않는 상황은 총 4 가지로 나눌 수 있다. (1) 같은 루프에서 빠져나가지 못 할때; (2) path explosion 이 일어나 메모리가 부족할 때; (3) 기존 solver 로 제약 조건을 풀지 못 할때; (4) 코드가 주어지지 않은 외부 함수를 사용할 때. 이 4 가지 상황에서는 기호 실행이 효율적이지 못하기에 인공 지능을 활용한다.

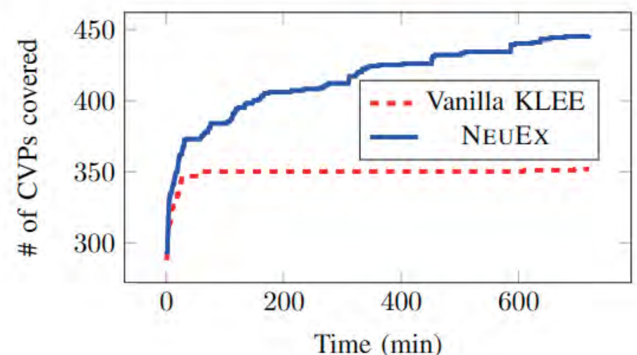
인공 지능을 사용할 때는, deep learning 을 사용하여 나머지 코드 부분을 추론한다. 기호 실행이 효율적이지 못한 코드 지점까지 생성된 기호 제약 조건을 풀어 해당 지점까지의 입력을 생성한다. 이 입력 값을 randomly mutate 하여 프로그램을 실행하고, CVP 로 도달하는 입력 값과 CVP 지점에서의 결과 값을 모아 training data 로 삼는다. 결과 값은 CVP 에서 버그를 발생시킬 수 있는 변수의 값이다. <그림 2>에서는 max 의 값이 결과 값이 된다. 이 training data 를 사용하여 나머지 코드 부분을 추론하는 neural network 를 생성한다.

이 Neural network 를 사용하면 해당 CVP 에서 보안 취약점이 존재하는지 확인할 수 있다. 먼저 랜덤 input 을 neural network 에 넣어 expected 결과 값을 확인한다. 이 결과 값과, CVP 에서 bug 를 trigger 하는 값을 비교하여 gradient-based algorithm 을 사용해 bug trigger 하는 값과 가장 가까운 값을 찾는다. 예를 들면 <그림 2>에서 max 값이 0 일 때, 보안 취약점이 발생한다. 따라서 neural network 에서 추론하는 max 값이 0 에 가장 가까운 input 을 찾는 방식이다. Local minima 가 존재할 수 있기 때문에 random input 을 여러번 바꿔가면서 해당 neural network 가 0 에 도달할 수 있는지 확인한다.

III. Performance

코드 분석에 사용된 프로그램은 cURL, SQLite, libTIFF, libsndfile, BIND, Sendmail, WuFTP 이다.

<그림 3> KLEE vs NeuEx [9]



<그림 3>은 기존 기호 실행 툴인 KLEE 와 인공 지능을 사용한 기호 실행 툴인 NeuEx 를 비교한 그래프이다. 전처리 과정에서 확인한 CVP 를 NeuEx 가 더 많이 확인한 것을 알 수 있다. 이는 특정 CVP 지점을 기존 기호 실행 툴로는 확인할 수 없는 반면에 NeuEx 를 사용하면 기존에 확인할 수 없는 code 부분도 확인할 수 있기 때문이다.

또한 아래 <표 1>을 보면 NeuEx를 사용하여 더 많은 보안 취약점을 찾은 것을 확인할 수 있다. 이는 NeuEx가 더 넓은 코드 커버리지를 보장하기 때문에 더 많은 취약점을 확인한 것으로 보인다.

<표 1> KLEE vs NeuEx

	KLEE	NeuEx
발견된 보안 취약점	18	34

5. 결론

본 논문에서는 인공 지능을 적용한 기호 실행과 이를 퍼징에 활용하여 더 많은 보안 취약점을 찾아낸 방법을 소개하였다. 기호 실행은 이론적으로 굉장히 강력한 코드 분석 기법이다. 프로그램 실행 경로에 따라 제약 조건을 확인하고, 제약 조건을 solver로 풀어 해당 경로를 위한 입력 값을 생성할 수 있다. 따라서 자동화 소프트웨어 테스트 기법인 퍼징과 연계하면 더 효율적으로 잠재적 보안 취약점을 확인할 수 있다. 하지만 화이트 박스 기법인 점, 기존 solver의 한계점, 그리고 실행 공간이 너무 커지면 비효율적인 점들이 문제가 되어 실제 코드를 분석하는데 빈번히 실패한다. 또한 퍼징과 같이 사용하기도 좋지 않다. 이를 해결하는 방법으로 인공 지능을 활용한 연구는 기존 기호 실행을 진행하다 한계점에 도달하면 나머지 코드 부분을 neural network로 추론한다. 이 neural network를 사용하여 잠재적 보안 취약점이 있는지 확인할 수 있다. 이 방법은 기존 기호 실행에서 분석 실패하던 코드 부분을 분석 가능하게 하여, 더 넓은 범위의 코드에서 보안 취약점을 찾을 수 있게 하였다.

6. ACKNOWLEDGEMENT

본 연구는 2020년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행되었으며 (NRF-2017R1A2A1A17069478), 2020년도 두뇌한국21 플러스사업, 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임. (No.2018-0-00230, (IoT 총괄/1세부) IoT 디바이스 자율 신뢰보장 기술 및 글로벌 표준기반 IoT 통합보안 오픈 플랫폼 기술개발 [TrusThingz 프로젝트])

참고문헌

[1] J. C. King, “Symbolic Execution and Program Testing,” Communications of the ACM, 1976.

[2] P. Godefroid, M. Y. Levin, D. A. Molnar et al., “Automated WhiteboxFuzz Testing,” in NDSS’08.

[3] B. Miller, L. Fredriksen, and B. So. "An empirical study of the reliabilityof unix utilities". Communications of the ACM, 33(12):32–44, 1990.

[4] N. Stephens, J. Grosen, C. Salls et al., "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". NDSS'16

[5] S. Gao, S. Kong, and E. M. Clarke, “dReal: An SMT Solver forNonlinear Theories over the Reals,” in CADE’13.

[6] Y. Zheng, X. Zhang, and V. Ganesh, “Z3-str: A Z3-Based String Solverfor Web Application Analysis,” in FSE’13.

[7] C. Cadar and K. Sen, “Symbolic Execution for Software Testing: ThreeDecades Later,” Comm of ACM’13.

[8] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. "Enhancing Symbolic Execution with Veritesting," ICSE' 14

[9] S. Shiqi, S, Shinde, S. Ramesh et al. "Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints." NDSS'19