(KSC2019 우수논문) 동적 함수 관련도를 이용한 퍼징 커버리지 향상 기법*

(Dynamic Function Relevance based Fuzzing for High Coverage)

이 아 청 †　　김 윤 호 ‡　　김 문 주§
(Ahcheong Lee)　　(Yunho Kim)　　(Moonzoo Kim)

요 약 커버리지 기반 퍼징 (Coverage Guided Fuzzing)은 테스트 케이스 생성 기법으로, 기법 자체가 간단하고, 큰 소프트웨어에도 적용이 가능하기 때문에 널리 이용되고 있다. 하지만, 기존의 퍼징 기법은 프로그램 내부의 시맨틱 정보를 쓰지 못하고 있다. 본 논문에서는 커버리지 향상을 위해 함수 관련도를 기반으로 변이할 바이트를 선택하는 새로운 2가지 휴리스틱을 제시한다. 두 함수 간의 함수 관련도는 두 함수가 같이 실행되는 테스트 케이스의 개수로 정의 되며, 높은 함수 관련도는 두 함수가 서로 높은 의존성을 가짐을 나타낸다. 어떤 타겟 함수의 커버리지 향상을 위해, 이 새로운 휴리스틱은 그 타겟 함수와 관련도가 높은 함수들이 읽고 쓰는 바이트만 변이하여 커버리지 향상을 꾀한다. 제시된 휴리스틱은 최신 퍼저 (Fuzzer)인 Angora와 FairFuzz를 기반으로 구현되었으며, 최신 퍼저들에서 사용된 실제 C 프로그램으로 평가하여 기존 퍼저 대비 각각 17.88%와 11.03%의 경로 커버리지 향상을 보였다.

키워드 : 퍼징, 동적 함수 관련도, 테스트 케이스 생성 기법, 커버리지 향상, 소프트웨어 테스팅

Abstract Coverage Guided Fuzzing (CGF) is one of the famous test case generation technique. The technique is actively researched and used based on its simplicity and scalability for large real software. However, most of the fuzzing techniques do not utilize valuable semantic information of target programs. This paper presents two new heuristics that use dynamic function relevance to select the appropriate input bytes which can be mutated to increase the coverage. The function relevance between the two functions is defined as the number of test cases that execute the functions together, and the high relevance means the two functions executing high dependency on each other. To improve coverage of a target function, the new heuristics determines bytes that are used by functions that are highly relevant to the target function, and only the valuable bytes are mutated. As these bytes have high data dependency on the variables in the target function, mutating them improves the coverage of the target function. We implemented the two heuristics on the top of the state-of-the-art fuzzers, Angora and FairFuzz, and evaluated on real-world C programs that are used by recent fuzzing works. The heuristics showed 17.88% and 11.03% path coverage improvement, respectively.

Key words : fuzzing, dynamic function relevance, test case generation, software testing

* 본 논문은 과학기술정보통신부의 재원으로 한국연구재단 차세대 정보 컴퓨팅기술개발사업의 지원 (NRF-2017M3C4A7068177), 과학기술정보통신부의 재원으로 한국연구재단의 지원 (NRF-2019R1A2B5B01069865), 과학기술정보통신부의 재원으로 한국연구재단 (NRF-2020R1C1C1013996), 삼성전자의 지원을 받아 수행한 연구임.

† 정회원 : KAIST 전산학부, ahcheong.lee@kaist.ac.kr

‡ 종신회원 : 한양대 컴퓨터소프트웨어학부, yunho.kim03@gmail.com

§ 종신회원 : KAIST 전산학부, moonzoo.kim@gmail.com

1. 서 론

커버리지 기반 퍼징 (Coverage Guided Fuzzing, CGF)은 테스트 케이스 생성 기술로, 그 기술 자체가 쉽게 적용하여 많이 사용되고 있다[1][2]. 퍼징은 심볼릭 실행(symbolic execution)[3]과 같이 오버헤드가 큰 기술을 피하는 대신, 각 테스트 케이스를 단순한 바이트의 나열로 읽은 후, 무작위의 바이트를 무작위하게 변이 시켜 새로운 테스트 케이스를 생성한다. 이때 가장 중요한 단계중 하나는, 주어진 테스트 케이스의 어떤 바이트를 변이할 것인지 고르는 것이다. 만약 너무 적은 바이트만을 선택하여 변이한다면, 의미 있는 바이트를 많이 놓치게 되고, 타겟 프로그램의 더 다양한 행동을 보일 수 있는 테스트 케이스를 생성하기 어려워 질 것이다. 마찬가지로 너무 많은 바이트를 선택하여 변이한다면, 바이트의 개수에 조합의 개수 만큼 큰 탐색 공간에서 제한된 시간 안에 의미있는 테스트 케이스의 생성이 어려워 진다. 따라서, 주어진 테스트 케이스의 어떤 바이트를 변이할 것인지 선택하는 다양한 기법들이 제시되었다. 최신 퍼징 기술 중 하나인 FairFuzz[4], Profuzzer[5] 및 GreyOne[6]의 경우 주어진 테스트 케이스를 타겟 프로그램에서 실행한 뒤, 실행 결과를 분석하여 주어진 테스트 케이스의 어떤 바이트를 변이할지 결정한다. 하지만 이러한 기법들은 실행 결과 또는 커버리지만 부분적으로 사용하기 때문에, 타겟 프로그램 내부의 시맨틱 정보를 활용하지 못하는 문제가 있다.

또다른 퍼징 기술인 Angora[7], Vuzzer[8] 및 Matryoshka[9]의 경우, 동적 오염 분석 (Dynamic Taint Analysis, DTA)을 적용하여 테스트 케이스의 각 바이트와 타겟 프로그램의 변수들 간의 데이터 의존성을 분석하여, 의존성을 띠는 바이트를 선택하여 변이한다. 더 자세하게, Angora는 먼저 달성하지 못한 분기(uncovered branch) 하나를 타겟으로 지정한 뒤 동적 오염 분석을 통해 해당 분기와 데이터 의존성을 띠는 테스트 케이스 바이트를 구하여 해당 바이트만을 변이한다. 이러한 바이트만을 선택하여 변이하는 것은 테스트 케이스의 전체 바이트를 변이하는 것에 비해 짧은 시간 안에 새로운 커버리지를 달성하는 테스트 케이스를 더 많이 생성할 수 있다. 하지만, 동적 오염 분석은 해당 분기를 달성하는데 필요한 바이트를 항상 모두 구하지 못하기 때문에, 동적 오염 분석을 통해 구한 바이트만을 변이할 때는 해당 분기를 달성하는데 실패할 수 있다.

따라서, 본 논문에서는 주어진 테스트 케이스의 어떤 바이트를 변이할지 조절할 수 있는 새로운 휴리스틱을 제시한다. 해당 타겟 분기가 포함된 함수를 f_t라 하였을 때, 1. f_t에 포함된 모든 분기를 타겟 분기와 높게 관련된 분기라 한다. 2. f_t와 관련도가 높은 함수들에 포함된 모든 분기도 타겟 분기와 높게 관련된 분기라 한다. 이타겟 분기와 높게 관련된 분기들은 타겟 분기의 변수

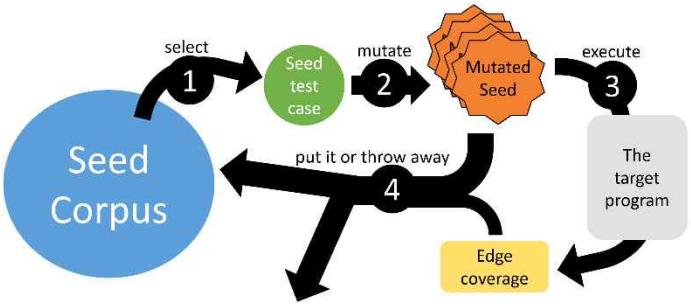

그림. 1 커버리지 기반 퍼징 프로세스
(Fig. 1 Coverage Guided Fuzzing process)

들을 읽고 쓸 가능성이 높기 때문에 해당 타겟 분기의 커버리지를 향상시키는데 사용될 수 있다. 또한, 타겟 분기 뿐만 아니라 타겟 분기와 높게 관련된 분기들도 새롭게 달성이 가능하다. 제시된 휴리스틱은 이 관련된 분기들에 영향을 미치는 바이트들만을 변이하여, 주어진 시간동안에 더 높은 커버리지를 달성할 수 있도록 한다.

본 논문에서는 이 높게 관련된 분기를 구하기 위해, 함수 관련도[10]를 퍼징에 새롭게 도입하였다. 기존의 함수 관련도는 Concolic 테스팅을 이용한 유닛 테스팅에서 오탐을 줄이면서도 빠른 테스팅이 가능하도록 관련도가 높은 함수를 묶어서 같이 테스팅하는 기술로 도입되었다. 본 논문에서는 이 함수 관련도를 퍼징 기술에서 분기 단위로 적용하여 커버리지 향상을 달성하였다.

본 논문에서는 이 휴리스틱을 최신 퍼징 기술 중 하나인 Angora와 FairFuzz에 적용하여 구현하였다. 최신 퍼징 연구들에서 사용된 실제 5개의 C 프로그램으로 평가하였으며, 각 휴리스틱은 5개 프로그램에서 평균 17.88%, 11.03%의 경로 커버리지 (path coverage) 향상을 보였다.

2. 커버리지 기반 퍼징 프로세스

그림 1은 커버리지 기반 퍼징의 일반적인 프로세스를 나타내고 있다. 0. 사용자가 기존에 가지고 있는 테스트 케이스 파일을 퍼저(Fuzzer)에게 전해주면, 퍼저는 이 테스트 케이스를 퍼저 자체의 시드 뭉치 (seed corpus)에 보관한다. 1. 시드 뭉치에서 하나의 테스트 케이스 파일 선택한 뒤, 2. 이를 무작위로 변이하여 여러 개의 새로운 테스트 케이스 파일을 생성한다. 그 후 3. 생성된 테스트 케이스 파일을 실행하여 각 테스트 케이스의 경로 커버리지를 구한 뒤, 4. 새로운 커버리지를 달성한 테스트 케이스만을 시드 뭉치에 넣는다. 이후 시드 뭉치에서 새로운 테스트 케이스를 선택하여 1번 단계부터 해당 프로세스를 반복한다.

단순한 프로세스이지만 구체적인 전략에 따라 퍼저의 성능이 달라진다. 1단계에서 시드 뭉치에서 어떤 테스트 케이스를 선택하여 변이할 것인지 결정해야하며, 2단계에서 어떤 테스트 케이스의 어떤 바이트를 선택하여 어떤 값으로 변이 할 것인지에 따라 새로 생성한 테스트 케이스가 커버리지를 높일 수 있는지가 결정된다. 본 논문에서는 2단계에서 테스트 케이스의 어떤 바이트를 선

택하여 변이 할 것인지에 대한 새로운 전략을 제안한다.

3. 함수 관련도 기반 퍼징

3.1 함수 관련도 정의

어떤 한 함수 f_1에 대한 다른 함수 f_2의 함수 관련도를 다음 수식(1)과 같이 정의한다.

$$Rel(f_1 \leftarrow f_2) = \frac{n_{f_1,f_2}}{n_{f_1}} \quad \dots (1)$$

n_{f_1}는 시드 뭉치에 존재하는 테스트 케이스 중 f_1을 실행하는 테스트 케이스의 수, n_{f_1,f_2}는 시드 뭉치에 존재하는 테스트 케이스 중 f_1과 f_2를 모두 실행하는 테스트 케이스의 수를 의미한다.

높은 함수 관련도는 두 함수를 모두 실행하는 테스트 케이스의 개수가 많음을 의미하고, 이는 곧 두 함수가 정보를 주고 받을 가능성이 높음을 의미한다. 이 프로그램 시맨틱 정보를 이용하여 퍼징을 진행하는 3.2절의 바이트 확장 기법, 3.3절의 바이트 축소 기법을 제시한다.

3.2 함수 관련도 기반 바이트 확장

Angora는 일단 달성하고자하는 목표 분기를 먼저 선택한 뒤 동적 오염 분석을 통해 테스트 케이스의 어떤 바이트가 해당 분기와 데이터 의존성을 보이는지 알아낸 다. 그 후, 이 바이트만을 변이하여 새로운 테스트 케이스를 생성한다. 테스트 케이스의 모든 바이트를 변이하는 것에 비해 짧은 시간 안에 커버리지를 높이는 테스트 케이스를 생성할 확률은 높지만, 동적 오염 분석으로 구한 바이트만을 변이하는 것이 해당 분기를 달성할 수 있다는 것을 보장하지 못한다. 동적 오염 분석은 하나의 테스트 케이스의 한번의 실행만을 분석하기 때문에, 실행되지 않은 부분의 정보를 얻지 못하며, 데이터 의존성만을 분석하기 때문에, 컨트롤 의존성(Control dependency) 정보는 전혀 사용하지 못한다. 따라서 동적 오염 분석으로 구한 바이트만을 계속해서 변이하는 것 보다, 해당 분기와 관련된 다른 바이트를 더 구해서 변이할 때 커버리지를 더 높일 수 있다.

본 논문에서는 함수 관련도를 이용하여 바이트를 확장하는 기법을 다음과 같이 제시한다.

1. 기존의 시드 뭉치 안의 테스트 케이스로 도달되었지만 (reached), 달성되지 않은 (uncovered) 분기 하나를 타겟으로 선택한다. 해당 분기에 도달하는 테스트 케이스의 어떤 바이트를 변이할 것인지 선택하기 위해 다음 단계를 수행한다.

2. 해당 타겟 분기를 포함하는 함수를 f_{target}이라 하였을 때, 관련도가 높은 함수들의 집합 $rel(f_{target})$을 구한다. $rel(f_{target})$는 수식 (2)와 같이 정의 된다.

$$rel(f_{target}) = \{ f \mid Rel(f_{target} \leftarrow f) > \alpha \} \quad \dots (2)$$

α는 사용자가 선택하는 역치 값으로, 본 논문에서는 0.7을 사용하였다.

3. $rel(f_{target})$의 모든 함수의 모든 분기를 타겟 분기와 관련도가 높은 분기라 하며, 동적 오염 분석을 관련도가 높은 분기들에 적용하여 해당 분기들에 데이터 의존성을 띠는 바이트를 모두 구한다.

4. 구한 바이트만을 변이하여 새로운 테스트 케이스를 생성한다.

기존 Angora는 해당 타겟 분기만을 선택하여 구한 바이트만을 변이했지만, 제시된 프로세스에서는 타겟 분기와 관련된 분기들에 대해서도 동적 오염 분석을 적용하므로 더 많은 바이트를 확장하여 변이하게 된다.

관련도가 높은 분기만을 골라 바이트를 확장하는 이유는 관련도가 높은 분기들이 타겟 분기와 같이 실행되는 경우가 많고, 따라서 타겟 분기와의 데이터 의존성이 높기 때문이다. 타겟 분기만을 동적 오염 분석하는 경우에는 타겟 분기를 달성할 수 있는 바이트를 구하지 못할 수 있는데, 이때 놓친 바이트를 관련된 분기에도 동적 오염 분석을 적용함으로써 구할 수 있으며, 동시에 테스트 케이스의 무작위 바이트를 변이하는 것 보다 짧은 시간 안에 해당 분기를 달성할 수 있다.

3.3 함수 관련도 기반 바이트 축소

또다른 퍼징 기법인 FairFuzz의 경우 테스트 케이스의 실행결과를 분석하는 방법으로 변이할 방법을 선택한다. FairFuzz는 모든 생성된 테스트 케이스를 실행하였을 때, 가장 적게 실행된 분기들을 타겟 분기로 정하고, 이러한 타겟 분기를 실행할 수 있는 테스트 케이스를 생성하는 것에 주력한다. 적게 실행된 분기들은 더 새로운 행동를 보일 수 있다는 것을 가정하고 있다. 더 자세하게, 하나의 타겟 분기가 주어 졌을 때, 기존의 테스트 케이스 중 이 타겟 분기를 실행하는 테스트 케이스 하나를 선택한 뒤, 이 테스트 케이스의 각 바이트가 변이할 가치가 있는지 조사한다. 각 바이트를 무작위로 한번씩 변이한 뒤 실행하면서, 이 무작위 변이를 한 뒤에도 여전히 타겟 분기를 실행한다면, 이 바이트를 변이하면 적게 실행된 분기를 실행하면서 새로운 행동을 보일 수 있을 것이라고 가정한다. FairFuzz는 이러한 바이트만을 선택적으로 변이하여 커버리지 향상을 꾀한다. 하지만, 실제로 이러한 바이트를 구하여 보았을 때 FairFuzz는 너무 많은 바이트를 고르는 문제가 있으며, 또한 테스트 케이스의 각 바이트마다 한번씩 변이하여 타겟 프로그램을 실행하기 때문에 바이트 분석에 필요한 오버헤드가 너무 큰 문제가 있다.

본 논문에서는 함수 관련도를 이용하여 변이할 바이트를 축소하는 기법을 다음과 같이 제시한다.

1. 기존 FairFuzz와 동일하게 실행 횟수가 적은 분기 하나를 타겟 분기로 선택하고, 해당 분기를 실행하는 테

표 1 실험 대상 프로그램 정보
(Table 1 The evaluation subject's information)

subject	version	Lines of Code
bison	3.5.2	43797
bsdtar	3.4.2	87592
cjpeg	2.0.4	4714
objdump	2.34	154286
readelf	2.34	60559

표 2 함수 관련도 기반 바이트 확장 기법 실험 결과
(Table 2 Results of function relevance-based byte extension experiment)

Path Coverage	Angora	**Angora-FR**	Angora-random
bison	12784.4	**19362.5 (+51.45%)**	17996.2 (+40.77%)
bsdtar	10953.2	**12915 (+17.91%)**	8901.6 (-18.73%)
cjpeg	3871.2	**4033.8 (+4.20%)**	3810.3 (-1.57%)
objdump	66671.7	**68131.8 (+2.19%)**	63823.1 (-4.27%)
readelf	16633.7	**18901.1 (+13.63%)**	15886 (-4.50%)
Average	-	**+17.88%**	+2.34%

스트 케이스 하나를 변이를 위해 선택한다.

2. 해당 타겟 분기를 포함하는 함수를 f_{target}이라 하고, 3.2절과 마찬가지로 수식(2)와 같이 관련도가 높은 함수들의 집합 $rel(f_{target})$를 구한다.

3. 1에서 선택한 테스트 케이스가 타겟 분기와 관련된 함수들을 얼만큼 실행하고 있는지를 수식(3)과 같이 평가한다.

$$Score(tc) = \frac{n_{rel}}{n_{func}} \in [0,1] \ldots (3)$$

이때 n_{rel}과 n_{func}의 값은 다음과 같다.

$$n_{rel} = |\{g, g \in rel(f_{target}), \text{function } g \text{ is executed by } tc\}|$$

$$n_{func} = |\{g, \text{function } g \text{ is executed by } tc\}|$$

*tc*는 평가하고자 하는 해당 테스트 케이스를 의미한다. 해당 점수가 높을 수록 해당 테스트 케이스는 타겟 분기와 관련도가 높은 함수들을 많이 실행하고 있음을 의미하며, 따라서 더 많은 바이트를 변이할 가치가 있다. 이 점수가 낮은 경우에는 시간을 많이 들여 변이할 가치가 떨어지는 테스트 케이스이며, 이 테스트 케이스는 기존에 비해 적은 일부 바이트만을 변이하도록 한다.

4. 3에서 구한 점수를 그대로 각 바이트를 조사할 확률로 사용한다. 1에서 선택한 테스트 케이스의 각 바이트의 변이 조사 여부를 확률로 결정 한뒤, 각 바이트에 대해 조사하는 것으로 결정되면, 기존 FairFuzz와 같이, 무작위로 해당 바이트를 한번 변이한 뒤 실행하여, 타겟 분기를 실행하는지 여부로, 이 바이트를 변이할지 여부를 결정한다. 조사 하지 않는 것으로 결정되면, 해당 바이트는 조사 및 변이에서 제외한다.

수식(3)에서 구한 점수가 높다면, 이는 해당 테스트 케이스가 타겟 분기와 관련된 분기들(함수들)을 많이 실행하고 있음을 의미한다. 따라서 이러한 테스트 케이스의 더 많은 바이트를 조사하고 변이한다면, 적게 실행된 분기를 실행하면서 새로운 행동을 보이는데 필요한 바이트를 더 많이 변이할 수 있다.

표 3 함수 관련도 기반 바이트 축소 기법 실험 결과
(Table 3 Results of function relevance-based byte shrinking experiment)

Path Coverage	FairFuzz	**FairFuzz-FR**	Random-0.25	Random-0.5	Random-0.75
bison	3606.3	**4278.2 (+18.77%)**	3903.8 (+8.37%)	3683.7 (+2.15%)	3528.8 (-2.15%)
bsdtar	4374.5	**4532.0 (+3.60%)**	4531.4 (+3.59%)	4425.9 (+1.17%)	4328.2 (-1.06%)
cjpeg	3123.6	**3341.0 (+6.96%)**	3193.2 (+2.23%)	3073.4 (-1.61%)	3182.2 (+1.88%)
objdump	5317.6	**6313.2 (+18.72%)**	5544.8 (+4.27%)	5358.4 (+0.77%)	5270.8 (-0.88%)
readelf	12895.7	**13812.2 (+7.11%)**	12784 (-0.87%)	13387.4 (+3.81%)	13574.8 (+5.27%)
Average	-	**+11.03%**	+3.52%	+1.26%	+1.04%

4. 평가 및 분석

4.1 실험 설정

본 논문에서는 제시한 2가지 함수 관련도 휴리스틱을 각각 Angora와 FairFuzz에서 구현하였으며, 이를 Angora-FR, FairFuzz-FR이라 하였다. 퍼징을 위한 기본적인 프로브와 함수 실행 결과를 수집하기 위한 프로브 생성을 위해 LLVM[11] compiler pass가 사용되었다. 이를 최신 퍼징 연구에서 사용된 5개의 실제 C 프로그램에서 평가하였다. 표 1에서 대상 프로그램에 대한 정보를 나타내었다. 각 퍼징의 실행은 24시간 동안 진행하였으며, 퍼징 자체에 무작위성이 존재하기 때문에 10번 수행을 반복한 뒤 평균값을 보고하였다. 다음 4개의 실험 문제 (Research Question)에 대해 답변하기 위해 각 퍼저가 달성한 평균 경로 커버리지 (달성된 경로(path)의 수)를 구하여 나타냈다.

RQ1. 함수 관련도를 이용하여 변이할 바이트를 확장하였을 때 더 높은 커버리지를 달성하는가?

RQ2. 함수 관련도를 이용하여 변이할 바이트를 축소하였을 때 더 높은 커버리지를 달성하는가?

또한 다음의 RQ3와 RQ4에 대해 평가하기 위해서 Angora-random와 FairFuzz-random을 구현하였다.

RQ3. 함수 관련도를 이용하여 바이트를 확장하는 것이 무작위로 바이트를 선택하여 추가하는 것보다 높은 커비지를 달성하는가?

RQ4. 함수 관련도를 이용하여 바이트를 축소하는 것이 무작위로 바이트를 축소하는 것보다 높은 커버리지를 달성하는가?

Angora-random의 경우, 일단 Angora-FR의 함수 관련도를 이용하여 얼만큼의 바이트를 확장하는지 구한다음, Angora-random에서 해당 숫자만큼 바이트를 무작위를 추가로 선택하여 변이하도록 하였다. FairFuzz-random의 경우, 각 바이트를 변이할지 조사

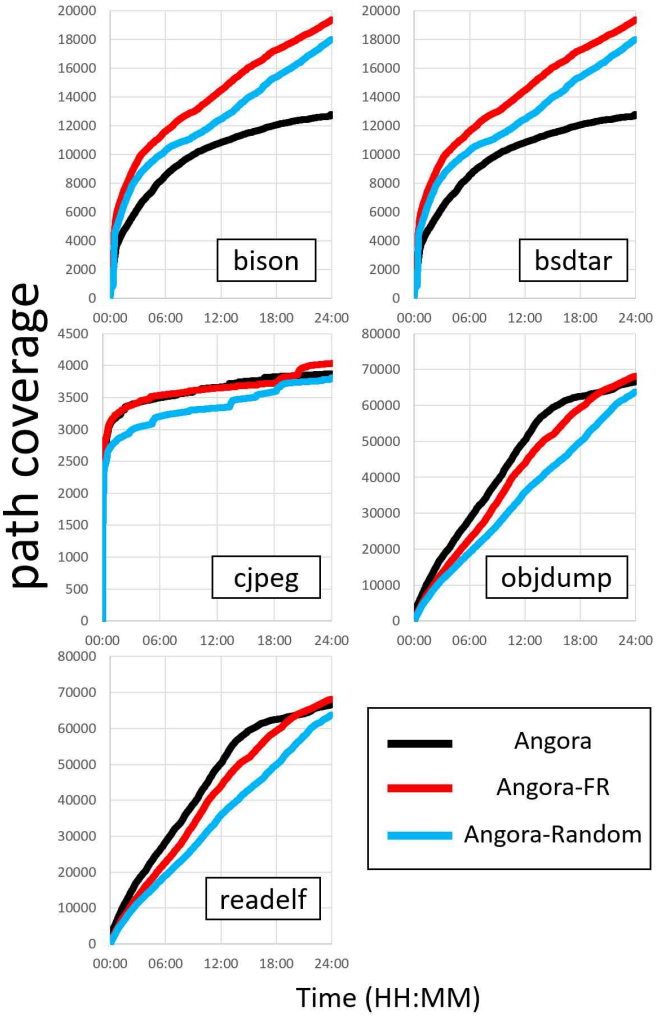

그림 2 함수 관련도를 이용한 바이트 확장 기법 결과
(Fig. 2 Results of function relevance-based byte extension experiment)

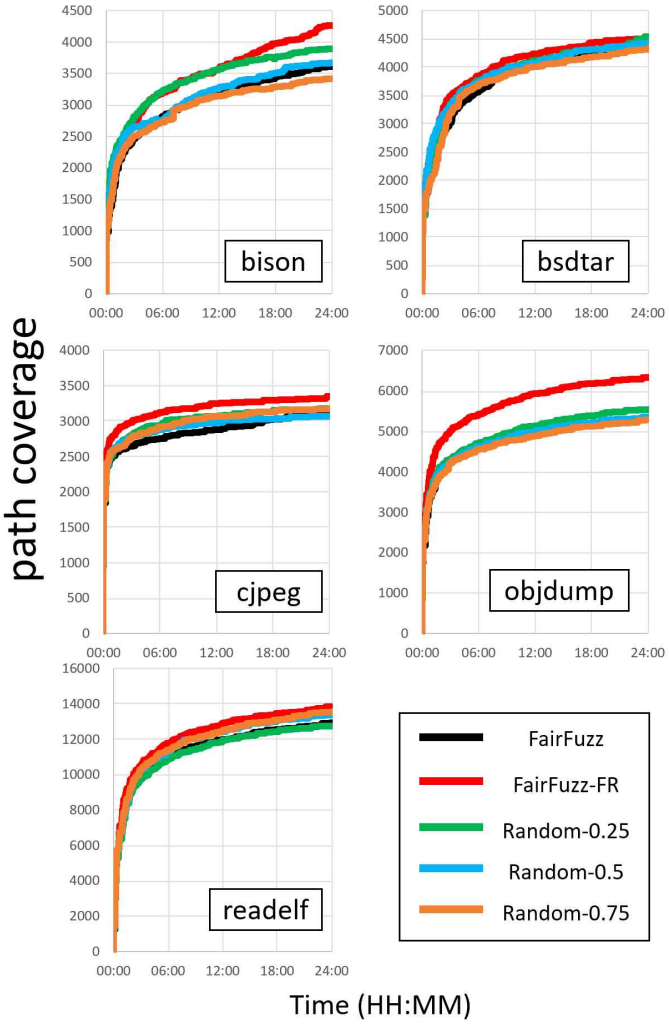

그림 3 함수 관련도를 이용한 바이트 축소 기법 결과
(Fig. 3 Results of function relevance-based byte shrinking experiment)

할 확률 (점수)를 계산하지 않고, 0.25, 0.5, 0.75로 고정하여 실험을 진행하였다. 즉, FairFuzz-FR의 경우 테스트 케이스의 전체 바이트 중, 함수 관련도를 이용하여 계산한 점수만큼의 바이트만 조사하여 변이할 가치가 있는지 조사하고, FairFuzz-random의 경우, 각각 25%, 50%, 75%의 바이트만 조사하여 변이할 가치가 있는지 확인한다.

추가적으로, 제시된 휴리스틱이 타겟 분기와 타겟 분기에 높게 관련된 분기들을 달성하도록 적용되었는지 알아보기위해 다음 실험 문제를 구성하였다.

RQ5. 함수 관련도를 이용하여 바이트를 확장하였을 때, 타겟 분기와 관련도가 높은 분기를 달성하는 경우가 증가하였는가?

이 실험 문제에 답하기 위해 바이트 확장이 적용된 Angora와 Angora-FR에 대해 다음과 같이 새로 달성된 분기 중 해당 달성된 분기가 타겟 분기 와 관련도가 높은 분기인 의 비율을 다음 수식(4)와 같이 구하였다.

$$\frac{\Sigma_{(b_t,b_c)\in B_{(target,covered)}} \left| \cup_{g\in F_t} \{b \in g\} \right|}{\left| B_{(target,covered)} \right|} \quad \cdots \quad (4)$$

이 때 $B_{(target,\,covered)}$는 퍼저에 의해 달성된 분기 b_c와, 해당 분기를 달성할 때 퍼저가 목표로 했던 타겟 분기b_t의 쌍 (pair)의 리스트를 의미한다. 또한, F_t는 각 타겟 분기 b_t에 대해, b_t가 포함된 함수 f_t와 관련도가 높은 함수들의 집합이다. 제시된 휴리스틱을 이용하여 테스트 케이스를 생성하였을 때, 이 비율값이 높다면, 어떤 달성된 분기가 타겟 분기와 관련도가 높은 분기일 가능성이 높은 것이고, 따라서 제시된 휴리스틱이 제대로 작동한 것이라고 볼 수 있다.

4.2 적용 결과

표 2와 그림 2는 변이할 바이트를 함수 관련도를 이용하여 확장하였을 때의 경로 커버리지 결과, 표 3와 그림 3는 변이할 바이트를 함수 관련도를 이용하여 축소하였을 때의 경로 커버리지 결과를 나타낸다. 표2와 표3에서 괄호안의 숫자는 각각 기존 Angora, FairFuzz

표 4 달성된 분기가 타겟 분기와 높게 관련된 분기인 비율
(Table 4 The ratio of covered branches that are highly related to the target branches)

	bison	bsdtar	cjpeg	objdump	readelf	Average
Angora	61.4%	78.7%	99.2%	86.6%	46.1%	74.4%
Angora-FR	**98.6%**	**89.0%**	**99.1%**	**96.7%**	**66.5%**	**90.0%**

의 결과와 비교하였을 때의 성능 향상 정도를 나타낸다. 5개 대상 프로그램에 대한 성능 향상 정도의 평균을 마지막 행에 나타내었다. 그림 2와 그림 3의 그래프는 각 대상 프로그램 별로 실험을 진행한 24시간 동안 달성한 경로 커버리지를 나타낸다.

RQ1. 표2에서, Angora-FR은 5개 모든 프로그램에서 가장 높은 경로 커버리지를 달성하였으며, 평균적으로 Angora에 비해 경로 커버리지는 17.88% 향상하였다. 그림 2에서, Angora에 비해 계속해서 경로 커버리지가 더 가파르게 증가하는 추세를 보였다.

RQ2. 표3에서, FairFuzz-FR은 5개 모든 프로그램에서 가장 높은 경로 커버리지를 달성하였으며, 평균적으로 FairFuzz에 비해 경로 커버리지는 11.03% 향상하였다. 그림 3에서, FairFuzz-FR은 모든 대상 프로그램에서 24시간 동안 꾸준히 우위를 점하는 것을 볼 수 있다.

RQ3. 표2에서, Angora-FR은 Angora-random 보다 5개 모든 프로그램에서 더 높은 성능을 보였으며, Angora-random은 Angora-FR과 같은 수의 바이트 만큼을 확장하였음에도, 한 대상 프로그램에서는 경로 커버리지가 18.73%까지 성능이 떨어지는 것을 보였다. 이는 실제로 함수 관련도를 이용하여 바이트를 선택하였을 때 더 가치 있는 바이트를 선택할 수 있었음을 의미한다. 그림 2에서, 모든 프로그램에서 24시간 동안 Angora-FR이 Angora-random에 비해 우위를 보이고 있다.

RQ4. 표3에서, FairFuzz-FR은 모든 FairFuzz-random보다 5개 모든 프로그램에서 높은 성능을 보였다. 이는 함수 관련도를 이용하여 각 테스트 케이스를 평가하여 변이할 바이트를 축소하는 것이 무작위하게 축소하는 것보다 성능 향상이 있었음을 의미한다. 경우에 따라 기존 FairFuzz보다 경로 커버리지가 감소하는 경우도 발생하였는데, 이는 무작위로 선택하는 전략이 새로운 커버리지를 달성하는데 필요한 바이트를 놓쳤음을 의미한다. 그림 3에서, 무작위로 각 바이트를 조사할 확률에 따라 경로 커버리지가 어떻게 변하는지 볼 수 있지만, 모든 프로그램에서 뚜렷하게 우위를 점하는 결과를 보이는 확률값은 없었다.

RQ5. 제시된 휴리스틱이 의도된 방향으로, 타겟 분기와, 타겟 분기와 높게 관련된 분기를 달성하도록 휴리스틱이 적용되었는지 알아보기위해 구성하였다. 표 4에 해당 결과가 나타나있다. 평균적으로, 제시된 휴리스틱이 적용되지 않은 Angora에서는 74.4%, 적용된 Angora-FR에서는 90.0%의 달성된 분기가 실제 타겟 분기와 관련도가 높은 분기로 나타났다. 따라서, 제시된 휴리스틱이 의도한 방향대로, 타겟 분기와, 타겟 분기와 관련된 분기를 더 많이 달성하도록 적용되어 커버리지가 증가하였음을 알 수 있다.

위 다섯개의 실험 문제로 제시된 휴리스틱이 실제로 성능 향상을 보였음을 알 수 있었다.

5. 결론

본 논문에서는 커버리지 기반 퍼징에서 커버리지를 높이기 위해 함수 관련도를 이용하여 바이트를 선택하는 2가지 휴리스틱을 제안하였다. 이를 최신 퍼저인 Angora와 FairFuzz에서 구현하였으며, 실제 5개의 C 프로그램에서 평가하였다. 제시된 휴리스틱은 각각 17.88%와 11.03%의 경로 커버리지 성능 향상을 보였다.

본 논문에서는 어떤 바이트를 선택하여 변이할 것인지에 대해서만 함수 관련도를 사용하였다. 하지만, 어떤 테스트를 골라서 변이할지, 각 테스트 케이스는 얼마나 많이 변이를 시도할지, 그리고 어떤 값으로 변이를 시도할지 등 많은 전략이 가능하기 때문에, 함수 관련도를 이용한 전략에 대해 향후 연구를 진행 할 것이다.

또한 시맨틱 정보를 나타내기 위해 함수 관련도뿐이 아니라 다양한 매트릭[12][13]이 제시되고 있다. 이러한 매트릭을 활용할 수 있는 퍼징 기술의 연구도 필요할 것이다.

또한 커버리지를 높이는 것 뿐만 아니라, 다른 다양한 목표로도 퍼징 기술이 개발되고 있다. 메모리 고갈 버그를 만들 수 있는 테스트 케이스를 만드는 퍼징 기술[14], 실행 시간이 오래 걸리는 테스트 케이스 생성 퍼징 기술[15] 등 다양한 목적으로 퍼징이 사용될 수 있기 때문에, 새로운 목적을 가지는 퍼징 기술에 대한 연구도 진행 할 것이다.

참고문헌

[1] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[2] V. Manès, H. Han, C. Han, S. Cha, M. Egele, E. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey.," *IEEE Transactions on Software Engineering*, 2019.

[3] R. Baldoni, E. Coppa, D. C. De’elia, C. Demetrescu, and I. Finocchi, “A Survey of Symbloic Execution Techniques,” *ACM Comput. Surv.* 51, 3, Article 50, 39 pages, May 2018.

[4] C. Lemieux, and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[5] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," *2019 IEEE Symposium on Security and Privacy (SP). IEEE*, 2019.

[6] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data Flow Sensitive Fuzzing," *29th USENIX Security Symposium,* 2020.

[7] P. Chen, and C. Hao, "Angora: Efficient fuzzing by principled search," *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," *The Network and Distributed System Security Symposium (NDSS).* Vol. 17. 2017.

[9] P. Chen, L. Jianzhong, and C. Hao. "Matryoshka: fuzzing deeply nested branches," *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019.

[10] Y. Kim, Y. Choi, and M. Kim, "Precise concolic unit testing of c programs using extended units and symbolic alarm filtering," *Proceedings of the 40th International Conference on Software Engineering. ACM*, 2018.

[11] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization),* 2004.

[12] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Transactions on Software Engineering 30, 8,* 2004.

[13] S. Chidamber and C. Kemerer, "Towards a Metrics Suite for Object Oriented Design," 1991.

[14] N. Coppik, O. Schwahn, and N. Suri, "MemFuzz: Using Memory Accesses to Guide Fuzzing," *12th IEEE Conference on Software Testing, Validation and Verification (ICST).ㄸhzz: Using Memory Accesses to Guide Fuzzing"SQ),ograms based on Information Flow"ack-directed and Runtime Optimization).*

[15] C. Lemieux, R. Padhye, K. Sen, and K. Song, "PerfFuzz: Automatically Generating Pathological Inputs," *In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

Ahcheong Lee
2019년 KAIST 전삭학부 졸업 학사
2019년 ~ 현재 KAIST 전산학부 석사 과정.
관심 분야는 퍼징, Concolic 테스팅

Yunho Kim
2007년 KAIST 전산학과 학사. 2009년 KAIST 전산학과 석사. 2017년 KAIST 전산학부 박사. 2018년 ~ 2020년 KAIST 전산학부 연구교수. 2020년 ~ 현재 한양대 컴퓨터소프트웨어학부 조교수. 관심분야는 자동화된 Concolic 유닛 테스팅, 변이 테스팅, 자동 오류 위치 추정, 정형 검증, 퍼징

Moonzoo Kim
1995년 KAIST 전산학과 학사. 2001년 Univ. of Pennsylvania 박사. 2002년 ~ 2004년 SECUi.COM 차장. 2004년 ~ 2006년 POSTECH 연구원. 2006년 ~ 2012년 KAIST 전산학과 조교수. 2012년 ~ 현재 KAIST 전산학부 부교수. 관심분야는 Concolic testing, 자동 오류 위치 추정, Concurrency 테스팅, 변이 테스팅, 정형 검증, 내장형 소프트웨어, 퍼징