동적 심볼릭 수행을 응용한 테스트 케이스 자동 생성 도구 비교

김윤호, 김문주

KAIST
대전광역시 유성구 373-1
kimyunho@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

요약: 우리가 살고 있는 사회가 점점 ubiquitous 컴퓨팅 사회로 발전되어감에 따라 소프트웨어의 신뢰성이 중요한 화두로 등장하고 있으며, 현재 소프트웨어 검증 방법 중 가장 널리 쓰이는 테스팅 기법의 성패는 효과적인 테스트 케이스 생성 여부에 좌우된다. 하지만 기존의 테스팅 기법은, 소프트웨어 엔지니어가 수동으로 생성하는 테스트 케이스를 주로 사용하고 있으며, 이로 인하여 테스트 케이스를 많이 생성하기도 힘들 뿐 더러, 복잡한 소프트웨어의 다양한 동작가운데 숨겨져 있는 버그를 찾아내는 데에 그 효과가 제한적인 단점이 있다.

이러한 단점을 극복하기 위하여 동적 심볼릭 수행(dynamic symbolic execution)을 통해 높은 커버리지를 달성하는 테스트 케이스를 자동으로 생성하는 테스팅기법이 최근에 제시되면서 관련학계 및 산업체의 관심을 받고 있다. 하지만 심볼릭 수행을 사용하는 테스팅 기법은 아직 표준화된 기법이 사용되기보다는 각각의 도구의 구현에 따라 특성이 좌우되는 만큼, 여러 도구들이 선택한 구현기법을 살펴보는 것이, 보다 발전된 기법을 연구하는 데에 도움이 되리라 생각된다. 따라서 본 논문에서는, open-source 로써 사용자의 필요에 따라 자유로이 확장을 할 수 있는 도구 중, CREST 와 KLEE 에 관해 비교연구를 수행함으로써, 각각의 기법에 대한 장단점을 분석하고 새로운 심볼릭 수행을 응용한 테스팅 기법의 나아갈 바를 알아보도록 한다.

핵심어: 심볼릭 수행, 소프트웨어 테스팅,

1. 서론

우리가 살고 있는 사회가 점점 ubiquitous 컴퓨팅 사회로 발전되어감에 따라 소프트웨어의 신뢰성이 중요한 화두로 등장하고 있다. 현대인들이 일상적으로 사용하는 이동전화와 같은 소형 기기부터 전력제어, 의료장비와 같은 대형 기기에 이르기까지 소프트웨어로 구동되면서 소프트웨어의 오류가 사회에 미치는 영향이 크고 다양해지고 있다. 예를 들어 윈도 XP 운영체제의 오류[1]로 인해 많은 기업과 개인이 업무를 보지 못하는 손실을 입는가 하면 의료장비 소프트웨어의 오류[2]로 인해 인명 피해가 발생하기도 하였다. 따라서 소프트웨어의 신뢰성을 검증하기 위한 효과적인 기법을 개발하는 것이 중요하다.

현재 소프트웨어 검증 방법 중 가장 널리 쓰이는 테스팅 기법의 성패는 효과적인 테스트 케이스 생성 여부에 좌우된다. 하지만 기존의 테스팅 기법은 소프트웨어 엔지니어가 수동으로 생성하는 테스트 케이스를 주로 사용하거나, 임의로 생성한 소프트웨어 입력을 사용하기 때문에, 테스트 케이스를 많이 생성하기 힘들고 복잡한 소프트웨어의 다양한 동작 가운데 숨겨져 있는 버그를 찾아내는 데에 그 효과가 제한적인 단점이 있다.

이러한 단점을 극복하기 위하여 동적 심볼릭 수행(dynamic symbolic execution)을 통해 높은 커버리지를 달성하는 테스트 케이스를 자동으로 생성하는 테스팅 기법[3-9]이 최근에 제시되면서 관련학계 및 산업체의 관심을 받고 있다. 하지만 심볼릭 수행을 사용하는 테스팅 기법은 아직 표준화된 기법이 사용되기보다는 각각의 도구의 구현에 따라 특성이 좌우되는 만큼, 여러 도구들이 선택한 구현기법을 살펴보는 것이, 보다 발전된 기법을 연구하는 데에 도움이 되리라 생각된다. 따라서 본 논문에서는, open-source 로써 사용자의 필요에 따라 자유로이 확장을 할 수 있는 도구 중, CREST[5]와 KLEE[9] 에 관해 비교연구를 수행함으로써, 각각의 기법에 대한 장단점을 분석하고 새로운 심볼릭 수행을 응용한 테스팅 기법의 나아갈 바를 알아보도록 한다.

2. 심볼릭 수행을 응용한 테스트 케이스 생성

심볼릭 수행을 응용한 테스트 케이스 자동 생성 기법은 크게 정적 테스트 생성 기법과 동적 테스트 생성 기법 두 가지로 나뉜다. 2 장에서는 각 기법을 소개하고 어떤 기준으로 비교 연구 대상을 선택했는지 설명한다.

2.1 정적 테스트 생성

정적 테스트 생성 기법[10-12]은 프로그램을 실행하지 않고 정적 분석을 통해 심볼릭 수행을 실행하고 특정 실행 경로를 따라가는 입력 값을 계산한다. 프로그램의 가능한 모든 입력 값에 따라 생성되는 프로그램의 실행 경로를 심볼릭 수행을 통해 면밀하게 검사하고 실제 실행 경로를 따라갈 수 있는 입력 값을 테스트 케이스로 생성한다. 정적 테스트 생성 기법은 프로그램의 각 조건문에서 가능한 모든 분기를 탐색하여 프로그램의 모든 가능한 실행 경로를 구하고, 각 실행 경로 π 에 대해 실행 경로상의 할당문과 실행된 조건 분기를 제약 조건으로 갖는 경로 제약 조건 φ_π 를 생성한다. φ_π 가 satisfiable 하면 π 가 실제로 실행 가능한 실행 경로이고 제약 조건을 만족하는 변수 값이 프로그램을 π 를 따라 실행하게 하는 입력 값이 된다.

하지만 이런 정적 테스트 생성 기법은 2 가지 한계가 있다. 첫번째 한계는 프로그램이 정적 심볼릭 수행을 통해 분석할 수 없는 구문을 갖는 경우다.

```
1 int check_pass(int inp, int orig){
2   if (inp != encrypt(orig)) return -1;
3   return 0;
4 }
```

함수 encrypt 는 외부 라이브러리에 구현되어 있는 함수라 가정하자. 정적 분석은 프로그램이 사용하는 외부 함수에 대한 분석을 수행할 수 없기 때문에 2 번째 줄에 있는 if 분기가 어느 쪽으로 수행되는 지 판단할 수 없다. 설령 encrypt 함수의 소스코드가 주어진다 해도 이와 같은 암호화 함수는 심볼릭 분석을 통해 제약 조건을 풀어내는 것이 매우 어렵게 설계되어 있어 정적 심볼릭 분석을 수행할 수 없다. 따라서 정적 심볼릭 수행 결과의 정확도가 크게 낮아지고 테스트 케이스를 효과적으로 생성하지 못하게 된다.

실제 프로그램 코드에는 이와 같이 정적 분석으로 분석할 수 없는 외부 함수 호출이나 심볼릭 수행으로 추론하기 어려운 복잡한 구문(포인터 연산, 비선형 표현식 등)이 많이 사용되고 있기 때문에, 정적 심볼릭 수행을 사용한 테스트 생성 기법이 실제 소프트웨어 검증에 활용되기에는 많은 어려움이 있다.

2.2 동적 테스트 생성

동적 테스트 생성 기법[3-9]은 주어진 입력을 갖고 실제 프로그램을 수행하면서 실행된 프로그램 경로를 따라 경로 제약 조건을 생성한다. 프로그램 수행이 종료되면 경로를 따라 생성된 경로 제약 조건을 활용하여, 이전에 실행된 경로와는 다른 실행 경로를 따르는 새로운 프로그램 입력을 생성한다. 이 과정을 모든 가능한 실행 경로가 수행되거나 사용자가 지정한 특정 조건을 만족할 때까지 반복한다.

동적 테스트 생성 기법은 실제 프로그램 수행을 통해 정적 테스트 생성 기법이 갖는 한계를 극복한다. 정적 심볼릭 수행으로 분석하기 어려운 구문이 있는 경우 실제 프로그램 수행에서 갖는 값을 활용하여 심볼릭 수행의 정확도를 최대한으로 높일 수 있다.

check_pass 함수를 다시 살펴보자. 프로그램을 실행하지 않고 2 번째 줄의 if 조건식이 만족되지 않는 경우(inp == encrypt(orig))를 계산하는 것은 불가능하지만 프로그램 실행을 통해 orig 변수의 특정 값에 대해 encrypt(orig) 의 값을 알게 되면 이 값과 같은 inp 변수 값을 생성할 수 있게 된다. 따라서 동적 심볼릭 실행을 활용한 테스트 생성 기법을 사용하면, 입력 변수 orig 의 값을 고정하고 inp 변수 값을 바꿈으로써 check_pass 의 모든 실행 경로를 면밀히 검사할 수 있는 테스트 입력을 생성할 수 있다.

이와 같이 동적 테스트 생성 기법은 심볼릭 수행을 통한 테스트 케이스 추론과 실제 프로그램 실행을 통한 프로그램 구문 간소화를 통해 정적 테스트 생성 기법이 갖는 정확도 문제를 극복하고 실제 프로그램 개발에 활용될 수 있을 정도의 유용성을 갖고 있다.

2.3 동적 테스트 생성 도구

2005 년 동적 심볼릭 수행(이하 특별한 언급이 없는 심볼릭 수행은 모두 동적 심볼릭 수행을 의미한다)을 활용한 테스트 자동 생성 기법이 제안된 이래 다양한 연구팀에서 이 기법을 사용한 도구를 개발하였다. 이 중 비교 연구에 사용할 도구를 선정하기 위해 다음 두 가지 기준을 사용하였다.

- C 언어로 작성된 프로그램 지원

소프트웨어는 용도와 개발 언어에 따라 복잡도가 크게 달라진다. 따라서 테스트 케이스 생성의 효과를 극대화하기 위해, 대상 프로그램의 특징에 따라 동적 테스트 생성 도구의 세부 생성 기법 및 구현을 최적화해야 한다.

본 연구진은 C 언어로 작성된 내장형 소프트웨어의 신뢰성 검증을 위한 동적 테스트 생성 기법 및 도구에 주력한다. 내장형 소프트웨어의 경우 하드웨어 및 외부입력과 직접적인 상호작용을 수행하기 때문에 복잡도가 높으며 한 번 기기와 함께 배포된 후에는 프로그램의 오류를 수정하기 어렵기 때문에 높은 신뢰도가 요구된다. 또한 내장형 소프트웨어의 경우 빠른 시간 안에 시장에 출시되어야 하기 때문에

효율적인 소프트웨어 신뢰성 검증 기법이 필요하다. 따라서 이런 문제들을 해결하기 위한 방법으로 심볼릭 수행을 활용한 동적 테스트 자동 생성 기법을 제안하고 각 도구를 비교 분석하도록 한다.

- Open-source

각 기법 및 도구의 실제 활용 가치를 비교하려면 각 도구의 알고리즘뿐만 아니라 도구 자체를 비교 분석해야 한다. 심볼릭 수행을 활용한 동적 테스트 자동 생성 기법은 개발된 지 오래 되지 않았기 때문에, 도구 구현이 실제 개발 환경에 적용할 수 있을 만큼 성숙하지 않았다. 따라서 실제 개발환경에 적용하기 위해 도구를 개선하려면 각 도구의 내부 구현을 분석하고 논문에 밝히지 않은 문제점들이 어떤 것들이 있으며 실제 개발환경에 적용하기 위해 어떤 숨겨진 문제점들이 있는지 파악하고 분석해야 한다.

따라서 비교 분석 도구는 소스코드가 공개되어 있고 자유롭게 사용할 수 있어야 한다. 또한 기 개발된 Open-source 도구를 발전 시켜 나감으로써 새로 도구를 개발하기 위한 노력을 줄여 쉽고 빠르게 도구를 개선해 나갈 수 있다는 장점이 있다.

위와 같은 두 가지 기준을 만족하는 도구는 CREST 와 KLEE 가 있다. 3 장에서는 CREST 를 설명하고 4 장에서는 KLEE 를 살펴보도록 한다.

3. CREST

CREST 는 UC Berkeley 대학의 K. Sen 교수 팀에서 개발한 동적 심볼릭 테스트 생성 도구이다. CREST 는 C 로 작성된 프로그램 소스코드를 입력으로 받아 가능한 모든 실행 경로를 탐색하는 테스트 입력을 생성한다. 먼저, 프로그램 상태 변화를 기록할 수 있도록 C Intermediate Language(CIL)[13] 소스 변환 도구를 사용해서 대상 프로그램을 수정(instrumentation) 하고 CREST 가 생성한 입력에 따라 프로그램을 수행시키면서 프로그램 실행 경로가 어떤 분기를 따라가는지 기록한 심볼릭 경로 조건식을 생성한다. 이전 프로그램 실행 경로와 다른 실행 경로를 탐색하기 위해 심볼릭 경로 조건식의 심볼릭 표현식을 부정하여 이전 경로와 다른 분기를 따라가게 될 새 경로 조건식을 만들고 선형 정수 조건식을 지원하는 SMT solver[14] 를 사용해서 새 조건식을 따라가게 할 테스트 입력을 구한다.

3.1 심볼릭 경로 조건식 생성

CREST 는 C 작성된 프로그램 소스코드를 입력으로 받아, 먼저 심볼릭 입력을 생성하고 프로그램 실행 경로에 따른 심볼릭 경로 조건식을 생성할 수 있도록 CIL 을 사용해서 프로그램 소스코드를 수정한다. 대상 프로그램의 구문마다 심볼릭 변수가 어떻게 바뀌고 어떤 실행 경로를 따라가는지 기록하기 위한 probe 를 삽입한다. 수정된 프로그램은 CREST 가 설정한 심볼릭 입력 값에 따라 실제로 실행되면서 실행 경로를 따르는 심볼릭 경로 조건식을 기록한다. 실행이 종료되면 실행 경로 조건식을 CREST 로 넘겨줘서 다음 입력 값을 계산하는데 활용하도록 한다. 첫 번째 실행의 입력값은 선언된 심볼릭 입력 변수 타입의 기본값을 갖고 그 다음부터는 CREST 가 계산한 입력값을 갖는다.

수정된 프로그램이 실행되면서 심볼릭 수행을 통해 심볼릭 경로 조건식을 생성한다. 각 심볼릭 변수는 심볼릭 맵을 통해 관리되며, 각 할당문 Sj 가 실행될 때 마다 할당문의 심볼릭 변수에 해당하는 심볼릭 표현식을 갱신한다. 프로그램이 조건 분기문을 실행하게 되면 해당 조건 분기의 조건식을 심볼릭 표현식으로 나타낸 심볼릭 경로 조건식을 계산한다. 수정된 대상 프로그램의 실행이 종료되면 프로그램 실행중에 계산한 심볼릭 경로 조건식을 전부 결합한, 실행 경로 π_i 에 대한 심볼릭 경로 조건식 $\varphi_{\pi i}$ 가 생성된다.

심볼릭 경로 조건식 $\varphi_{\pi i}$ 는 다음 입력값을 생성하기 위해 사용된다. CREST 는 $\varphi_{\pi i}$ 의 심볼릭 제약 조건 중 하나를 선택해서 부정하고(negate) 부정한 제약 조건 이후의 제약 조건을 지움으로써 새로운 경로 조건식 $\varphi_{\pi i}$ 를 생성한다. $\varphi_{\pi i}$ 를 풀어 생성한 입력값으로 프로그램을 수행하면 이전 프로그램 실행 경로와는 다른 경로를 실행하게 된다. 만약 depth-first search(DFS) 를 사용해서 프로그램 실행 경로를 탐색한다면 $\varphi_{\pi i}$ 의 가장 마지막 심볼릭 제약 조건을 선택해서 부정한다.

CREST 는 모든 가능한 프로그램 실행 경로를 수행하거나 사용자가 지정한 제한 값에 도달할 때까지 프로그램 실행과 다음 입력 값 생성을 반복한다.

3.2 대상 프로그램

CREST 는 C 로 작성한 소스코드를 입력으로 받아 CIL 을 사용해서 probe 를 삽입하고 실제 실행될 수 있는 바이너리를 생성하여 실행한다. 이와 같은 방법은 구현이 쉽고 실제 바이너리가 CPU 바로 위에서 실행되기 때문에 속도가 빠른 장점이 있다.

반면 다음과 같은 단점이 있다. 첫째, C 로 작성한 소스코드를 입력으로 받아 probe 를 삽입하기 때문에 CIL 이 모든 복잡한 C 구문을 정확하게 해석해서 probe 를 삽입할 수 있어야 한다. 만약 CIL 이 복잡한 구문 분석에 실패한다면 해당 구문의 심볼릭 제약 조건을 정확하게 구할 수 없기 때문에 전체 테스트 생성이 부정확해 진다. 둘째, probe 삽입이 C 구

문 단위로 수행되기 때문에 C++ 혹은 다른 언어로 도구를 확장하기 어렵다. 다른 언어로 확장하기 위해서 probe 와 이를 삽입하기 위한 구문 모두 해당 언어에 맞게 새로 작성하여야 한다.

현재 CREST 는 심볼릭 메모리 구조를 완전히 지원하지 않기 때문에 심볼릭 포인터나 심볼릭 배열 참조를 지원하지 않는다.

```
1 int func(int n){
2   int a[4]={1,2,3,4};
3   if (a[n] == 3) return -1;
4   return 0;
5 }
```

위 예제에서 n 을 심볼릭 입력 값이라고 가정하자. CREST 는 심볼릭 메모리 구조를 지원하지 않기 때문에 변수 n 의 값을 어떻게 선택해야 a[n]==3 을 만족하는지 추론할 수 없다. 따라서 a[n] 은 프로그램 실행 당시의 실제 값으로 치환되고 심볼릭 경로 조건에 들어가지 않게 된다. 포인터를 심볼릭하게 사용하는 경우도 이와 비슷하며 따라서 CREST 는 배열과 포인터를 사용하는 프로그램의 테스트 케이스를 생성하는 데 있어 약점이 있다.

3.3 선형 정수 심볼릭 경로 조건식

CREST 가 생성하는 심볼릭 경로 조건식은 선형 정수 표현식으로 표현된다. 선형 정수 표현식에서 각 변수와 상수의 타입은 크기 제한이 없는 정수이고 각 표현식은 일차식의 형태를 갖는다. 즉, 사칙 연산 중 변수와 변수의 곱셉 연산이나 나눗셈 연산은 처리할 수 없으며, C 에서 사용하는 bit-wise 연산(&, |, ^등) 을 지원하지 않는다. CREST 가 프로그램을 실행하여 심볼릭 경로 조건식을 만들 때 선형 정수 표현식으로 표현할 수 없는 C 표현식을 만나면 실행 당시 계산된 실제 변수 값을 사용해서 선형 정수 표현식으로 간소화한다.

선형 정수 심볼릭 경로 조건식을 사용했을 때 갖는 가장 큰 장점은 상대적으로 빠른 시간 안에 풀 수 있다는 점이다. 선형 정수 조건식을 효율적으로 풀기 위한 알고리즘[15]이 많이 연구되어 왔으며 이를 구현한 SMT solver 튜닝 기술 또한 발전하였다. 실제 수학 연산을 이진 회로로 인코딩해서 표현하는 것이 아니라 CPU 연산으로 계산하기 때문에 bit-vector 를 사용하는 것 보다 더 빨리 조건식을 풀 수 있다.

반면 선형 정수 심볼릭 경로 조건식은 실제 프로그램의 수행을 정확하게 표현하지 못한다. 먼저 실제 프로그램에서 사용하는 제한된 크기의 정수 변수가 아니라 제한 없는 정수 변수를 사용하기 때문에 오버플로나 언더플로와 같은 경우를 고려할 수 없다. 또한 bit-wise 연산자도 지원하지 않기 때문에 이와 같은 연산을 사용한 프로그램에서 높은 커버리지를 갖는 테스트 케이스를 생성하지 못할 수 있다.

4. KLEE

KLEE 는 Stanford 대학의 D. Engler 교수 팀에서 개발한 심볼릭 수행 도구이다. KLEE 는 LLVM[16] bytecode 의 해석기를 기반으로, LLVM bytecode 로 컴파일 된 프로그램을 입력으로 받아서 명령어를 수행하고 심볼릭하게 표현된 프로그램의 상태 변경을 기록한다. 프로그램 수행 가운데 분기를 만나게 되면 현재 프로그램의 심볼릭 상태에 따라 분기할 수 있는 경우를 계산하고 가능한 실행 경로를 따라 계속해서 프로그램을 실행한다. 현재 프로그램의 심볼릭 상태가 특정 분기를 실행할 수 있는지 검사하기 위해 KLEE 는 bit-level 정확도를 갖는 조건식을 생성하고 bit-vector 로직을 지원하는 SMT solver 를 사용해서 조건식을 풀어낸다.

4.1 심볼릭 경로 조건식 생성

KLEE 는 주어진 심볼릭 입력을 갖고 LLVM bytecode 를 수행한다. KLEE 는 실행할 심볼릭 프로세스를 선택하고 명령어를 해석해서 실행하는 것을 주어진 실행 시간이 다 되거나 더 이상 실행할 수 있는 명령어가 없을 때까지 반복한다.

일반적인 프로세스에서 레지스터, 스택, 힙 메모리가 데이터 값을 갖는 것과 달리 심볼릭 프로세스의 레지스터, 스택, 힙 메모리는 심볼릭 표현식을 나타낸다. 각 심볼릭 표현식의 말단 노드는 심볼릭 상수와 변수를 나타내고 중간 노드는 bytecode 명령어를 실행 후 나온 결과를 표현하는 심볼릭 표현식을 나타낸다. 예를 들어

%dst = add i32 %src0,%src1

과 같은 식이 있다면 %src0, %src1 에 해당하는 심볼릭 표현식을 가져와서 add 연산을 수행한 심볼릭 표현식을 새로 만든 다음 %dst 에 새로 만든 표현식을 저장한다.

조건 분기 명령어는 Boolean 표현식을 받아서 그 값에 따라 프로그램 카운터의 값을 바꾼다. KLEE 가 명령어 해석중에 조건 분기 명령어를 만나면 조건 분기 명령어의 Boolean 표현식을 나타내는 심볼릭 표현식이 현재 주어진 심볼릭 입력에 대해 참인지 거짓인지 알아내기 위해 SMT solver 를 통해 풀어낸다. 만약 어느 한 쪽으로만 진행이 가능하다면 KLEE 는 가능한 쪽으로 계속해서 명령 해석을 수행한다. 현재 주어진 심볼릭 입력값에 대해 양 쪽 모두 분기 진행 가능성이 있으면 현재 진행중인 심볼릭

프로세스를 복제(clone) 하고 각 분기에 맞게 프로그램 카운터를 설정한 다음 양쪽 모두를 해석하게 된다. 각 심볼릭 프로세스의 심볼릭 상태를 저장하기 위한 메모리 사용을 최소로 하기 위해 Copy-on-Write 기법을 사용해서 변경되는 심볼릭 상태만 따로 저장하고 공통된 부분은 서로 공유한다.

4.2 대상 프로그램

KLEE 는 LLVM 의 bytecode 로 컴파일된 프로그램을 입력으로 받고 bytecode 명령어를 직접 해석하여 실행한다.

이와 같은 방법은 CREST 에서 사용한 C 프로그램을 대상으로 하는 것과 비교하여 다음과 같은 세 가지 장점이 있다. 첫째, 프로그래밍 언어에 독립적인 도구를 만들 수 있다. CREST 의 경우 C 언어를 대상으로 하였기 때문에 프로그램 수정이나 심볼릭 상태 기록이 모두 C 언어에서 사용되는 경우만 가정하고 있다. 그렇기 때문에 이를 C++ 이나 다른 언어로 확장하기 위해서는 추가 작업이 필요하다. 그러나 LLVM bytecode 는 프로그래밍 언어와 독립적이기 때문에, KLEE 는 LLVM 이 지원하는 언어로 작성된 모든 프로그램을 대상으로 할 수 있다. 둘째, 대상 프로그램 실행을 세밀하게 제어해서 중복 실행되는 부분을 막을 수 있다. CREST 의 경우 심볼릭 경로 조건식을 생성하기 위해서 생성된 입력에 따라 프로그램을 온전히 실행시켜야 한다. 이 과정에서 이전 수행과 현재 수행에서 분기가 갈라지기 이전까지의 부분이 중복하여 실행된다. 반면 KLEE 는 LLVM 의 인터프리터로서 동작하여 실제 분기가 수행되는 부분에서 심볼릭 실행을 분기함으로써 프로그램 각 부는분이 한번씩만 수행되도록 할 수 있다. 세번째 장점은 실제 프로그램이 수행되는 바이너리를 직접 검증하는 점이다. CREST 의 경우 컴파일러가 생성한 바이너리에 문제가 발생하는 경우 오류를 발견하지 못할 수 있다. 그러나 KLEE 는 컴파일이 끝난 LLVM bytecode 를 검사하기 때문에 실제 수행되는 바이너리 프로그램이 야기할 수 있는 오류까지 검증할 수 있다.

반면 KLEE 는 대상 프로그램의 bytecode 를 직접 해석해야 하기 때문에 실제 프로그램을 실행하는 CREST 보다 수행 시간이 오래 걸리는 단점이 있다. CREST 에서 대상 프로그램과 추가된 probe 가 중복 실행되긴 하지만 이는 CPU 가 직접 실행하기 때문에 KLEE 에서 bytecode 를 해석하는 것 보다 실행 속도가 더 빠르다.

4.3 심볼릭 외부 환경 모델링

프로그램은 실행하면서 다양한 외부 환경에서 입력값을 가져온다. 외부 환경은 실행 인자, 네트웍 패킷, 파일 입력, 환경 변수 와 같은 것들을 포함한다. 즉 심볼릭 수행을 통해 프로그램 테스트 커버리지를 높이려면 이러한 외부 환경을 심볼릭 입력으로 모델링 할 수 있어야 한다. 또한 심볼릭 외부 환경을 프로그램이 변경하는 경우, 이 변경이 해당 프로그램 실행의 이후 부분에 영향을 미쳐야 한다.

KLEE 는 입력 파일이나 환경 변수와 같은 외부 환경을 심볼릭 입력으로 생성할 수 있다. 외부 환경의 값을 가져오고 기록하려면 POSIX 라이브러리 함수의 호출을 사용해야 한다. KLEE 는 uClibc 를 수정하여 외부 환경과 통신하는 함수 및 시스템 호출이 심볼릭 환경을 모델링 할 수 있도록 하였다. 예를 들어 open 시스템 콜의 인자로 심볼릭 변수로 선언된 파일 이름을 넘겨주면 open 시스템 콜 수행 부분에서 해당 이름의 파일이 있는 경우, 파일이 없는 경우로 분기하게 되어 프로그램이 파일이 존재하는 경우와 존재하지 않는 경우를 모두 가정하고 작성되었는지 테스트 할 수 있다. 뿐만 아니라, 생성된 파일의 크기 및 내용도 심볼릭하게 모델링 할 수 있다.

4.4 Bit-level 심볼릭 경로 조건식

KLEE 는 LLVM 의 해석을 수행하면서 bit-level 의 심볼릭 표현식을 계산한다. KLEE 에서 조건 분기 명령어를 해석할 때 주어진 심볼릭 입력과 심볼릭 표현식을 bit-level 심볼릭 조건식으로 표현한다. Bit-level 의 심볼릭 조건식은 각각의 변수를 주어진 길이의 bit-width 를 갖는 bit-vector 로 표현하고 CPU 회로에서 처리하는 것과 동일한 로직을 사용해서 사칙연산 및 bit-wise 연산(&, |, ^등)을 처리한다. 따라서 bit-wise 연산을 지원하지 않고 선형 조건식만 처리할 수 있는 선형 정수 심볼릭 조건식보다 더 정확하게 프로그램의 상태를 표현할 수 있다.

그러나 bit-level 의 심볼릭 조건식은 일반적으로 선형 정수 조건식보다 더 풀기 어려운 것으로 알려져있다. bit-level 조건식과 선형 정수 조건식 모두 다항 시간 알고리즘은 없지만 선형 정수 조건식의 경우 branch-and-bound, cutting-pane 알고리즘을 사용하여 빠른 시간안에 풀 수 있다. 또한 bit-level 의 심볼릭 조건식은 선형 정수 조건식과 달리 변수와 변수의 곱셈 연산과 나눗셈 연산과 같은 복잡한 조건식이 생성될 수 있고 사칙 연산이 이진 게이트 수준의 저 수준으로 표현되기 때문에 조건식을 푸는 시간이 상대적으로 더 길다.

5. CREST 와 KLEE 의 비교

표 1 에서 볼 수 있듯, CREST 와 KLEE 가 사용한 심볼릭 경로 조건식 생성 방법과 표현형태에 따라 각 도구의 성능과 정확도에서 큰 차이를 보였다. CREST 의 경우 심볼릭 경로 조건식 표현 방법으로

선형 정수 표현식을 사용함으로써 프로그램의 실행

	CREST	KLEE
입력 형식	C 소스코드	LLVM bytecode
심볼릭 경로 조건식 생성	CIL 을 사용해서 probe 코드 삽입	LLVM bytecode 를 직접 해석
심볼릭 경로 조건 표현형태	선형 정수 표현식	Bit-level 표현식
장점	빠른 실행 속도	Bit-level 정확도, 바이너리 수준 정확도 보장, 심볼릭 외부 환경 모델링 지원
단점	선형 정수 연산만 지원	느린 실행 속도

표 1 CREST 와 KLEE 의 비교

경로를 정확하게 묘사할 수 없었고 C 로 작성된 소스코드를 입력으로 함으로써 바이너리 레벨이 아닌 소스코드 레벨의 정확도만을 보장할 수 있었다. 반면 KLEE 는 LLVM bytecode 를 입력으로 하고 심볼릭

경로 조건식을 bit-vector 를 사용해서 표현함에 따라 프로그램이 실제 CPU 에서 수행되는 것을 정확하게 따라가고 표현할 수 있었다.

하지만 도구의 정확도가 높을수록 성능이 낮아졌다. 선형 정수 표현식은 bit-vector 표현식에 비해 표현이 간단하기 때문에 더 쉽게 풀릴 수 있었다. 또한 CREST 가 실제 프로그램 실행을 CPU 에서 수행한 것과 다르게 KLEE 는 모든 명령어를 하나하나 해석해야 했기 때문에 심볼릭 경로 조건식을 푸는 시간뿐 만 아니라 프로그램을 실행하는 시간에서도 CREST 에 크게 뒤쳐지는 결과를 보여줬다.

6. 결 론

본 연구진은 심볼릭 수행을 통한 테스트 케이스 자동 생성 기법을 소개하고 기법을 구현한 두 가지 도구 CREST 와 KLEE 를 비교 분석하였다. 두 도구 모두 동적 심볼릭 수행을 통해 정적 심볼릭 수행이 갖는 문제를 해결하고 C 로 작성된 프로그램의 테스트 케이스를 자동으로 생성할 수 있었다. 또한 두 도구 모두 open-source 로 개발됨에 따라 도구 내부 이해를 정확하게 할 수 있고, 새로운 도구를 개발할 필요 없이 기존의 도구를 발전시켜 나감으로써 새로운 기법을 빨리 구현할 수 있는 장점이 있다.

하지만, 비교분석 결과 두 도구 모두 내장형 소프트웨어를 검증하는 데 활용되기 위해서는 정확도와 성능 면에서 아직 한계가 있음을 발견하였다. 따라서 정확도와 성능 모두 만족할 수 있는 심볼릭 경로 조건식의 표현 방법을 새로 정의하고 심볼릭 경로 조건식을 효율적으로 풀기 위한 개선된 기법이 필요하다. 또한 현재의 DFS 방식의 탐색 방법으로는 복잡한 프로그램을 대상으로 비용 효율적이지 못하기 때문에 실행 경로 나무를 비용 효율이 높게 탐색할 수 있는 향상된 탐색 기법의 개발이 필요하다.

참고문헌

[1] M. Bailey, E. Cooke, D. Watson, F. Jahanian, and J, Nazario, “The Blaster Worm: Then and Now,” IEEE Security and Privacy, Vol. 3, No. 4, pp.26-31, 2005

[2] N. Leveson and C. S. Turner, “An investigation of the therac-25 accidents,” IEEE Computer, Vol. 26, No.7, pp.18-41, 1993

[3] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing,” ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, 2005

[4] K. Sen, D. Marinov, and G. Agha, “CUTE: A Concolic Unit Testing Engine for C,” International Symposium on the Foundations of Software Engineering, 2005

[5] CREST Project Page http://code.google.com/p/crest

[6] P. Godefroid, M. Y. Levin, and D. Molnar, “Automated Whitebox Fuzz Testing”, Annual network and Distributed System Security Symposium, 2008

[7] N. Tillmann, and J. Halleux, “Pex-White Box Test Generation for .NET,” International Conference on Tests and Proofs, 2008

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “EXE: Automatically Generating Inputs of Death”, ACM Conference on Computer and Communications Security, 2006

[9] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”, USENIX Symposium on Operating System Design and Implementation, 2008

[10] J. C. King “Symbolic Execution and Program Testing,” Journal of the ACM, Vol.19, No. 7, pp. 385-394, 1976

[11] C. Csallner and Y. Smaragdakis, “Check’n Crash:

Combining Static Checking and Testing", International Conference on Software Engineering, 2005

[12] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution," Tools And Algorithms for Construction and Analysis of Systems, 2005

[13] G. C. Necula, S. McPeak, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," International Conference on Compiler Construction, 2002

[14] SMT-LIB: The Satisfiability Modulo Theories Library, http://combination.cs.uiowa.edu/smtlib/.

[15] B. Dutertre and L. Moura, "A Fast Linear-arithmetic Solver for DPLL(T)," International Conference on Computer Aided Verification, 2006

[16] C. Lattner, and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation", International Symposium on Code Generation and Optimization, 2004