바이너리 코드에 대한 그레이 박스 Concolic 테스트

최재승 KAIST 대전, 대한민국
jschoi17@kaist.ac.kr

장준 운 Samsung Research Seoul, 대한민국
joonun.jang@samsung.com

한충우 네이버랩스 대한민국 성남
cwhan.tunz@gmail.com

차 상길 KAIST 대전, 대한민국
sangkilc@kaist.ac.kr

개요 - 우리는 화이트 박스 퍼징과 그레이 박스 퍼징의 장점을 결합한 새로운 경로 기반 테스트 케이스 생성 방법인 그레이 박스 concolic 테스트를 제시합니다. 높은 수준에서 우리 기술은 화이트 박스 퍼징(concolic testing)과 같이 테스트 중인 프로그램의 실행 경로를 체계적으로 탐색하면서 그레이 박스 퍼징의 단순성을 포기하지 않습니다. SMT 솔버에 의존합니다. 우리는 Eclipser라는 시스템에서 우리의 기술을 구현했으며, 이를 최첨단 그레이박스 fuzzer(AFLFast, LAF-intel, Steelix, VUzzer 포함) 및 Symbolic executor(KLEE)와 비교했습니다. 우리의 실험에서 우리는 더 높은 코드 커버리지를 달성했고 다른 도구보다 더 많은 버그를 발견했습니다.

색인 용어 - 소프트웨어 테스트, concolic 테스트, 퍼징

I. 서론

퍼즈 테스팅(줄여서 퍼징)은 폐쇄 바이너리 코드에서 보안 취약점을 찾기 위한 사실상의 표준이었습니다[1]. 보안 실무자들은 항상 증거와 함께 버그를 찾기 때문에 퍼징을 높이 평가합니다. Microsoft 및 Google과 같은 주요 소프트웨어 회사는 현재 제품의 보안을 보장하는 수단으로 소프트웨어 개발 수명 주기에서 퍼징을 사용합니다[2], [3].

가장 주목할만한 것은 AFL[4], AFLFast[5], Steelix[6], VUzzer[7], Angora[8], CollAFL[9], T-Fuzz[10]와 같은 그레이 박스 fuzzer가 떠오르고 있습니다. 최신 버그 찾기. Grey-box fuzzing은 진화적인 프로세스로 테스트 케이스를 생성합니다. 특히 테스트 케이스를 실행하고 적합성 함수(목적 함수라고도 함)를 기반으로 평가합니다. 그런 다음 더 나은 적합성을 가진 사람들의 우선 순위를 지정하고 목표를 충족하는 테스트 사례를 찾도록 발전시키고 프로그램 충돌을 유발하는 버그가 있는 경로를 실행하기를 희망하면서 전체 프로세스를 계속 반복합니다.

현재의 그레이박스 퍼저는 피트니스 기능으로 코드 커버리지를 사용합니다. 따라서 커버리지 기반 퍼저(coverage-based fuzzer)[5], [7]라고도 합니다. 예를 들어, AFL [4]과 그 후속 모델 [5], [6], [11]은 근사치 형태의 분기 커버리지를 사용하는 반면 VUzzer [7]는 가중 기본 블록 적중 수를 피트니스 함수로 사용합니다. 코드 적용 범위를 최대화하면 테스트 중인 프로그램(PUT)의 흥미로운 실행 경로를 실행할 가능성이 높아진다는 것은 자명합니다.

그러나 기존의 그레이박스 퍼저는 코드 커버리지가 입력 돌연변이에 대해 민감하게 변경되지 않기 때문에 커버리지 기반 지침에도 불구하고 새로운 분기를 실행하는 데 어려움을 겪습니다. 특히, 두 개의 다른 입력을 가진 두 개의 프로그램 실행은 실행에서 비교된 조건 분기의 값이 서로 다르더라도 동일한 코드 적용 범위를 달성할 수 있습니다. 즉, 코드 커버리지는 피드백을 제공할 수 있습니다.

조건부 분기가 무작위로 생성된 입력으로 침투하지만 그러한 입력을 생성하는 데 직접적인 도움이 되지 않는 경우에만. 이러한 감도 부족으로 인해 PUT이 입력을 특정 매직 값과 비교할 때와 같이 일부 상황에서 그레이 박스 fuzzer가 높은 범위의 테스트 케이스를 생성하기가 어렵습니다. AFLGo[11], Steelix[6], VUzzer[7]와 같은 현재의 최첨단 그레이 박스 fuzzer도 거의 같은 문제를 가지고 있습니다.

결과적으로, 그레이박스 퍼징은 취약점을 찾는 효과에도 불구하고 유일한 테스트 케이스 생성 알고리즘이 될 수 없다고 널리 알려져 있습니다. 따라서 그레이 박스 퍼저는 동적 기호 실행 [10], [12] 및 미세한 taint 분석 [7], [8], [13]과 같은 고비용 화이트 박스 분석에 의해 또는 초기 테스트 케이스 생성 프로세스를 지시하는 시드 입력 [14], [15].

예를 들어 Angora[8]와 Driller[12]는 각각 세분화된 taint 분석과 동적 기호 실행을 활용하여 그레이박스 퍼징의 코드 적용 범위를 개선합니다.

한편, 화이트 박스 퍼징(동적 기호 실행 또는 concolic 테스트라고도 함)[16]–[21]은 분기 조건을 해결하여 테스트 케이스를 체계적으로 생성할 수 있지만 고전적인 경로 폭발 문제를 제쳐두고 확장성에 의해 근본적으로 제한됩니다. 첫째, 화이트 박스 퍼저는 PUT의 모든 단일 명령을 분석합니다. PUT의 모든 단일 명령어를 계측하기 때문에 모든 퍼징 반복에는 상당한 계산 비용이 수반됩니다. 둘째, 기호 실행은 모든 실행 경로에 대한 기호 경로 제약 조건을 구축합니다.

SMT 솔버[22]로 이러한 제약을 푸는 것은 계산적으로 비용이 많이 듭니다. 또한 기호 입력의 영향을 받는 모든 단일 메모리 셀에 대한 기호 표현식을 저장하려면 상당한 메모리 공간이 필요합니다.

이 문서에서 우리는 그레이 박스 concolic 테스트라고 하는 새로운 테스트 케이스 생성 기술을 제안하고 여기에서 Eclipser라고 하는 도구에서 구현합니다. Grey-box concolic 테스트는 단순함을 잃지 않으면서 화이트박스 퍼징과 같은 분기 조건을 충족하는 테스트 케이스를 효율적으로 생성합니다. 값비싼 프로그램 분석 기술에 의존하지 않습니다. 따라서 회색 상자 퍼징과 같이 실제 응용 프로그램으로 확장됩니다.

우리의 접근 방식은 화이트 박스 퍼징[19, 23]에서 널리 사용되는 검색 전략인 세대 검색과 유사합니다. 여기서 단일 프로그램 실행은 실행 중에 발생하는 모든 조건 분기를 해결하여 테스트 케이스 세대를 생성합니다. 그레이 박스 concolic 테스트는 경로 기반 테스트 케이스 생성도 수행하지만 그레이 박스 방식으로 조건부 분기를 해결하려고 시도합니다.

테스트 케이스를 생성하기 위해 PUT 및 실행 동작을 관찰합니다.

그레이 박스 concolic 테스트와 화이트 박스 퍼징의 주요 차이점은 우리의 접근 방식이 PUT의 각 실행 경로를 실행하기 위한 입력 조건을 부분적으로 설명하는 대략적인 형태의 경로 제약 조건에 의존한다는 것입니다. 근사 경로 제약 조건은 SMT 해결과 같은 CPU 또는 메모리 집약적 작업에 의존하지 않고 조건부 분기를 통과할 수 있는 입력을 찾는 데 도움이 됩니다. 당연히 그레이 박스 concolic 테스트에서 생성된 경로 제약 조건은 정확하지 않지만 실제로는 다양한 실행 경로를 빠르게 탐색할 수 있을 정도로 정확합니다. 여기에서 주요 설계 결정은 단순함과 정확성을 절충하는 것입니다.

물론 정밀도가 부족하면 PUT에서 경로를 불완전하게 탐색하게 되지만 Eclipser는 Driller[12]에서와 같이 그레이 박스 concolic 테스트와 고전적인 그레이 박스 퍼징 사이를 번갈아 가며 이를 보완합니다. 그레이 박스 concolic 테스트가 PUT의 조건부 분기를 완전히 다루지는 않지만 그레이 박스 퍼징 모듈은 계속해서 새로운 경로와 분기를 다루며 그 반대의 경우도 마찬가지입니다. 우리는 실제로 이 설계 결정이 취약점을 찾고 높은 코드 범위에 도달한다는 점에서 현재의 최첨단 그레이 및 화이트 박스 fuzzer를 넘어 Eclipser의 기능을 효과적으로 확장한다는 것을 발견했습니다.

우리는 최신 fuzzer에 대해 Eclipser를 평가했습니다. 테스트 케이스 생성기로서의 우리 시스템의 실용성은 주어진 소스 코드에서 높은 커버리지로 테스트를 생성하는 데 탁월한 것으로 알려진 최첨단 기호 실행기인 KLEE에 대해 수행한 실험에 의해 확인되었습니다[18], [24]. 실험에서 Eclipser는 SMT 솔버의 도움 없이 테스트 케이스 생성 알고리즘을 평가하는 데 사용되는 잘 알려진 벤치마크인 GNU coreutils에서 KLEE보다 8.57% 더 높은 코드 적용 범위를 달성했습니다.

Eclipser를 버그 찾기 도구로 평가하기 위해 우리는 Eclipser를 AFLFast[5], LAF-intel[28], Steelix[6] 및 VUzzer[7]와 같은 여러 최첨단 그레이 박스 fuzzer와 비교했습니다.
우리는 또한 Debian 9.1에서 추출한 22개의 바이너리에서 Eclipser를 실행했고 17개 프로그램에서 40개의 고유한 버그를 발견했습니다. 발견한 모든 버그를 개발자에게 보고했습니다. 요약하면 이 논문은 다음과 같은 기여를 합니다.

1) 우리는 경량 계측을 활용하여 높은 커버리지 테스트 케이스를 생성하는 그레이 박스 concolic 테스트라고 하는 새로운 경로 기반 테스트 케이스 생성 알고리즘을 소개합니다.

2) Eclipser를 구현하여 AFLFast, LAF-intel, Steelix, VUzzer 등의 최첨단 fuzzer에 대한 다양한 벤치마크로 평가한다. 평가에 따르면 Eclipser는 코드 커버리지와 버그 발견 모두에서 그들에 비해 뛰어납니다.

3) 22개의 실제 Linux 애플리케이션에서 Eclipser를 실행했고 40개의 이전에 알려지지 않은 버그를 발견했습니다. 그 중 8개에 CVE 식별자가 할당되었습니다.
4) 오픈 사이언스를 위해 Eclipser의 소스 코드를 공개합니다: https://github.com/SoftSec-KAIST/Eclipser.

II . 배경 및 동기

A. 그레이박스 퍼징

퍼징은 본질적으로 생성된 테스트 케이스로 테스트 중인 프로그램(PUT)을 반복적으로 실행하는 프로세스입니다. 그레이 박스 퍼징[4]–[6], [29]은 피드백 루프 내에서 테스트 케이스를 발전시키며, 여기서 각 테스트 케이스에 대한 PUT의 실행은 적합성 함수라고 하는 기준에 의해 평가됩니다. 특정 구현이 다를 수 있지만 대부분의 그레이 박스 fuzzer는 적합성 기능으로 코드 적용 범위를 사용합니다. 예를 들어 AFL[4]은 분기 적용 범위(일부 노이즈 모듈로)를 사용하여 다음에 퍼지해야 하는 입력을 결정합니다.

최근의 성공에도 불구하고 커버리지 기반 그레이박스 퍼저는 퍼징 프로세스가 특정 분기를 실행하는 테스트 케이스를 찾기에 너무 많은 불필요한 시도를 포함한다는 주요 결점과 연결되어 있습니다. 이는 주로 퍼징에 사용되는 피트니스 함수의 둔감함 때문입니다.

비공식적으로 말하면, 피트니스 기능은 입력 값의 작은 수정으로 피트니스가 쉽게 변할 수 있는 경우 민감합니다. 노드 커버리지 및 분기 커버리지와 같은 코드 커버리지 메트릭은 true 및 false 브랜치를 커버하는 두 실행 사이에 중간 적합성이 없기 때문에 민감하지 않습니다. 따라서 주어진 분기 조건을 뒤집는 입력을 찾기가 어렵습니다.

민감한 적합도 기능의 필요성은 테스트 케이스 생성이 최적화 문제로 고려되는 검색 기반 소프트웨어 테스팅[30]에서 널리 인식되고 있습니다. 한 가지 주목할만한 적합성 함수는 조건부 분기의 피연산자 값 사이의 거리인 분기 거리[31], [32]입니다.

퍼징 커뮤니티는 최근 아이디어를 사용하기 시작했습니다. 앙고라[8]는 퍼징 성능을 개선하기 위해 분기 거리를 활용했습니다. Eclipser는 유사한 통찰력을 활용하지만 메타 휴리스틱에 의존하지 않고 감도를 사용하여 근사 분기 조건을 직접 추론하고 해결합니다. 두 접근 방식은 서로 직교하고 상호 보완적입니다.

B. 표기법 및 용어

우리는 실행을 유한한 명령어 시퀀스로 둡니다. 예를 들어 무한 루프가 있는 프로그램 실행은 고려하지 않습니다. fuzzer는 일반적으로 fuzzer의 매개변수인 특정 시간이 지나면 PUT을 강제 종료하기 때문에 fuzzing에서는 문제가 되지 않습니다. 우리는 $\sigma_p(i)$ 에 의해 입력 i로 프로그램 p의 실행을 나타냅니다 . 우리 모델에서 입력은 바이트 시퀀스이지만 비트 문자열을 나타내기 위해 쉽게 확장할 수 있습니다. 주어진 입력 i에 대해 i[n]을 i의 n번째 바이트 값이라고 합니다. i[n]을 $i[n \leftarrow v]$만큼 v가 되도록 수정하여 파생된 입력을 나타냅니다. 문서 전체에서 테스트 케이스와 테스트 입력이라는 용어를 같은 의미로 사용합니다. 입력 필드를 입력의 연속적인 하위 시퀀스라고 합니다. 주어진 입력에 대해 많은 입력 필드가 있을 수 있으며 입력 필드가 겹칠 수 있습니다.

대략적인 경로 제약. 기호 실행[19]에서 경로 제약 조건은 실행 경로가 실행 가능하면 해당 경로 조건이 충족되도록 입력에 대한 술어입니다. 우리의 접근 방식은 경량화를 시도하기 때문에 정확한 경로 조건을 추적하지 않고 대략적인 경로 제약 조건이라고 하는 대략적인 버전을 추적합니다.

```
1 int vulnfunc(int32_t intInput, char * strInput) {
2         if (2 * intInput + 1 == 31337)
8             if (strcmp(strInput, "나쁜!") == 0)
4                 크래시();
5     }
6 int main(int argc, char* argv[]) {
7         문자 버퍼[9];
8         int fd = open(argv[1], O_RDONLY);
9         읽기(fd, buf, sizeof(buf) - 1);
10        버퍼[8] = 0;
11        vulnfunc(*((int32_t*) &buf[0]), &buf[4]);
12        반환 0;
13    }
```

(a) C로 작성된 예제 프로그램. 오류 처리 루틴은 의도적으로 단순화를 위해 표시되지 않았습니다.

퍼저	버전	풀어 주다	클래스 바이너리 히트			시간
이클립서	1.0	2019년 5월 25일		✓	✓	0.64초
클리 [18]	1.4.0	2017년 7월 22일	G# #		✓	0.32초
LAF-인텔 [28]	8b0265	2016년 8월 23일	G#		✓	430초
AFL [4]	2.51b	2017년 8월 30일	G#			-
AFFast [5]	15894a	2017년 10월 28일	G#			-
AFLGo [11]	d650de	2017년 11월 24일	G#	✓✓		-

(b) 예제 프로그램에서 최신 fuzzer 간의 비교.
G# 및 #은 각각 그레이 박스 및 화이트 박스 방법론을 나타냅니다. 그만큼
다섯 번째 열은 fuzzer가 바이너리 코드를 처리할 수 있는지 여부를 보여줍니다. 여섯 번째 열은 fuzzer가 1시간 내에 충돌을 발견했는지 여부를 나타냅니다.

그림 1. 동기를 부여하는 예제와 다른 fuzzer의 비교.

씨앗. 이 논문에서 우리는 seed를 다음과 같은 데이터 구조로 둡니다.
특정 프로그램에 대한 입력을 나타냅니다. 우리는 종자를 나타냅니다
프로그램 p를 sp로, p를 시드로 실행
sp 를 σp(sp)로 표시합니다. 시드 sp 의 n번째 바이트는 sp[n] 으로 표시됩니다 .
시드의 모든 바이트는 "constr" 필드로 태그가 지정됩니다.
근사 경로 제약 조건의 독립적인 하위 집합입니다.
바이트와 관련하여. 대략적인 경로에 접근할 수 있습니다.
점 표기법을 사용 하는 시드 sp 의 n번째 바이트 제약 :
sp[n].constr. 주어진 시드 sp에 대해 시드 의 n번째 바이트
sp[n]은 sp[n].constr을 만족해야 합니다.
σp(sp) 와 동일한 실행 경로 .

C. 동기

그림 1a는 동기를 부여하는 예제 프로그램을 보여줍니다.
연구. 우리 시스템은 원시 바이너리 실행 파일에서 작동하지만 설명의 편의를 위해 C 표현을 사용합니다.
파일을 입력으로 받아 처음 4바이트를 사용합니다.
파일을 정수로, 나머지 4바이트를 5바이트 문자열로
끝에 NULL 문자를 추가합니다(10행). 이 두
값은 vulnfunc 함수에 대한 매개변수로 사용됩니다. ~ 안에
Line 4에서 충돌을 찾으려면 32비트를 제공해야 합니다.
정수 15,668 및 문자열 "Bad!" 함수에 대한 입력으로.
현재의 그레이박스 퍼저가 트리거하는 테스트 입력을 찾을 수 있습니까?
이 충돌? 그러한 회색 상자 fuzzers를 찾는 데 얼마나 효과적인지
단순한 버그? 이러한 질문에 답하기 위해 예제를 퍼징했습니다.
6개의 최첨단 fuzzer와 Eclipser를 사용한 프로그램
Intel Xeon E3-1231 v3의 단일 코어에서 각각 1시간 동안
프로세서(3.40GHz). AFL[4], AFLFast[5], AFLGo[11] 및
LAF-인텔1 [28]. 우리는 또한 인기있는 상징적 집행자를 선택했습니다.

1Steelix가 열려 있지 않기 때문에 Steelix [6] 대신 LAF-intel을 선택했습니다.
출처. Steelix를 LAF-intel의 개선된 버전으로 생각할 수 있습니다.

즉, 화이트 박스 fuzzer, KLEE [18]. 일부 주의
fuzzer, 즉 KLEE, LAF-intel 및 AFLGo는
소스 코드에. 따라서 소스와 함께 실행했지만
컴파일된 바이너리에서 다른 fuzzer를 실행했습니다. 예를 들어,
우리는 QEMU 모드에서 AFL을 실행했습니다[33]. AFLGo를 실행하기 위해 우리는
4행을 목표 위치로 지정하여 안내합니다.

그림 1b는 결과를 요약합니다. 모든 그레이박스 퍼저
LAF-intel이 버그가 있는 테스트 케이스를 찾지 못한 것을 제외하고. LAF-인텔
다중 바이트 비교를 분해하기 때문에 성공했습니다.
명령문을 여러 단일 바이트 비교로 변환하여 코드 적용 범위 메트릭을 입력 돌연변이에
효과적으로 민감하게 만듭니다.
그러나 LAF-intel은 Eclipser보다 671배 느렸습니다.
소스 기반 계측으로도 버그를 찾는데,
바이너리 레벨 계측보다 낮은 오버헤드를 수반합니다.

특히 결과는 KLEE와 비슷했습니다. 이클립서
버그를 찾는 데 KLEE보다 두 배 느리지만 Eclipser
KLEE는 소스가 필요한 반면 바이너리 코드에서 직접 실행됩니다.
암호. 또한 기호 실행이 빠르게 느려집니다.
SMT 때문에 더 많은 조건부 분기를 만나기 때문에
복잡한 경로 조건은 크게
Eclipser의 성능에 영향을 미칩니다. 실제로 Eclipser는
GNU coreutils에서 KLEE보다 훨씬 더 높은 코드 적용 범위
§ VC에서 논의하고 Eclipser가 확장할 수 있음을 보여줍니다.
§ VE에서 대규모 실제 응용 프로그램을 처리합니다.

이 예는 회색 상자 복통의 가능성을 강조합니다
테스트. 우리의 기술이 정확도를 손상시키는 동안
화이트박스 퍼징, 운동을 위한 테스트 케이스를 빠르게 생성
에 의존하지 않고 PUT의 다양한 고유 실행 경로
모든 고비용 분석.

III. GREY-BOX CONCOLIC 테스트

Grey-box concolic testing은 테스트 케이스를 생성하는 방법입니다.
주어진 시드 입력에서. 높은 수준에서는 유사하게 작동합니다.
세대 검색을 사용한 동적 기호 실행
전략 [19], [23], 여기서 시드가 있는 PUT 실행
가능한 모든 것을 확장하여 테스트 케이스 생성
실행 경로의 분기 조건. 회색 상자 복통
테스트는 유사한 방식으로 작동하지만 선택적으로 해결합니다.
의존하지 않는 동안 경로에서 발생한 분기 조건
SMT 해결에 대해.

우리 접근 방식의 핵심 측면은 각 입력마다 대략적인 경로 제약 조건의 독립적인 하위 집합을 유지하는 것입니다.
시드의 바이트. 제약 조건은 고유한 테스트 사례를 생성하는 데 도움이 됩니다.
동일한(또는 유사한) 실행을 실행하는 데 사용할 수 있는
제약 조건을 해결하여 PUT의 경로. 그런 시험으로
경우에 따라 조건부 분기 중 일부가
경로는 고유한 입력 값을 비교합니다.
동일한 실행 경로. 우리는 그러한 실행 행동을 사용합니다
회색 상자 방식으로 조건부 분기를 관통합니다. 우리의
이 기술은 시스템을 유지하면서 화이트 박스 퍼징(예: concolic 테스트)과 같은 분기
조건을 효과적으로 해결합니다.
그레이 박스 퍼징(gray-box fuzzing)처럼 가볍고 확장 가능합니다.

가. 개요

Grey-box concolic testing은 네 가지 주요 기능인 SPAWN, IDENTIFY, SELECT 및 SEARCH로 작동합니다. 핵심

알고리즘 1: Grey-box Concolic Testing.

```
1 기능 GreyConc(p, sp, k)
2     pc ← {} // 대략적인 경로 제약 조건
3     싸앗 ← ∅
4     execs ← SP AW N(p, sp, k)
5     conds ← ID ENTIF Y(p, execs) for
6     cond in SE LEC T(conds) do c ← SE
7         ⁰초,_ ARC H(p, k, pc, exec, cond)
8         싸앗 ← 싸앗 + s p
9         pc ← pc ∧ c // 두 제약 조건 병합
10    반환 싸앗
```

Gray-box concolic 테스트의 비율은 이러한 기능을 사용하여 알고리즘 1로 표현됩니다.

SPAWN (p, sp, k) → 실행
SPAWN 은 프로그램 p, 시드 sp 및 바이트 오프셋 k를 입력으로 받습니다. 먼저 sp 의 k번째 바이트를 수정하여 Nspawn 고유 입력 세트를 생성합니다 . 여기서 Nspawn 은 사용자 매개변수입니다. 그런 다음 생성된 입력으로 p를 실행하고 실행(execs)을 반환합니다(§ III-C 참조).

IDENTIFY (p, execs) → conds
IDENTIFY 는 프로그램 p와 실행 집합(execs)을 입력으로 받습니다. k번째 입력 바이트의 영향을 받는 일련의 조건문(conds)을 식별합니다(§ III-D 참조).

SELECT (conds) → conds0
SELECT 는 주어진 조건문의 시퀀스에서 하위 시퀀스를 반환합니다. 현재 Eclipser 구현에서 이 단계는 Nsolve 가 사용자 매개변수(§ III-E 참조) 인 경우 무작위로 선택된 최대 Nsolve 조건문의 하위 시퀀스를 반환합니다.

검색 (p, k, pc, exec, cond) → s
SEARCH 는 주어진 조건문 cond를 관통하려고 하고 cond에서 새 분기를 실행할 수 있는 새 시드 s를 반환합니다. ⁰cp,_ 중(제약 조건을 만족하지 않는 분기입니다. 제약 조건 c는 현재 실행 σp(sp)를 따르기 위한 입력 조건을 나타냅니다.

생성된 시드는 σp(sp) 와 동일한 실행을 cond까지 수행하고 cond 에서 반대 분기를 실행합니다(§ III-F 참조).

높은 수준에서 그레이 박스 concolic 테스트는 프로그램 p, 시드 입력 sp 및 바이트 위치 k를 입력으로 사용하고 σp(sp) 와 다른 실행 경로를 포함하는 테스트 케이스 세트를 출력합니다 . 일반적인 concolic 테스트와 달리 우리의 접근 방식은 관심 있는 입력 바이트 위치를 지정하기 위해 추가 매개변수 k를 사용합니다. 이는 오프셋 k에 위치한 단일 입력 필드에만 초점을 맞춰 그레이 박스 concolic 테스트 프로세스를 단순화하기 위한 것입니다. 우리의 초점은 단일 입력 필드에 있지만, 우리의 전략은 한 번에 하나의 입력 필드에 대해 만족스러운 할당을 찾을 수 있기 때문에 조건이 여러 입력 필드의 영향을 받는 조건부 분기를 통과하는 것이 여전히 가능합니다. 또한 SEARCH 가 만족할 만한 솔루션을 찾지 못하더라도 Eclipser는 임의의 돌연변이를 수행하여 오류를 보상합니다(§ IV). 이러한 경우를 일반적인 방식으로 처리하는 것은 이 문서의 범위를 벗어납니다.

변수 pc는 대략적인 경로 제약 조건을 나타냅니다.

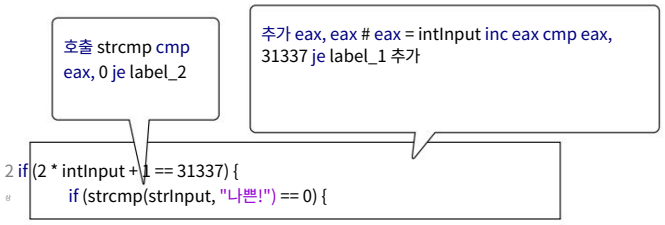


그림 2. 실행 중인 예제 스니펫.

실행 σp(sp). 구체적으로 말하면, pc는 sp 의 바이트에서 해당 바이트에 대한 독립 제약 조건으로의 맵이며, 이는 처음에 알고리즘 1의 2행에 있는 빈 맵입니다. 대략적인 경로 제약 조건은 실행에서 조건문을 만나면 커집니다. 이 데이터 구조는 [18], [34]에서 사용된 독립적인 공식에서 영감을 얻었습니다.

Grey-box concolic 테스트는 실행의 모든 비교 명령을 계측하지만 제약 조건 pc를 구축하기 위해 라인 6에서 이들 중 일부만 선택하여 대략적인 경로 제약 조건을 생성합니다. 선택한 각 조건문에 대해 해당 공식을 pc에 추가합니다(9행). 이 프로세스는 경로 제약 조건의 대략적인 하위 집합을 유지한다는 점을 제외하고 동적 기호 실행과 동일합니다.

나. 예

우리의 기술을 설명하기 위해 § II-C의 동기 부여 예를 다시 살펴보겠습니다. 그림 2는 예제에서 가져온 코드 조각과 해당 바이너리 코드를 보여줍니다. (1) 초기 시드 파일 sp 가 8개의 연속 0으로 구성되고, (2) Nspawn 이 3으로 설정되고, (3) 현재 오프셋 k가 0이라고 가정합니다.

Eclipser는 § IV에서 설명한 대로 퍼징 캠페인 전체에서 이 오프셋 k를 이동하여 작동합니다.

SPAWN 이 세 개의 입력 sp[0 ← 10], sp[0 ← 50] 및 sp[0 ← 90]을 생성하고 입력으로 p를 실행하여 σp(sp[0 ← 10]), σp(sp[0 ← 50]) 및 σp(sp[0 ← 90]). 그런 다음 IDENTIFY 는 실행에서 첫 번째 cmp 명령이 정수 31,337을 eax의 21, 101 및 181의 세 가지 다른 값과 비교하는 것을 관찰합니다.

3개의 중첩 실행 접두어에서 IDENTIFY 는 비교 명령어 쌍과 다음 조건부 점프 명령어를 반환합니다. 다음으로, SELECT 는 쌍을 취하고 고려할 항목이 하나만 있으므로 단순히 반환합니다. 마지막으로 SEARCH 는 중첩 실행에서 세 값(10, 50, 90)과 해당 비교 값(21, 101, 181) 간의 관계를 확인합니다. 이 경우 SEARCH 는 eax = 2 × sp[0] + 1과 같은 선형 관계를 추론합니다.

이 방정식을 풀면 첫 번째 조건을 만족하는 intInput의 값인 15,668(0x3d34)을 얻습니다.

그러나 솔루션은 1바이트에 맞지 않습니다. 따라서 첫 번째 바이트(k = 0이므로)와 인접 바이트를 포함하는 해당 입력 필드의 크기를 유추해야 합니다. 우리는 크기 2부터 시작하여 최대 8바이트의 입력 크기를 고려합니다. 이 경우 2바이트 솔루션이 작동하고 테스트 케이스를 생성하는 데 사용됩니다(s 2바이트 sp, 결과적으로 다음 8바이트 파일

⁰피) 첫 번째를 대체하여

16진수 표현: 34 3d 00 00 00 00 00 00. SEARCH 는 이 입력으로 PUT을 실행하여 조건 분기를 통과할 수 있는지 확인합니다. 새 분기를 실행할 수 있으므로 이 분기에 대한 대략적인 경로 제약 조건을 포함하는 생성된 시드를 반환합니다. {sp[0] 7→ [0x34, 0x34], sp[1] 7→ [0x3d, 0x3d]}, 여기서 대괄호는 닫힌 간격을 나타냅니다. § IV-C에서 대략적인 경로 제약 조건을 인코딩하는 방법을 설명합니다.

Eclipser는 이제 k를 증가시키면서 s 새 시드를 사용하여 위의 프로세스를 반복합니다. k = 4일 때 SPAWN 은 σp(s [4 ← 50]) 및 다섯 번째 입력 바이트(k = 4)와 eax 사이의 세 가지 실행을 반환합니다 . $s'_{피}$ [4 ← 10]), σp($s'_{피}$ [4 ← 90]). IDENTIFY 는 서신을 찾습니다.

SEARCH 는 eax 값이 단조롭다는 것을 알아냅니다[4]. k번째 입력 바이트를 변경하여 s에 대해 증가 $s'_{피}$ 이진 검색을 수행하고 입력 바이트가 0x42('B')에서 0x43('C')으로 변경될 때 eax가 -1에서 1로 변경됨을 찾습니다. eax를 0으로 만드는 솔루션을 찾지 못했기 때문에 입력 필드 크기를 1 확장하고 0x4200과 0x4300 사이에서 다른 이진 검색을 수행합니다. PUT이 조건문의 진정한 분기를 실행하도록 하는 솔루션 "Bad!"를 찾을 때까지 이 프로세스를 반복합니다. 마지막으로 SEARCH 는 "Bad!" 문자열을 포함하는 시드를 생성합니다.

C. 스폰

SPAWN 은 sp[k].constr 제약 조건에 따라 시드 sp 의 k번째 바이트를 변경하여 테스트 입력을 생성하고 생성된 입력과 관련하여 p 실행을 반환합니다.

여기서 주요 목표는 σp(i1) ≈ σp(i2) ≈ · · ≈ σp(iN)가 되도록 N 테스트 입력 세트 {i1, i2, · · · iN } 를 생성하는 것 입니다.
SMT 솔버를 사용하여 이러한 입력을 찾는 것은 실제로 가능하지만 설계 목표 중 하나는 대략적인 경로 제약 조건을 가벼운 방식으로 해결할 수 있다는 점을 기억하십시오.

Eclipser는 간격을 사용하여 대략적인 경로 제약 조건을 나타냅니다(§ IV-C 참조). 따라서 대략적인 경로 제약 조건을 충족하는 입력을 찾는 것은 간격 내에서 값을 선택하는 것만큼 쉽습니다. 제약 조건 sp[k].constr이 기호 실행에서와 같이 정확하다면 PUT의 정확히 동일한 경로를 실행하는 데 사용할 수 있는 고유한 테스트 입력을 항상 생성할 수 있습니다. 즉, σp(i1) = σp(i2) = · · · = σp(iN). 그러나 우리의 접근 방식은 sp[k].constr의 불완전성으로 인해 실제 경로 제약 조건을 충족하지 않는 잘못된 입력을 생성할 수 있습니다 . 우리

IDENTIFY 의 초점 이 겹치는 실행 접두사에 있으므로 이것은 심각한 문제가 아닙니다 .

Nspawn 으로 SPAWN 에서 반환할 최대 실행 횟수를 나타냅니다 . 즉, N = Nspawn 입니다. 분석가가 구성할 수 있는 매개변수입니다. Eclipser의 현재 구현에서는 이 값을 기본적으로 10으로 설정합니다. 이 값은 § VB의 경험적 연구를 기반으로 선택됩니다. SPAWN 은 주어진 시드에 대해 PUT Nspawn 시간을 실행하는 반면 기존 기호 실행은 PUT을 한 번만 실행합니다. 이것은 확장 가능한 fuzzer를 설계하기 위해 수용해야 하는 주요 절충안입니다.

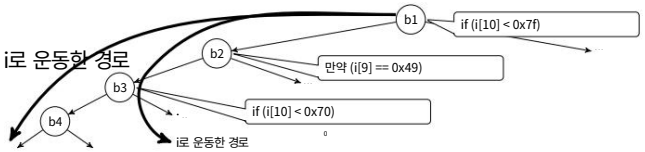


그림 3. A CFG 여기서 b1, . . . , b4 는 조건부 분기입니다. 두 가지 실행) 조건 분경로, σp(i) 및 σp(i 오른 ^{0}기 b3에서 분기합니다. 왼쪽과
쪽 분기는 각각 참 및 거짓 분기에 해당합니다.

D. 식별

IDENTIFY 의 주요 목표는 오프셋 k의 입력 바이트와 σp(sp) 의 조건문 사이의 일치성을 결정하는 것 입니다. sp[k] 의 영향을 받는 모든 조건문을 포함하는 σp(sp) 의 하위 시퀀스를 반환합니다 .

목표를 달성하기 위해 세분화된 오염 분석을 사용할 수 있습니다.
그러나 각 입력 바이트에 식별자를 할당하고 주어진 입력의 영향을 받는 모든 표현식에 대해 이러한 ID 집합을 유지하기 때문에 메모리가 부족한 프로세스입니다. Fine Grained taint 분석의 공간 효율성을 줄이기 위한 여러 연구가 있지만 [35], [36] 집합 요소 간에 상당한 중첩이 있다고 가정합니다. 또한 오염 분석은 PUT의 모든 단일 명령을 계측하므로 계산 비용이 많이 들고 퍼징에 너무 느릴 수 있습니다.

PUT을 여러 번 실행하는 간단하고 확장 가능한 접근 방식을 사용합니다. SPAWN 은 s의 k번째 바이트를 변경하여 생성된 테스트 입력을 기반으로 Nspawn 실행을 반환합니다 . 실행의 동작 차이를 관찰하여 실행에서 k번째 바이트와 조건부 분기 간의 대응 관계를 식별할 수 있습니다.

특히, 먼저 겹치는 실행 접두사의 동일한 위치에서 조건문 세트를 추출합니다. 그런 다음 b 결정의 차이를 관찰하여 조건문 b가 시드의 k번째 바이트에 의해 영향을 받는지 여부를 결정합니다. 이 간단한 접근 방식은 실행의 어떤 조건부 분기가 입력 바이트의 영향을 받는지에 대한 민감한 피드백을 제공합니다.

부분적으로 겹치는 실행이 항상 있을 수 있으므로 대략적인 경로 제약 조건의 부정확성은 여기에서 문제가 되지 않습니다. 또한 SPAWN 은 돌연변이에 의해 입력을 생성하기 때문에 생성된 실행 중 일부는 완전히 다른 실행 경로를 실행하여 PUT의 흥미로운 경로를 커버할 수 있습니다. Eclipser는 이러한 부산물로부터 이점을 얻을 수 있습니다.

그림 3은 오프셋 10에서 바이트 값만 다른 두 개의 입력 i와 i로 프로그램 p를 실행하는 경우를 보여 줍니다 . 실행 σp (i) 및 σp(i 는 b1 및 b3 에 대한 비교 값이 실행에서 다

^{0}). 이 예에서 우리는 다음을 관찰할 수 있습니다.

따라서 우리는 11번째 입력 바이트(i[10] 및 i
^{0}[10]) b1 및 b3에 대응합니다.

마. 선택

IDENTIFY 동안 처리할 조건문이 너무 많을 수 있습니다. 이 현상은 종종 언급됩니다.

```
1 # mov ebx, f(입력) 2 cmp ebx, 20 3
je 레이블
```

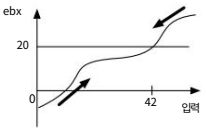


그림 4. 모노토닉 입출력 관계.

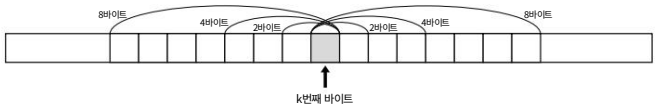


그림 5. 입력 필드 인식.

동적 기호 실행에서 경로 폭발 문제로.
예를 들어, 다음 for 루프를 고려하십시오. 여기서 inp는 사용자 제공 입력을 나타냅니다. for (i=0; i<inp; i++) { /* 생략 */ }

이 경우 사용자 입력에 따라 임의의 수의 조건문이 발생할 수 있습니다. IDENTIFY에서 반환된 모든 단일 명령문을 처리하는 경우 시스템은 주어진 시간 동안 흥미로운 경로를 탐색하지 않을 수 있습니다.

이 문제에 대처하기 위해 SELECT 는 표시 순서를 유지하면서 주어진 조건문의 시퀀스에서 Nsolve 조건문을 무작위로 선택 합니다. 프로그램 실행을 따라 대략적인 경로 제약 조건을 구축해야 하기 때문에 순서는 동일하게 유지되어야 합니다. Eclipser의 현재 구현에서는 경험적으로 결정되는 Nsolve = 200을 사용합니다(§ VB).

Sage [19] 및 KLEE [18]와 같은 동적 기호 실행 프로그램도 동일한 문제를 처리하기 위해 여러 경로 선택 휴리스틱을 사용합니다.

F. 검색

SEARCH 는 새 분기를 포함하도록 분기 조건을 해결합니다.
주어진 조건문에서 cond. 결과적으로 현재 실행 경로 σp(sp)를 따르기 위해 간격(§ IV-C)으로 근사한 분기 조건과 함께 새 시드를 반환합니다. 여기서 주요 과제는 SMT 솔버의 도움 없이 대략적인 경로 제약 조건을 해결하는 것입니다.

IDENTIFY 는 k번째 입력 바이트와 관계가 있는 조건문을 반환 한다는 점을 기억 하십시오. 이 관계를 데이터 흐름 추상화로 나타낼 수 있습니다. 여기서 sp[k]는 입력이고 각 조건문의 피연산자 중 하나는 출력입니다. SEARCH 의 핵심 직관은 이러한 입력-출력 관계를 실현함으로써 대략적인 경로 제약의 잠재적인 솔루션을 추론할 수 있다는 것입니다.

특히 SEARCH 는 입력 출력 관계가 선형 또는 단조인 경우에 중점을 둡니다. 이 설계 선택은 우리 자신의 경험적 관찰뿐만 아니라 다양한 이전 연구 [37]-[39]에 의해 뒷받침됩니다. 우리는 실제 프로그램의 많은 조건 분기가 선형 또는 단조 제약 조건을 갖는 경향이 있음을 관찰했습니다(§ V-C1 참조).

SEARCH 는 (1) 현재 분기 조건 공식화 및 해결(§ III-F1), (2) 해당 입력 필드 인식 (§ III-F2), (3) 침투할 수 있는 새 시드 생성의 세 단계로 실행됩니다. 조건문(§ III-F3).

1) 분기 조건 풀기: cond의 두 피연산자 중 하나만 입력 i의 영향을 받고 피연산자가 oprnd(i)로 표시된다고 wlog를 가정해 보겠습니다. cond의 분기 조건이 선형이라고 결정할 수 있습니다.

i1, i2, i3 가 존재한다면 $\frac{\text{oprnd (i1)-oprnd(i2)}}{\text{i1 i2}} = \frac{\text{oprnd (i2) oprnd (i3)}}{\text{i2 i3}}$. 이 경우 선형 방정식이나 부등식을 직접 구성하고 해결할 수 있습니다. 반면, cond는 oprnd가 cond를 실행한 관찰된 모든 입력 i1, i2, ...에 대해 단조 in (n ≥ 3) 함수인 경우 단조 분기 조건을 갖습니다. 그림 4는 2바이트 입력 필드 (ebx) 사이에 단조로운 입출력 관계가 있는 예를 보여줍니다. 이러한 단조로운 관계(input 값과 output 값 사이)에 대해 솔루션을 찾기 위해 이진 검색을 수행합니다.

2) 입력 필드 인식: 지금까지 우리의 초점은 입력 바이트, 즉 sp[k]에 있었습니다. 그러나 많은 분기 조건은 입력 바이트뿐만 아니라 입력 필드(예: 32비트 정수 또는 64비트 정수)에 의해 제한됩니다.

이는 SEARCH 가 임의 크기의 입력 필드를 처리할 수 있어야 함을 의미합니다. 게다가 SEARCH 의 방정식 풀이 는 임의 정밀도 정수에서 작동하므로 바이트에 맞지 않는 솔루션을 제공할 수 있습니다. 우리는 더 많은 입력 후보로 PUT을 실행함으로써 자연스럽게 SEARCH 의 기능을 확장할 수 있습니다. 구체적으로, 특정 크기의 솔루션을 고려하면서 얻은 솔루션으로 seed를 교체합니다. 선형 방정식이나 부등식을 풀 때 그림 5에서 설명하는 것처럼 가능한 모든 후보를 시도하기 위해 최대 7개의 경우를 고려합니다. 단조로운 조건에 대한 이진 검색의 경우 입력 필드의 크기를 1로 고려하여 검색을 시작한 다음 현재 구현에서 8로 설정되는 임계값까지 크기를 점차 증가시킵니다.

3) 시드 생성: 새 경로를 실행하는 새 시드를 생성하려면 먼저 현재 분기의 제약 조건을 근사화하고 새로 생성된 시드의 constr 필드로 인코딩해야 합니다. 특히, 분기 조건을 사전 c로 변환하여 입력 바이트 위치 i를 간격으로 표시되는 근사 제약 조건 c[i]에 매핑합니다. c의 모든 바이트 위치 i에 대해 sp[i].constr을 ¬c[i] ∧ pc[i]로 업데이트합니다. 여기서 ∧는 두 간격의 결합을 나타냅니다. sp[i] 의 구체적인 값 도 간격 ¬c[i] 내에 있는 값으로 업데이트됩니다. 현재 실행에서 가지 않는 경로를 따르기를 원하기 때문에 분기 조건 ¬c[i] 각각을 부정합니다. 즉, 새 시드가 PUT으로 실행될 때 반대 분기를 가져와야 합니다. SEARCH 는 c를 반환하고 이를 사용하여 pc를 구축합니다.

발견된 분기 조건을 근사화하는 방법에 대한 자세한 내용은 § IV-C를 참조하십시오.

IV. 이클립서 아키텍처

그레이 박스 concolic 테스트 자체가 주어진 시드 sp 와 바이트 위치 k에서 p에 대한 체계적인 테스트 케이스 생성을 가능하게 하지만, 다양한 바이트 위치와 다양한 바이트 위치로 그레이 박스 concolic 테스트를 실행하는 방법을 고안해야 합니다.

알고리즘 2: Eclipser의 주요 알고리즘.

```
// p: PUT, seed: 초기 seed, t: 시간 제한 1 function Eclipser(p, seed, t)
2 Q ← InitQueue(시드)
3 T ← ∅
4   getTime() < 하는 동안
5       RG, RR ← 일정()
6       Q, T ← GreyConcolicLoop(p, Q, T, RG)
7       Q, T ← RandomFuzzLoop(p, Q, T, RR)
8   리턴 T
```

흥미로운 경로를 탐색하기 위해 씨앗. 이 섹션에서는 Eclipser 설계에서 이러한 문제를 해결하는 방법을 설명합니다.

A. 주요 알고리즘

§III-F에서 회상하면 현재 회색 상자 concolic 테스트는 선형 및 단조 제약 조건에 초점을 맞추고 있으며 여러 입력 필드가 포함된 일부 복잡한 분기 조건을 처리하지 못할 수 있습니다. 이러한 문제에 대처하기 위해 Eclipser는 고전적인 그레이 박스 퍼징 전략을 사용합니다. 우리의 목표는 그레이 박스 concolic 테스트와 그레이 박스 퍼징을 교대로 사용하여 둘 다의 능력을 최대화하는 것입니다. 퍼징 전략을 번갈아 사용한다는 아이디어는 이전에 제안된 바 있으며 [12], [40], [41], 우리와 보완적입니다.

알고리즘 2는 Eclipser의 전체 절차를 설명합니다.
Eclipser는 프로그램 p, 시간 제한 t 및 초기 시드 시드 세트를 입력으로 받아 퍼징 캠페인 동안 생성된 테스트 케이스 T 세트를 반환합니다. Eclipser는 먼저 제공된 초기 시드 시드로 우선순위 큐 Q를 초기화하고 시간 제한 t가 만료될 때까지 while 루프에서 실행합니다. 5행에서 Schedule은 RG(gray-box concolic testing) 및 RR(gray-box fuzzing)을 위한 리소스를 할당합니다. 그런 다음 두 가지 퍼징 전략, 즉 그레이 박스 con colic 테스팅(GreyConcolicLoop)과 그레이 박스 퍼징(RandomFuzzLoop)은 할당된 모든 리소스를 사용할 때까지 새로운 테스트 케이스를 교대로 생성합니다. 자원 관리에 대한 자세한 내용은 §IV-B를 참조하십시오. Eclipser는 GreyConcolicLoop 및 RandomFuzzLoop에서 Q 및 T를 업데이트합니다. 단순히 새로 생성된 테스트 케이스, 즉 시드를 Q 및 T에 각각 추가합니다. T는 나중에 퍼징 캠페인이 끝나면 메인 알고리즘에 의해 반환됩니다(라인 8).

우선 순위 대기열. 생성된 각 테스트 입력에 대해 Eclipser는 코드 적용 범위를 기반으로 적합성을 평가하고 이를 Q에 추가합니다. 특히, 우리는 모든 새 노드를 포함하는 시드에 높은 우선순위를 부여하고 새 경로를 포함하는 시드에 낮은 우선순위를 부여합니다. 코드 적용 범위를 개선하지 않는 시드를 삭제합니다.

Eclipser는 사용할 k의 다음 값과 함께 시드를 큐에 삽입합니다. Eclipser는 현재 k를 k 1 및 k + 1로 만들고 두 위치에서 시드를 두 번 푸시합니다.

우선 순위 대기열의 한 가지 중요한 측면은 두 가지 퍼징 전략이 시드를 공유할 수 있다는 것입니다. 그레이 박스 concolic 테스트는 현재 새로운 테스트 케이스를 생성할 때 주어진 시드의 크기를 확장하지 않는 반면 그레이 박스 퍼징은 가능합니다. 그레이박스 퍼징 모듈이 길이를 확장하여 흥미로운 시드를 생성하면 우선순위 큐 Q를 통해 그레이 박스 concolic 테스트 모듈과 공유됩니다.

B. 리소스 스케줄링

두 퍼징 전략을 번갈아 사용할 때 각 전략에 얼마나 많은 리소스를 할당해야 하는지 결정해야 합니다. Eclipser에서 리소스는 허용된 프로그램 실행 수입니다. 전략이 허용된 수보다 PUT을 더 많이 실행하면 Eclipser가 전략을 전환합니다. 전환 시기를 결정하기 위해 Eclipser는 각 퍼징 전략의 효율성을 평가하고 효율성에 비례하여 시간을 할당합니다. Nexec 를 알고리즘 2의 4행에 있는 while 루프의 1회 반복에 대한 총 프로그램 실행 횟수라고 합시다 . 효율성 f = Npath/Nexec를 정의합니다. 여기서 Npath는 새 실행 경로를 실행한 고유한 테스트 케이스의 수입니다. 즉, Eclipser는 더 많은 새로운 경로를 탐색하는 전략에 더 많은 리소스를 할당합니다.

C. 대략적인 경로 제약

그레이 박스 concolic 테스트는 간격으로 경로 제약 조건을 근사화한다는 것을 기억하십시오. 대략적인 경로 제약 조건은 입력 바이트에서 해당 간격 제약 조건으로의 맵입니다. 각 제약 조건을 닫힌 간격으로 나타냅니다. [l, u]를 제약 조건 l ≤ x ≤ u라고 합니다. 그런 다음 두 구간의 교집합으로 두 제약 조건의 논리적 결합을 표현할 수 있습니다. [l1, u1] ∧ [l2, u2] = [max(l1, l2), min(u1, u2)].

SEARCH 가 분기 조건을 해결 했다고 가정해 보겠습니다.
n 바이트 입력 필드 x와 연관되어 결과적으로 동일 조건 x = k를 얻었습니다. 이 조건은 정밀도 손실 없이 각 바이트에 대한 간격으로 표현될 수 있습니다. {x0 7→ [k0, k0], x1 7→ [k1, k1], · · · xn 1 7→ [kn 1, kn 1]}, 여기서 ki = (k (8 * i)) & 0xff 및 x0, xn 1 은 각각 x의 최하위 바이트와 최상위 바이트입니다.

해결된 분기 조건이 부등식 조건 l ≤ x ≤ u라고 가정합니다. 이 경우 조건은 x의 최상위 바이트에 대한 간격 제약으로 근사됩니다: {xn 1 7→ [ln 1, un 1 + 1]}. "정수" 유형으로 표시되는 간격을 과도하게 근사하기 위해 여기에서 가장 중요한 바이트만 선택합니다. Eclipser는 요소별 결합을 수행하여 알고리즘 1의 9행에서 pc에 이 근사 제약 조건을 추가합니다.

D. 시행

우리는 4.4k 라인의 F# 코드로 Eclipser의 메인 알고리즘을 구현하고 QEMU(2.3.0) [33]에 800라인의 C 코드를 추가하여 Eclipser의 바이너리 계측 로직을 구현했습니다.
우리는 본질적으로 AFL의 단순화된 버전인 F#으로 Eclipser의 그레이 박스 퍼징 모듈을 작성했습니다[4]. 우리는 AFL에서 사용되는 돌연변이 연산과 퍼징 캠페인 동안 시드 수를 최소화하기 위해 greedy set-cover 알고리즘 [14], [42]을 사용했습니다. 바이너리 실행에서 실행 피드백을 얻기 위해 다양한 아키텍처를 처리하도록 Eclipse를 쉽게 확장할 수 있는 QEMU 사용자 모드 에뮬레이션을 사용했습니다. 현재 Eclipser는 널리 사용되는 세 가지 아키텍처인 x86, x86-64 및 ARMv7을 지원합니다. Eclipser 구현은 GitHub에서 공개적으로 사용할 수 있습니다. https://github.com/SoftSec-KAIST/Eclipser.2

2ARMv7 버전은 IP 문제로 인해 오픈 소스가 되지 않습니다.

V. 평가

다음 질문에 답하기 위해 Eclipser를 평가했습니다. 1) Eclipser의 구성 매개변수가 성능에 어떤 영향을 줍니까? (§ VB)

2) 그레이 박스 concolic 테스트가 일반적인 테스트 케이스 생성 알고리즘이 될 수 있습니까? 그렇다면 기존 화이트박스 퍼저와 비교하면 어떤가요? (§ VC)
3) Eclipser가 버그를 찾는 데 있어 최첨단 그레이 박스 fuzzers를 이길 수 있습니까? (§ VD)
4) Eclipser가 실제 애플리케이션에서 새로운 버그를 찾을 수 있습니까? 그레이 박스 concolic 테스트는 그렇게 크고 복잡한 프로그램을 처리할 만큼 충분히 확장 가능합니까? (§ VE)

A. 실험 설정

64개의 VM으로 구성된 프라이빗 클러스터에서 실험을 실행했습니다. 각 VM에는 단일 Intel Xeon E5-2699 V4(2.20GHz) 코어와 8GB 메모리가 장착되었습니다. 우리는 세 가지 벤치마크에서 실험을 수행했습니다. (1) GNU coreutils 8.27의 95개 프로그램; (2) LAVA-M 벤치마크의 4개 프로그램; (3) Debian 9.1 패키지에 포함된 22개의 실제 프로그램.

먼저 KLEE [18] 및 기타 화이트 박스 fuzzer [25], [27]가 성능을 평가하기 위해 이 벤치마크를 사용하기 때문에 우리는 Eclipser와 KLEE를 비교하기 위해 GNU coreutils를 선택했습니다. 둘째, 기존의 많은 fuzzer[6],[7],[10]를 평가하는 데 사용되는 LAVA-M 벤치마크[43]에서 Grey-box fuzzer에 대한 Eclipser의 버그 찾기 기능을 평가했습니다. 마지막으로, Eclipser의 실질적인 영향을 측정하기 위해 Debian 9.1에서 선택한 실제 응용 프로그램을 퍼징했습니다.

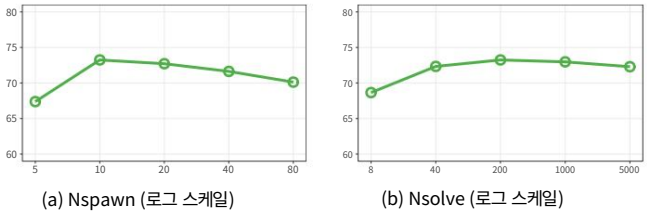
비교 대상. 우리는 비교를 위해 AFLFast[5]와 LAF-intel[28]과 같은 두 개의 기존 그레이 박스 fuzzer를 선택했습니다. ELF 바이너리에 대한 현재 지원이 제한되어 있으므로 Driller [12]를 생략했습니다. VUzzer[7]는 상용 제품인 IDA pro에 의존하기 때문에 실행할 수 없었습니다. Steelix[6], T-Fuzz[10], Angora[8]도 공개되지 않았기 때문에 생략했습니다.

B. Eclipser 구성

§ III에서 회상하면 Eclipser는 Nspawn 및 Nsolve의 두 가지 사용자 구성 가능한 매개변수를 사용합니다. 이 매개변수는 각각 그레이 박스 concolic 테스트로 식별하고 침투할 분기의 수를 결정합니다. 매개변수의 영향을 추정하기 위해 첫 번째 벤치마크(coreutils 8.27)의 각 프로그램에서 한 시간 동안 다양한 구성과 측정된 코드 적용 범위 차이로 Eclipser를 실행했습니다. 특히 각 매개변수에 대해 기하급수적으로 증가하는 5개의 값을 선택했습니다.

그림 6은 결과를 요약한 것입니다. Nspawn 이 너무 작으면 IDENTIFY 가 흥미로운 조건부 분기를 식별하지 못하고 결과적으로 적용 범위가 줄어들었지만 Nspawn 이 너무 크면 Eclipser가 불필요한 프로그램 실행에 너무 많은 시간을 소비하게 되었습니다. 마찬가지로 Nsolve를 너무 작게 만들면 Eclipser 에서 몇 가지 흥미로운 조건부 분기를 놓치기 시작했지만 너무 크게 만들면 경로 폭발로 인해 더 적은 노드를 다루기 시작했습니다.

이 결과에서 Nspawn = 10 및 Nsolve = 200을 Eclipser의 기본 매개변수 값 세트로 사용하기로 결정하고 나머지 실험에 사용했습니다.



(a) Nspawn (로그 스케일) (b) Nsolve (로그 스케일)

그림 6. Nspawn 과 Nsolve의 영향.

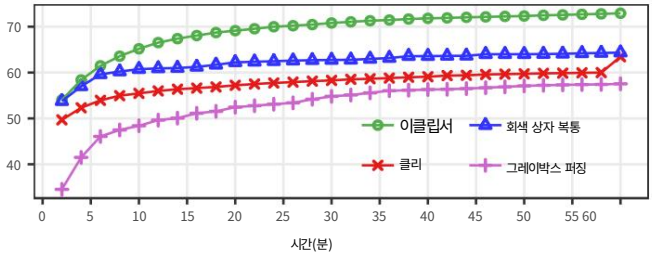


그림 7. coreutils에 대해 시간 경과에 따른 Eclipser 및 KLEE에 의해 달성된 라인 커버리지.

C. 화이트박스 퍼징과의 비교 테스트 케이스 생성 알고리즘으

로서 그레이박스 concolic 테스트의 효율성을 평가하기 위해 작성 당시 최신 버전인 KLEE 버전 1.4.0과 비교했다. KLEE [18]의 원본 문서에서 사용된 coreutils를 벤치마크로 선택했습니다. coreutils 8.27의 107개 프로그램 중 kill 및 rm과 같이 퍼징 프로세스 자체에 영향을 줄 수 있는 8개 프로그램과 KLEE에서 처리되지 않은 예외를 발생시킨 4개 프로그램을 제외했습니다. 나머지 95개 프로그램을 각각 1시간 동안 테스트했습니다. 또한 KLEE 웹사이트[44]에 보고된 명령줄 옵션을 사용하여 KLEE를 실행했습니다. 공정한 비교를 위해 Eclipse를 실행할 때 입력 크기에 동일한 제한을 설정했습니다. 여기에 보고된 모든 숫자는 8회 반복에 대한 평균입니다.

우리는 여기에서 세 가지 질문에 답하려고 합니다. (1) 그레이 박스 퍼징 모듈 없이 그레이 박스 concolic 테스트 자체가 코드 커버리지 측면에서 KLEE를 능가할 수 있습니까? (2) 그레이 박스 퍼징과 그레이 박스 concolic 테스트를 번갈아 가며 사용할 수 있습니까? (3) Eclipser가 coreutils에서 현실적인 버그를 찾을 수 있습니까? KLEE와 비교하면 어떤가요?

1) Grey-box Concolic Testing 효과: 우리는 Eclipse를 두 가지 다른 모드로 실행했습니다. 그림 7의 파란색과 분홍색 선은 각각의 경우에 대한 적용 범위를 나타냅니다. 총 32,223개의 소스 라인 중 그레이박스 concolic 테스트는 20,737라인(64.36%), 그레이박스 퍼징 모듈 단독 사용은 18,540라인(57.54%), KLEE는 20,445라인(63.45%)3 .

이 결과는 회색 상자 복통 테스트만 KLEE와 비교할 수 있음을 분명히 나타냅니다. 우리 도구는 바이너리 실행 파일에서 직접 실행되는 반면 KLEE는 소스 코드에서 실행됩니다. 이것

3 우리는 KLEE의 라인 커버리지가 약 60분 동안 급격히 증가했다고 해서 KLEE가 해당 지점 주변에서 코드를 빠르게 탐색하기 시작한다는 것을 의미하지는 않습니다.
시간 제한이 만료되면 KLEE는 기호 실행이 완료되지 않은 경우에도 메모리에 남아 있는 테스트 케이스를 출력합니다. 실제로 KLEE를 6시간 이상 더 실행했지만 적용 범위는 2.10% 증가에 그쳤습니다.

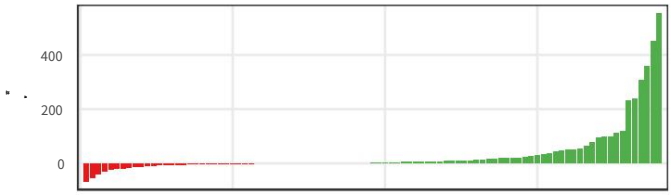


그림 8. Eclipser와 KLEE가 커버하는 라인 수의 차이.

표 1
LAVA-M에서 발견된 버그의 수.

프로그램	AFF빠른 0	LAF-인텔 40	부저 17 1	스틸릭스	이클립서
base64	0	0		43	46
md5sum	0			28	55
uniq who	0	26	27	7	29
		3	50	194	1135
총	0	69	95	272	1265

결과에 초점을 맞춘 설계 선택을 경험적으로 정당화합니다.
선형 또는 단조 분기 조건 해결.

2) 두 가지 전략 간의 전환: 녹색 선
그림 7은 다음으로 달성한 소스 라인 커버리지를 보여줍니다.
Eclipser는 두 가지 다른 전략을 번갈아 사용합니다.
우리의 디자인 선택이 실제로
시너지 효과 달성: Eclipser는 23,499개 라인(72.93%) 커버,
코드 커버리지 측면에서 KLEE를 능가합니다. 표준
Eclipser의 적용 범위의 편차는 0.54%인 반면,
KLEE의 커버리지는 0.49%였습니다. 또한 그림 8은
각각에 대한 Eclipser와 KLEE의 적용 범위 차이
프로그램. x축은 테스트된 프로그램을 나타내고 y축은
Eclipser가 더 많은 추가 라인을 포함했는지 나타냅니다.
KLEE보다. 맨 왼쪽 프로그램은 stty입니다. 여기서 KLEE는
66줄을 더 포함하고 가장 오른쪽 프로그램은 vdir입니다.
여기서 Eclipser는 554개 이상의 라인을 처리했습니다.

3) coreutils의 실제 버그: GNU core utils의 프로그램은 많은 테스트를 거쳤습니다. Eclipser가 여전히 의미 있는 것을 찾을 수 있습니까?
그들에 버그? 실험 과정에서 Eclipser는
각각 충돌할 수 있는 이전에 알려지지 않은 두 개의 버그를 찾았습니다.
b2sum 및 stty 각각. 반면 KLEE는
실험 중에 버그 중 하나만 찾을 수 있습니다. 이것
결과는 실제로 우리 시스템의 실용성을 강조합니다.

D. Grey-box Fuzzer와 비교

Eclipser는 최신 그레이 박스 fuzzer와 어떻게 비교됩니까? 에게
이 질문에 답하기 위해 우리는 의 버그 찾기 능력을 비교했습니다.
LAVA-M의 최첨단 그레이 박스 퍼저에 대한 Eclipser.
§ VA에서 우리는 Steelix와 VUzzer를 실행할 수 없었음을 상기합니다.
이 실험을 위해. 대신에 보고된 숫자를 사용했습니다.
다른 fuzzers와 비교하기 위해 그들의 논문. 공정하기 위해,
Steelix가 사용한 것과 유사한 설정으로 fuzzer를 실행했습니다. 우리
[6]에서 사용한 것과 동일한 초기 시드를 사용하고 실험을 실행했습니다.
같은 시간(5시간) 동안

표 I은 LAVA-M에서 발견된 버그의 수를 보여줍니다.
기준. 숫자는 8번의 반복된 실험에 대한 평균입니다. Eclipser는 18.3배, 13.3배 및 4.7
배 더 많은 버그를 발견했습니다.
LAF-intel, VUzzer 및 Steelix보다 각각. AFFast
실험 중에 버그를 찾지 못했습니다. 일부에서는
프로그램에서 Eclipser는 작성자가 발견한 버그도 찾을 수 있었습니다.
LAVA의 번식에 실패했습니다. 예를 들어 base64에서
LAVA의 작성자는 [43]에서 44개의 버그만 재현할 수 있습니다.

LAF-intel은 소스 기반 도구이므로
바이너리 기반 도구에 비해 계측 오버헤드가 적습니다.
예를 들어, LAVA-M 벤치마크에서 AFL을 실행했을 때,
소스 기반의 초당 실행 횟수
계측은 바이너리 기반보다 9.3배 더 높았습니다.
평균적으로 계측. 이러한 단점에도 불구하고,
Eclipser는 LAF-intel보다 훨씬 더 많은 버그를 발견했습니다. 이 결과는 보여줍니다
그레이 박스 복통 테스트는 복잡한 문제를 효과적으로 해결할 수 있습니다.
LAVA에 의해 주입된 버그를 유발하는 조건.

E. 현실 세계에서의 퍼징

우리는 다양한 프로그램에서 시스템을 추가로 평가했습니다.
현실 세계에서. 구체적으로, 우리는 22개의 프로그램을 수집했습니다.
다음 단계가 있는 데비안 OS. 먼저, 우리는 부채 태그를 사용했습니다.
C 프로그램이 포함된 패키지 검색
명령줄 인터페이스를 통해 이미지, 오디오 또는 비디오. 다음으로 우리는
데비안을 기반으로 인기 있는 상위 30개 패키지를 선택했습니다.
인기 콘테스트[45]. 그런 다음 수동으로
(1) 파일을 입력으로 받아들이고, (2) 컴파일할 수 있는 패키지
LAF-intel을 사용하여 (3) AFLFast 없이 퍼지할 수 있습니다.
오류. 마지막으로 다음에서 최대 두 개의 프로그램을 추출했습니다.
각각의 패키지는 총 22개의 프로그램을 얻습니다. 우리
더미 시드로 각 프로그램을 24시간 동안 퍼징했습니다.
16개의 연속 NULL 바이트로 구성됩니다.

표 II는 그 결과를 보여준다. 전반적으로 Eclipser는 1.43×
(1.44x) 및 1.25x (1.25x) 더 많은 노드(분기)
AFLFast 및 LAF-intel. 조사하다가
결과, 우리는 Eclipser의 그레이 박스 concolic 테스트가
실제로 높은 적용 범위를 달성하는 데 중요한 역할을 했습니다. ~ 안에
예를 들어, oggenc는 Eclipser보다 3.8배 더 많은 노드를 처리했습니다.
AFLFast as 회색 상자 복통 테스트가 성공적으로 생성됨
처음부터 FLAC 또는 RIFF 형식에 대한 유효한 서명.

발견된 충돌을 추가로 조사하고 수동으로
51개의 고유한 버그를 식별했습니다. 전체적으로 Eclipser, AFLFast 및
LAF-intel은 각각 40, 10, 25개의 고유한 버그를 발견했습니다. 우리
결과를 추가로 분석한 결과 회색 상자 복통이 발견되었습니다.
테스트는 실제로 버그를 찾는 데 중요한 역할을 했습니다. 우리가 달렸다면
그레이박스 퍼징 모듈만을 사용한 동일한 실험
바닐라 AFL[4]에 가까운 Eclipser의
24시간 후에 8개의 고유한 버그만 발생합니다. 즉, 회색 상자
concolic 테스트는 Eclipser가 5배 더 많은 고유 버그를 찾는 데 도움이 되었습니다. 우리
Eclipser가 발견한 모든 버그를 개발자에게 보고했으며,
작성 당시 총 8개의 새로운 CVE가 할당되었습니다. 우리
이 결과는 Eclipser의 실질적인 영향을 확인시켜줍니다.

VI. 논의

그레이박스 배앓이 검사의 현재 디자인은 다음 사항에 중점을 둡니다.
비교의 피연산자가 다음의 선형 또는 단조 함수로 표현될 수 있는 경우 분기 조건 풀기

입력 필드. Eclipser는 현재 복잡한 회색 상자가 있는 분기를 관통하기 위해
전통적인 그레이 박스 퍼징에 의존하고 있음을 기억하십시오.
제약. 이것은 해결하기 때문에 큰 단점이 아닙니다.

표 II
코드 적용 범위가 달성되었고 데비안 프로그램에서 발견된 고유한 버그의 수.

프로그램	패키지	장소	AFFast			LAF-인텔			이클립서		
			노드 코브.	브랜치 코브.	# 유니크. 벅스	노드 코브.	브랜치 코브.	# 유니크. 벅스	노드 코브.	브랜치 코브.	# 유니크. 버그
advmng	어드밴스컴프	22,615	2,517	3,219	0	2,516	3,215	3,046	3,742	3,701	2,421
advzip			2,572	3,310	0	2,886	6,490	5,887	8,117	5,025	3,676
dcparse	드크로	11,328	2,006	2,621	0	1,880	4,636	27,542	18,933	30,454	19,228
dcraw			5,004	7,082	0	4,712	2,991	2,369	2,707	2,327	5,505
fig2dev	fig2dev	35,027	5,489	7,718	6	5,626	3,002	2,087	7,304	5,941	1,790
gifdiff	gifsicle	15,212	1,381	1,608	1	2,823	3,395	7,422	2,366	2,156	1,775
gifsicle			3,365	4,269	1	4,693	4,421	34,612	36,772	181,335	161,669
gnuplot	gnuplot	113,368	14,560	21,016	0	18,769		1			26,402
gocr	gocr	17,719	19,281	30,059	1	19,457		1			29,864
icotool	icutils	31,337	2,182	2,758	0	2,250		0			3,507
변기			1,805	2,205	0	2,344		1			2,015
jhead	jhead	4,099	1,886	2,758	0	2,208		0			2,561
optipng	optipng	82,107	3,885	2,205	0	4,087		0			6,088
ldactoasc	섹스트랙터	39,083	1,200	2,758	0	1,228		0			3,727
sndfile-info	sndfile-프로그램 30,141		2,751	2,205	0	3		0			5 10,186
sndfile-play			6 2,689	2,286	0	1,742		0			28,1
ufraw-batch	ufraw-배치	66,487		5,201	0	15,92		2			삼
oggenc	보비스 도구	30,141		1,397	0			1			2
vorbiscomment				3,616	0			0			0
wavpack	wavpack	32,923		9 3,281	0			1			8
wvunpack x264					0			0			0
	x264	70,382			1			2			삼
총		571,828			10			25			40

비선형 제약 조건은 어쨌든 어렵습니다. 그러나
§ VII에서 논의한 메타휴리스틱 기반 알고리즘을 채택합니다.
Eclipser는 현재 소스 없이 다양한 프로그램을 테스트하기 위해 바이너리
기반 계측을 사용하고 있습니다.
암호. 그러나 이진 기반 계측은 실험 중 하나에서 관찰한 바와 같이 상당한 오버
헤드를 발생시킵니다.
§ VD에서. 의 성능을 향상시키는 것은 간단합니다.
소스 기반 계측을 채택하여 Eclipser.

VII. 관련 작업

Eclipser는 그 자체로 fuzzer가 아니지만 fuzzing을 사용합니다.
기준 치수. 따라서 모든 위대한 연구는 퍼징에 관한 것입니다[4]–
[7], [10], [11], [13], [14], [28], [29], [46]–[50]은 실제로
우리의 보완.
그레이 박스 배얇이 테스트는 화이트 박스에서 영감을 받았기 때문에
퍼징, 그것은 자연스럽게 경로 폭발 문제를 겪는다.
대응하기 위해 다양한 탐색 전략이 제안되었다.
문제. 예를 들어 KLEE[18]는 임의 경로를 채택합니다.
다른 사람들 [19], [23], [27], [51]–[53] 우선 순위
덜 이동된 실행 경로 또는 노드, 또는 정적 활용
검색을 안내하는 분석 [54]. Eclipser는 다음을 따르지만
[19]에서와 유사한 접근 방식, 우리는 더 많은 채택을 믿습니다.
복잡한 전략은 유망한 미래 작업입니다. 한편,
예를 들어 상태 병합[25], [55], [56]과 같이 화이트 박스 퍼징의 확장성을 증가
시키려는 여러 시도가 있습니다. ~ 안에
대조적으로, 우리의 작업은 주로 근본적인 완화에 중점을 둡니다.
기호 공식을 구성하고 해결하기 위한 오버헤드.
값비싼 데이터 흐름 없이 프로그램을 분석하는 아이디어
분석은 다양한 맥락에서 연구되었습니다. 예를 들어,
MUTAFLOW [57] 오염 없는 정보 흐름 감지
소스 포인트에서 입력 데이터를 단순히 변형하여 분석
및 싱크 포인트에서 출력 데이터에 영향을 미치는지 관찰합니다.
Helium [37]은 회귀 분석을 사용하여 관계를 추론합니다.
코드 세그먼트의 입력과 출력 사이. 그런
동적 분석은 기호 실행을 보완하는 데 사용됩니다.
알 수 없는 라이브러리 기능 또는 루프가 있는 경우. 우리의
작업은 이러한 아이디어를 확장하고 더 적극적으로 적용합니다.
일반적인 테스트 케이스 생성 알고리즘을 고안합니다.
Angora[8]와 SBF[58]는 우리와 가장 가까운 퍼저입니다.
그들은 검색 기반 소프트웨어 테스팅[30]의 아이디어를 채택하고,
[59]–[63]에서 논의된 분기 침투 문제를 해결하기 위해
§ II-C. 특히, Angora는 최소화하는 입력을 찾으려고 합니다.
조건부 분기의 분기 거리. 그러나 사용
에 영향을 미치는 입력 바이트를 식별하기 위한 세분화된 taint 분석
대상 조건부 분기인 반면 Eclipser는 PUT를 반복적으로 실행하여 이러한 관계
를 동적으로 추론합니다. 따라서,
우리는 두 가지 접근 방식이 서로 보완적이라고 믿습니다.
예를 들어, 먼저 그레이 박스 복통 테스트를 적용할 수 있습니다.
간단한 가지 조건을 뚫고 앙고라로
더 복잡한 조건을 처리하기 위한 전략.

VIII. 결론

이 논문은 디자인 공간의 새로운 요점을 제시합니다.
퍼징. 제안된 기법인 그레이박스 복통 검사,
SMT에 의존하지 않고 화이트 박스 퍼징을 효과적으로 어둡게 합니다.
여전히 경로 기반 테스트를 수행하면서 해결합니다. 우리는 Eclipser라는 시스
템에서 우리의 기술을 구현하고 coreutils, LAVA-M,

뿐만 아니라 데비안의 22개 프로그램. 우리는 우리의 기술을 보여주었다
의 현재 최첨단 도구에 비해 효과적입니다.
코드 적용 범위와 발견된 버그 수에 대한 조건입니다.

승인

피드백을 주신 익명의 검토자에게 감사드립니다. 이것
이 작업은 다음이 자금을 지원하는 정보통신기술진흥원(IITP) 보조금으로 부분
적으로 지원되었습니다.
한국정부(과기정통부) (No.B0717-16-0109, 빌딩
바이너리 코드 분석을 통한 자동화된 리버스 엔지니어링 및 취약성 감지를 위한
플랫폼) 및 보조금 지원
삼성리서치(Binary Smart Fuzzing) 제공.

[46] R. Swiecki 및 F. Grobert, "honggfuzz," https://github.com/google/
홍퍼즈.