Optimuzz: 안전한 컴파일러를 위한 오번역 자동 탐지 기술

한국과학기술원 ▌장봉준·허기홍*

1. 서 론

컴파일러는 소프트웨어 개발에서 중추적인 역할을 한다. 컴파일러는 개발자가 작성한 프로그램을 하드웨어가 실행할 수 있는 프로그램으로 번역한다. 따라서 컴퓨터에서 구동되는 모든 소프트웨어는 컴파일러를 거친다 해도 과언이 아니다. 운영체제부터 데이터베이스, 인공지능까지 모든 프로그램은 컴파일러를 거쳐 실행가능한 프로그램이 된다.

```
int foo (void) {
  signed char x = 1;      // 0x00000001
  unsigned char y = 255; // 0x000000ff
  return x > y;
}
```

그림 1 x86 아키텍처 Ubuntu Linux 8.04.1 버전에 포함된 gcc에서 발생한 오번역 사례. 위 함수 foo는 항상 1을 반환하도록 잘못 최적화되었다. 이 오번역 버그는 C 컴파일러 마구실행기 CSmith가 발견했다.

컴파일러는 개발자가 작성한 프로그램을 같은 의미의 기계어로 변환해야 한다. 그러나 번역 과정에서 원래 프로그램과 의미가 다른 프로그램을 생성하기도 하는데, 이를 오번역(miscompilation)이라고 부른다. 오번역의 사례를 이해하기 위해 그림 1에서 gcc의 예를 보자[1]. 함수 foo는 두 상수 x와 y를 비교하고 있다. x와 y는 모두 32비트 정수로 변환되는데, y는 부호가 없기 때문에 최상위비트를 무시하므로 `0x000000ff`로 변환된다. 여기서 비교 연산자 >은 부호를 고려하므로, 0을 반환해야 한다. 그러나 이 함수는 최적화 후 항상 1을 반환하는 함수로 번역되었다.

이러한 오번역 버그는 컴파일러 최적화 과정에서 주로 발생한다. 컴파일러는 번역 과정에서 입력 프로그램을 분석해, 의미가 같은 더 빠른 프로그램으로 바꾸며 이를 최적화 과정이라고 한다. 컴파일러를 작성한 개발자가 실수로 번역 전후 의미가 달라지는 경우를 미처 생각하지 못하면 오번역을 일으키는 컴파일러 최적화 코드를 작성하게 된다.

컴파일러 오번역 버그는 중대한 보안 위협으로 이어질 수 있다. WebAssembly 컴파일러인 Cranelift[1]의 예를 들어보자. WebAssembly 프로그램은 허용된 메모리 영역 이외에 접근할 수 없다[2]. 사용자 브라우저에서 실행되는 특성을 고려해 샌드박스를 통해 보안을 강화한 것이다. 하지만 2023년 3월, Cranelift의 오번역 버그로 인해 번역 후 WebAssembly 프로그램이 허용된 영역을 벗어난 메모리접근이 가능하다는 보고가 공개되었다[3]. Cranelift 컴파일러를 사용해 WebAssembly 프로그램을 번역할 경우, 공격자는 허용된 영역 밖의 메모리 영역을 임의로 읽거나 변조하는 심각한 공격을 할 수 있었다. 이렇듯 컴파일러는 오번역을 통해 심각한 보안 취약점을 만들 수 있으며, 컴파일러는 모든 소프트웨어 개발에 사용되므로 그 파급력이 매우 크다.

그러므로 오번역 버그를 사전에 탐지하는 기술이 필수적이다. 하지만 널리 사용되는 컴파일러(LLVM, GCC, V8)에서 오번역 버그를 탐지하는 것은 여간 어려운 일이 아니다. 코드 규모가 매우 방대하고 수시로 업데이트되고 있기 때문이다. LLVM의 경우 150만줄 이상의 코드로 이루어져 있으며, 2024년에 37,000번이 넘는 업데이트(커밋)이 있었다. 이렇게 크고 빠르게 변화하는 프로그램을 사람이 일일이 검수하기란 불가능에 가깝다.

이 문제를 해결하기 위해 번역 검산(translation validation)과 마구실행(fuzzing) 기술이 컴파일러에 도입되고 있다. 번역 검산은 컴파일 전후에 프로그램 의미가 달라졌는지 확인하는 기술이다. 컴파일러의 번역 전, 번역 후

* 종신회원

1) https://cranelift.dev/
2) https://webassembly.github.io/spec/core/intro/introduction.html
3) https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-ff4p-7xrq-q5r8

프로그램만 사용하므로 컴파일러 내부 구현에 상관 없이 작동한다. 또한 프로그램 의미를 자동증명기(automated theorem prover)를 통해 자동으로 검사하므로 빠르게 개발되는 컴파일러의 버그 탐지를 자동화하는 데 적합하다.

이에 더해, 마구실행 기술은 컴파일러의 입력 프로그램을 자동으로 만들어내는 역할을 한다. 번역 검산기는 컴파일러의 번역 전후 입출력 프로그램이 필요하므로, 입력 프로그램이 없다면 오번역 버그를 탐지할 수 없다. 보통은 컴파일러 개발진이 작성하는 유닛 테스트를 사용하지만, 개발자가 미처 생각하지 못한 입력 프로그램이 오번역을 일으키는 경우가 있을 수 있다. 따라서 다양한 입력 프로그램을 자동으로 생성할 수 있는 마구실행 기술이 필수적이다.

그러나 마구실행을 컴파일러에 바로 사용하는 것은 적절치 않다. 위에서 언급한 것처럼, 컴파일러는 규모가 매우 크고 빠르게 업데이트되기 때문이다. 일반적인 마구실행기는 프로그램 전체를 대상으로 하기 때문에 규모가 큰 컴파일러에 적용하면 짧은 시간 안에 원하는 입력 프로그램을 얻기 힘들다. 이런 상황에서는 마구실행이 다 끝나기도 전에 새로운 업데이트가 등장할 것이다. 또한, 오번역 버그는 최적화기에서 주로 발생하기 때문에 컴파일러 전체에 대해 마구실행하면 불필요한 시간이 허비되는 점, 컴파일러 앞단을 통과하기 위해 문법과 타입 규칙을 고려한 입력변형기를 만들어야 하는 등 추가적으로 해결해야 할 문제가 존재한다.

Optimuzz는 마구실행을 컴파일러와 번역 검산에 적용할 때 발생하는 문제들을 해결한 컴파일러 최적화 지향성 마구실행 프레임워크다. 오번역이 자주 일어나는 컴파일러 최적화기를 대상으로, 의심되는 컴파일러 최적화 규칙이 있는 지점에 도달하는 입력 프로그램을 효과적으로 생성한다. LLVM과 같은 거대하고 복잡한 컴파일러에 대해서 1시간 안에 오번역을 유발하는 입력 프로그램을 생성하는 데 성공하여 빠르게 업데이트되는 컴파일러 생태계의 특성도 극복했다. 이에 더해 JIT(Just-In-Time) 번역 기술이 적용된 컴파일러인 TurboFan에서도 6시간 안에 오번역을 유발하는 입력 프로그램을 생성하는 데 성공하여 여러 종류의 컴파일러에 적용 가능한 기술임을 입증하기도 하였다.

Optimuzz는 컴퓨팅 분야 최고 학술지인 ACM SIGPLAN PLDI 2025에서 6월 16일 발표되었다[6]. 또한 현재 프로젝트 웹사이트[4]를 통해 소스코드 및 발견한 버그 현황, 발견한 버그에 대한 설명을 제공하고 있다. 이 글에서는 번역 검산 기술과 Optimuzz의 컴파일러 최적화를 위한 지향성 마구실행 기술을 소개하고 Optimuzz의 성과를 공유하고자 한다.

2. 개 요

2.1 번역 검산(translation validation)

번역 검산은 컴파일러의 입력 프로그램과 출력 프로그램을 보고, 프로그램의 의미가 보존되었는지 확인하는 기술이다[7]. 예를 들어, 컴파일러가 아래 src 프로그램을 tgt 프로그램으로 최적화했다고 해보자. (a / b) * b에서 (b / b)는 항상 1이므로, 계산을 생략해도 된다고 컴파일러 개발자가 잘못 생각한 경우를 가정한다.

```
int src(int a, int b) {
    return (a / b) * b;
}

int tgt(int a, int b) {
    return a;
}
```

번역 검산기는 번역 전 프로그램(src)와 번역 후 프로그램(tgt)의 의미 동등성을 검사한다. 여기서 자동증명기가 사용된다. src와 tgt을 모두 자동증명기의 논리식으로 변환하여 가능한 모든 입력에 대하여 출력이 같은지 확인한다. 예를 들어, 번역 검산기는 아래 수식이 참인지 자동증명기를 통해 검사한다.

$$\forall a, b.\ (a / b) \times b = a$$

이 경우, src(4, 3)은 3인데, tgt(4, 3)은 4가 되므로 동등성이 지켜지지 않았다는 반례를 찾을 수 있다. 이렇게 번역 검산기는 오번역을 탐지할 수 있다.

번역 검산기는 대상 언어에 정의되지 않은 행동(undefined behavior)가 있을 경우, 번역 검산기는 동등성이 아니라 구체화(refinement relation)를 검사한다. 구체화는 동등성보다 완화된 조건이다. 동등성을 따지려면 두 프로그램이 입출력이 하나하나 같은지를 판별해야 하는데, 프로그램에 미정의 행동이 있다면 출력이 하나로 결정되지 않기 때문이다. 이 경우에는 번역 후 프로그램이 출력할 수 있는 값의 범위가 번역 전 프로그램보다 좁은지 확인한다. LLVM IR을 대상으로 하는 번역 검산기 Alive2가 대표적인 예이다[8].

4) https://prosys.kaist.ac.kr/optimuzz/

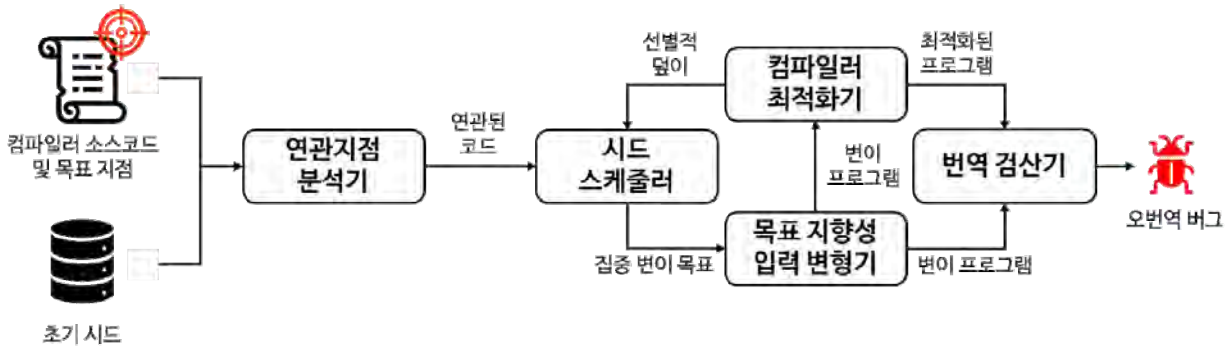

그림 2 컴파일러 최적화 지향성 마구실행 프레임워크 Optimuzz의 시스템 개요도

```
int src(int x) {
    return x + 1 > x;
}

int tgt(int x) {
    return 1;
}
```

구체화 검사의 이해를 돕기 위해 위의 예를 살펴보자. C 프로그램에서는 부호가 있는 정수의 값이 넘치는 경우(Signed Integer Overflow)가 정의되지 않았기 때문에, 값넘침이 일어나는 경우 프로그램의 출력은 아무 값이나 될 수 있다. 입력 프로그램 src을 보면 x가 int 타입의 최대값일 때 값넘침이 일어나며, 최대값이 아닐때는 항상 x + 1 > x가 성립한다. 따라서 x가 int 타입의 최대값일 때는 어떤 값이나 출력해도 상관없고, 최대값이 아닐 때는 항상 1을 출력해야 한다. 결론적으로 모든 입력 x에 대해 항상 참을 출력할 수 있다. 그러므로 번역 검산기는 위 예에서 올바른 번역이 일어났다고 결론을 내린다.

번역 검산기는 이와 같이 자동으로 오번역이 일어났는지 알려주는 강력하면서도 편리한 도구이다. 이러한 장점 때문에 컴파일러 커뮤니티에서는 번역 검산기를 적극 활용하고 있다. LLVM의 경우에는 새로운 컴파일러 최적화를 추가할 때 해당 최적화를 실행하는 유닛 테스트를 제출해야 하며, Alive2 번역 검산기를 이용하여 해당 유닛 테스트에서 오번역이 일어나지 않음을 보여야 한다[5].

2.2 지향성 마구실행(directed fuzzing)

지향성 마구실행은 프로그램의 특정 지점을 실행시키는 입력을 만드는 기술이다. 프로그램의 규모가 거대해지고, 자주 수정되면서 전체 프로그램을 대상으로 오랫동안 하는 마구실행은 현실적으로 사용하기 어려워졌다. 따라서 프로그램의 일부분을 대상으로 집중적으로 검사해 빠르게 오류를 찾아내는 지향성 마구실행의 중요성이 더욱 대두되었다.

AFLGo, DAFL 등 대표적인 지향성 마구실행기들은 시드 입력의 프로그램 덮이(coverage)를 관찰하여 점수를 부여하는 방식으로 지향성을 달성한다[3][4]. AFLGo는 실행 흐름 그래프 상에서 목표 지점에 가깝게 도달할수록 더 많은 점수를 부여하며, DAFL은 정의 사용 그래프(def-use graph)를 이용해 목표 지점에 사용되는 변수들과 연관된 지점을 더 많이 지날수록 더 높은 점수를 부여한다. 마지막으로, 높은 점수를 받은 만큼 더 많은 변종을 생성해 목표 지점에 도달할 가능성이 높은 입력을 더 많은 생성한다.

또한 시드 입력을 변이하는 방식에서 지향성을 달성할 수도 있다. WindRanger의 경우, 변이된 입력이 만드는 덮이가 달라지는 것을 통해 입력에서 조건식에 쓰인 바이트를 식별해내는 방법을 사용한다[5]. 조건식에 영향을 미치는 입력 일부분을 특정하여 집중적으로 변이함으로써 더욱 효과적인 마구실행을 꾀하는 방식이다.

Optimuzz 또한 시드 입력이 만드는 덮이를 관찰하고, 시드 입력을 변형시킬 때 특정 지점에 집중함으로써 지향성을 달성한다. 다음 장에서는 Optimuzz가 취하는 전략을 살펴보며, 어떻게 지향성을 달성하는지 탐구한다.

3. Optimuzz 프레임워크

Optimuzz는 번역 검산과 지향성 마구실행을 결합한 컴파일러 최적화 지향성 마구실행 프레임워크다. 목표 최적화 규칙이 존재하는 지점이 주어지면 Optimuzz는

5) https://llvm.org/docs/InstCombineContributorGuide.html#proofs

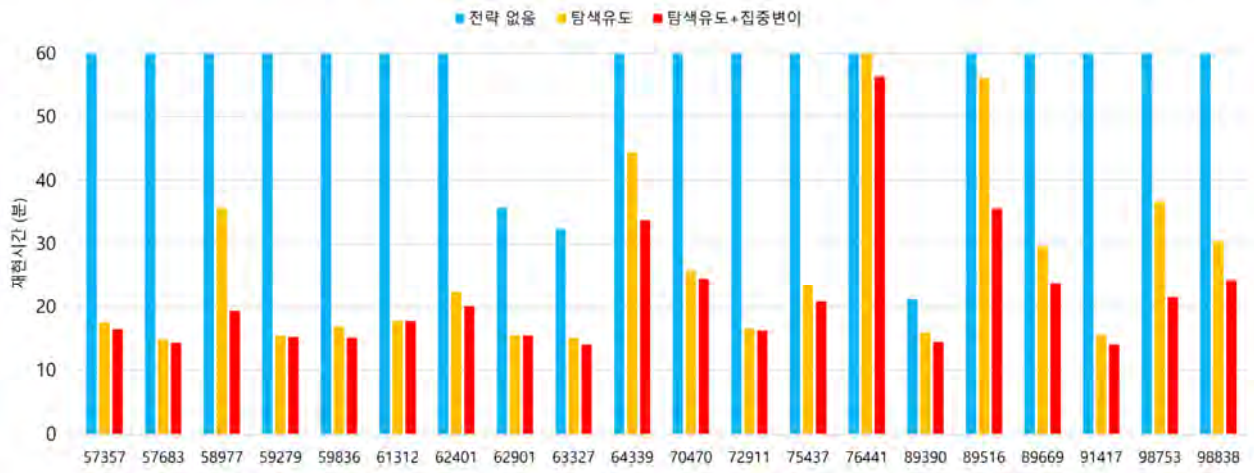

그림 3 사용한 전략 별로 LLVM 오번역 버그를 재현하는 데 걸린 시간. 24개의 오번역 버그 중에서 사용한 전략에 상관없이 항상 실패하거나 20분안에 모두 성공하는 경우는 제외하였다. 탐색 유도 전략과 집중 변이 전략을 사용할 때마다 재현 시간이 감소하며 재현 가능한 오번역 버그가 늘어남을 확인할 수 있다.

지향성 마구실행을 통해 목표 최적화 규칙을 일으키는 다양한 입력 프로그램을 만든다. 그리고 해당 입력 프로그램들을 번역 검산을 통해 오번역이 일어나는지 확인한다. 번역 검산기가 오번역을 탐지하면 이를 컴파일러 개발자에게 제보하여 컴파일러를 수정한다.

Optimuzz가 사용하는 전략들을 하나씩 살펴보자.

3.1 탐색 유도 전략 (Guided Search Strategy)

Optimuzz는 탐색 유도 전략을 통해 지향성 마구실행을 달성한다. 목표 최적화 지점이 주어지면 Optimuzz는 대상 컴파일러 소스코드에서 연관된 지점을 추려낸다. 이 후 입력 프로그램을 실행할 때, 연관된 지점에서만 선별적으로 덮이를 관찰한다.

이 때, Optimuzz는 컴파일러 최적화기에서 일반적으로 발견되는 구조를 이용한다. 먼저 컴파일러 최적화는 최적화가 일어나기 위한 조건을 하나씩 검사한다. 이후 모든 조건이 맞아 떨어지면 최종적으로 최적화가 일어나는 방식으로 작성된다. 이 때 최적화 조건이 하나라도 틀리면 최적화가 일어나지 않기 때문에 모든 최적화 조건이 만족되는 것이 중요하다.

이 구조를 이용하여 실행 흐름 그래프 상의 도달 가능성으로 목표 최적화와 연관된 지점을 판단한다. 실행 흐름 그래프의 지점이 목표 최적화 지점에 도달 가능하다면 연관된 지점으로 추려내고, 그렇지 않으면 걸러내는 방식이다. 따라서 목표 최적화와 관련없는 최적화 조건 검사를 무시하여, 만족시켜야 할 조건에만 집중할 수 있다.

또한 Optimuzz는 시드 프로그램에 거리 점수를 부여한다. 연관된 지점만 남은 실행 흐름 그래프에서 목표 지점에 가까운 지점에 도달할수록 시드 프로그램이 목표에 가깝다고 판별한다. 목표 지점에 가까운 시드 프로그램일수록 더 많은 변이 프로그램을 만들어낸다. 그 결과, 최적화 조건을 더 많이 만족시킬수록 변이 프로그램을 더 많이 생성하기 된다.

탐색 유도 전략을 통해 목표와 무관한 최적화 조건만 만족하는 입력 프로그램은 시드 집합에서 제외된다. 그리고 목표 최적화와 연관된 최적화 조건을 만족하는 프로그램을 더 많이 만들게 된다.

따라서 마구 실행하는 동안, Optimuzz는 대부분의 시간을 목표 최적화를 실행하는 유용한 입력 프로그램을 생성하는데 투자할 수 있다. 그 결과, 목표 지점에 상관없이 입력 프로그램을 생성하는 다른 컴파일러 마구실행 전략에 비해 훨씬 성능이 향상되는 것이 확인되었으며 오번역을 더욱 효과적으로 탐지할 수 있다.

3.2 집중 변이 전략 (Targeted Mutation Strategy)

Optimuzz는 시드 입력 프로그램을 변형하는 과정에서 목표 최적화과 연관된 시드 프로그램의 중요 지점을 집중적으로 변형한다. 중요 지점은 가장 최근에 덮이를 증가시킨 변이 지점과 정의-사용 관계 상 이웃 지점으로 식별한다. 이는 컴파일러 최적화가 입력 프로그램 상에서 정의-사용 관계로 연관된 값들을 검사하여 최적화 여부를 결정한다는 사실을 이용한 것이다. 이를 통해 Optimuzz는 목표 최적화와 연관된 중요 지점을 더 많이 변이시켜, 더 빠르게 목표 최적화에 도달할 수 있다.

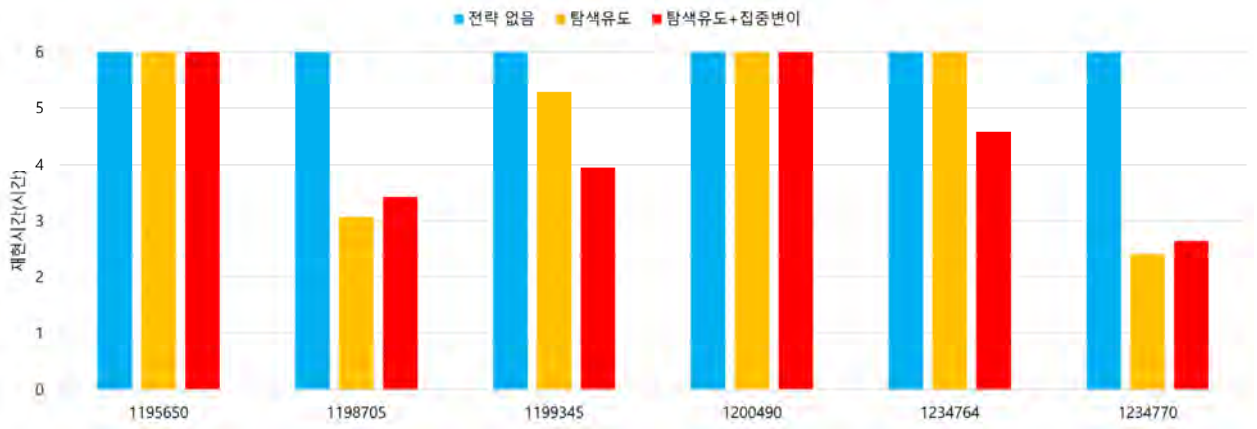

그림 4 사용한 전략 별로 TurboFan 오번역 버그를 재현하는 데 걸린 시간. 탐색 유도 전략과 집중 변이 전략을 사용할 때 가장 많은 오번역 버그를 재현할 수 있음을 알 수 있다.

```
1: int src(int x) {   // 정의: x
2:  int y = 5;
3:  int k = y / 4;
4:  int exp = 5; // 정의: exp
5:  int r = x * exp; // 사용: x, exp
6:  return r + k;
7: }

Instruction *I opt(Instruction *I) {
  if (I is Mul) // 조건 (1)
    if (I.X is V) // 조건 (2)
      if (I.Y is PowerOf2) // 조건 (3)
        return (X << log2(Y));
  return null;
}
```

위의 예를 들어, X * (2 ** C)를 X << C로 최적화하는 경우를 생각해보자. 시드 프로그램 X + (2 ** C)에서 5번째 줄의 연산자가 더하기에서 곱하기로 바뀌어 X * (2 ** C)로 변이가 일어난 상황이다. 이 변이를 통해 입력 프로그램 src를 만들었다. 최적화 opt는 제일 먼저, 곱하기가 있는지 확인할 것이므로 덮이가 증가한다. 그 다음 opt는 곱하기의 좌우변을 검사하는데, 곱하기의 우변(Y)이 2의 제곱수인지 확인하는 것에 주목하자. 이 때, 곱하기와 그 우변은 서로 정의-사용 관계상 이웃이다.

이러한 컴파일러 최적화기의 특징을 이용해 최근 덮이가 증가(조건 1)한 변이 지점(줄 5)에서 정의-사용 관계로 연관된 지점(줄 1, 4)을 집중적으로 변이하면 덮이(조건 2, 3)를 증가시킬 수 있다. 4번째 줄에서 exp의 값이 2의 제곱수로 바뀌면 opt의 덮이가 다시 증가할 것이고, 목표 최적화가 최종적으로 실행된다. 또한, 이 과정에서 목표 최적화와 연관되지 않은 2, 3번째 줄은 자연스럽게 변이 기회를 덜 받게 된다.

Optimuzz는 이런 전략을 구사하여 중요 지점을 식별함으로써 목표 최적화에 도달하는 입력 프로그램을 효과적으로 생성한다. Optimuzz는 집중 변이 전략을 통해 탐색 유도 전략만으로는 재현할 수 없었던 오번역 버그를 재현했으며, 재현에 걸리는 시간 또한 단축할 수 있었다.

3.3 LLVM과 V8을 대상으로 적용

그림 3과 그림 4는 Optimuzz가 LLVM과 V8에 과거에 알려졌던 버그를 재현하는 실험의 결과를 보여준다. LLVM의 경우, 2023년부터 번역 검산기(Alive2)를 통해 탐지 가능한 모든 오번역 버그 24개를 선별하였다. Optimuzz는 이 중 23개를 1시간 안에 재현하는 데 성공하였다. V8 TurboFan에 경우, 번역 검산기(TurboTV) 논문에 사용된 9개의 버그 중에서 자바스크립트 마구실행기 Fuzzilli[6]로 재현가능한 버그 6개를 선별하였다. Optimuzz의 두 전략을 구현한 Fuzzilli는 이 6개 중 4개를 6시간 안에 재현하는 데 성공하였다.

Optimuzz는 다른 컴파일러 마구실행기와 비교해 뛰어난 성능을 보여주기도 하였다. LLVM 컴파일러 마구실행기 FLUX, Alive-Mutate와 Optimuzz의 전략을 구현하지 않은 Fuzzilli는 같은 제한시간 동안 선별된 버그를 모두 재현하지 못했다[9][10]. 이는 Optimuzz의 우수한 성능과 함께 지금까지 시도되지 않았던 컴파일러 지향성 마구실행의 가능성을 입증한 것이다.

3.4 새로운 오번역 버그 탐지

Optimuzz는 LLVM에서 56개의 새로운 오번역 버그

6) https://github.com/googleprojectzero/fuzzilli

를 찾아내기도 하였다. 이중 22개의 버그가 패치되어 LLVM의 보안에 기여했다.

먼저 Optimuzz는 최신 버전 LLVM에서 목표 지점을 수집하여 검사하는 방식으로 LLVM을 검사했다. LLVM 최적화기의 소스코드를 분석하여 최적화 규칙을 검사하는 목표 지점을 식별하고 컴파일러 마구실행과 번역 검산을 이용해 46개의 오번역 버그를 탐지하였다.

또한 Optimuzz는 LLVM 최적화기를 업데이트마다 검사하기도 하였다. 업데이트에서 변경된 코드 지점에서 목표 지점을 식별하여 같은 과정을 통해 8개의 오번역 버그를 탐지하였다. 이 과정을 통해 Optimuzz는 빠르게 개발 속도의 거대한 컴파일러 프로젝트에서 그 유효성을 입증하였다.

4. 결 론

이 글에서는 번역 검산과 지향성 마구실행을 결합해 컴파일러 오번역 버그를 자동으로 탐지하는 Optimuzz 프레임워크에 대하여 소개하였다. 기존 번역 검산 도구는 입력 프로그램 생성의 한계로 오번역 버그를 탐지하지 못하는 한계가 있었으나 Optimuzz는 지향성 마구실행을 활용하여 그 문제를 해결하였다.

또한 Optimuzz가 보여준 지향성 마구실행 기술은 실제 대규모 컴파일러 프로젝트에서 그 유효성을 입증하기도 하였다. Optimuzz와 같은 지향성 마구실행이 번역 검산과 함께 여러 컴파일러의 오번역 버그를 빠르게 탐지하여 더 견고한 컴파일러 생태계를 만들기 위해 기여할 수 있기를 기대해 본다.

참고문헌

[1] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr, “Finding and understanding bugs in C compilers,” SIGPLAN Vol. 46, Issue 6, p. 283 – 294, 2011.

[2] Zhide Zhou, Zhilei Ren, Guojun Gao, He Jiang, “An empirical study of optimization bugs in GCC and LLVM,” Journal of Systems and Software, Volume 174, 2021.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury, “Directed Greybox Fuzzing,” In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), ACM, pp. 2329 – 2344, 2017.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury, “Directed Greybox Fuzzing,” In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), ACM, pp. 2329 – 2344, 2017.

[5] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao, “WindRanger: a directed greybox fuzzer driven by deviation basic blocks,” In Proceedings of the 44th International Conference on Software Engineering (ICSE '22), ACM, pp. 2440 – 2451, 2022.

[6] Jaeseong Kwon, Bongjun Jang, Juneyoung Lee, and Kihong Heo, “Optimization-Directed Compiler Fuzzing for Continuous Translation Validation,” Proc. ACM Program. Lang. 9, PLDI, Article 172, 24 pages. 2025.

[7] Pnueli, A., Siegel, M., Singerman, E., “Translation validation,” Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol 1384. Springer, Berlin, Heidelberg, 1998.

[8] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr, “Alive2: bounded translation validation for LLVM,” In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021), ACM, pp. 65 – 79, 2021.

[9] Eric Liu, Shengjie Xu, and David Lie, “FLUX: Finding Bugs with LLVM IR Based Unit Test Crossovers,” In Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23), IEEE Press, pp. 1061 – 1072. 2024.

[10] Yuyou Fan and John Regehr, “High-Throughput, Formal-Methods-Assisted Fuzzing for LLVM,” In Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '24), IEEE Press, pp. 349 – 358. 2024.

약 력

장 봉 준

2025 한국과학기술원 전산학부 졸업(학사)
2025~현재 한국과학길원 전산학부 재학(석사)
Email : bongjun.jang@kaist.ac.kr

허 기 홍

2009 서울대학교 컴퓨터공학부 졸업(학사)
2017 서울대학교 컴퓨터공학부 졸업(박사)
2017~2019 미국 University of Pennsylvania, Postdoctoral Researcher
2020~2024 한국과학기술원 전산학부 조교수
2024~현재 한국과학기술원 전산학부 부교수
Email: kihong.heo@kaist.ac.kr