차분 퍼징을 활용한 아키텍처별 컴파일러 최적화 버그 탐지

신재형* 백규민* 김동건* 안서현** 김진영***

*중부대학교 *아주대학교 *부경대학교 (학부생) **서울여자대학교 (학부생)
***성균관대학교 (대학원생)

Architecture-specific compiler optimization bug detection using differential fuzzing

Jae-Hyeong Shin* Gyu-Min Baek* Dong-Geon Kim*

Seo-Hyeon Ahn** Jin-Yeong Kim***

*Joongbu *Ajou *Pukyong University(Undergraduate student)
**SeoulW University(Undergraduate student)
***Sungkyunkwan University(graduate student)

요 약

컴파일러는 소스 코드를 실행 가능한 프로그램으로 변환시키는데, 이때 프로그램의 실행 시간 성능을 향상하기 위해 최적화 과정을 수행한다. 다양한 아키텍처가 목적에 적합하게 설계 및 개발되는 현대 컴퓨팅 환경에서 아키텍처별 요구사항을 충족시키기 위해 컴파일러의 역할이 더욱 중요하다. 각 아키텍처는 고유한 특성이 있어, 같은 소스 코드와 컴파일러를 사용하더라도 컴파일러 최적화 과정에서 다양한 동작을 수행하는 바이너리를 생성할 수 있다. 이러한 다양성은 명시적인 오류 메시지 없이 예상치 못한 버그를 유발할 수 있다. 본 연구는 차분 퍼징(Differential Fuzzing) 기법을 활용하여 다양한 아키텍처에서 컴파일러 최적화 버그를 효과적으로 탐지하는 방법을 제안한다. 제안한 시스템의 Code Generator에서 생성한 코드를 Executor에서 병렬적으로 실행한다. 이후 Analyzer에서 버그를 탐지하고, Validator에서 버그를 검증하여 버그 탐지의 신뢰도를 향상하였다. 본 연구에서는 다양한 컴파일러와 아키텍처를 대상으로 실험을 수행하였으며, 그 결과 16건의 컴파일러 최적화 버그를 식별하여 보고하였다.

I. 서론

컴파일러 최적화 모듈은 프로그램의 런타임 성능을 향상하거나 크기를 줄이기 위해 다양한 최적화 기능을 지원하며[1], 대부분의 소프트웨어 제품은 프로그램을 출시하기 전에 필수적으로 컴파일러 최적화를 수행한다[2].

현대 컴퓨팅 환경에서는 사용자의 목적과 요구사항에 맞춰 다양한 아키텍처를 활용하고 있으며, 각 아키텍처는 고유의 명령어 집합 구조(ISA), 코드 오프셋 및 함수 호출 규약(Calling Convention)을 가지고 있어, 상이한 기계어 코드를 실행해야 한다[3].

하지만, 아키텍처의 다양성은 컴파일러 최적화 과정에서 예측할 수 없는 많은 문제를 야기할 수 있다. 컴파일러가 동일한 최적화 기능을 각 아키텍처에 맞게 상이하게 구현해야 하기 때문이다. 그러므로 다양한 환경에서 일관된 성능과 정확성을 유지하기 위해 컴파일러 최적화 버그를 식별하고 수정하는 것이 중요하며, 이를 통해 컴파일된 프로그램의 안정성과 신뢰성을 보장하여 잠재적인 취약점을 방지할 수 있다.

본 연구에서는 Multi-Processing을 활용하여 컴파일 한 결과를 실행하고 비교하여 아키텍처별 컴파일러 최적화 버그를 자동으로 탐지할 수 있는 접근법을 제안한다. 해당 접근법을 기반으로 MSVC, LLVM, GCC 컴파일러 대상 차분 퍼즈 테스트(Differential Fuzz Test)를 수행하였다. 실험 결과 X86-64, MIPS64, MIPS64EL, RISC-V, ARM64, S390x 아키텍처에서 총 16건의 버그를 탐지하였다.

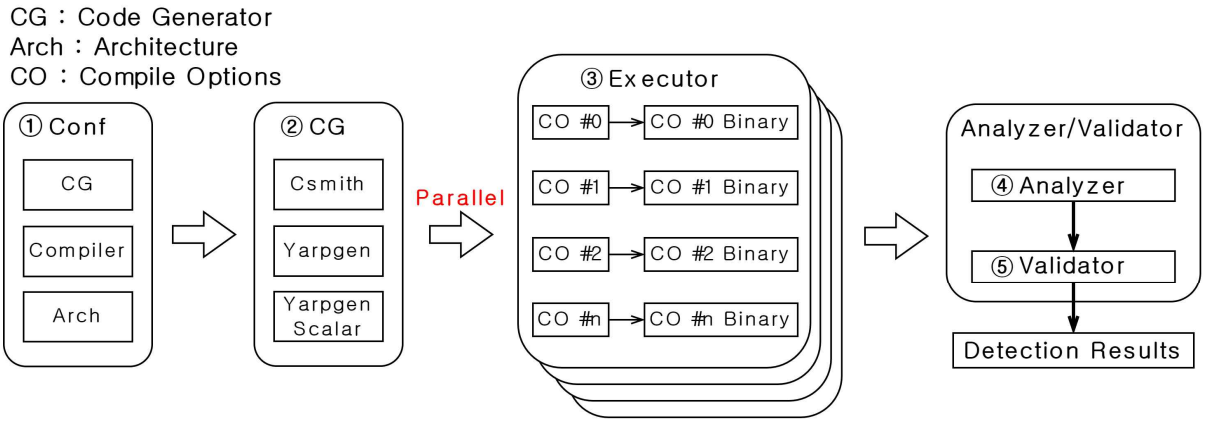

Fig. 2. Differential Fuzzer Overview

II. 배경지식

2.1 컴파일러와 아키텍처

컴파일러는 일반적으로 세 가지 컴포넌트로 구성되어있다: Front-, Middle-, Back-End[4]. 대표적인 컴파일러인 LLVM의 각 컴포넌트에서 수행하는 역할은 Fig 1과 같다.

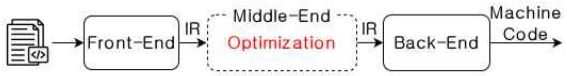

Fig. 1. LLVM Compile Process

Front-End에서는 소스 코드를 분석하여 구문과 형식을 검사하고 이를 IR(중간 표현)으로 변환한다. Middle-End에서는 사용자가 지정한 최적화 옵션에 맞는 다양한 기능을 적용하기 위해, IR을 기반으로 최적화를 진행한다. Back-End에서는 최적화된 IR을 아키텍처에 부합하는 기계어 코드로 변환한다.

MOCKINGBIRD[3]를 사용한 실험 결과, 다양한 아키텍처에서 컴파일된 바이너리는 각 아키텍처의 고유한 특성 때문에 컴파일러 코어가 같더라도 ARM, MIPS, X86 등 각 아키텍처 사이에서 정확도의 차이를 보였다. 이로 인해 서로 다른 아키텍처에서 동일한 코드가 예상치 못한 버그를 발생시킬 가능성이 있다.

2.2 컴파일러 버그 탐지 방법론

버그 탐지는 소프트웨어 개발 분야에서 중요한 주제로 다루어져 왔다. 정적 분석과 동적 분석은 버그 탐지 주요 방법론으로, 각각 코드를 실행하지 않고 분석하는 방식과 실행 중인 프로그램의 동작을 검사하는 방식을 의미한다[6].

컴파일러 버그 탐지에 있어 동적 분석을 수행하려면 언어의 문법을 준수하는 소스 코드가 필요하다. 버그 탐지를 위해 개발자가 수동으로 테스트 프로그램을 작성하였으나, 임의의 코드를 생성하는 코드 생성기가 개발되어 자동화된 테스트를 가능하게 하였다[7]. 그러나 임의로 생성된 코드는 복잡하고 방대하여, 버그가 발생한 부분을 식별하는 데 많은 자원을 소모하게 되었다. 이 문제를 해결하기 위해 본 연구에서는 사용자 정의 옵션을 설정하고, 아키텍처별로 소스 코드를 병렬로 컴파일 및 실행하여 결과를 비교하는 방식의 차분 퍼징을 수행하였다.

III. 시스템 개요

본 연구에서 제안하는 퍼저는 Conf, Code Generator, Executor, Analyzer, Validator로 구성되어 있으며, 구조는 Fig 2와 같다.

① 퍼저를 실행하기 전 테스트에 적용할 코드 생성기와 컴파일러 및 아키텍처를 사용자가 지정한다. 아키텍처의 엔디안 타입에 따른 데이터 처리 방식 차이로 False Positive의 위험을 높일 수 있기에, 엔디안 옵션을 구분하여 적용한다. ② Code Generator에서 테스트케이스를 생성하며 ③ Executor에서 테스트케이스를 컴파일 및 실행한 뒤, ④ Analyzer에서 실행 결과를 비교하여 버그를 식별한다. ⑤ Validator에서는 버그 중복 여부 검증을 통해 False Positive를 제거한다.

IV. Code Generator: Test-Case 생성

Code Generator는 C 표준을 준수하는 코드 생성기인 Csmith[8], Yarpgen[9], Yarpgen Scalar[9]를 사용한다. 각 코드 생성기는 다음과 같은 특징을 가지고 있다:

- Csmith: C언어 문법의 다양한 기능을 활용한 임의의 코드를 생성한다.
- Yarpgen: 복잡한 반복문 연산을 수행하는 임의의 코드를 생성한다.
- Yarpgen Scalar: 복잡한 산술 연산 및 형 변환을 수행하는 임의의 코드를 생성한다.

퍼징 속도를 증진하고 버그 분석 과정의 효율성을 개선하기 위해, 사용자가 코드 복잡도를 설정할 수 있도록 퍼저를 구현하였다.

V. Executor: 컴파일 및 실행

Executor는 사용자가 퍼즈 테스트 수행 전 설정한 다양한 옵션에 따라 컴파일러 및 아키텍처별로 동작한다. Multi-Processing을 통해 병렬적으로 코드를 컴파일하고 프로그램을 실행한 뒤, 성공 여부와 에러에 관한 정보를 Analyzer로 전달한다.

VI. Analyzer/Validator: 결과 분석

6.1 Analyzer: 컴파일러 버그 탐지

Analyzer는 컴파일 및 실행 결과에 대한 분석을 수행한다. 최적화를 수행하지 않는 CO #0(i.e., 최적화 미적용)을 Ground Truth로 정의하여, 다양한 아키텍처와 컴파일러 간의 결과 일관성을 검증한다. Ground Truth와 상이한 결과라면, 이를 잠재적인 버그로 간주하고 Table 1에 따라 분석 우선순위를 부여한다.

Table 1. Bug Analysis Priority

Priority 1	Different CheckSum
Priority 2	Crash
Priority 3	Compile Failed

6.2 Validator: 중복된 버그 제거

신규 발견된 버그가 이전 테스트에서 발견되고 제보되었지만, 아직 패치 되지 않은 버그일 가능성이 있어 버그 중복 검사는 필수적이다. Validator는 탐지된 버그들에 대한 정밀한 검증 및 중복 제거를 수행하며, 세부 컴포넌트는 Fig 3과 같다. 최초 발견된 버그는 Validator Manager로 전달되며, 내부 버그 정보 DB와 비교하여 버그의 중복 여부를 확인한다.

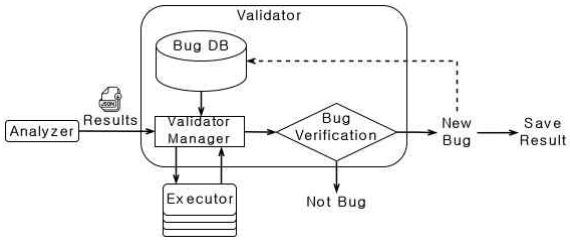

Fig. 3. Validator Structure

Validator는 버그 탐지의 신뢰도를 높이는 핵심 도구이며, 그 기능은 계속 확장될 수 있다. 새로운 버그 유형 발견 시, Validator 로직에 추가하여 미래의 유사한 버그를 효과적으로 판별할 수 있다. 시스템은 다음과 같은 3개의 검사 로직을 적용한다:

- Logic 1 Ground Truth에서 타임아웃이 발생하고 다른 옵션에서 타임아웃이 발생하지 않으면, 해당 문제는 무한 루프 최적화 버그와 중복되는 것으로 판단한다.
- Logic 2 아키텍처별 Data Model의 다양성으로 인해 탐지되는 케이스를 검증하기 위해 공통된 크기의 자료형으로 치환하여 해당 코드를 재컴파일 및 실행한다.
- Logic 3 소스 코드 내에 존재할 수 있는 무작위 요소로 인해 다른 결과를 출력할 가능성이 있다. 이를 검증하기 위해 여러 번 재컴파일 및 실행하여 결과의 일관성을 확인한다.

VII. 실험 결과

7.1 실험 환경

Intel(R) Core(TM) i7-1260P, 16GB 메모리 성능에 준하는 머신 12대를 활용하여 4주간 실험을 진행하였다. MSVC 19.37.32824.0 (X86-64), LLVM Trunk(X86-64, ARM64, MIPS64(EL), PowerPC64(EL), MIPS64, PowerPC64, S390x), GCC 11.4.0(X86-64, ARM64, MIPS64(EL), PowerPC64(EL), MIPS64, PowerPC64, S390x)를 대상으로 실험을 진행하였으며, 이를 위해 Python 기반의 퍼저(2,039 LoCs)를 개발하였다.

7.2 식별한 컴파일러 버그

본 연구에서는 퍼저를 활용해, Table 2와같이 다양한 아키텍처에서 총 16건의 컴파일러 최적화 버그[10]를 탐지하고 제보하였다.

Table 2. Bug Type

Compiler	Arch	Bug Type	Count
MSVC	X86-64	Crash	2
MSVC	X86-64	Incorrect operation	3
LLVM	MIPS64 (EL)	Incorrect operation	2
GCC	S390x	Incorrect operation	2
MSVC	X86-64	Wrong addresses interpretation	1
LLVM	X86-64	Infinite loop	1
LLVM	RISC-V	Signed extension	1
MSVC	X86-64	Signed extension	2
MSVC	X86-64	Missing local variable stack initialization	1
LLVM	ARM64	Skip pointer dereference	1

VIII. 결론

현대 컴퓨팅 환경에서 아키텍처의 다양성은 컴파일러 최적화 과정 중 예상치 못한 오류나 시스템의 취약점이 발생할 가능성을 내포한다.

본 연구에서는 이러한 문제를 해결하기 위해 다양한 아키텍처에서 컴파일러 최적화 버그를 탐지하는 새로운 방법론을 제안하였다. 자동화된 퍼징 기법을 활용하여 4주 동안 테스트를 수행한 결과, 다양한 아키텍처에서 총 16건의 버그를 성공적으로 식별하고 보고하였다.

향후 후속 연구에서는 탐지된 버그가 실제로 보안 취약점을 유발하는지 검증할 것이다. CodeQL과 같은 정적 분석 도구를 활용하여, 컴파일러 최적화 버그가, 실제 취약점으로 연결될 가능성이 존재하는지 분석할 예정이다.

[참고문헌]

[1] CHEN, Junjie, et al. Efficient compiler autotuning via bayesian optimization. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021. p. 1198-1209.

[2] J Yang et al. Isolating Compiler Optimization Faults via Differentiating Finer-grained Options 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 2022, pp. 481-491, doi: 10.1109/SANER53432.2022.00065.

[3] HU, Yikun, et al. Cross-architecture binary semantics understanding via similar code comparison. In: *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2016. p. 57-67.

[4] MA, Haoyang. A Survey of Modern Compiler Fuzzing. *arXiv preprint arXiv:2306.06884*, 2023.

[5] GEORGIOU, Kyriakos, et al. Lost in translation: Exposing hidden compiler optimization opportunities. *The Computer Journal*, 2022, 65.3: 718-735.

[6] SAMARASEKARA, Piyumika, et al. A Comparative Analysis of Static and Dynamic Code Analysis Techniques. 2023.

[7]CHEN, Junjie, et al. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, 2020, 53.1: 1-36.

[8] csmith, https://github.com/csmith-project/csmith

[9] yarpgen, https://github.com/intel/yarpgen

[10] https://github.com/BoBpiler/MultiArchCompilerOptimizationBugs