# 컴파일러 최적화 버그 자동 탐지 체계

백규민 [O1] 신재형 [1] 김동건 [1] 김대영 [1] 김진영*
[O1] 아주대학교 사이버보안학과 [1] 중부대학교 정보보호학과 [1] 부경대학교 컴퓨터공학과
[1] 세명컴퓨터고등학교
*성균관대학교 소프트웨어학과 박사과정
guminb@ajou.ac.kr, 201811420@pukyong.ac.kr, 22s302h0659@sonline20.sen.go.kr,
shinwogud12@naver.com, ckatoqlenfrl@gmail.com

# Automatic Detection of Compiler Optimization Bug

Gyu-Min Baek[O1] Jae-Hyeong Shin[1] Dong-Geon Kim[1] Dae-young Kim[1] Jin-Yeong Kim*
[O1] Department of Cyber Security, Ajou University
[1]Department of Information Security, Joongbu University
[1]Department of Computer Science and Engineering, Pukyong University
[1]Semyeong Computer Highschool (Highschool student)
*Department of Software, Sungkyunkwan University

**요 약**

컴파일러는 컴파일 과정에서 프로그램의 런타임 성능을 향상하기 위한 최적화를 수행한다. 그러나 복잡한 최적화 과정으로 인해 개발자의 의도와 다른 동작을 수행하는 버그를 내포할 가능성이 존재한다. 이러한 버그는 잠재적으로 프로그램의 안정성에 악영향을 줄 수 있기 때문에, 반드시 컴파일러 개발 과정에서 해결해야 한다. 컴파일러 최적화 버그는 오류 메시지 없이 발생하기 때문에 Domain Knowledge가 없다면 버그의 종류를 식별하고 탐지하는 데에 어려움이 있다. 개발자들이 Domain Knowledge를 학습하기 위해서는 컴파일러의 내부 동작과 구조를 깊게 이해해야 하므로 상당한 시간과 노력이 필요하다. 본 연구에서는 사용자가 지정한 옵션을 바탕으로, Domain Knowledge에 의존하지 않고도 컴파일러 버그를 자동으로 식별하고 탐지할 수 있는 확장성 있는 접근 방식을 제안한다.

최적화를 수행하지 않은 바이너리의 실행 결과를 Groundtruth(i.e.,, 올바른 결과)로 정의하였고, 최적화 옵션별로 Groundtruth에 위배되었는지 검증하였다. 실험 결과, 기존에 발견되지 않은 5개의 버그 종류를 새롭게 식별하였고, 케이스 스터디를 통해 본 연구에서 탐지한 여러 버그를 자세히 분석하였다.

## 1. 서 론

컴파일러의 최적화는 프로그램 성능 향상을 위해 필수적인 컴포넌트이며, 대부분의 소프트웨어 제품은 컴파일러 최적화 단계를 수행한 후에 출시된다[1]. 그러나 컴파일러에 버그가 있을 경우 최적화 과정에서 개발자의 의도와 다른 동작을 프로그램에 포함시킬 위험성이 존재한다[2]. 이러한 문제는 오류 메시지 없이 발생하기 때문에 Domain knowledge가 없는 일반 사용자는 버그를 탐지하기 어렵다. 하지만 높은 수준의 Domain knowledge를 학습하기 위해서는 프로그래밍 언어, 아키텍처, OS 등의 지식을 구비해야 하므로, 문제 해결을 위한 전문가 양성에 많은 자원이 소모된다. 본 연구에서는, Domain knowledge 없이 자동으로 컴파일러 최적화 버그를 탐지할 수 있는 접근법을 제안한다. 우리는 최적화가 이루어지지 않은 바이너리의 실행 결과를 Groundtruth로 정의하였고, 다양한 최적화 옵션을 사용하여 사전 정의한 규칙들을 기반으로 Groundtruth에 위배되었는지 확인하여 Domain Knowledge의 의존성을 배제하였다. 탐지 성능을 평가하기 위해, 사용자가 지정한 다양한 옵션에 따라 유연하게 적용될 수 있는 확장성 있는 자동화된 프로토타입을 제작했다. 이를 통해 퍼즈 테스트(Fuzz Test)를 수행한 결과 MSVC 19.37.32824에서 5개의 버그를 발견하였다.

## 2. 배경지식

### 2.1 컴파일러 최적화

소프트웨어 코드의 품질은 성능에 결정적인 역할을 하며, 컴파일러의 최적화 과정에 크게 의존한다. 최적화를 통해 코드의 메모리 사용을 최소화하고 자원을 효율적으로 활용하면, 시스템의 안정성 향상과 오버헤드 감소와 같은 이점을 획득할 수 있다. 그러나 최적화 과정에서의 버그 발생 가능성을 주의해야 한다. 2.2 에서 관련된 사례를 소개한다.

### 2.2 컴파일러 최적화 버그 사례

**그림 1** 은 컴파일러의 데드 스토어 제거(Dead Store Elimination) 최적화로 인해 문제가 발생한 사례이다[3].

```
1  void GetData(char *MFAddr) {
2    char pwd[64];
3    if (GetPassword(pwd, sizeof(pwd))) {
4      if (ConnectMainframe(MFAddr, pwd)) {
5        // Mainframe 에서 pwd 버퍼 사용
6      }
7    }
8    memset(pwd, 0, sizeof(pwd));
9  }
```

그림 1. CWE-14 PoC

최적화를 수행하지 않는다면 memset() 함수 호출은 개발자가 의도한 대로 pwd 버퍼를 0 으로 덮어쓴다(8 번째 라인). 그러나 최적화를 수행하면서, 컴파일러는 5 번째 라인 이후 pwd 버퍼가 더 이상 사용되지 않기 때문에 pwd 버퍼에 값을 덮어쓰는 행위가 불필요하다고 판단한다. 이에 따라 컴파일러는 8 번째 라인을 데드 스토어 제거(Dead Store Elimination) 기법을 통해 memset() 함수 호출 코드를 제거한다. 해당 버그는 pwd 변수 초기화를 막기 때문에, 메모리 누수(Memory Leak)가 발생하여 공격자에 의해 중요 정보(pwd)가 탈취될 수 있다. 이처럼 최적화 과정에서 개발자의 의도와 다른 동작을 수행하는 버그를 포함한 바이너리를 생성할 가능성이 있다.

### 2.3 컴파일러 최적화 버그 탐지 방법론

컴파일러 테스팅의 역사를 살펴보면, 초기에는 주로 소스 코드 분석과 실행 기반의 단순한 테스트가 주를 이뤘다. 하지만 컴파일러의 복잡성 증가와 최적화 기술의 발전으로 인해 보다 정교하고 체계적인 테스팅 접근 방식이 필요해졌다. 이에 따라 CHEN 등[4]은 컴파일러 테스트를 하기 위해 RDT(Random Differential Testing, 무작위 차등 테스팅), DOL(Different Optimization Levels, 다른 최적화 수준), EMI(Equivalence Modulo Inputs, 입력에 대한 등가성) 세 가지 기법을 제안하였다.

본 논문에서는 최적화 수준에 따른 컴파일러 버그를 탐지하는 데 특화된 DOL 방법론 기반 퍼저를 개발하였다. 기존 연구와의 차별화되는 점으로, 버그 탐지와 분류의 효율성을 높이기 위해 Analyzer 와 Validator 로직을 추가하였다. 또한, 확장 가능성을 목표로 설계되어 다양한 사용자 설정을 지원한다. 이러한 확장성은 사용자가 다양한 컴파일러 환경과 최적화 설정에 맞춰 퍼저를 유연하게 적용할 수 있도록 한다.

## 3. 컴파일러 최적화 버그 탐지 체계

### 3.1 Random Code Generator

본 논문에서는 컴파일러 퍼즈 테스트를 위해 임의 코드를 생성하는 Csmith[5]와 Yarpgen[6]을 사용하였다. Csmith 는 C 표준을 엄격히 준수하면서 다양한 C 언어 기능을 활용해 코드를 생성한다. Yarpgen 은 정의되지 않은 내부 행위를 배제하면서도 표현력이 높은 코드를 생성한다. 두 가지 코드 생성기는 컴파일러 버그 탐지를 위해, 사용자가 알아보기 어려운 복잡한 코드를 생성한다. 그러나 대부분 컴파일러 최적화 버그들의 개념증명은 45 줄 미만의 비교적 작은 C 코드에서 발견된다[7]. 따라서 코드 생성기의 소스 코드를 수정하여 테스트하는 코드의 복잡도를 감소시켰다.

### 3.2 퍼저 구조

우리는 컴파일러의 최적화 과정에서 Domain Knowledge 에 의존하지 않고 다양한 버그를 탐지할 수 있는 퍼저를 설계하였다. 우리는 확장성을 고려하여 다음과 같은 구조를 제안하였으며, 그 결과 다양한 종류의 컴파일러를 대상으로 쉽게 적용할 수 있게 만들었다. **그림 2** 는 컴파일러 최적화 버그 탐지를 위한 퍼저의 구조를 보여준다. 제안한 퍼저는 Generators, Executor, Analyzer, Validator 로 구성되어 있으며, 각각의 역할은 다음과 같다.

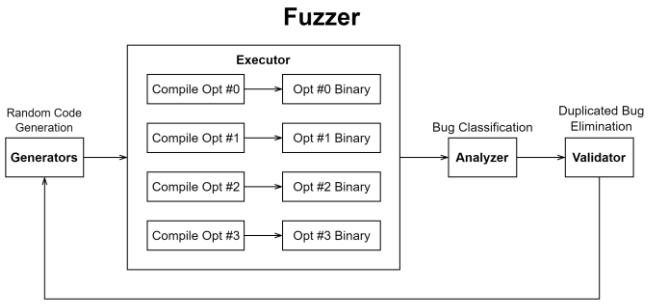


**그림 2. 퍼저 구조**

- **Generators**: 사용자가 지정한 코드 생성기 및 옵션을 기반으로 임의 소스 코드를 생성한다.
- **Executor**: multi-processing 을 통해 병렬적으로 타겟 컴파일러의 최적화 옵션별로 소스 코드를 컴파일하고 바이너리를 실행한다. 이때, 컴파일 성공 여부 및 오류에 대한 정보를 기록하고, 생성된 바이너리를 실행하여 결과를 저장한다.
- **Analyzer**: 하나의 소스 코드에 대해 수행한 모든 결과를 기반으로 크게 컴파일 타임과 런타임 두 가지 측면에서 버그를 탐지한다. 특히 컴파일러의 최적화 옵션에 따라 일관된 결과가 출력되었는지를 검증한다. 최적화를 수행하지 않은 결과를 Groundtruth 로 지정하고, 컴파일과 런타임에서 이에 위배되는 결과를 탐지한다. **표 1** 을 기반으로 탐지된 버그에 분석 우선 순위 규칙을 적용함으로써, 사용자가 중요한 버그부터 분석할 수 있도록 하였다.
- **Validator**: 중복되거나 알려진 패턴의 버그를 자동으로 제거하여, 탐지된 케이스를 검증한다. 이 기능은 컴파일러 최적화 과정에서 발견되는 버그들을 더 정확하게 식별할 수 있도록 도와준다.

**표 1. 분석 우선 순위**

| 우선 순위 | Groundtruth 위배 케이스 |
|---|---|
| High | Checksum이 다른 경우 |
| Medium | 부분적인 컴파일 실패 및 비정상 실행 |
| Low | 부분적인 타임아웃(컴파일, 바이너리 실행) |

## 4. 실험

### 4.1 실험 환경

우리는 Windows 11 Pro (OS Build 22621.2283) OS 를 사용하여 실험을 진행하였다. Intel(R) Core(TM) i7-1260P, 16GB 메모리 성능에 준하는 머신 9 대를 활용하였다. MSVC 19.37.32824 (x64)를 대상으로 실험을 진행하였으며, 이를 위해 Python 기반의 퍼저(1,051 LoCs)를 개발하였다.

### 4.2 실험 결과

실험에서는 퍼저를 통해 탐지한 버그를 분석하여, **표 2** 와 같이 5 개의 버그 유형으로 분류하였다. 반복문, 형 변환(casting) 및 다양한 대입 연산이 포함된 복잡한 표현식에서 버그가 발생하였다.

**표 2. 버그 유형**

| Case | Compiler | Status | Bug Type |
|---|---|---|---|
| 1[8] | MSVC | Fixed | 루프로 인한 Side Effect |
| 2[9] | MSVC | UI | 잘못된 부호 처리 |
| 3[10] | MSVC | UC | 잘못된 변수 주소 인식 |
| 4[11] | MSVC | UC | 부정확한 최적화 연산 |
| 5[12] | MSVC | UC | 부정확한 함수 호출 순서 |

UI†: Under Investication, UC‡: Under Consideration

## 5. 케이스 스터디

**그림 3** 은 MSVC 컴파일러에서 발견한 버그 중 하나로, 특정 최적화 옵션에서 Groundtruth 를 위배한다.

```
1   int globalVal = 0;
2   int *globalPtr = &globalVal;
3   void func() {
4     int localVal = 0;
5     int i;
6     for (i = 0; i < 1; i++) {
7       // 버그를 유발하는 반복문
8     }
9   for (i = 0; (i > (-2)); i--) {
10    localVal ^= (i > ((uint64_t)(-2)));
11    globalVal = i;
12  }
13    *globalPtr = localVal;
14  }
15  int main () {
16    func();
17    printf("globalVal: %d\n", globalVal);
18    return 0;
19  }
```

**그림 3. Case 1 PoC**

main() 함수에서 호출하는 func() 함수의 결과로 globalVal 변수에 '1'이 저장되어야 한다. 하지만 특정 최적화 옵션에서는 반복문이 알 수 없는 부작용(Side Effect)을 발생시켜(6~8 번째 라인), func() 함수에서 globalVal 변수에 '0'을 저장한다.

## 6. 결론

컴파일러 최적화 과정은 소프트웨어의 성능 향상에 필수적이지만, 컴파일러 자체의 불완전성으로 인해 예기치 않은 버그가 발생할 수 있다. 최적화 관련 버그는 대체로 눈에 띄지 않으며, 심각한 프로그램 오류나 잠재적인 취약점으로 연결될 수 있다[13].

본 논문에서는 Domain Knowledge 없이 자동화된 방법을 활용하여 컴파일러 최적화 버그를 효과적으로 탐지하는 방법을 제시하였다. MSVC 컴파일러를 실험 대상으로 선정하고, 3 주간의 실험을 통해 여러 최적화 버그를 성공적으로 식별하고 보고하였다.

향후에는 다양한 컴파일러 및 아키텍처를 퍼즈 테스트 환경에 적용할 수 있도록 확장성을 개선하여, 보다 많은 버그를 탐지할 수 있도록 할 예정이다.

## 참고 문헌


[1] 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.

[2] KELLAS, Andreas D., et al. Divergent Representations: When Compiler Optimizations Enable Exploitation. In: *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2023. p. 337-348.

[3] https://cwe.mitre.org/data/definitions/14.html

[4] CHEN, Junjie, et al. An empirical comparison of compiler testing techniques. In: Proceedings of the 38th International Conference on Software Engineering. 2016. p. 180-190.

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

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

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

[8] https://developercommunity.visualstudio.com/t/Compiler-bug-causing-unknown-behavior/10481332?sort=newest

[9] https://developercommunity.visualstudio.com/t/Signed-variable-value-extended-in-an-uns/10478879?sort=newest&q=Signed+variable+value+extended+in+an+unsigned+manner&page=3

[10] https://developercommunity.visualstudio.com/t/Memory-reference-error-due-to-excessive/10477735?sort=newest&page=1

[11] https://developercommunity.visualstudio.com/t/It-optimizes-the-and-operation-into-x/10481313

[12] https://developercommunity.visualstudio.com/t/O1-Optimization-Leads-to-Incorrect-Funct/10469220?sort=newest

[13] XU, Jianhao, et al. Silent Bugs Matter: A Study of {Compiler-Introduced} Security Bugs. In: 32nd USENIX Security Symposium (USENIX Security 23). 2023. p. 3655-3672.