# Compiler Provenance using Frequency Analysis

Yongwoo Lee
KAIST
Korea
ywlee97@kaist.ac.kr

## 요 약

바이너리를 보고 어떤 컴파일러로 컴파일 되었는지 알아내는 기술은 여러 분야에서 활용될 수 있다. 그 예로 악성코드 분석, 성능 분석, 바이너리 코드 계측 등을 들 수 있다.

그러나 심볼이 제거된 바이너리를 보고 어떤 컴파일러로 컴파일 되었는지 아는 것은 쉽지 않은 일이다. 따라서 본 논문에서는 바이너리에 사용된 명령어들의 빈도수를 분석하여 어떤 컴파일러로 컴파일 되었는지 알아내는 방법을 제시한다.

## CCS CONCEPTS

• **Software and its engineering → Compilers**.

## KEYWORDS

compiler provenance, frequency analysis

**ACM Reference Format:**

Yongwoo Lee. 2021. Compiler Provenance using Frequency Analysis. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1. INTRODUCTION

바이너리를 보고 어떤 컴파일러로 컴파일 되었는지 알아내는 기술은 여러 분야에서 활용될 수 있다. 예를 들어 악성코드 분석에서는 악성코드의 제작자를 알아내기 위해 컴파일러 정보가 사용될 수 있다. 혹은 바이너리 코드 계측을 할 때 컴파일러 정보를 알고 있다면 더 효율적으로 계측을 할 수 있다.



*Conference'17, July 2017, Washington, DC, USA*



ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

그러나 모든 심볼을 포함한 메타 데이터가 없는 바이너리에서는 컴파일러의 정보를 알아내기 쉽지 않다. 메타 데이터가 제거된 바이너리에서 컴파일러 정보를 얻기 위해 SVM과 같은 전통적인 머신 러닝 기법을 활용하는 연구 [6, 7]나 GNN (Graph Neural Network)를 활용하는 연구 [1]가 존재한다. 그러나 이런 연구들은 1) CFG (Control Flow Graph)를 필요로 한다는 점과 2) 모델을 학습시켜야 한다는 한계점이 있다.

본 논문에서는 바이너리에서 사용된 명령어의 빈도수를 분석하여 유사도를 측정하고 이를 기반으로 컴파일러 정보를 얻어내는 방법을 소개한다. 이 방법은 바이너리의 CFG를 사용하지 않고 모델을 학습시키는 과정이 없기 때문에 앞서 설명한 방법들보다 훨씬 간단하고 효율적이다. 이를 통해 주어진 45개의 바이너리를 대상으로 약 97.8%의 정확도를 얻을 수 있었고 이 방법이 여러 컴파일 옵션과 아키텍쳐에서의 환경에도 적용 가능함을 보인다. 그리고 모든 구현은 깃헙[1] 에 올려두었다. 이는 과제 제출기한이 모두 지난 후 public으로 전환하였다. 과제 제출에는 용량 제한이 걸려 있어서 코드만 첨부했으며 실험을 위해서는 깃헙에 올려진 데이터를 함께 사용해야 한다.

2장에서는 풀고자 하는 문제의 정의를 포함한 배경 지식을 설명한다. 3장에서는 제안하는 방법에 대한 구체적인 설명을 하고 4장에선 평가 및 결과를 서술한다. 5장에서는 관련 연구를 소개하고 마지막으로 6장에 결론을 서술하며 논문을 마친다.

## 2. MOTIVATION

본 장에서는 빈도수 분석을 통한 컴파일러 특정 방법의 동기를 설명하고 구체적인 문제 상황을 정의한다.

### 2.1 Compiler Preferred Instructions

서로 다른 컴파일러들은 같은 소스코드여도 다른 바이너리 코드를 생성한다. 이 때 같은 로직을 컴파일하는 방법도 다양한데 예를 들어 조건문을 만났을 때 조건을 만족시키면 점프하는 방법이 있을 수 있고 만족시키지 못했을 때 점프하는 방법이 있을 수 있다. 이 외에도 다양한 구현의 차이가 있고 이 과정에서 각 컴파일러가 자주 사용하는 명령어가 서로 다를 것이라 생각했다.

[1] https://github.com/yongwoo36/compiler_provenance

그림 1은 gcc, icc, clang으로 각각 컴파일 한 `elfedit` 바이너리의 `unix_lbasename` 함수의 내용이다. 분명 같은 동작을 하는 함수이지만 컴파일러마다 생성해내는 어셈블리 코드는 모두 다름을 볼 수 있다. 사용하는 명령어의 종류 및 개수는 표 1에 정리했다.

### 2.2 Problem Definition

본 장에서는 문제 상황을 구체적으로 정의한다. 먼저 컴파일러는 gcc, icc, clang 이 세가지만 사용한다고 가정한다. 바이너리 데이터는 주어진 총 45개의 바이너리를 사용했다[2]. 그리고 Binkit[5]에 올려져 있는 다양한 아키텍쳐에서 gcc와 clang으로 컴파일 된 바이너리들을 추가적으로 평가에 사용했다.

이 때 바이너리에 존재하는 어떠한 메타 데이터도 사용할 수 없다고 가정한다. 예를 들어 어떠한 컴파일러로 컴파일 되었는지 적혀있는 `.comment` 섹션 을 포함한 모든 디버깅 심볼 또한 제거되었다고 가정했다.

## 3. DESIGN

바이너리 명령어 빈도수 분석은 아래와 같이 총 3단계로 이루어진다. 이 방법은 비지도 학습의 군집화에서 착안한 방법이다.

(1) 빈도수 벡터 생성
우선 빈도수 벡터란 각 opcode가 전체 중 몇 퍼센트를 차지하고 있는지를 나타내는 벡터이다. `.text` 섹션에 명령어들의 opcode들을 모두 추출하고 한 opcode가 전체 중 얼마를 차지하는지 백분율로 계산하여 기록한다. 여기서 백분율로 나타내는 이유는 같은 컴파일러로 컴파일 된 바이너리라도 크기가 다르다면 opcode의 개수도 달라지기 때문에 개수가 아닌 백분율로 나타냈다.

(2) 컴파일러의 평균 빈도수 벡터 계산
각 컴파일러마다 모든 바이너리의 빈도수 벡터의 평균을 구한다. 그리고 이 평균 빈도수 벡터를 컴파일러의 빈도수 벡터로 할당한다.

(3) 예측
컴파일러를 알 수 없는 새로운 바이너리가 들어오면 우선 해당 바이너리의 빈도수 벡터를 계산한다. 그리고 세가지 컴파일러의 빈도수 벡터와 각각 유클리드 거리를 계산 한 후 가장 가까운 컴파일러로 예측하게 된다.

이 방법은 충분히 다른 아키텍쳐, 다른 컴파일러에 대해 일반화 할 수 있다고 생각한다.

## 4. EVALUATION

빈도수 분석을 평가하기 위해 아래와 같이 세 가지 기준을 세웠다. 그리고 각 장에서 이에 대한 분석 결과를 설명한다.

- RQ1. 분석 및 예측 시간은 얼마나 걸리는가?
- RQ2. 예측이 정확한가?
- RQ3. 다른 조건 (아키텍쳐, 컴파일러 옵션 등)의 바이너리에 대해서도 잘 동작하는가?

### 4.1 RQ1

분석 및 예측에 소요되는 시간을 평가했다. 바이너리 하나의 빈도수 벡터를 생성하는 시간, 주어진 45개의 데이터를 모두 사용해서 각 컴파일러의 빈도수 벡터를 계산하는 시간, 그리고 입력 바이너리가 주어졌을 때 예측까지 걸리는 시간을 초 단위로 표 2에 나타냈다.

컴파일러 빈도수 벡터를 계산하는 시간은 바이너리의 개수와 거의 정비례함을 알 수 있다. 이는 병렬화를 전혀 고려하지 않은 설계이므로 컴파일러 빈도수 벡터를 계산할 때 병렬화를 도입한다면 시간을 많이 단축시킬 수 있을 것으로 보인다.

### 4.2 RQ2

우선 주어진 45개의 바이너리를 대상으로 정확도를 평가했다. 우선 같은 컴파일러로 컴파일된 바이너리는 비슷한 빈도수 벡터를 갖는가에 대한 실험을 해보았다. 같은 컴파일러로 컴파일된 두 바이너리를 랜덤하게 200쌍 뽑아 그 빈도수 벡터의 유클리드 거리를 계산해 보았다. 그 결과 평균적인 거리는 약 **4.155e-2**였다. 그리고 서로 다른 컴파일러에서 랜덤하게 뽑은 두 바이너리 200쌍의 평균 유클리드 거리는 **7.284e-2**였다. 두 값 모두 랜덤하게 뽑았기 때문에 결과가 매번 달라지기는 했으나 거의 모든 경우에 같은 컴파일러에서의 유클리드 거리와 다른 컴파일러에서의 유클리드 거리는 확연히 구분할 수 있었다.

그리고 3장에서 설명한 방법으로 주어진 45개의 바이너리를 예측해 보았다. 이때 컴파일러 빈도수 벡터 계산에 포함되는 데이터와 검증을 위한 데이터, 즉 학습 데이터와 테스트 데이터를 분리하기 위해 학습 데이터의 개수를 각 컴파일러마다 5개, 10개로 설정하고 각각 실험해 보았다. 5개로 설정했을 때의 남은 총 30개 중 평균적으로 **19**개의 정답을 맞힐 수 있었다. 학습 데이터를 10개로 했을 경우에는 남은 총 15개 중 **14**개의 정답을 맞힐 수 있었다. 그리고 모든 데이터를 사용해 컴파일러 빈도수 벡터를 계산했을 때 컴파일러의 빈도수 벡터와 각 바이너리들의 유클리드 거리의 평균은 표 3에 나타냈다. 가장 작은 유클리드 거리를 밑줄로 표시했다.

세로축은 정답을 나타내고 가로축은 컴파일러 빈도수 벡터와의 거리를 타나낸다. 예를 들어 gcc (gt) - clang 칸은 gcc로 컴파일한 바이너리들과 clang의 빈도수 벡터의 평균 유클리드 거리를

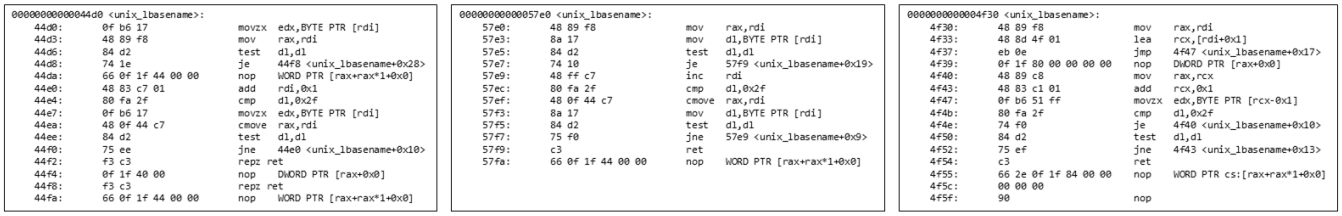


**그림 1: 왼쪽부터 차례대로 gcc, icc, clang으로 컴파일한 결과**

**표 1: 컴파일러마다 사용한 명령어의 빈도수**

| | add | cmove | cmp | inc | je | jmp | jne | lea | mov | movzx | nop | repz | ret | test |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **gcc** | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 0 | 2 |
| **icc** | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 3 | 0 | 1 | 0 | 1 | 2 |
| **clang** | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 2 | 1 | 3 | 0 | 1 | 1 |

**표 2: 시간 측정**

| | one frequency vector | compiler frequency vector | prediction |
|---|---|---|---|
| **time (s)** | 2.744 | 158.29 | 3.619 |

**표 3: 평균 유클리드 거리**

| | gcc | icc | clang |
|---|---|---|---|
| **gcc (gt)** | 0.765e-2 | 2.555e-2 | 1.440e-2 |
| **icc (gt)** | 2.630e-2 | 1.467e-2 | 2.221e-2 |
| **clang (gt)** | 1.599e-2 | 2.269e-2 | 0.821e-2 |

나타낸다. 이 경우 45개의 데이터 중 **44**개의 정답을 맞힐 수 있었다. 이를 통해 빈도수 분석을 통한 군집화가 제법 유의미한 결과를 도출할 수 있음을 알 수 있다.

## 4.3 RQ3

본 장에서는 빈도수 분석을 통한 방법이 다른 컴파일 옵션, 다른 아키텍쳐에서도 잘 동작하는지 살펴본다. 이때 사용한 데이터 셋은 Binkit[2]에 올라와 있는 바이너리를 사용했다. gcc와 clang의 다양한 버전과 다양한 컴파일 옵션으로 컴파일된 바이너리들 중 binutils 바이너리들을 사용했다. icc로 컴파일된 바이너리는 찾지 못해 추가 실험에 사용하지 못했다.

*4.3.1 Compile options.* 우선 컴파일 옵션에 따라 달라지는 결과를 살펴봤다. 이를 위해 아키텍쳐는 x86_64로 고정하고 gcc-8.2.0 버전, clang-7.0 버전의 바이너리들만 사용했으며 컴파일러 옵션은 O0, O1, O2, O3를 사용했다. 그 이유는 컴파일러의 버전마다 사용하는 명령어의 빈도수가 많이 달라질 것이라 생각했기 때문이다. 그럴 경우 앞서 주어진 45개의 바이너리를 통해 구한 컴파일러 빈도수 벡터와 바이너리의 빈도수 벡터가 많이 달라질 수 있다고 생각했다. 따라서 Binkit에 존재하는 gcc와 clang 중 가장 최신 버전의 컴파일러로 컴파일된 바이너리를 사용했다. 그 결과 총 112개의 바이너리 중 **83**개의 정답을 맞힐 수 있었다. 따라서 컴파일러 옵션이 달라지더라도 빈도수 분석을 통한 컴파일러 예측이 유의미함을 알 수 있었다.

*4.3.2 Architectures.* 아키텍쳐가 다른 경우 사용하는 명령어 셋이 다르다. 따라서 아키텍쳐마다 컴파일러 빈도수 벡터를 새로 계산해야 한다. 이 추가 실험에서 사용할 아키텍쳐로는 ARM_64와 MIPS_64를 선택했다. 아키텍쳐마다 gcc-8.2.0버전과 clang-7.2버전으로 컴파일 된 바이너리들을 사용했으며 컴파일러 빈도수 벡터를 계산하기 위해 각각 무작위로 선택한 15개의 바이너리를 사용했다. 이를 바탕으로 남은 바이너리들의 컴파일러를 예측해본 결과 ARM_64에서는 총 82개 중 **71**개의 정답을, MIPS_64에서는 82개 중 **82**개의 정답을 맞힐 수 있었다. 이를 통해 빈도수 분석을 통한 컴파일러 예측이 다른 아키텍쳐에서도 충분히 가능함을 보였다.

## 4.4 Discussion

여러 실험을 거치면서 이 방법을 개선시킬 수 있는 몇가지 방법에 대해 생각해 보았다. 우선 첫 번째 방법은 속도 개선이다. 앞서 4.1장에서 언급했듯이 지금의 구현체는 컴파일러 빈도수 벡터를 계산할 때 병렬화를 전혀 고려하지 않았다. 또한 한 바이너리의

빈도수 벡터를 계산할 때 B2R2 라이브러리를 매번 불러오는 식으로 동작하는데 이 과정에서 시간이 오래걸린다. 이러한 구현적인 부분을 개선한다면 계산 속도가 지금보다 훨씬 개선될 수 있을 것이다.

두 번째는 opcode 뿐만 아니라 레지스터 정보도 함께 활용하는 것이다. 컴파일러마다 자주 사용하는 레지스터가 다르다거나 레지스터의 사용 순서가 다르다는 정보를 함께 활용한다면 더 정확한 컴파일러 예측을 할 수 있을 것이다.

세 번째로 개선시킬 수 있는 부분으로는 다양한 컴파일러 버전 및 아키텍쳐에 대한 통합이다. 지금의 구현은 아키텍쳐마다 분리해서 실험을 진행했는데 여러 아키텍쳐에 대해 통합된 컴파일러 빈도수 벡터를 구해서 다양한 아키텍쳐의 바이너리가 섞인 테스트 데이터로 예측을 진행해 본다면 어떠한 결과가 나올지 궁금하다. 개인적으로는 사용하는 명령어 셋이 다르기 때문에 아키텍쳐마다의 컴파일러 빈도수 벡터를 단순히 합하는 것 만으로도 충분히 의미가 있을 수 있다고 생각한다. 또한 컴파일러 버전이 달라짐에 따라 빈도수 벡터가 달라질 것으로 생각했는데 4.3.1 장의 실험 결과를 보면 컴파일러 버전이 달라짐에도 얼추 정확한 예측을 할 수 있었다. 따라서 이 부분에 대한 추가 실험 및 구현도 재미있는 향후 연구가 될 수 있을 것이다.

마지막으로 이 방법의 한계점은 분류하고자 하는 컴파일러로 컴파일된 바이너리 데이터 셋이 필요하다는 점이다. 예를 들어 4.3 장의 실험에서 icc 로 컴파일된 바이너리가 입력으로 들어온다면 우리는 이를 icc로 구분해 낼 수 없다.

## 5. RELATED WORK

앞서 1장에서 소개했듯이 바이너리를 보고 컴파일러를 알아내는 연구[1, 6, 7]가 몇몇 존재한다. 앞서 언급한 세가지 논문 모두 전통적인 혹은 최신의 머신 러닝 기술을 사용한다. 반면 우리는 학습을 사용하지 않고 컴파일러를 알아내는 방법을 제시했다.

바이너리를 대상으로 명령어 빈도수 분석을 진행한 연구들이 몇몇 존재한다. 예를 들어 Gamayunov et al.[3]은 명령어 빈도수 분석을 통해 쉘코드를 탐지하는 연구를 하였다. 또한 Han et al.[4]은 명령어 빈도수 분석을 통해 악성코드를 탐지하는 연구를 진행했다. 우리는 명령어 빈도수 분석 방법을 컴파일러를 알아내는 문제에 적용했으며 컴파일러마다 생성하는 명령어들의 빈도수가 유의미하게 다르고 이를 통해 컴파일러를 알아낼 수 있음을 보였다.

## 6. CONCLUSIONS

본 논문에서는 바이너리의 명령어 빈도수 분석을 통한 컴파일러 예측 방법을 소개했다. 이는 기존의 연구들과는 달리 CFG 분석 혹은 모델 학습의 과정을 필요로 하지 않는 간단한 방법이다. 우리는 컴파일러마다 자주 사용하는 명령어가 다르다는 사실을 실험적으로 보였고 이를 활용해 바이너리의 명령어 빈도수 분석을 통해 어떤 컴파일러로 컴파일되었는지 알아낼 수 있음을 보인다. 이 방법은 컴파일 옵션이 달라지더라도 유의미한 결과를 도출할 수 있고 다른 아키텍쳐에도 동일하게 적용 가능함을 보였다. 아직은 본 연구가 미흡해서 현존하는 방법들에 비해 정확도는 부족하지만 더 많은 데이터 셋을 활용해 4.4 장에서 언급한 향후 연구를 진행한다면 유의미한 결과를 얻을 수 있다고 생각한다.

## 참고 문헌


[1] Tristan Benoit, Jean-Yves Marion, and Sébastien Bardin. 2021. Binary level toolchain provenance identification with graph neural networks. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 131–141. https://doi.org/10.1109/SANER50967.2021.00021

[2] Sang Kil Cha. 2021. *Is561 samples binarys.* https://softsec.kaist.ac.kr/depot/sangkilc/is561-samples.bz2

[3] Dennis Gamayunov, Nguyen Thoi Minh Quan, Fedor Sakharov, and Edward Toroshchin. 2009. Racewalk: Fast Instruction Frequency Analysis and Classification for Shellcode Detection in Network Flow. In *2009 European Conference on Computer Network Defense*. 4–12. https://doi.org/10.1109/EC2ND.2009.9

[4] Kyoung Soo Han, Boojoong Kang, and Eul Gyu Im. 2011. Malware Classification Using Instruction Frequencies. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation* (Miami, Florida) *(RACS '11)*. Association for Computing Machinery, New York, NY, USA, 298–300. https://doi.org/10.1145/2103380.2103441

[5] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. [n. d.]. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. ([n. d.]). arXiv:2011.10749 [cs.SE]

[6] Nathan Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2011. Recovering the Toolchain Provenance of Binary Code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 100–110. https://doi.org/10.1145/2001420.2001433

[7] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2010. Extracting Compiler Provenance from Program Binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. Association for Computing Machinery, New York, NY, USA, 21–28. https://doi.org/10.1145/1806672.1806678