Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components

USENIX 2014
Manuel Egele, Maverick Woo, Peter Chapman, David Brumley

김영철
2017. 1. 13.

Introduction

두 바이너리 코드 사이의 Semantic similarity를 결정하는 것이 보안 분야에서 중요한 문제

- Automatic patch-based exploit generation
  - 패치된 취약점 찾기
- Malware 샘플에서 비슷한 malware 찾기
  - 기록된 실행 패턴에 대하여 클러스터링 하는 방법

Introduction

Semantic binary differencing(diffing)

- BinDiff
  - 그래프 비교를 통한 탐지, 코드 블록 강조
  - 그래프 비교 문제는 시간이 오래 걸리지만, 휴리스틱을 이용하여 빠르게 수행
  - 같은 바이너리의 CFG가 비슷할 때, 사용 가능
  - 최적화 레벨에 따른 실험에서 정확도가 25%까지 하락

➔ 그래프 비교 방법 등 기존의 기술들을 사용하지 않은 새로운 binary diffing 알고리즘 제시

Introduction

New binary diffing algorithm

- 최적화나 난독화에 상관없이 비슷한 코드는 의미적으로 비슷한 실행 행동을 보인다.
- 고레벨에서 같은 input으로 바이너리를 직접 실행하고 관찰된 행동을 비교하여 유사도를 도출한다.
- Polynomial identity testing(PIT) problem을 적용하기 위해 semantics를 7개의 정보로 나타낸다.
  - full coverage를 위해 함수 내의 모든 명령어가 적어도 한 번씩 실행될 때까지 실행되지 않은 명령어부터 반복적으로 실행

Introduction

BLEX : dynamic equivalence testing system

- heap과 stack에 read/write 하는 값 (4개)
- library call
- system call
- 분석되는 함수의 실행이 끝날 때까지 eax 레지스터에 저장된 값들

- 위의 7가지 항목을 추출하여 가중치가 적용된 Jaccard 합을 계산

Blanket execution

Introduction

Contributions

1. full-coverage dynamic analysis 제안
2. 7개의 바이너리 코드 의미 추출기 제안
3. 문법적으로 차이가 있는 바이너리 사이에서 BinDiff 보다 성능이 좋음

Problem Setting and Challenges

Problem setting

- debug symbol이나 source code가 없는 바이너리 코드만 주어진 상황
- 바이너리 코드는 packing 되지 않음
- 컴파일러와 최적화 옵션을 고려함

Problem Setting and Challenges

Problem setting

```
1   static int strcmp_name(
2       V a, V b
3   )
4   {
5   return cmp_name(a, b, strcmp);
6   }
7
8   static inline int
9   cmp_name (
10    struct fileinfo const *a,
11    struct fileinfo const *b,
12    int (*cmp) (
13       char const *,
14       char const *)
15  )
16  {
17    return
18        cmp (a->name, b->name);
19  }
```

```
1   407ab9 <strcmp_name>:
2      ab9: push %rbp
3   ...
4      ad1: mov $0x402710,%edx
5   ... PLT entry of strcmp
6      ad6: mov %rcx,%rsi
7      ad9: mov %rax,%rdi
8      adc: callq 406fa1 <cmp_name>
9      ae1: leaveq
10     ae2: retq
11
12  406fa1 <cmp_name>:
13     fa1: push %rbp
14  ...
15     fcd: callq *%rax
16  ... call function pointer (e.g., strcmp)
17     fcf: leaveq
18     fd0: retq
```

```
1   4053e0 <strcmp_name>:
2       e0: mov (%rsi),%rsi
3       e3: mov (%rdi),%rdi
4       e6: jmpq 402590 <strcmp@plt>
```

Problem Setting and Challenges

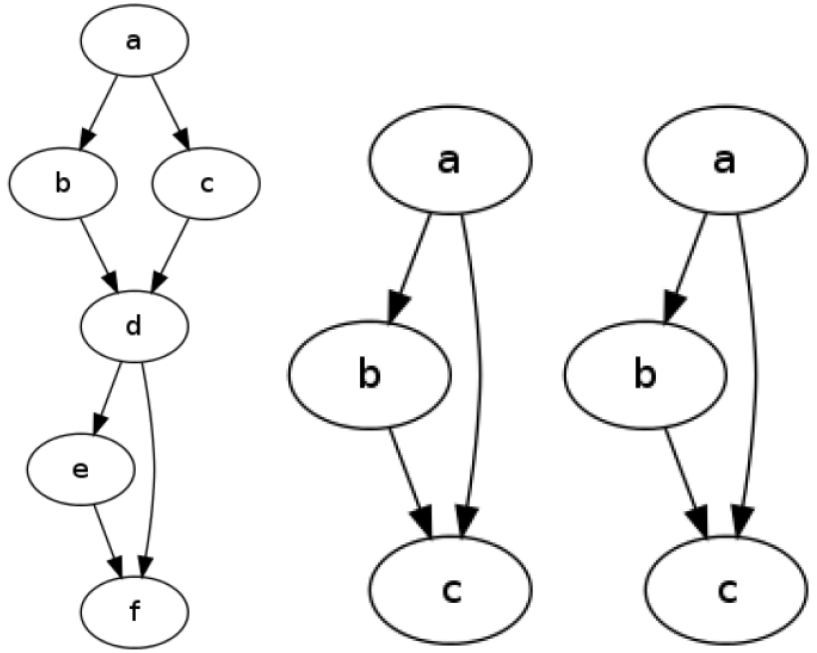
Challenges

- 의미적으로 비슷한 함수들이 문법적으로 비슷하지 않을 수 있음
- 메모리를 어떤 방법으로 read/write를 수행했는지 확인할 필요가 있음
- Inter-procedural and context sensitive analysis 수행해야함

Problem Setting and Challenges

Challenges

- 문법적으로 접근한 방법으로는 식별할 수 없음
- 그래프 비교를 통한 방법으로도 식별할 수 없음



(a) md5_finish_ctx (unoptimized) (b) md5_finish_ctx (optimized) (c) xstrxfrm (optimized)

Approach

Blanket Execution : 제시하는 dynamic analysis 방법

- 함수 f의 Blanket Execution
  - f를 반복적으로 실행하여 f의 명령어들이 적어도 한 번 실행됨을 보장하는 것
- full-coverage를 위해 실행되지 않은 명령어부터 연속적으로 실행
  - dynamic runtime information 들을 기록
- 함수 *f* 와 environment *env* 를 입력 받아 vector v(features)를 결과물로 내줌

Approach

1. Environment

environment : blanket execution이 발생하는 환경

- 바이너리를 실행하기 위해서 모든 레지스터와 메모리 정보가 있어야 함.
- 맵핑되지 않은 메모리에 대해서 무작위이지만 고정된 더미 메모리 페이지를 지정
- environment는 위 두 가지를 합친 것

Approach

2. Blanket Execution

함수 f와 environment env가 주어졌을 때,
env 환경에서 함수 f의 모든 명령어가 적어도 한 번씩 실행될 때까지 실행되지 않은 명령어부터 반복적으로 실행하는 것을 말함.

blanket execution run (be-run) : 실행 한 번

blanket execution campaign (be-campagin) : 같은 env에서의 be-run 집합

Approach

2. Blanket Execution

input : Function binary f, Environment env
output : Feature vector $\vec{v}(f, env)$ of f in env
$\mathbb{I} \leftarrow getInstructions(f)$
$fvec \leftarrow emptyVector()$
while $\mathbb{I} \neq \emptyset$ **do**
  $addr \leftarrow minAddr(\mathbb{I})$
  $(covered, v) \leftarrow be\text{-}run(f, addr, env)$
  $\mathbb{I} \leftarrow \mathbb{I} \setminus covered$
  $fvec \leftarrow pointwiseUnion(fvec, v)$
end
return $fvec$

수행되지 않은 명령어 중에서 주소가 작은 것부터 blanket execution

feature 집계

Approach

3. Assessing Semantic Similarity

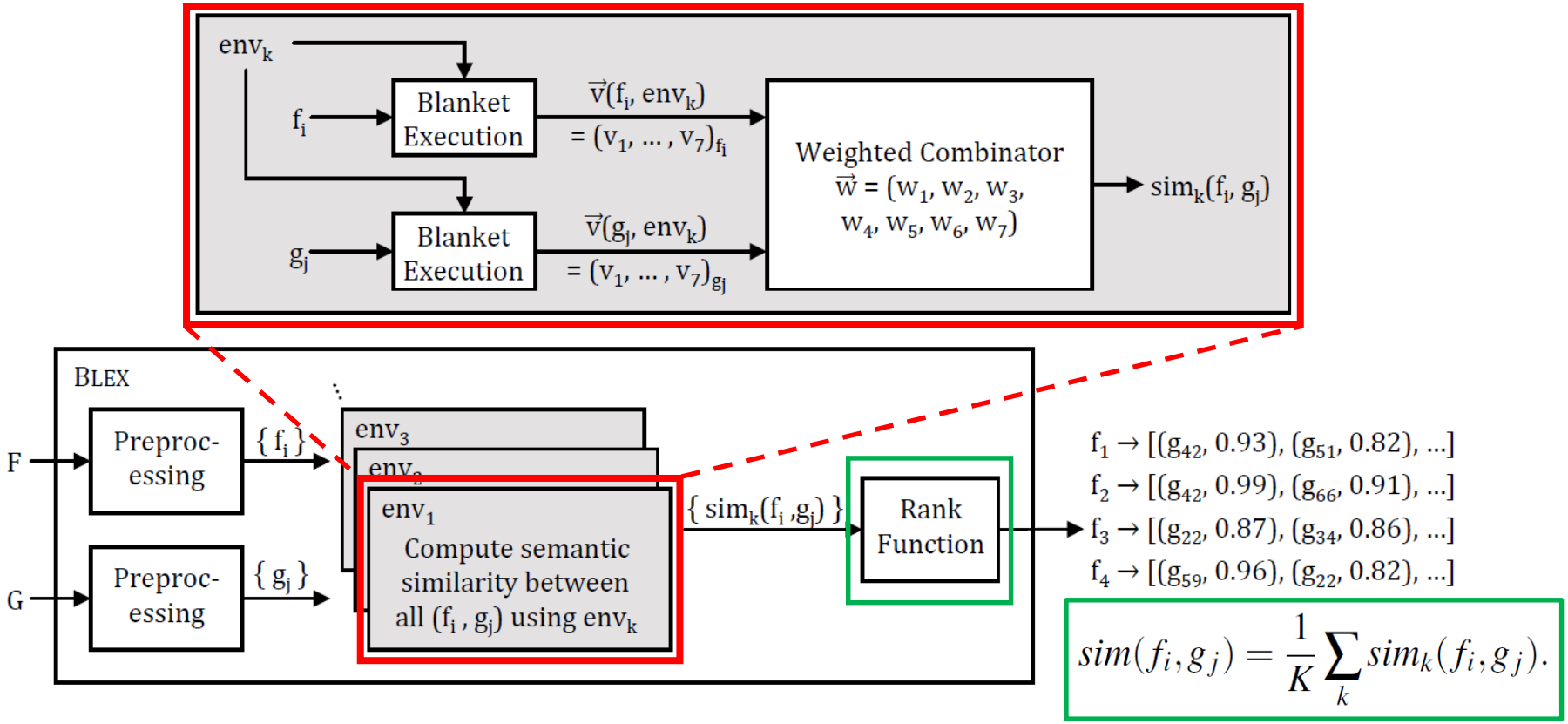
$\vec{v}(f, env)$: env 환경에서 함수 f의 blanket execution으로 생성되는 N개의 feature(value의 set)

$$sim_k(f,g) = \sum_{i=1}^{N} \left(w_i \times \frac{|v_i(f, env_k) \cap v_i(g, env_k)|}{|v_i(f, env_k) \cup v_i(g, env_k)|} \right) / \sum_{\ell=1}^{N} w_\ell .$$

➔ Jaccard index의 합을 유사도로 결정, 0과 1 사이의 값으로 나타남
➔ 각각의 feature들의 중요도가 다르기 때문에 가중치를 적용 (5.1.2 설명)

Approach

4. Binary Diffing with Blanket Execution



Implementation

1. Inputs to Blanket Execution

- input : 두 개의 바이너리
- 바이너리를 함수 단위로 쪼개는 작업을 거친 후, 각각의 함수를 getInstructions()으로 구성하는 명령어로 쪼갬
- function boundary를 식별하는 것은 중요하지만 연구에서 다루는 부분이 아님
  ➔ IDA pro 사용

Implementation

2. Performing a BE-Run

- blanket execution은 env환경에서 주어진 address i부터 함수를 실행하는 것
  ➔ 하지만 특정 주소부터 실행하도록 할 수 없음
  ➔ 로더가 프로그램 entry point에 넘겨준 제어권을 be-run을 할 주소 i로 넘겨주게 함
- dynamic linker에 의해 만들어지는 side effect 제거
  ➔ LD_BIND_NOW 환경변수를 설정하여 해결
- 임의의 환경에서 실행하는 코드가 맵핑되지 않은 메모리에 접근할 수도 있음
  ➔ 더미 메모리 페이지로 대체

Implementation

2. Performing a BE-Run

BE-run이 완료되는 조건

- 실행으로 함수의 끝에 도달했을 때,
- Exception 발생 or Terminal signal을 받았을 때,
- 설정한 명령어 수만큼 실행됐을 때,
- 설정한 시간이 경과했을 때

Implementation

3. Instrumentation

함수 유사도를 정하는데 수집하는 정보들

- 힙으로부터 읽은 값 (v_1)
- 힙에 쓴 값 (v_2)
- 스택으로부터 읽은 값 (v_3)
- 스택에 쓴 값 (v_4)

➔ 메모리에 접근하기 전에 effective address인지 검증

➔ 맵핑되지 않은 메모리는 더미 메모리 페이지로 맵핑됨

Implementation

3. Instrumentation

- PLT를 통해 임포트 된 라이브러리 함수 콜 (v_5)
  ➔ PLT section의 정보를 추출
- 실행 중 발생한 시스템 콜 (v_6)
  ➔ pin을 이용하여 시스템 콜이 호출되기 전에 정보 추출
- 분석중인 함수 완료 시, rax 레지스터에 저장된 값 (v_7)

➔ 각각의 정보들은 중요도가 다르기 때문에 가중치를 적용할 수 있도록 함 (5.1.2 절)

➔ Jaccard index를 이용하여 유사도 측정

Evaluation

1. Dataset / 2. Experimental Setup

- coreutils-8.13의 95개 유틸리티 대상
- gcc4.7.2 icc14.0.0 clang3.0-6.2 / optimization -O0~O3 (12가지)
- 함수 이름이 같으면 같은 함수로 판정
- w1=0.3846, w2=2.4979, w3=0.3786, w4=0.4052, w5=0.1082, w6=0.8775, w7=0.3222 가중치 적용
- 실행 명령어 최고치 10,000 개, 실행시간 제한 3초
- 195,560개의 함수를 11개의 environment에서 be-run
  - 총 1,590,773번 (9,756 timeout, 604,491 명령어 제한)

Evaluation

3. Comparing Semantically Equivalent Implementations

의미적으로 같은 함수 간의 비교를 위한 실험

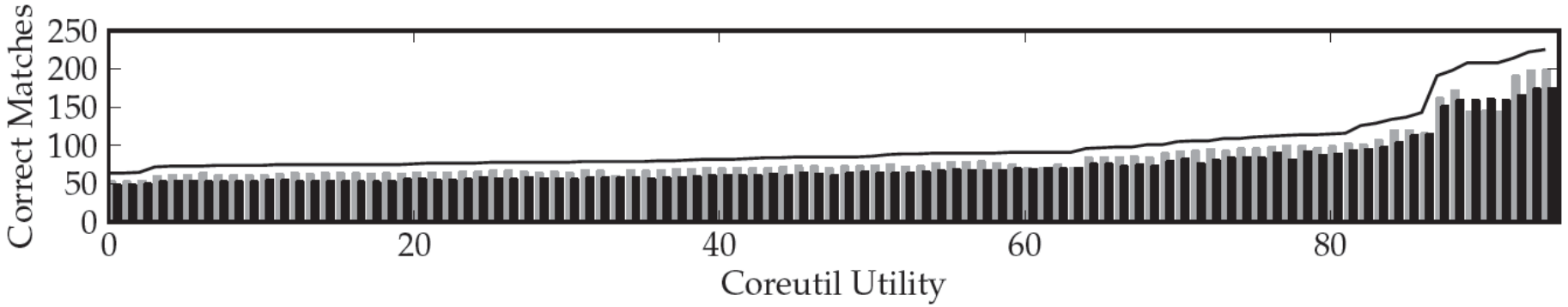
- Newlib과 uclibc의 ffs 함수를 gcc -O2 컴파일
  - Newlib의 경우, 4개의 베이직 블록
  - uclibc의 경우, 11개의 베이직 블록

  ➔ 바이너리에서 큰 차이를 보임
- 13개의 env에서 BLEX를 통해 분석
  - 분석 결과 두 함수가 같다고 판정됨

  ➔ 서로 다른 소스라도 의미적으로 비슷하면 BLEX를 통해 탐지 가능

Evaluation

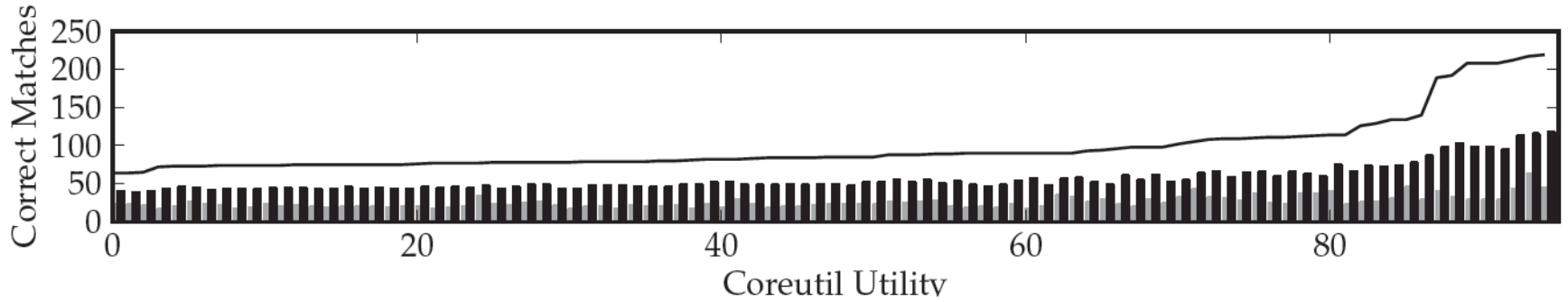
4. Function Similarity across Compiler Configurations
최적화 옵션에 따른 비교(-O2와 –O3 간의 비교)



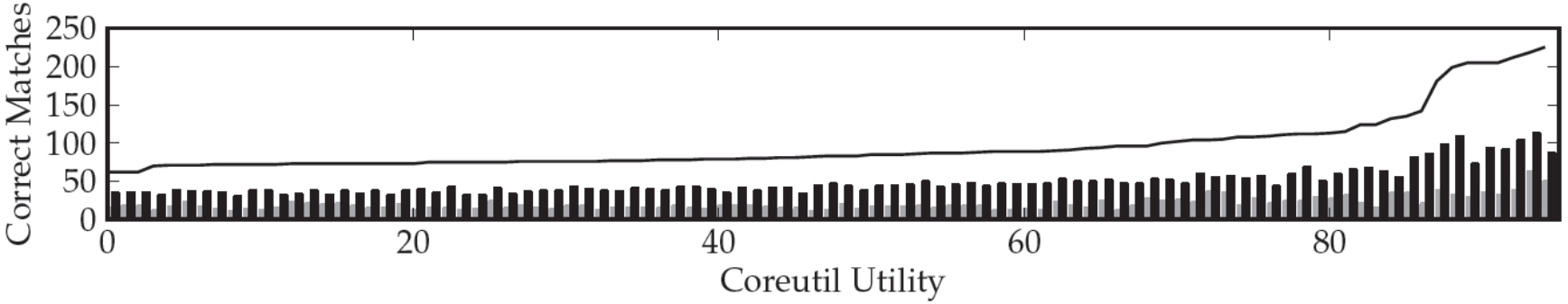
- gcc -O2 & -O3 컴파일 된 coreutils 간의 매칭
- 회색 막대 : BinDiff / 검은 막대 : BLEX

Evaluation

4. Function Similarity across Compiler Configurations
최적화 옵션에 따른 비교(-O0와 –O3 간의 비교)



gcc -O0 & gcc -O3



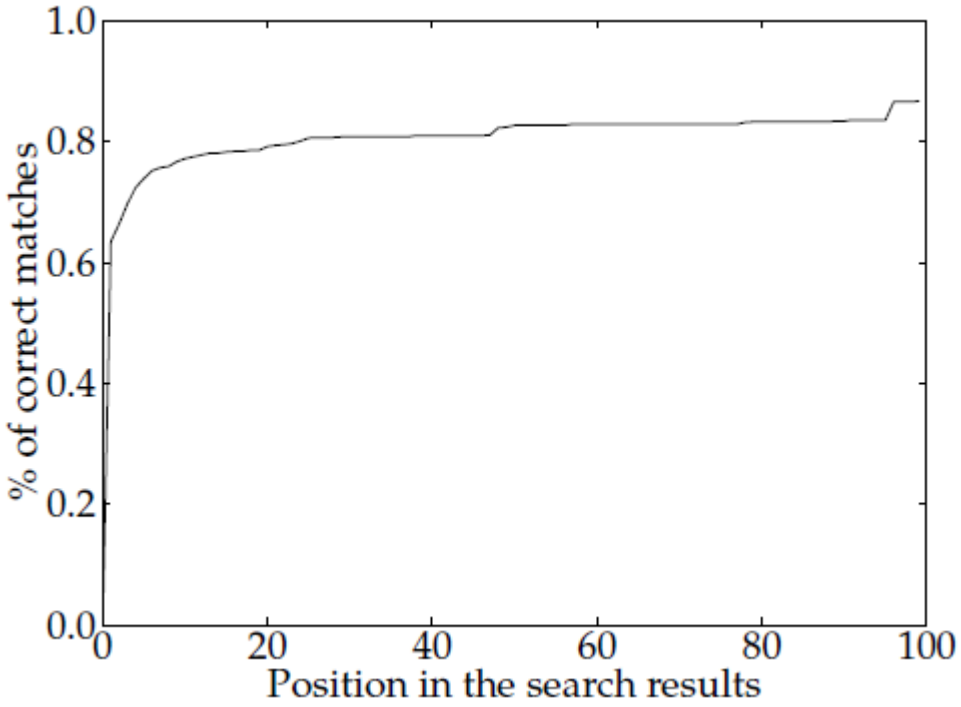
gcc -O0 & icc -O3

Evaluation

5. BLEX as a Search Engine

gcc -O0 1000 functions & gcc -O1~O3 29,015 functions

이 부분에서 말하는 결과가
뭔지 잘 모르겠음...



Conclusion

- BinDiff
  - 정적인 정보를 가지고 함수의 매칭을 판단함
  - 비교하는 두 바이너리를 만들 때, 컴파일러가 다르거나 최적화 옵션에 따라 달라지면 성능 하락

- BLEX
  - dynamic similarity testing system
  - BinDiff 보다 컴파일러나 최적화 옵션에 따른 성능이 더 좋음