석 사 학 위 논 문

Master's Thesis

동적 테인트 분석을 활용한 효과적인 동적 기호 실행 탐색 전략

Effective Concolic Search Strategy Using Dynamic Taint Analysis

2020

박 건 우 (朴 乾 雨 Park, Kunwoo)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

동적 테인트 분석을 활용한 효과적인 동적 기호 실행 탐색 전략

2020

박 건 우

한 국 과 학 기 술 원

전산학부

동적 테인트 분석을 활용한 효과적인 동적 기호 실행 탐색 전략

박 건 우

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2019년 12월 16일

심사위원장 김 문 주 (인)

심 사 위 원 유 신 (인)

심 사 위 원 고 인 영 (인)

Effective Concolic Search Strategy Using Dynamic Taint Analysis

Kunwoo Park

Advisor: Moonzoo Kim

A thesis submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in School of Computing

Daejeon, Korea
December 16, 2019

Approved by

Kim, Moonzoo
Associate Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1)].

1) Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my dissertation contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS 20183202

박건우. 동적 테인트 분석을 활용한 효과적인 동적 기호 실행 탐색 전략. 전산학부. 2020년. 38+iv 쪽. 지도교수: 김문주. (한글 논문)

Kunwoo Park. Effective Concolic Search Strategy Using Dynamic Taint Analysis. School of Computing. 2020. 38+ivpages. Advisor: Moonzoo Kim. (Text in Korean)

초 록

Concolic 테스팅은 대상 프로그램의 모든 경로를 탐색하는 것을 목표로 테스트 케이스를 생성하는 소프트웨어 자동화 테스팅 기법이다. 하지만 프로그램의 모든 경로를 탐색하는 것은 많은 비용이 들기 때문에, 분기 커버리지를 높이 달성할 수 있는 가장 유망한 경로부터 먼저 살펴보는 것이 중요하다. 어떤 경로부터 탐색할지 결정하는 전략을 Concolic 탐색 전략이라고 부른다. 기존의 Concolic 탐색 전략들(DFS, rev-DFS, CFG, random)은 대상 프로그램에 있는 함수들 간의 데이터 의존도를 고려하지 않고 방문할 분기를 결정하기 때문에 분기 커버리지가 낮게 나오는 경우가 있다.

본 연구는 동적 테인트 분석을 통해 함수 간 데이터 의존도를 함수 간 데이터 흐름을 기반으로 정의하고 함수 간 데이터 의존도를 바탕으로 새로운 concolic 탐색 전략 Taint를 설계하였다. Taint는 타겟 함수 g의 커버하지 못한 분기를 커버하고자 할 때, 함수 f에 대한 함수 g의 데이터 의존도가 높은 경우(즉, 함수 f에서 함수 g로 변수의 값들을 많이 보내고), 함수 f의 분기를 부정하거나 함수 g의 분기를 부정해서 함수 g에서 커버하지 못한 분기를 커버하도록 하는 휴리스틱이다. 이러한 접근 방식은 함수 f가 함수 g로 보낸 변수의 값들이 함수 g의 실행 경로를 결정한다는 직관에서 비롯된다. 또한, 함수 간 데이터 의존도를 계산할 때, 고려해야 할 여러 가지 요소들 중에서 3가지 요소들을 고려하였다. 동적 테인트 분석은 함수 간 데이터 의존도의 측정에 기반이 되는 함수 간 데이터 흐름을 추출하기 위해 사용되었다. Taint는 기존의 Concolic 탐색 전략들을 사용했을 때보다 1.2%p~7.1%p 더 높은 분기 커버리지를 달성할 수 있다.

핵 심 낱 말 Concolic 테스팅, Concolic 탐색 전략, 동적 테인트 분석

Abstract

Concolic testing is a software automated testing with the goal of visiting all the paths of the target program. Since exploring all the paths of a program is expensive, it is important to look at the most promising paths first to achieve high branch coverage. The strategy that determines which path to visit is called concolic search strategy. Conventional concolic search strategies (e.g., DFS, rev-DFS, CFG, random) sometimes have low branch coverage because they decide which path to visit without considering the correlation between functions in the target program.

In this paper, we defined data dependency between functions based on the data flow between functions through dynamic taint analysis and designed a new concolic search strategy called Taint based on data dependency between functions. If target function g has an uncovered branch that needs to be covered and function g has a high data dependency on function f (i.e., function f sends many values of variables to function g), Taint guides concolic testing to negate the branch in function f or function g to cover the uncovered branch in function g. This approach stems from the intuition that the values of variables that function f sends to function g determine the execution path of function g. Three design choices were taken into account when measuring the data dependency between functions. Dynamic taint analysis is used to extract def-use chains between functions and measure data dependency between functions. Taint can achieve 1.2%p~7.1%p higher branch coverage than using conventional concolic search strategies.

Keywords Concolic Testing, Concolic Search Strategy, Dynamic Taint Analysis

차 례

표 차례

그림 차례

제 1 장 머리말

일상 생활에서 소프트웨어가 널리 사용됨에 따라 소프트웨어의 신뢰성이 중요한 이슈가 되었다. 소프트웨어 테스팅은 이러한 소프트웨어의 신뢰성을 확인하는 주요 방법이다. 소프트웨어 테스팅은 일련의 테스트 케이스들로 대상 프로그램을 실행하고 각 실행 결과가 사전에 정의된 필수 요건을 충족하는지 그리고 SW 내부에 결함이 존재하는지 확인한다.

성공적인 소프트웨어 테스팅을 위해서는 대상 프로그램의 모든 실행 경로를 탐색하는 고품질의 충분한 수의 테스트 케이스들을 만드는 것이 중요하다. 하지만, 소프트웨어의 크기와 복잡성이 커짐에 따라 개발자가 프로그램의 모든 실행 경로를 처리하는 테스트 케이스를 수작업으로 만드는 것은 거의 불가능하기 때문에 SW 내부에 존재하는 결함을 발견하기 어렵다. 따라서, 수작업 테스팅 인력을 줄이고 고품질의 테스트 케이스들을 생성하기 위해 random 테스팅이나 심볼릭 실행과 같은 다양한 소프트웨어 자동화 테스팅 기술들이 개발되었다.

Concolic (CONCrete + symbolic) 테스팅 기법[45]은 그 중 실제 산업 분야에서 널리 사용되고 있는 테스팅 기법이기 때문에[14-20], 이 테스팅 기법을 개선 및 보완하는 것은 중요한 연구 주제이다. 소프트웨어 테스팅의 목표 중 하나는 충분한 수의 테스트 케이스를 생성하여 높은 커버리지를 달성하고 프로그램의 신뢰성을 높이는 것이기 때문에, concolic 테스팅 기법을 개선하여 같은 프로그램에 대해 더 높은 커버리지를 달성하는 것이 중요하다. (2.1절 참조)

더 높은 커버리지를 달성하도록 concolic 테스팅 기법을 개선하는 방법으로는 concolic 탐색 전략을 개선하는 것이 존재한다. Concolic 탐색 전략은 새로운 테스트 케이스를 생성하기 위해 어떤 분기를 방문할지 결정하는 전략을 말한다. 기존 concolic 탐색 전략으로는 대표적으로 DFS, Rev-DFS, CFG, random 탐색 전략이 있다 (2.1.1절 참조).

1.1 기존 접근의 문제점

기존 탐색 전략(DFS, rev-DFS, CFG, random)들의 문제점은 한 함수가 사용하는 변수들의 값이 어떤 함수에서 왔는지에 대한 정보가 새로운 분기를 커버하는데 중요한 역할을 할 수 있음에도 불구하고, 그 정보를 고려하지 않고 방문할 분기를 탐색하기 때문에 높은 코드 커버리지를 달성할지 못한다는 점이다. DFS와 Rev-DFS 탐색 전략은 정해진 순서에 맞게 방문할 분기를 결정하기 때문에 복잡한 반복문이나 재귀 함수 안에 빠지면 그 바깥에 있는 분기를 커버하지 못한다. Random 탐색 전략은 무작위성에 의존한다. CFG 탐색 전략은 현재의 커버리지 정보를 고려해서 방문할 분기를 결정하는 휴리스틱이지만 한 함수가 사용하는 변수들의 값이 어떤 함수에서 왔는지에 대한 정보를 고려하지 않는다.

한 함수 f에서 다른 함수 g로의 def-use chain이 많은 경우, 즉 g에서 사용(use)한 변수 값들이 f에서 많이 정의(def)된 경우 함수 f에 대한 함수 g의 데이터 의존도가 높다고 말한다. 이를 f가 g의 실행에 끼치는 영향력이 크다고 해석할 수 있다. 함수 g와 데이터 의존도가 높은 함수 f의 분기를 부정했을 때, 함수 g의 실행 경로에 변화가 생겨 새로운 분기를 커버할 수 있다. 기존 탐색 전략들은 함수 간 데이터 의존도를 고려하지 않기 때문에 분기 커버리지가 낮게 나올 수 있다. 따라서, 함수 간 데이터 의존도를 고려한 새로운 concolic 탐색 전략을 설계하여 기존 탐색 전략들이 커버하지 못하는 분기를 커버하고 분기 커버리지를 높일 필요가 있다.

1.2 제안하는 접근 방식

함수 간 데이터 의존도를 활용하는 새로운 concolic 탐색 전략을 설계한다. 새로운 concolic 탐색 전략은 타겟 함수 g의 커버하지 못한 분기를 커버하고자 할 때, g의 분기 조건을 부정하거나 g에 영향력이 큰 (즉, 데이터 의존도가 큰) 함수 f의 분기 조건을 부정해서 커버리지를 높인다. g에 영향력이 큰 함수 f의 분기 조건을 부정할 경우, g의 실행 경로에 변화가 생겨서 커버리지를 높이는 것을 기대할 수 있다.

새로운 탐색 전략에 활용되기 때문에 두 함수 f와 g에 대해서 함수 f에 대한 함수 g의 데이터 의존도를 함수 f에서 함수 g로의 데이터 흐름 기반으로 정의하고 이를 측정하기 위해 동적 테인트 분석을 사용한다. 동적 테인트 분석은 타겟 프로그램을 실행해서 관찰하고자 하는 변수의 데이터 흐름(def-use chain)을 분석하는 기술이다. 동적 테인트 분석은 기본적으로 사용자가 관찰할 변수를 테인트 소스(taint source)로 지정하고 프로그램의 실행 경로를 따라 그 변수값이 전파되어 최종적으로 테인트 싱크(taint sink)로 지정된 변수에 도달할 경우, 테인트 소스와 테인트 싱크 사이에 데이터 흐름을 추출하고 분석할 수 있다. (2.2절 참조) 동적 테인트 분석을 사용해 모든 함수 쌍 사이에 존재하는 def-use chain들을 추출하고 이들을 바탕으로 모든 함수 쌍에 대해서 데이터 의존도를 측정한다.

1.3 논제 및 기여

이 논문의 논제(thesis statement)는 다음과 같다.

함수 간 데이터 의존도를 활용한 concolic 탐색 전략은 효과적으로 더 높은 분기 커버리지를 달성할 수 있다.

또한 이 논문은 다음과 같은 기여를 한다:

- 함수 간 데이터 의존도를 측정하기 위해 동적 테인트 분석 도구를 구현하고 활용하였다.

- 3가지 design choice를 바탕으로 함수 간 데이터 의존도를 정의하고, 이를 활용해 새로운 concolic 탐색 전략 Taint를 설계하였다.
  - 함수 f에서 함수 g로의 데이터 흐름의 크기를 f에서 g로의 def-use chain 개수의 총합으로 수치화하였다. (3.1.3.1절 참조)
  - 함수 사이에 존재하는 모든 def-use chain들 중에서 조건문의 조건식에 사용되는 변수들을 use로 하는 def-use chain들만 고려해서 데이터 의존도를 계산하였다. (3.1.3.2 참조)
  - 프로그램의 실행에서 한 함수가 여러 번 호출되어서 def-use chain이 중복으로 여러 번 등장하는 경우, 중복된 def-use chain들을 1개의 def-use chain으로 처리하여 데이터 의존도를 계산하였다. (3.1.3.3절 참조)
- Taint는 실제 산업 분야에서 사용하는 프로그램 7개에 적용했을 때 기존의 concolic 탐색 전략들을 사용했을 때보다 1.2%~7.1%p 더 높은 분기 커버리지를 달성하였다.

1.4 논문의 전체 구성

이하에 기술되는 논문의 내용은 다음과 같이 구성되어 있다.

제 2장에서는 concolic 테스팅과 concolic 테스팅에서 사용하는 기존 탐색 전략의 문제점, 그리고 동적 테인트 분석에 대해 설명한다.

제 3장에서는 연구에 사용된 동적 테인트 분석 도구에 대해 설명하고, 도구를 사용해서 함수 간 데이터 의존도를 측정하는 방법에 대해 설명한다. 그리고 3개의 design choice들을 바탕으로 함수 간 데이터 의존도를 정의하는 방법에 대해 설명한다. 마지막으로 동적 테인트 분석을 활용한 새로운 concolic 탐색 전략 Taint를 설명한다.

제 4장에서는 실험을 통해 Taint가 기존의 다른 concolic 탐색 전략에 비해서 얼마나 효과적으로 분기 커버리지를 높이는지를 보여준다.

제 5장에서는 관련 연구를 설명한다.

제 6장에서는 이 논문의 결론과 향후 연구에 대해 기술한다.

제 2 장 연구 배경

제 2 장에서는 본 연구를 수행한 배경인 concolic 테스팅 기법과 concolic 테스팅 기법에서 사용하는 기존 탐색 전략의 문제점, 그리고 동적 테인트 분석에 대해 설명한다.

2.1 Concolic 테스팅

Concolic (CONCrete + symbOLIC) 테스팅은 concrete 실행 (특정 테스트 입력값으로 대상 프로그램을 실행)과 심볼릭 실행(프로그램의 입력 변수를 심볼릭 변수로 프로그램의 실행 경로를 동시다발적으로 탐색할 수 있는 테스팅 기법)[24, 25]을 결합한 소프트웨어 테스팅 기법으로 높은 코드 커버리지를 갖는 테스트 케이스들을 자동으로 생성한다. Concolic 테스팅에서는 먼저 심볼릭 변수로 설정할 입력 변수들을 설정해서 구체적인 테스트 케이스를 심볼릭 변수에 대입해 테스트 대상 프로그램을 실행하고, 실행 경로가 통과한 분기 조건들을 수집해서 심볼릭 경로 수식(symbolic path formula)을 생성한다. 실행이 끝나면 심볼릭 경로 수식의 분기 조건 중 하나를 부정하여 새로운 심볼릭 경로 수식을 생성하고, SMT solver로 이 새로운 심볼릭 경로 수식을 만족하는 테스트 케이스를 구한다. 예를 들어, 세 변수 x,y,z가 심볼릭 변수이고 새로운 심볼릭 경로 수식이 $x > 5 \wedge y \neq 3 \wedge z = 18$일 경우, SMT solver는 (x,y,z) = (9, 0, 18)라는 새로운 테스트 케이스를 생성한다. 이 일련의 과정은 실행 가능한 경로를 모두 실행했거나 사용자가 지정한 시간이 초과될 때까지 반복된다. Concolic 테스팅은 이론상 모든 실행 가능한 경로를 찾아 테스트하기 때문에 수작업으로 테스트 케이스를 만드는 것보다 효과적으로 코드 커버리지를 높일 수 있다.

2.1.1 Concolic 탐색 전략

Concolic 테스팅은 대상 프로그램을 실행하면서 아직 탐색하지 않은 분기를 방문하는 것을 목표로 테스트 케이스를 생성한다. 새로운 테스트 케이스를 생성하기 위해 다음 iteration에서 어떤 분기를 방문할지 결정하여야 그에 알맞은 테스트 케이스를 생성할 수 있는데, 제한된 시간 동안 어떤 분기부터 방문해서 커버리지를 높일지 결정하는 전략을 탐색 전략이라고 부른다. C 프로그램을 대상으로 하는 대표적인 Concolic 테스팅 도구인 CROWN[22]과 CREST[23]에서는 4개의 탐색 전략 DFS, rev-DFS, CFG, random을 제공한다.

그림 2.1은 삼각형 세 변의 길이를 양의 정수로 입력 받아 삼각형의 종류를 판별해주는 프로그램이며, CROWN을 이용해 Concolic 테스팅을 수행할 수 있도록 심볼릭 변수 설정을 완료한 상태다(6, 8번째 줄). Concolic 테스팅에서 구체적인 테스트 케이스에 대해 프로그램의 한 경로 e를 실행하면 심볼릭 변수가 만족해야 하는 심볼릭 경로 수식 p가 수집된다. 즉, 심볼릭 변수들이 심볼릭 경로 수식 p를 만족할 때, 프로그램의 경로 e를 실행할 수 있다. 여기서 탐색 전략에 따라 다음 테스트 케이스 생성을 위한 새로운 심볼릭 경로 수식을 만드는 방식이 달라진다.

```
 1  // triangle.c
 2  #include <crown.h>
 3  #include <stdio.h>
 4  int main() {
 5      int a,b,c, match=0;
 6      SYM_int(a); SYM_int(b); SYM_int(c);
 7      // filtering out invalid inputs
 8      if(a<=0 || b<=0 || c<=0) exit(-1);
 9      printf("a,b,c = %d,%d,%d:",a,b,c);
10
11      //0: Equilateral, 1:Isosceles,
12      //2: Not a traiangle, 3:Scalene
13      int result=-1;
14      if(a==b) match = match + 1;
15      if(a==c) match = match + 2;
16      if(b==c) match = match + 3;
17      if(match==0) {
18          if(a+b <= c) result = 2;
19          else if(b+c <= a) result = 2;
20          else if(a+c <= b) result = 2;
21          else result=3;
22      } else {
23          if(match == 1) {
24              if(a+b <= c) result = 2;
25              else result=1;
26          } else {
27              if(match == 2) {
28                  if(a+c <= b) result = 2;
29                  else result = 1;
30              } else {
31                  if(match == 3) {
32                      if(b+c <= a) result = 2;
33                      else result = 1;
34                  } else result = 0;
35      }   }   }
36      printf("result=%d\n",result);
37  }
```

그림 2. 1: 삼각형 판별 프로그램 예시

DFS (깊이 우선 탐색, Depth First Search) 탐색 전략은 새로운 심볼릭 경로 수식을 만들 때 뒤에 있는 조건부터 부정한다. 만약 그 조건을 부정했을 때 새로운 입력 값을 만들 수 없다면, 그 앞의 조건을 부정하여 심볼릭 경로 수식을 만든다. DFS 탐색 전략은 이론상 프로그램의 가능한 모든 실행 경로를 탐색할 수 있다. 표 2.1은 처음 입력 값이 (1,1,1)이라고 가정 했을 때 그림 2.1의 코드를 DFS 탐색 전략으로 concolic 테스팅한 경우 생성되는 테스트 케이스들의 예시이다. 총 11개의 테스트 케이스를 생성한 후 프로그램의 모든 실행 가능한 경로를 커버했기 때문에 concolic 테스팅이 종료되었다.

표 2. 1: 삼각형 판별 프로그램을 DFS 옵션으로 입력 값 생성하는 예시

TC	입력값 (a,b,c)	현재 심볼릭 경로 수식	다음 심볼릭 경로 수식	다음 심볼릭 경로 수식을 만족하는 입력 값
1	1,1,1	$a = b \wedge a = c \wedge b = c$	$a = b \wedge a = c \wedge b \neq c$	UNSAT
			$a = b \wedge a \neq c$	1,1,2
2	1,1,2	$a = b \wedge a \neq c \wedge b \neq c \wedge a + b \leq c$	$a = b \wedge a \neq c \wedge b \neq c \wedge a + b > c$	2,2,3
3	2,2,3	$a = b \wedge a \neq c \wedge b \neq c \wedge a + b > c$	$a = b \wedge a \neq c \wedge b = c$	UNSAT
			$a \neq b$	2,1,2
4	2,1,2	$a \neq b \wedge a = c \wedge b \neq c \wedge a + c > b$	$a \neq b \wedge a = c \wedge b \neq c \wedge a + c \leq b$	2,5,2
5	2,5,2	$a \neq b \wedge a = c \wedge b \neq c \wedge a + c \leq b$	$a \neq b \wedge a = c \wedge b = c$	UNSAT
			$a \neq b \wedge a \neq c$	1,2,2
6	1,2,2	$a \neq b \wedge a \neq c \wedge b = c \wedge b + c > a$	$a \neq b \wedge a \neq c \wedge b = c \wedge b + c \leq a$	4,2,2
7	4,2,2	$a \neq b \wedge a \neq c \wedge b = c \wedge b + c \leq a$	$a \neq b \wedge a \neq c \wedge b \neq c$	1,2,3
8	1,2,3	$a \neq b \wedge a \neq c \wedge b \neq c \wedge a + b \leq c$	$a \neq b \wedge a \neq c \wedge b \neq c \wedge a + b > c$	3,1,2
9	3,1,2	$a \neq b \wedge a \neq c \wedge b \neq c \wedge a + b > c \wedge b + c \leq a$	$a \neq b \wedge a \neq c \wedge b \neq c \wedge a + b > c \wedge b + c > a$	1,3,2
10	1,3,2	$a \neq b \wedge a \neq c \wedge b \neq c \wedge a + b > c \wedge b + c > a \wedge a + c \leq b$	$a \neq b \wedge a \neq c \wedge b \neq c \wedge a + b > c \wedge b + c > a \wedge a + c > b$	3,4,5
11	3,4,5	$a \neq b \wedge a \neq c \wedge b \neq c \wedge a + b > c \wedge b + c > a \wedge a + c > b$	n/a	n/a

Rev-DFS 탐색 전략은 DFS 탐색 전략의 역순으로 조건을 부정한다. 즉, 새로운 심볼릭 경로 수식을 만들 때 가장 앞에 있는 조건부터 부정하며, 그 조건을 부정했을 때 새로운 테스트 케이스를 만들 수 없다면, 그 뒤의 조건을 부정하여 심볼릭 경로 수식을 만든다. Rev-DFS 탐색 전략은 DFS 탐색 전략과 동일하게 모든 가능한 경로를 탐색할 수 있으며, DFS보다 초기에 더 많은 branch를 커버할 수 있다. 이는 심볼릭 경로 앞부분의 조건을 바꾸는 경우가 뒷부분의 조건을 바꾸는 경우보다 새로운 코드 영역을 수행할 가능성이 더 높기 때문이다.

DFS와 rev-DFS 탐색 전략이 CFG가 모든 경로를 탐색하는 것을 목표로 한다면 CFG 탐색 전략은 분기 커버리지를 빠르게 증가시키는 것을 목표로 한다[21]. 이 탐색 전략은 코드의 CFG (Control Flow Graph)를 분석해서 커버되지 않은 분기를 먼저 커버하도록 새로운 테스트 케이스를 생성하는 휴리스틱이다. 휴리스틱이기 때문에 모든 경우에 항상 잘 동작하는 것은 아니며 기법 특성 상 모든 경로를 완전히 탐색하는 것이 어렵다. 또한, 중복되는 테스트 케이스를 생성하기도 한다.

Random 탐색 전략은 새로운 심볼릭 경로 수식을 만들 때 부정할 조건을 무작위로 선택한다. 무작위로 한 조건을 선택하여 심볼릭 경로 수식을 만들었을 때 새로운 테스트 케이스를 만들 수 없다면 다시 무작위로 조건을 선택하여 부정한다.

```
1  void g(int z, char *str) {
2    int i;
3    if (z == 1) {
4      for (i = 0; i < 100; i++) {
5        if (str[i] == 'B')
6          printf("if\n");
7        else
8          printf("else\n"); } }
9   else Error(); }
```

그림 2. 2: 경로의 폭발적 증가를 설명하기 위한 예제 함수

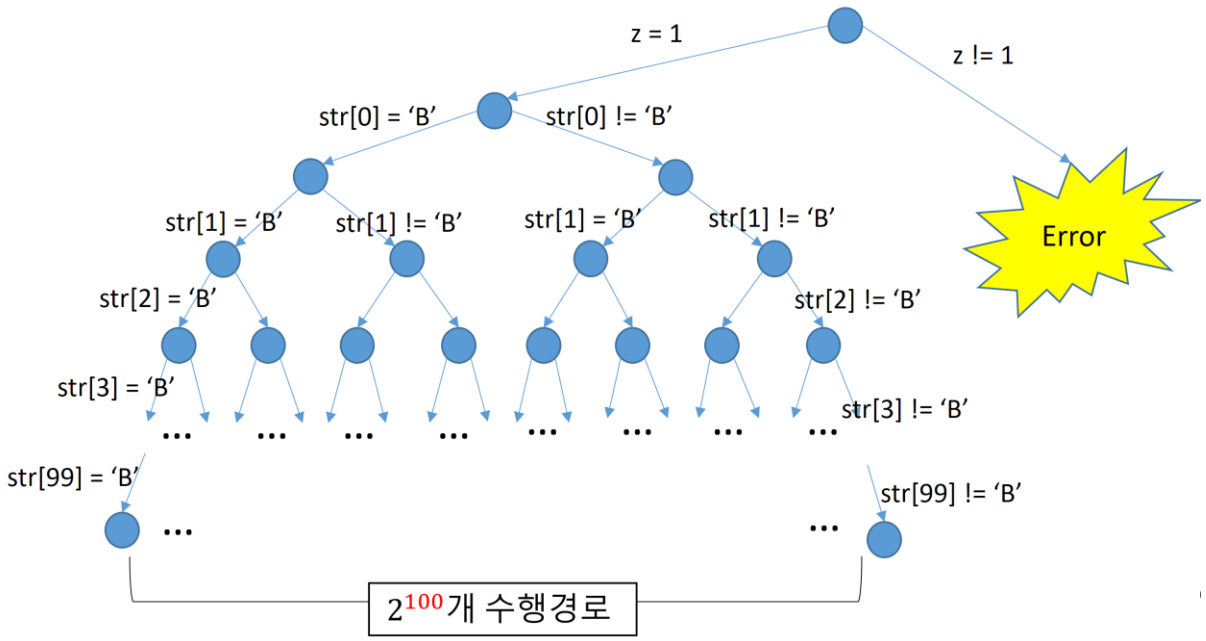


그림 2. 3: 그림 2.2 예제 함수의 실행 트리

2.1.2 기존 Concolic 탐색 전략의 문제점

기존의 concolic 탐색 전략들(DFS, rev-DFS, CFG, random)은 대상 프로그램의 함수 간 데이터 의존도를 고려하지 않고 방문할 분기를 결정하기 때문에 경로의 폭발적 증가(path explosion) 문제에 대해 높은 코드 커버리지를 달성하지 못한다는 문제가 있다.

그림 2.2는 경로의 폭발적 증가를 설명하기 위한 예제 함수이고, 그림 2.3은 그림 2.2 예제 함수의 실행 트리이다. 해당 예제 함수의 파라미터인 int 타입 z와 길이가 100인 char 타입 str(즉, str[0], str[1], … str[99])을 심볼릭 변수로 설정하였다고 가정하자. 임의의 초기 테스트 케이스로 {z = 1, str[0] = '\0', str[1] = '\0', …, str[99] = '\0'}을 사용해서 concolic 테스팅을 수행했을 때, 구체적 실행은 3번째 줄의 if 분기 조건이 참이 되어 4-8번째 줄의 반복문을 실행하게 된다. 동시에

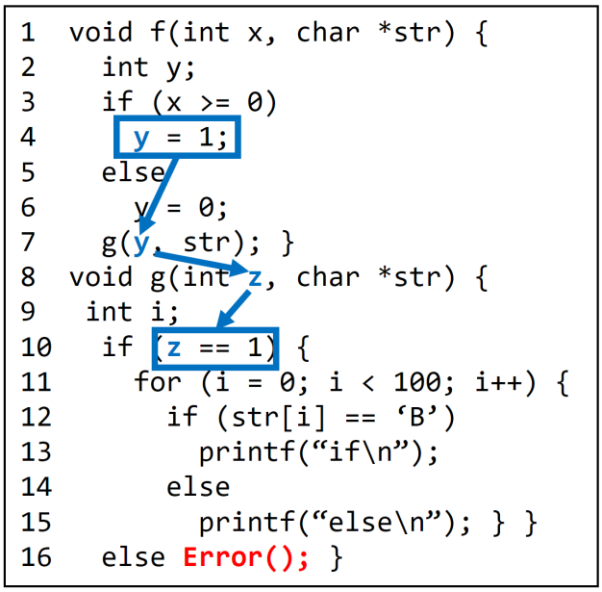


그림 2. 4: 함수 간 데이터 의존도를 활용한 경로의 폭발적 증가 문제 해결

심볼릭 실행은 $z = 1 \wedge str[0] != 'B' \wedge str[1] != 'B' \wedge \ldots \wedge str[99] != 'B'$의 심볼릭 경로 수식을 생성한다. 즉, 그림 2.3에서 z = 1 분기 밑에 있는 경로 하나를 실행하게 된다. 기존 concolic 탐색 전략인 DFS 탐색 전략은 현재 심볼릭 경로 수식의 맨 뒤에 있는 조건부터 부정해서 새로운 심볼릭 경로 수식을 만들기 때문에 그림 2.3에서 z = 1 분기 밑에 있는 모든 경로를 다 탐색하기 전까지 z != 1 분기 밑에 있는 경로를 탐색하지 못한다. 문제는 z = 1 분기 밑에 있는 경로의 개수가 2^{100}개이기 때문에 현실적으로 제한된 시간과 자원으로 z = 1 분기 밑에 있는 경로를 다 탐색하고 z != 1 분기 밑에 있는 에러를 검출하지 못한다.

함수 간 데이터 의존도를 활용한 새로운 concolic 탐색 전략은 경로의 폭발적 증가 문제를 해결할 수 있다. 그림 2.4는 그림2.2 예제 함수 g에 g를 호출하는 함수 f가 추가되었다. 초기 테스트 케이스가 {x = 1, str[0] = '\0', str[1] = '\0', ..., str[99] = '\0'}일 때 3번째 if 분기 조건이 참이 되어 y = 1이 되고 정의된 y값이 g의 파라미터로 넘어가서 10번째 분기 조건이 참이 된다. Concolic 테스팅을 수행하면 $x \geq 0 \wedge str[0] != 'B' \wedge str[1] != 'B' \wedge \ldots \wedge str[99] != 'B'$의 심볼릭 경로 수식을 생성한다. 따라서 DFS 탐색 전략을 사용하면 아까와 마찬가지로 경로의 폭발적 증가 문제에 빠진다. 여기서 새로운 concolic 탐색 전략은 함수 f에서 정의한 변수 y 값이 g에 사용이 되었기 때문에 f가 g에 영향력을 끼친다고 해석하고 f의 분기 조건을 부정한 새로운 심볼릭 경로 수식 $x != 1$을 생성한다. 이를 만족하는 테스트 케이스(예. {x = -1, str[0] = '\0', str[1] = '\0', ..., str[99] = '\0'}를 생성해서 프로그램을 실행하게 되면 16번째 줄을 커버하고 에러를 검출할 수 있게 된다.

2.2 테인트 분석

테인트 분석은 관찰하고자 하는 변수의 데이터 흐름(def-use chain)을 추출하고 분석하는 기술이다. 사용자가 테인트 소스(taint source)에 해당하는 변수와 테인트 싱크(taint sink)에 해당하는 변수를 지정해서 테인트 소스 변수 값이 전파되어 최종적으로 테인트 싱크 변수에 도달하는지 확인하고, 이를 바탕으로 테인트 소스와 테인트 싱크 사이의 def-use chain을 추출할 수 있다. 이 때 관찰할 변수 값은 프로그램의 테스트 케이스, 시스템 호출의 결과 또는 그 밖에 사용자가 지정한 데이터 등이다. 소프트웨어 취약점 분석, 악성코드 분석과 같은 보안 분야에서 널리 쓰이고 있다.

테인트 분석에는 정적 테인트 분석과 동적 테인트 분석[12]이 있고, 본 연구에서는 동적 테인트 분석을 활용하였다. 정적 테인트 분석은 프로그램을 실행하지 않고 프로그램의 소스 코드를 분석해서 데이터의 흐름을 분석한다. 프로그램을 실행하지 않기 때문에 분석 속도가 빠르고 코드 커버리지가 높지만, 정확도가 보장되지 않고, 포인터 연산처럼 복잡한 연산에 대해서는 데이터의 흐름을 잘 찾지 못하는 문제가 있다. 동적 테인트 분석은 주어진 시스템 케이스로 프로그램의 다양한 실행 경로를 수행하면서 테인트 소스와 테인트 싱크 사이의 데이터 흐름을 동적으로 추출하고 분석하기 때문에 정적 테인트 분석보다 코드 커버리지는 낮을 수 있어도 정확도가 뛰어나다[36,37,38].

그림 2.5은 동적 테인트 분석을 사용해 함수 f에서 함수 g로의 데이터 흐름(def-use chain)을 추출하는 과정을 설명하기 위한 프로그램 예시이다. 대상 프로그램을 실행하면 프로그램에서 사용된 각 변수들의 메모리 주소와 할당된 메모리 크기를 동적으로 추출할 수 있다. 함수 f 5번째 줄에서 할당된 배열 arr의 메모리 주소가 0x0000, 메모리 크기가 12 바이트 (4 바이트 정수 3개)라고 가정하자. 함수 f 5번째 줄, 6번째 줄에서 각각 0x0000~0x000c, 0x0000~0x0004 메모리 공간에 대한 쓰기 동작이 일어났다. 함수 g 2번째 줄에서 *param 값이 할당된 주소 0x0000, 크기 4바이트인 메모리 공간에 대해 읽기 동작이 일어났고 해당 메모리 공간이 함수 f 6번째 줄에서 에서 쓰기 동작이 일어난 메모리 공간과 일치하기 때문에, 함수 f와 함수 g 사이의 def-use chain을 수집할 수 있다.

테인트 분석은 분석의 방향에 따라 두 가지로 나눌 수 있다. 정방향 테인트 분석은 테인트 소스가 프로그램의 어느 지점까지 영향을 주는지에 대해 분석한다[35]. 예를 들어, 그림 2.5에서 쓰기 동작이 일어난 6번째 줄의 arr[0]이 테인트 소스일 때, arr[0]의 값에 영향을 받는 모든 변수들(예. *param)을 찾고 분석하는 방법이다. 반대로 역방향 테인트 분석은 테인트 싱크로부터 거슬러 올라가며 분석한다. 예를 들어, 그림 2.5에서 읽기 동작이 일어난 2번째 줄의 *param이 테인트 싱크일 때, *param의 값에 영향을 주는 모든 변수들(예. arr[0])을 실행 경로를 거슬러 올라가며 찾고 분석한다. 본 연구에서는 역방향 테인트 분석을 활용해서 특정 변수 값의 원출처가 되는 변수를 역으로 추적해서 def-use chain을 수집한다.

```
1 void g(int *param) {
2   if (*param == 2); }
3
4 void f() {
5   int arr[3] = {1,2,3};
6   arr[0] = 4;
7   g(arr);}
8
9  int main() {f(); return 0;}
```

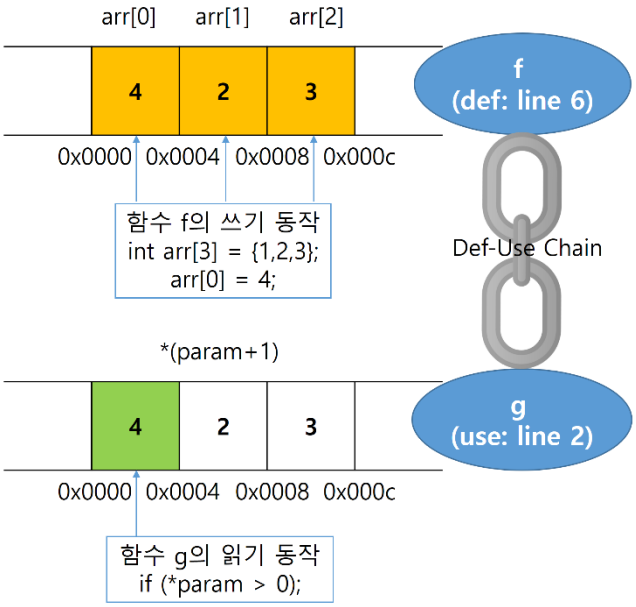


그림 2. 5: 동적 테인트 분석을 활용한 def-use chain 추출 예시

제 3 장 동적 테인트 분석을 활용한 Concolic 탐색 전략

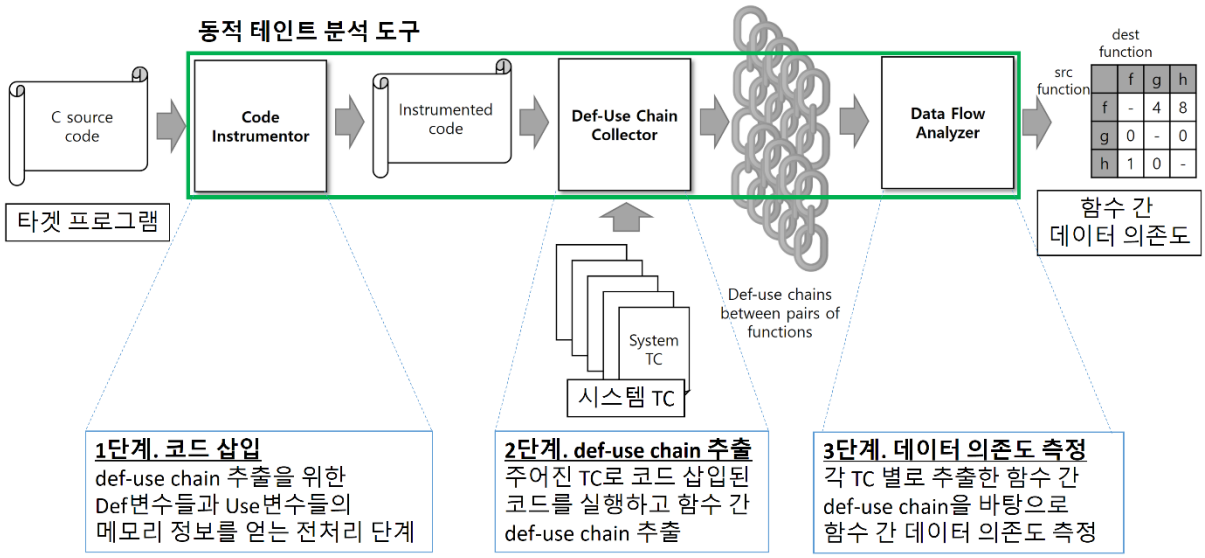


그림 3. 1: 동적 테인트 분석 도구

3.1 동적 테인트 분석 도구

함수 간 데이터 의존도를 측정하기 위해 함수 간 def-use chain들을 추출하는 목적으로 구현된 기존 동적 테인트 분석 도구가 존재하지 않기 때문에 본 연구에서 동적 테인트 분석 도구를 새롭게 구현하였다. 동적 테인트 분석 도구는 타겟 프로그램과 시스템 테스트 케이스를 입력받아 함수 간 데이터 의존도를 측정하고 출력한다. 개발한 동적 테인트 분석 도구는, 대상 변수의 메모리 주소값을 추적하고, 대입문 및 함수 인자를 넘길 때 좌변 (LHS)과 우변 (RHS)의 관계를 추적하여, Sound 한 분석결과를 도출한다 (3.1.1.2절 참조)). 동적 테인트 분석 도구의 구성과 개략적인 설명은 다음과 같다.

1. (3.1 절) 코드 삽입기 (Code Instrumentor): 대상 프로그램의 변수에 대해 읽기/쓰기 동작이 일어났을 때, 해당 변수의 정보를 동적으로 추출하기 위해 소스 코드 레벨에서 코드를 삽입한다. 여기서 해당 변수의 정보란 변수의 메모리 주소, 변수가 할당된 메모리 크기, 그리고 변수가 사용된 함수 이름이다.
2. (3.2 절) Def-Use Chain 수집기 (Def-Use Chain Collector): 주어진 시스템 테스트 케이스들로 코드가 삽입된 대상 프로그램을 실행하고, 각 시스템 테스트 케이스 별로 한 함수에서 변수를 메모리 공간에 할당하고 다른 함수에서 할당된 메모리 공간을 읽는 def-use chain들을 수집한다. 즉, 테인트 소스는 쓰기 동작이 일어나는 변수들이고, 테인트 싱크는 읽기 동작이 일어나는 변수들이다.
3. (3.3 절) 데이터 흐름 분석기 (Data Flow Analyzer): 각 테스트 케이스 별로 수집한 함수 간 def-use chain들을 바탕으로 함수 간 데이터 의존도를 측정한다.

3.1.1 코드 삽입기 (Code Instrumentor)

코드 삽입기는 프로그램의 변수에 대해 읽기/쓰기 동작이 일어났을 때, 해당 변수의 정보를 동적으로 추출하기 위해 소스 코드 레벨에서 코드를 삽입한다. 코드 삽입기를 소스 코드 레벨로 구현한 이유는 삽입한 코드가 변수 이름과 줄 번호와 같은 고레벨의 소스 코드 정보를 그대로 가져올 수 있어서 분석하기 쉽고 추후에 도구를 수월하게 개선할 수 있기 때문이다. 삽입한 코드는 읽기/쓰기 동작이 일어난 해당 변수의 메모리 주소와 할당된 메모리 크기(바이트) 정보를 추출한다. 추출한 메모리 공간 (즉, 메모리 주소와 메모리 크기) 정보는 쓰기 동작이 일어나는 변수를 테인트 소스로 하고 읽기 동작이 일어나는 변수를 테인트 싱크로 하는 def-use chain들을 수집하는데 사용된다.

3.1.1.1절에서는 소스 코드 레벨에서의 읽기/쓰기 동작을 정의하고, 3.1.1.2절에서는 읽기/쓰기 동작이 일어나는 메모리 주소와 메모리 크기를 추출하는 코드를 삽입하는 방법에 대해 설명한다.

3.1.1.1 소스 코드 레벨에서 읽기/쓰기 동작

C 소스 코드 레벨에서 읽기/쓰기 동작이 일어난 경우를 다음과 같이 정의하였다.

- 쓰기 동작이 일어나는 경우
  - 대입 구문에서 대입 연산자 또는 복합 대입 연산자의 왼쪽 피연산자
  - 단항 증감 연산자(--, ++)의 피연산자
  - 메모리 공간을 할당하는 특정 C 라이브러리 함수의 전달인자 (예. memset 함수의 1번째 전달인자, strcpy 함수의 1번째 전달인자)
  - 함수의 리턴값 (예. 함수 f의 리턴값이 함수 g에서 사용되는 경우 함수 f에서 쓰기 동작이 일어났다고 간주하고 함수 f와 g 사이에 def-use chain을 생성한다)
- 읽기 동작이 일어나는 경우
  - 쓰기 동작이 일어나는 경우에 해당하지 않는 모든 변수들 (예. 대입 연산자와 복합 대입 연산자의 오른쪽에 등장하는 변수들)
  - 단항 증감 연산자(--,++)의 피연산자

읽기/쓰기 동작이 일어나는 변수가 포인티 변수 (예. **param)이거나 구조체의 멤버 변수 (a.b.x)인 경우, 해당 변수뿐만 아니라 추가로 읽기 동작이 일어나는 경우가 존재한다. 가령, **param 변수에 대해 쓰기 동작이 일어났을 때(예. **param = 2), param 포인터 변수와 *param 포인터 변수에 대해서는 읽기 동작이 일어났다고 보았다. 그 이유는 C컴파일러가 **param 포인티 변수의 메모리 공간에 값을 쓰기 위해 접근하는 과정에서 param 포인터와 *param 포인터의 메모리 주소를 읽기 때문이다. 구조체의 멤버 변수 역시 동일한 이유로 a.b.x 변수에 대해 쓰기 동작이 일어났을 때(예. a.b.x = 1), a와 a.b에 대해서도 추가적인 읽기 동작이 일어났다고 보았다.

```
1 void g(int *param) {
2   if (*param == 2); }
3
4 void f() {
5   int arr[3] = {1,2,3};
6   arr[0] = 4;
7   g(arr);}
8
9  int main() {f(); return 0;}
```

그림 3. 2: 코드 삽입 대상 프로그램 예시

```
2       if  ((fprintf(__FP,  "USE:  %p\t%lu\t%s\n",  &*param,  sizeof(*param),
"g"),*((fprintf(__FP, "USE: %p\t%lu\t%s\n", &param, sizeof(param), "g"),param)))
== 2); }
```

그림 3. 3: 쉼표 연산자를 사용한 코드 삽입 예시

```
5   int arr[3] = {1,2,3};fprintf(__FP, "DEF: %p\t%lu\t%s\n", &arr, sizeof(arr),
"f");
```

그림 3. 4: 변수 초기화 구문에 대한 코드 삽입 예시

```
6   (fprintf(__FP, "DEF: %p\t%lu\t%s\n", &arr[1], sizeof(arr[1]), "f"),arr[0] =
4);
```

그림 3. 5: 변수 정의 구문에 대한 코드 삽입 예시

3.1.1.2 소스 레벨 코드 삽입 방법

기존 코드의 의미를 바꾸지 않고 코드를 삽입하기 위해 C의 쉼표 연산자를 사용하였다. 쉼표 연산자는 첫 번째 피연산자를 실행하고 실행 결과값을 버린 다음 두 번째 피연산자를 실행하고 그 실행 결과값을 리턴하는 이진 연산자이다. 가령, b = (printf("1"),4); 라는 구문을 실행하면 문자열 1이 출력되고 변수 b의 값은 4가 된다. 어떤 변수의 메모리 정보를 추출하고자 할 때, 첫 번째 피연산자를 대상 변수의 메모리 정보를 추출하는 코드 (fprintf 함수 사용)로 하고 두 번째 피연산자를 대상 변수로 하는 쉼표 연산자를 삽입하였다.

그림 3.2는 코드 삽입 방법을 설명하기 위한 프로그램 예시이다. 그림 3.3는 그림 3.2 2번째 줄에서 읽기 동작이 일어나는 두 변수 param, *param에 대해 코드 삽입기가 코드(붉은색)를 삽입한 결과이다. 첫 번째 fprintf문은 함수 g에서 *param 포인터 변수에 해당하는 메모리 공간에 읽기 동작이 일어났다는 정보를 추출하고, 두 번째 fprintf문은 함수 g에서 param 포인터 변수에 해당하는 메모리 공간에 읽기 동작이 일어났다는 정보를 추출한다. 출력 문구 앞 부분에 있는 "USE" 문자열은 해당 출력 문구가 읽기 동작이 일어난 메모리 공간 정보임을 알려준다. 반대로 출력 문구가 쓰기 동작에 대한 정보일 경우 "DEF" 문자열이 출력 문구 앞 부분에 위치한다.

대입 연산자, 복합 대입 연산자의 왼쪽 피연산자의 경우 (그림 3.2 5번째 줄의 arr, 6번째 줄의 arr[1]), 그림 3.3과 같은 방법으로 했을 때 문법 오류가 발생하는 특별한 케이스이기 때문에 두 가지 다른 방법으로 코드를 삽입하였다.

```
1 void g(int *param) {
2   if ((fprintf(__FP, "USE: %p\t%lu\t%s\n", &*param, sizeof(*param), "g"),*((fprintf(__FP,
"USE: %p\t%lu\t%s\n", &param, sizeof(param), "g"),param))) == 2); }
3
4 void f() {
5   int arr[3] = {1,2,3};fprintf(__FP, "DEF: %p\t%lu\t%s\n", &arr, sizeof(arr), "f");
6   (fprintf(__FP, "DEF: %p\t%lu\t%s\n", &arr[1], sizeof(arr[1]), "f"),arr[0] = 4);
7   g((fprintf(__FP, "USE: %p\t%lu\t%s\n", &arr, sizeof(arr), "f"),arr));}
8
9  int main() {f(); return 0;}
```

그림 3. 6: 그림 3.2 소스 코드의 코드 삽입 결과

```
DEF: 0x7ffc267f5b30 12  f
DEF: 0x7ffc267f5b34 4   f
USE: 0x7ffc267f5b30 12  f
USE: 0x7ffc267f5b30 4   g
USE: 0x7ffc267f5b18 8   g
```

그림 3. 7: 그림 3.6에서 삽입된 코드의 출력 결과

방법1. 그림 3.2 5번째 줄과 같은 변수의 초기화 구문에서는 세미콜론(;) 바로 뒤에 코드를 삽입하였다. 그림 3.4는 그림 3.2 5번째 줄에서 쓰기 동작이 일어나는 변수 arr에 대해 코드 삽입기가 코드(붉은색)를 삽입한 결과이다. fprintf문은 함수 f에서 arr 변수에 해당하는 메모리 공간에 쓰기 동작이 일어났다는 정보를 추출한다.

방법2. 그림 3.2 6번째 줄과 같은 변수의 정의 구문에서는 그림 3.3처럼 쉼표 연산자를 사용하는데 두 번째 피연산자를 해당 변수가 아닌 구문 전체로 하는 쉼표 연산자를 사용하였다. 그림 3.5는 그림 3.2 6번째 줄에서 쓰기 동작이 일어나는 변수 arr[0]에 대해 코드 삽입기가 코드(붉은색)를 삽입한 결과이다. fprintf문은 함수 f에서 arr[0] 변수에 해당하는 메모리 공간에 쓰기 동작이 일어났다는 정보를 추출한다.

그림 3.6는 그림 3.2의 소스 코드에 코드를 삽입한 결과이고, 그림 3.7은 그림 3.6의 프로그램을 실행했을 때, 삽입한 코드가 출력하는 결과물이다. 그림 3.7에서 1번째 열은 메모리 공간에 대한 쓰기 동작("DEF")인지 읽기 동작("USE")인지 보여주고, 2번째 열과 2번째 열은 각각 해당 메모리 공간의 주소와 크기(바이트)를 나타낸다. 4번째 열은 읽기/쓰기 동작이 일어난 함수 이름을 보여준다.

전역 변수나 포인터가 가리키는 메모리 공간에 대한 읽기/쓰기 동작(그림 3.6)에 대해서는 앞서 설명한 코드 삽입 방법만으로 충분히 def-use chain을 추출할 수 있다. 읽기 동작이 일어나는 메모리 공간과 쓰기 동작이 일어나는 메모리 공간 사이에 겹치는 부분이 존재하는지 여부만 확인하면 되기 때문이다. 그림 3.7에서의 출력 결과에서 쓰기 동작이 일어난 1번째 줄의 메모리 공간(그림 3.6 5번째 줄에 삽입된 코드가 출력하는 배열 arr에 대한 정보)과 읽기 동작이 일어난 5번째 줄의 메모리 공간(그림 3.6 2번째 줄에 삽입된 코드가 출력하는 *param에 대한 정보)이 4바이트만큼 겹치기 때문에 함수 f에서 함수 g로 데이터가 흐르는 def-use chain이 존재한다는 사실을 쉽게 알 수 있다.

```
1  void g(int z) {
2    int y = z;
3    if (y > 0); }
4
5  void f() {
6    int x = 10;
7    g(x); }
8
9  int main() {f(); return 0;}
```

그림 3. 8: 변수 간 대입 연산자와 pass-by-value를 사용하는 프로그램 예시

```
1  void g(int z) {fprintf(__FP, "PARAM: %d\t%p\t%lu\t%s\n", 1, &z, sizeof(z), "g");
2    int y = (fprintf(__FP, "ASSIGN: %p\t%lu\t%p\t%lu\t%s\n", &y, sizeof(y), &z,
sizeof(z), "g"),fprintf(__FP, "USE: %p\t%lu\t%s\n", &z, sizeof(z), "g"),z);
fprintf(__FP, "DEF: %p\t%lu\t%s\n", &y, sizeof(y), "g");
3    if ((fprintf(__FP, "USE: %p\t%lu\t%s\n", &y, sizeof(y), "g"),y) > 0); }
4
5  void f() {
6    int x = 10; fprintf(__FP, "DEF: %p\t%lu\t%s\n", &x, sizeof(x), "f");
7    g((fprintf(__FP, "USE: %p\t%lu\t%s\n", &x, sizeof(x), "f"),fprintf(__FP,
"ARG: %s\t%d\t%p\t%lu\t%s\n", "g", 1, &x, sizeof(x), "f"),x)); }
8
9  int main() {f(); return 0;}
```

그림 3. 9: 그림 3.8 소스 코드의 코드 삽입 결과

```
DEF: 0x7fff27301b74 4   f
USE: 0x7fff27301b74 4   f
ARG: g  1   0x7fff27301b74  4   f
PARAM: 1    0x7fff27301b4c  4   g
ASSIGN: 0x7fff27301b54  4   0x7fff27301b4c  4   g
USE: 0x7fff27301b4c 4   g
DEF: 0x7fff27301b54 4   g
USE: 0x7fff27301b54 4   g
```

그림 3. 10: 그림 3.9에서 삽입된 코드의 출력 결과

하지만 대입 연산자 또는 pass-by-value로 함수 f에서 정의한 변수 x의 값을 함수 g에서 사용된 변수 y로 넘긴 경우, 쓰기 동작이 일어난 x의 메모리 공간과 읽기 동작이 일어난 y의 메모리 공간이 다르기 때문에 앞서 설명한 코드 삽입 방법만으로는 x와 y 사이의 def-use chain을 추출하지 못한다. 상기한 문제를 해결하기 위해 각 변수들 간의 연결고리 정보(예. x의 값이 y에 대입되었다는 정보)도 추출할 수 있도록 추가적으로 코드를 삽입하였다.

그림 3.8은 변수 간 대입 연산자와 pass-by-value를 사용하는 프로그램 예시이다. 함수 f 6번째 줄에서 쓰기 동작이 일어난 변수 x와 함수 g 3번째 줄에서 읽기 동작이 일어난 변수 y가 서로 메모리 주소는 다르지만 def-use 관계가 있다는 것을 육안으로 알 수 있다. 그림 3.9은 그림 3.8의 프로그램에 추가적인 코드 삽입(파란색)을 포함해 코드를 삽입한 결과 프로그램이고, 그림 3.10는 그림 3.9의 코드를 실행했을 때, 삽입한 코드가 출력하는 결과물이다. 그림 3.10의 "ARG"로 시작하는 출력문과 "PARAM"으로 시작하는 출력문은 한 쌍을 이루어 pass-by-value를 통해 함수 f의 변수 x의 값이 함수 g의 1번째 파라미터 z로 값이 전달되었다는 정보를 준다. "ARG"로 시작하는 출력문에서 1번째 열과 2번째 열은 함수 g의 1번째 전달인자 x에 대한 정보라는 것을 나타내고, 3번째 열과 4번째 열은 x의 메모리 주소와 크기를 나타내며, 5번째 열은 함수 g를 호출한 함수 이름이다. 그림 3.10의 "PARAM"으로 시작하는 출력문에서 1번째 열은 1번째 파라미터 z에 대한 정보라는 것을 나타내고, 2번째 열과 3번째 열은 z의 메모리 주소와 크기를 나타낸다. 그림 3.10의 "ASSIGN"으로 시작하는 출력문은 그림 3.8의 2번째 줄에서 대입 연산자에 의해 변수 z의 값이 변수 y에 전달되었다는 정보를 준다. 해당 출력문에서 1번째 열과 2번째 열은 각각 변수 y의 메모리 주소와 크기를 나타내고, 3번째 열과 4번째 열은 각각 변수 z의 메모리 주소와 크기를 나타낸다. 3.1.2절에서 추가적인 정보를 바탕으로 def-use chain을 수집하는 방법을 설명한다.

3.1.2 Def-Use Chain 수집기 (Def-Use Chain Collector)

Def-use chain 수집기는 앞서 언급했듯이 주어진 시스템 테스트 케이스들로 코드가 삽입된 대상 프로그램을 실행하고, 각 시스템 테스트 케이스 별로 프로그램의 모든 함수 쌍에 대해 한 함수에서 변수를 메모리 공간에 할당하고 다른 함수에서 할당된 메모리 공간을 읽는 def-use chain들을 수집한다. 기본적으로 def-use chain 수집기는 삽입 코드의 출력 결과를 위에서부터 순서대로 읽고, 읽기 동작("USE") 출력문일 때 이전에 등장했던 모든 쓰기 동작("DEF") 출력문들과 비교해서 겹치는 메모리 공간이 있는지 여부로 두 함수 간 def-use chain들을 수집한다. (그림 3.7 예시)

만약 삽입 코드의 출력 결과에 "ARG" 출력문과 "PARAM" 출력문 쌍이나 "ASSIGN" 출력문이 있어서 변수 x의 값을 변수 y에 넘긴다는 정보를 얻은 경우, x와 y의 메모리 공간을 연결하는 연결 리스트를 생성한다. 그래서 이후 y의 메모리 공간에 읽기 동작("USE")이 일어났을 때, 연결 리스트에 의해 y의 메모리 공간과 연결된 x의 메모리 공간도 읽기 동작("USE")이 일어난 것으로 간주해 이전에 등장했던 모든 쓰기 동작("DEF") 출력문들과 비교해서 겹치는 메모리 공간이 있는지 여부로 서로 다른 두 함수 간 def-use chain들을 수집한다.

각 def-use chain 별로 데이터 흐름의 크기 (즉, 바이트 수)도 수집하였다. 데이터 흐름의 크기는 쓰기 동작이 일어난 메모리 공간과 읽기 동작이 일어난 메모리 공간의 교집합에 해당하는 크기이다. 예를 들어, 앞서 그림 3.7에서 추출한 def-use chain 1개의 데이트 흐름의 크기는 4바이트이다.

3.1.3 데이터 흐름 분석기 (Data Flow Analyzer)

데이터 흐름 분석기는 각 테스트 케이스 별로 수집한 함수 간 def-use chain들을 바탕으로 함수 간 데이터 의존도를 측정한다. 함수 f에 대한 함수 g의 데이터 의존도는 주어진 시스템 TC를 실행했을 때 g가 f로부터 받는 데이터 양의 평균으로 계산된다. 대상 프로그램에 두 함수 f와 g가 존재하고, 주어진 전체 n개 시스템 테스트 케이스 중 함수 g를 실행하는 시스템 테스트 케이스가 n_g개 있다고 가정하자. 주어진 n개 시스템 테스트 케이스들로 대상 프로그램을 실행해서 수집한 함수 f에서 함수 g로의 def-use chain들을 바탕으로 함수 f에 대한 함수 g의 데이터 의존도 $\mathrm{data}(g \leftarrow f)$를 다음과 같이 계산하였다.

$$\mathrm{data}(g \leftarrow f) = \frac{\sum_{k=1}^{n_g} num_k(f,g)}{n_g}$$

$\mathrm{num}_k(f,g)$란 k번째 시스템 테스트 케이스로 수집한 함수 f에서 g로의 데이터 흐름의 크기이다.

효과적인 concolic 탐색 전략을 위해 함수 간 실제 데이터 의존도를 반영하는 $num_k(f,g)$를 구하기 위해 3가지 design choice들을 고려하였다.

3.1.3.1 Design Choice 1: 함수 간 데이터 흐름의 크기를 수치화하는 방법

함수 f에서 g로의 데이터 흐름의 크기 $num_k(f,g)$를 수치화할 때, 다음과 같은 2가지 방법을 생각할 수 있다.

방법1. 함수 f에서 함수 g로의 def-use chain 수의 총합으로 하는 방법
방법2. 함수 f에서 함수 g로의 def-use chain 바이트 수의 총합으로 하는 방법

방법2는 함수 f에서 함수 g로 데이터를 보낼 때, 구조체 같이 크기가 큰 데이터를 보내는 경우와 단일 문자 같이 크기가 작은 데이터를 보내는 경우를 구분해서 크기가 큰 데이터를 보냈을 때 함수 f에 대한 함수 g의 데이터 의존도 $\mathrm{data}(g \leftarrow f)$가 더 높게 계산된다는 점에서 방법1과 차이가 있다.

그림 3.11은 같은 프로그램에 대해서 방법1을 사용했을 때와 방법2를 사용했을 때 함수 간 데이터 흐름의 크기가 어떻게 달라지는지 보여준다. 방법1을 사용할 경우, 함수 f1에서 함수 g로 2개의 def-use chain이 존재하고, 함수 f2에서 함수 g로 1개의 def-use chain이 존재하므로, f1이 f2보다 g의 실행에 끼치는 영향력이 크다고 해석할 수 있다. 반면, 방법2를 사용할 경우에는 반대로 f2가 f1보다 g의 실행에 끼치는 영향력이 크다고 해석할 수 있다. 함수 f1에서 정의된 2개 변수가 모두 1바이트 크기의 char 타입 변수이기 때문에 f1에서 g로 2바이트의 def-use chain이 존재하고, 함수 f2에서 정의한 변수는 4바이트 크기의 int 타입 변수이기 때문에 f2에서 g로 4바이트의 def-use chain이 존재한다.

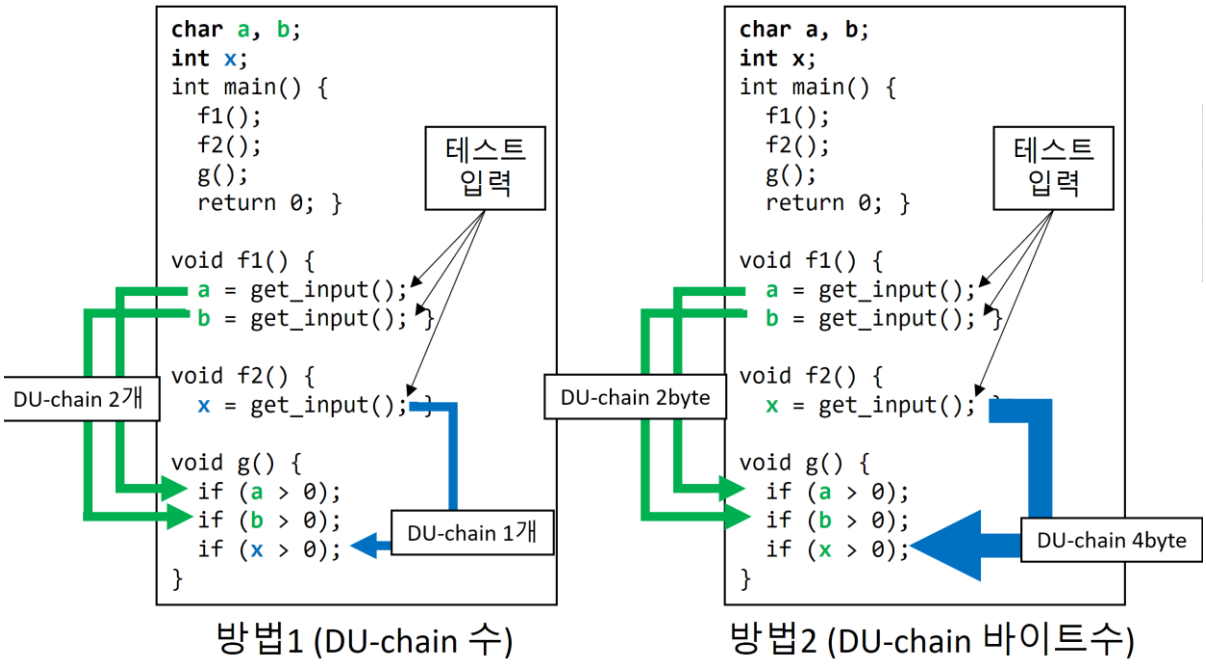


그림 3. 11: Design Choice 1 설명 예시

본 논문에서는 방법1을 채택하였다. Concolic 테스팅 관점에서 g의 수행 경로를 결정하는 변수는 a, b, x 총 3개인데 그 중 f1에서 정의된 변수가 2개로 f2보다 많기 때문에, f1이 f2보다 g에 끼치는 영향력이 더 크다고 보는 것이 적절하다.

Design Choice 1 $\text{num}_k(f,g)$는 k번째 시스템 테스트로 수집한 함수 f에서 함수 g로의 def-use chain 수의 총합으로 계산한다.

3.1.3.2 Design Choice 2: 모든 def-use chain을 고려해야 하는가

$num_k(f,g)$ 를 계산할 때, 다음과 같은 2가지 방법을 생각할 수 있다.

방법1. def-use chain 수집기가 수집한 모든 def-use chain들을 고려해서 계산한다.

방법2. 조건문의 조건식에 사용된 변수들을 use로 하는 def-use chain들만 고려해서 계산한다.

방법1은 함수 f에서 g로의 모든 def-use chain들이 g의 실행에 영향을 끼친다고 해석하고, 방법2는 그 중 일부만 g의 실행에 영향을 끼친다고 해석한다.

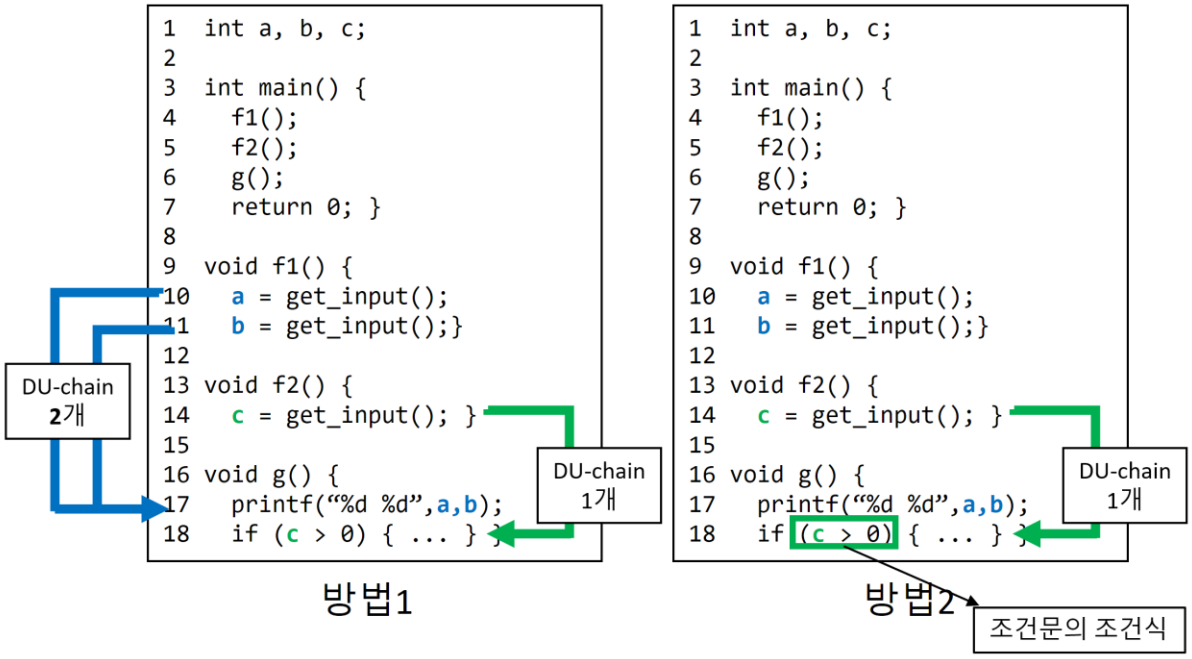


그림 3. 12: Design Choice 2 설명 예시

그림 3.12는 같은 프로그램에 대해서 방법1을 사용했을 때와 방법2를 사용했을 때 함수 간 데이터 흐름의 크기가 어떻게 달라지는지 보여준다. 방법1을 사용할 경우, 함수 f1에서 함수 g로 2개의 def-use chain이 존재하고 함수 f2에서 함수 g로 1개의 def-use chain이 존재하므로, f1이 f2보다 g의 실행에 끼치는 영향력이 크다고 해석할 수 있다. 방법2를 사용할 경우, 조건문의 조건식에 사용된 변수가 c 하나뿐이므로, f1에서 g로의 def-use chain은 존재하지 않고 f2에서 g로의 def-use chain 1개만 존재한다. 따라서, f2가 f1보다 g의 실행에 끼치는 영향력이 더 크다고 해석할 수 있다.

본 논문에서는 방법2를 채택하였다. 함수 f1에서 정의한 두 변수 a와 b는 g의 조건문의 조건식에 사용되지 않기 때문에 g의 수행 경로를 결정하지 않지만, 함수 f2에서 정의한 변수 c는 g의 수행 경로를 결정하므로, f2가 f1보다 g에 끼치는 영향력이 더 크다고 보는 것이 적절하다.

Design Choice 2
모든 읽기 동작 중 조건문의 분기 조건에 일어난 읽기 동작만 고려해 def-use chain을 수집한다.

3.1.3.3 Design Choice 3: 함수가 여러 번 호출되어 def-use chain이 중복 등장하는 케이스 처리

한 execution에서 한 함수가 여러 번 호출되는 경우, 같은 메모리 공간을 대상으로 def-use chain이 중복해서 여러 번 등장하는 경우가 생긴다. 그림 3.13에서 함수 f가 반복문을 통해 함수 g를 10000번 호출하고 있고, 호출할 때마다 변수 i 값을 다르게 정의해서 g의 파라미터로 전달하고 있다. 여기서 $num_k(f,g)$를 계산하는 2가지 방법을 다음과 같이 생각해볼 수 있다.

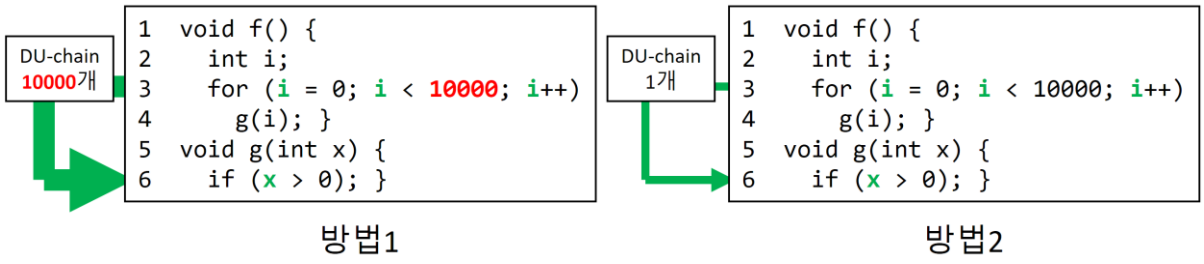


그림 3. 13: Design Choice 3 설명 예시

방법1. 함수를 호출할 때마다 발생하는 def-use chain들의 모든 개수를 고려한다.

방법2. 중복된 def-use chain들을 1개로 처리한다.

방법1을 사용할 경우에는 f에서 g로의 def-use chain 수가 10000개이고, 방법2를 사용할 경우에는 10000개의 def-use chain을 1개로 처리한다.

본 논문에서는 방법2를 채택하였다. 방법1을 사용하면 $num_k(f,g)$의 값이 매우 커져서 f가 g에 끼치는 영향력이 매우 크다고 해석되지만, concolic 테스팅 관점에서 서로 다른 10000개의 변수 값이 모두 g의 분기 커버리지를 높이는데 관여하지 않는다. 또한, def-use chain이 10000개라고 해서 def-use chain이 1개일 때보다 g에 10000배 더 큰 영향력을 끼친다고 보기 어렵기 때문에 방법2를 채택하는 것이 적절하다.

Design Choice 3
같은 메모리 공간에 대해서 발생한 중복된 데이터 흐름을 데이터 흐름 1번으로 처리한다.

3.1.4 동적 테인트 분석 도구의 구현

코드 삽입기(1682 LOC)는 Clang-4.0으로 구현하였다. Clang은 C/C++ 프로그램을 추상 구문 트리 (Abstract Syntax Tree, AST)로 변환하고 그 추상 구문 트리를 수정하는 기능을 제공하는 라이브러리이다. Clang을 사용하면 C/C++ 프로그램을 소스 코드 레벨에서 손쉽게 수정할 수 있다. 예를 들어, 어떤 조건문이 실행되는지 확인하기 위해 프로그램의 모든 조건문 앞에 printf문을 삽입하고자 할 때, 혹은 널 포인터 역참조(Null Pointer Dereference)가 일어나는지 확인하기 위해 포인터 변수 ptr을 참조하기 직전에 assert(ptr != null)를 삽입하고자 할 때, Clang으로 구현된 코드 삽입기를 사용하면 기계적으로 원하는 코드 구문을 삽입할 수 있다.

def-use chain 수집기(602 LOC)와 데이터 흐름 분석기(171 LOC)는 파이썬 2.7로 구현하였다. def-use chain 수집기에서 쓰기 동작이 일어난 메모리 구간([메모리 주소, 메모리 주소 + 크기])과 읽기 동작이 일어난 메모리 구간 사이에 겹치는 구간을 찾을 때 속도를 개선하기 위해 interval tree 자료 구조를 사용하였다. Interval tree는 구간들을 저장하는 트리 자료 구조로, 주어진 구간과 겹치는 모든 구간들을 효율적으로 찾는데 사용한다. Interval tree에 저장된 구간들의 개수가 n개이고, 주어진 구간과 겹치는 구간들이 interval tree 속에 m개 있을 때, $O(\log n + m)$의 시간으로 m개의 겹치는 구간들을 찾는다[11].

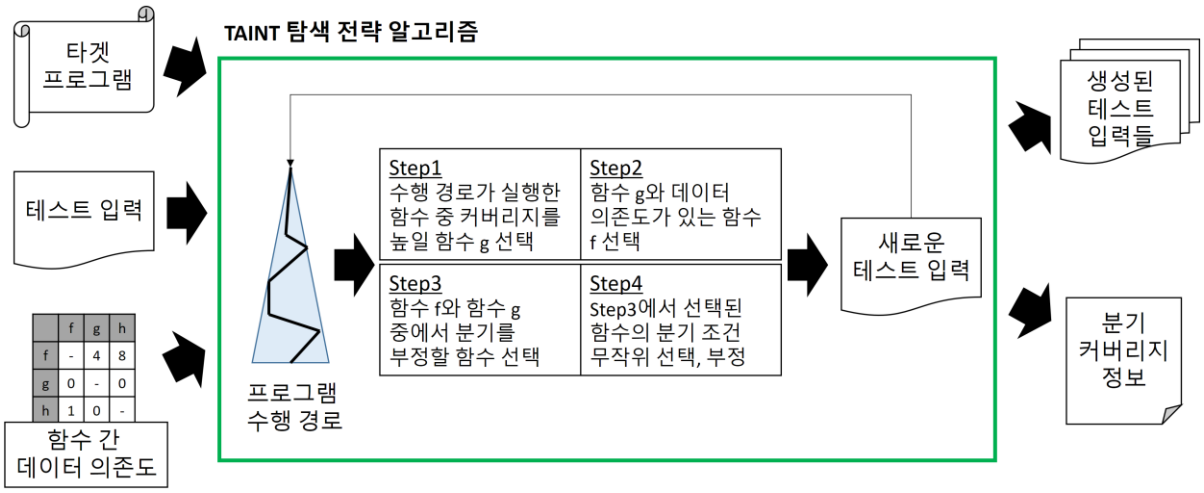


그림 3. 14: Taint 탐색 전략의 전체 흐름

3.2 동적 테인트 분석을 활용한 Concolic 탐색 전략(Taint)의 알고리즘

3가지 design choice들을 바탕으로 정의한 함수 간 데이터 의존도를 활용해 새로운 concolic 탐색 전략 Taint를 설계하였다. Taint는 타겟 함수 g의 커버하지 못한 분기를 커버하기 위해, 함수 g의 분기 조건을 부정(negate)하거나 g에 영향력이 큰 함수의 분기 조건을 부정해서 커버리지를 높인다. 그림 3.14는 Taint 탐색 전략 알고리즘의 전체적인 흐름을 보여준다. 타겟 프로그램, 초기 테스트 케이스, 그리고 측정한 함수 간 데이터 의존도를 입력으로 4개의 step의 반복 실행해서 생성된 테스트 케이스들과 분기 커버리지 정보를 출력한다. 대상 프로그램이 초기 테스트 케이스를 가지고 있을 경우, 초기 테스트 케이스로 함수 간 데이터 의존도를 측정하고, 그렇지 않을 경우 random 탐색 전략을 사용한 concolic 테스팅을 수행해 10개의 초기 테스트 케이스를 생성해서 함수 간 데이터 의존도를 측정한다. 4개의 step에 대한 자세한 설명은 다음과 같다.

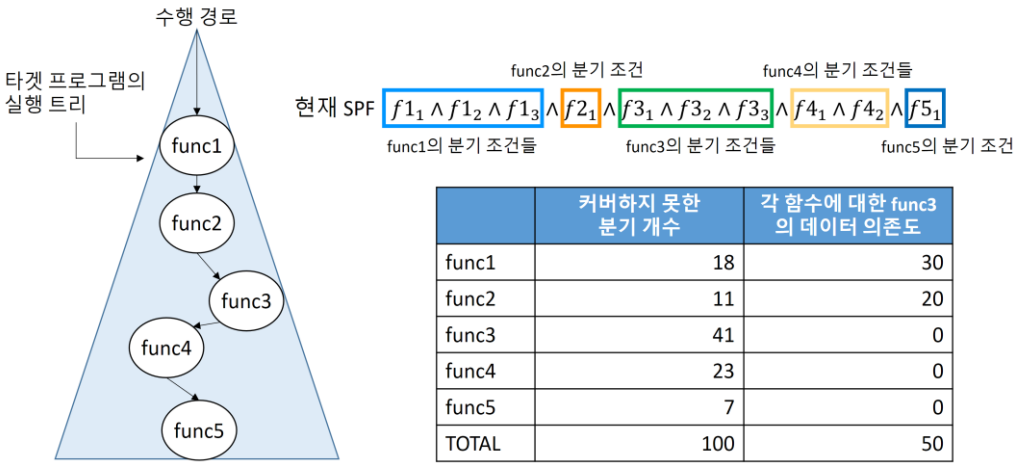
Step1. 테스트 케이스의 수행 경로가 실행한 함수들 중 커버리지를 높일 타겟 함수(g라고 함) 선택

타겟 함수 g를 선택하는 기준은 함수 별 커버하지 못한 분기 개수에 비례한 확률로 무작위 선택한다. 커버하지 못한 분기 개수가 많은 함수의 우선 순위가 높은 이유는 커버하지 못한 분기가 많을수록 커버리지를 높일 가능성이 크기 때문이다.

현재 심볼릭 경로 수식이 실행한 n개 함수 $f_1, f_2, \dots, f_n$가 커버하지 못한 분기 개수가 각각 $b_1, b_2, \dots, b_n$이라고 할 때, 함수 f_k $(1 \le k \le n)$가 타겟 함수 f로 선택될 확률은 $\frac{b_k}{\sum_{i=1}^{n} b_i}$이다.

Step2. g에 영향력을 끼치는 (즉 데이터 의존도가 있는) 함수들 중 한 함수 (f라고 함) 선택

함수 f를 선택하는 기준은 g를 선택하는 기준은 g의 데이터 의존도 수치(즉, g에 영향력을 끼치는 정도)에 비례한 확률로 무작위 선택한다. g의 데이터 의존도가 큰 함수의 우선 순위가 높은 이유는 데이터 의존도가 클수록 그 함수의 분기를 부정했을 때 g의 실행 경로에 큰 영향을 끼치기 때문이다.



	커버하지 못한 분기 개수	각 함수에 대한 func3의 데이터 의존도
func1	18	30
func2	11	20
func3	41	0
func4	23	0
func5	7	0
TOTAL	100	50

그림 3. 15: Taint 탐색 전략 설명을 위한 예시

현재 심볼릭 경로 수식이 실행하고 타겟 함수 g로의 def-use chain이 존재하는 m개 함수가 $f_1, f_2, \ldots, f_m$이고, 함수 f_k $(1 \le k \le m)$에 대한 타겟 함수 g의 데이터 의존도 $data(g \leftarrow f_k)$를 d_k라고 할 때, 함수 f_k가 g에 영향력이 큰 함수 f로 선택될 확률은 $\frac{d_k}{\sum_{i=1}^{m} d_i}$이다.

Step3. 두 함수 f와 g 중 분기 조건을 부정할 함수 1/2 확률로 무작위 선택

Step3에서 f가 선택될 경우 f의 분기 조건을 부정해서 g의 분기 커버리지 증가를 기대할 수 있고, g가 선택될 경우 g의 분기 조건을 부정해서 g의 분기 커버리지 증가를 기대할 수 있다. 만약 g에 영향력을 끼치는 함수가 존재하지 않아 Step2에서 f가 선택되지 않은 경우에는, Step3에서 타겟 함수 g가 자동으로 선택된다.

Step4. 현재 심볼릭 경로 수식의 분기 조건 중 Step3에서 선택한 함수의 분기 조건 하나를 균등한 확률로 무작위 선택해서 부정한 새로운 심볼릭 경로 수식 생성

현재 심볼릭 경로 수식의 분기 조건들 중에 Step3에서 선택한 분기 조건을 부정할 함수의 분기조건이 총 x개 있다고 할 때, 1/x의 확률로 부정할 분기 조건을 무작위 선택해서 부정한다. 분기 조건을 부정한 새로운 심볼릭 경로 수식을 만족하는 새로운 테스트 케이스를 생성하였을 경우, 생성된 테스트 케이스로 프로그램을 실행해서 함수 별 커버하지 못한 분기 개수를 갱신하고 다시 Step1을 수행한다.

그림 3.15는 테스트 케이스의 수행 경로와 심볼릭 경로 수식, 그리고 수행 경로가 실행하는 각 함수 별 커버하지 못한 분기 개수와 g의 데이터 의존도를 보여준다. 이 예시를 이용해 각 Step을 설명하면 다음과 같다.

Step1. 커버리지를 높일 타겟 함수 g로 func3가 41/100의 확률로 선택되었다.

Step2. g에 영향력이 큰 함수 f로 func1이 30/50의 확률로 선택되었다.

Step3. func1과 func3 중 1/2의 확률로 func1이 분기를 부정할 함수로 선택되었다.

Step4. 현재 심볼릭 경로 수식에서 func1의 분기 조건 3개 중 마지막 분기 조건인 $f1_3$이 1/3의 확률로 선택되었다. 따라서, 새로운 심볼릭 경로 수식은 $f1_1 \wedge f1_2 \wedge \neg f1_3$가 된다.

제 4 장 실험

4.1 타겟 프로그램 코드 정보

본 연구는 두 논문[57][58]의 실험에서 사용한 대한민국 국방 회사에서 자체 개발한 7개 안전 필수 C 프로그램을 대상으로 실험을 진행하였다. 7개 프로그램 모두 DFS 탐색 전략으로 concolic 테스팅했을 때 1분 이상이 지나도 모든 실행 경로를 탐색하지 못하는 프로그램들이다. 표 4.1은 각 타겟 프로그램에 대한 코드 정보이다.

표 4. 1: 타겟 프로그램 코드 정보

Target Program	# Func	LOC	# Branch
Prog1	2	29	40
Prog2	28	149	46
Prog3	32	588	390
Prog4	20	294	203
Prog5	10	136	50
Prog6	18	309	165
Prog7	26	461	316
Avg	19.4	280.8	172.8

4.2 연구 문제

Taint 탐색 전략의 효과성을 측정하기 위해 다음과 같은 연구 질문을 작성하였다.

RQ1. Taint 탐색 전략이 기존의 concolic 탐색 전략(DFS, rev-DFS, CFG, random)에 비해 같은 수의 테스트 케이스로 분기 커버리지를 얼마나 높이 달성했는가?

RQ2. Taint 탐색 전략이 기존의 concolic 탐색 전략(DFS, rev-DFS, CFG, random)에 비해 같은 시간 동안 분기 커버리지를 얼마나 높이 달성했는가?

RQ3. Design choice 1이 분기 커버리지를 높이는데 얼마나 효과적인가?

4.3 실험 설계

타겟 프로그램들은 초기 테스트 케이스를 가지고 있지 않기 때문에 random 탐색 전략을 사용한 concolic 테스팅을 수행해 10개의 초기 테스트 케이스를 생성하고 이를 바탕으로 함수 간 데이터 의존도를 측정하였다.

RQ1: 각 타겟 프로그램에 대해서 CROWN을 사용해 1만개 테스트 케이스를 생성하도록 각 concolic 탐색 전략 별로 concolic 테스팅 실험을 진행했다. 동일한 입력 변수 동일한 심볼릭 변수)를 대상으로 concolic 테스팅을 진행하였고, CFG, random, Taint 탐색 전략의 경우 자체적으로 불확실성을 갖기 때문에 10번 실험을 진행해 달성 커버리지를 평균 내었다.
RQ2: 각 타겟 프로그램에 대해서 CROWN을 사용해 10분의 시간만큼 각 concolic 탐색 전략 별로 concolic 테스팅 실험을 진행했다. Taint 탐색 전략을 사용했을 때, 10분의 시간 안에 동적 테인트 분석 도구를 적용한 시간(평균 20초)도 포함하였다. 즉, 실제로 concolic 테스팅을 돌린 시간이 다른 concolic 탐색 전략보다 짧다. 동일한 입력 변수 (즉, 동일한 심볼릭 변수)를 대상으로 concolic 테스팅을 진행하였고, CFG, random, Taint 탐색 전략의 경우 자체적으로 불확실성을 갖기 때문에 10번 실험을 진행해 달성 커버리지를 평균 내었다.
RQ3: 각 프로그램에 대해서 3.1.3.1절에서 언급한 방법1(Taint 탐색 전략)과 방법2(Taint-bytes 탐색 전략)을 사용해 같은 수의 테스트 케이스일 때 (1만개 테스트 케이스 생성), 그리고 같은 시간 동안 concolic 테스팅을 돌렸을 때 (10분) 달성한 분기 커버리지의 차이를 비교하였다. Taint와 Taint-bytes 모두 자체적으로 불확실성을 갖기 때문에 10번 실험을 진행해 달성 커버리지를 평균 내었다.
실험은 i5-9600K CPU @ 3.70GHz의 CPU를 가진 Ubuntu Linux 16.04 LTS 서버에서 진행하였고, Concolic 테스팅 도구로 CROWN을 사용하였다.

4.4 실험 결과

4.4.1. RQ1에 대한 결론

표 4.2는 각 대상 프로그램에 대해 DFS, rev-DFS, CFG, random, 그리고 Taint 탐색 전략을 사용하여 1만 개 테스트 케이스를 생성했을 때, 달성한 분기 커버리지와 걸린 시간을 보여준다. Prog1에 대해 DFS와 rev-DFS 탐색 전략을 사용한 경우, 1만 개보다 적은 6144개 테스트 케이스로 모든 경로를 커버하였다.

같은 테스트 케이스 수로 Taint 탐색 전략이 다른 4개의 탐색 전략보다 효과적으로 분기 커버리지를 높였다. Taint 탐색 전략은 7개의 대상 프로그램에 대해 평균 74.4% 분기 커버리지를 달성하여, 다른 4개의 탐색 전략이 달성한 분기 커버리지보다 1.8%p~7.1%p 더 높이 달성했다. Prog4와 Prog6을 제외한 5개 프로그램에 대해 Taint 탐색 전략이 가장 높은 분기 커버리지를 달성했다.

4.4.2. RQ2에 대한 결론

표 4.3은 각 대상 프로그램에 대해 DFS, rev-DFS, CFG, random, 그리고 Taint 탐색 전략을 사용하여 10분 동안 테스트 케이스를 생성했을 때, 달성한 분기 커버리지와 걸린 시간을 보여준다.

표 4. 2: 1만개 TC 생성 시 각 Concolic 탐색 전략의 달성 분기 커버리지

Target	DFS		rev-DFS		CFG		random		Taint	
	Cov (%)	Time (s)	Cov (%)	Time (s)	Cov (%)	Time (s)	Cov (%)	Time (s)	Cov (%)	Time (s)
Prog1	100.0	102	100.0	101	100.0	182	100.0	188	100.0	171
Prog2	87.0	117	87.0	118	87.0	119	87.0	121	87.0	135
Prog3	77.4	373	79.2	373	80.0	347	71.3	377	81.0	388
Prog4	30.5	182	46.8	182	64.5	177	67.5	171	66.5	197
Prog5	64.0	179	68.0	179	70.0	209	62.0	188	70.0	192
Prog6	67.9	278	69.1	278	64.8	231	61.2	261	67.3	281
Prog7	44.0	189	42.4	189	41.8	225	48.1	185	49.1	204
Avg.	67.3	202.9	70.4	202.9	72.6	212.9	71.0	213.0	74.4	224.0

표 4. 3: 10분 동안 TC 생성 시 각 Concolic 탐색 전략의 달성 분기 커버리지

Target	DFS		rev-DFS		CFG		random		Taint	
	Cov (%)	Time (s)	Cov (%)	Time (s)	Cov (%)	Time (s)	Cov (%)	Time (s)	Cov (%)	Time (s)
Prog1	100.0	102	100.0	101	100.0	600	100.0	600	100.0	600
Prog2	87.0	312	87.0	310	87.0	600	87.0	600	87.0	600
Prog3	79.7	600	80.8	600	81.0	600	72.6	600	81.5	600
Prog4	41.9	600	62.1	600	68.5	600	69.5	600	70.4	600
Prog5	70.0	600	72.0	600	72.0	600	72.0	600	72.0	600
Prog6	69.1	600	69.7	600	68.5	600	68.5	600	67.9	600
Prog7	45.3	600	48.4	600	44.6	600	49.1	600	50.9	600
Avg.	70.4	487.7	74.3	487.3	74.5	600.0	74.1	600.0	75.7	600.0

Prog1에 대해 DFS와 rev-DFS 탐색 전략을 사용한 경우, 102초만에 6144개 테스트 케이스를 생성하여 모든 경로를 커버하였다.

같은 테스팅 시간 동안 Taint 탐색 전략이 다른 4개의 탐색 전략보다 효과적으로 분기 커버리지를 높였다. Taint 탐색 전략은 7개의 대상 프로그램에 대해 평균 75.7% 분기 커버리지를 달성하여, 다른 4개의 탐색 전략이 달성한 분기 커버리지보다 1.2%p~5.3%p 더 높이 달성했다. 또한, Prog6을 제외한 6개 프로그램에 대해 Taint 탐색 전략이 가장 높은 분기 커버리지를 달성했다.

DFS, rev-DFS, CFG, random, Taint 탐색 전략이 생성한 평균 TC 수는 각각 24765.6, 24285.7, 26264.1, 25576.4, 24247.9개로 RQ1에서 생성한 10000개 TC보다 많다. 이는 concolic 테스팅을 오래 돌려도 여전히 Taint 탐색 전략이 다른 탐색 전략보다 높은 분기 커버리지를 달성한다는 것을 보여준다.

표 4. 4: 1만 개 TC 생성 시 Taint와 Taint-bytes의 달성 분기 커버리지

Target	Taint		Taint-bytes	
	Cov (%)	Time (s)	Cov (%)	Time (s)
Prog1	100.0	171	100.0	195
Prog2	87.0	135	87.0	126
Prog3	81.0	388	79.0	369
Prog4	66.5	197	65.5	166
Prog5	70.0	192	72.0	165
Prog6	67.3	281	68.5	334
Prog7	49.1	204	47.2	176
Avg.	74.4	224.0	74.2	218.7

표 4. 5: 10분 동안 TC 생성 시 Taint와 Taint-bytes의 달성 분기 커버리지

Target	Taint		Taint-bytes	
	Cov (%)	Time (s)	Cov (%)	Time (s)
Prog1	100.0	600	100.0	600
Prog2	87.0	600	87.0	600
Prog3	81.5	600	80.0	600
Prog4	70.4	600	68.5	600
Prog5	72.0	600	72.0	600
Prog6	67.9	600	69.7	600
Prog7	50.9	600	51.3	600
Avg.	75.7	600.0	75.5	600.0

4.4.3. RQ3에 대한 결론

표 4.4은 각 대상 프로그램에 대해 Taint와 Taint-bytes 탐색 전략을 사용하여 1만 개 테스트 케이스를 생성했을 때, 달성한 분기 커버리지와 걸린 시간을 보여준다. 표 4.5는 각 대상 프로그램에 대해 Taint와 Taint-bytes 탐색 전략을 사용하여 10분 동안 테스트 케이스를 생성했을 때, 달성한 분기 커버리지와 걸린 시간을 보여준다.

같은 테스트 케이스 수일 때와 같은 테스팅 시간일 때 모두 Taint 탐색 전략이 Taint-bytes 탐색 전략보다 분기 커버리지가 높게 나왔다. Taint 탐색 전략은 7개의 대상 프로그램에 대해 같은 테스트 케이스 수일 때 평균 74.4% 분기 커버리지를 달성하였고, 같은 캐스팅 시간일 때 평균 75.7% 분기 커버리지를 달성하여, Taint-bytes 탐색 전략이 달성한 분기 커버리지보다 각각 0.2%p, 0.2%p 높이 달성했다.

```
1  unsigned f(struct node *n, char **parentKey, char *key, char *value) {
2    while (n != NULL) {
3      ...
4      if (n->isLocal) {
5        if (n->isSocket) {
6          result = h(key, value, “isSocket”);
7          ... }
8        else {
9          int i;
10         for (i = 0; i < 20; i++) {
11           if (!SYM_strcmp(parentKey[i], “cli”)) { ... }
12           else if (!SYM_strcmp(parentKey[i], “relay”)) { ... }
13         } }
14       target_g(n, key, value); }
15     ... }
16 }
17 void target_g(struct node *n, char *key, char *value) {
18   ...
19   while (...) {
20     if (!SYM_strcmp(key, “type”)) {
21       if (!SYM_strcmp(value, “serial”)) { ... };
22       else if (!SYM_strcmp(value, “tcp”)) { ... };
23       else if (!SYM_strcmp(value, “inet”)) { ... };
24       else if (!SYM_strcmp(value, “socket”)) {/* many branches uncovered*/};
25       else { ... };
26     }
27   ...
28 }}
29 int SYM_strcmp(const char *s1, const char *s2) {
30   while (*s1 && (*s1 == *s2))
31     s1++,s2++;
32   return *(const unsigned char*)s1-*(const unsigned char*)s2;
33 }
34 unsigned h(char *key, char *value, char *str) {
35   ...
36   if (!SYM_strcmp(str, “isSocket”)) value = “socket”;
37   ...
38 }
```

그림 4. 1: Prog3 소스코드 예시

4.4.4. Taint가 다른 탐색 전략보다 분기 커버리지를 높게 달성한 사례 분석

그림 4.1은 Taint 탐색 전략이 기존의 다른 탐색 전략보다 분기 커버리지를 더 높게 달성한 Prog3 소스코드 예시이다. 그림 4.1의 소스코드에서 DFS, rev-DFS, CFG, Random, Taint-bytes 탐색 전략들은 함수 f의 5번째 줄 if 분기 조건과 함수 target_g의 24번째 줄 else-if 분기 조건이 참이 되는 테스트 케이스를 생성하지 않지만 Taint는 그러한 테스트 케이스를 생성하여 커버리지가 더 높게 나왔다.

기존 탐색 전략들이 f의 5번째 줄 if 분기 조건이 참이 되는 테스트 케이스를 생성하지 못하는 이유는 5번째 줄 if 분기 조건이 거짓이 되는 수행 경로에서 10-13번째 줄에서 경로의 폭발적 증가에 빠지기 때문이다. 10-13번째 줄의 반복문은 20번 반복 실행하면서 문자열 비교를 하기 때문에 수행 경로 개수가 매우 많다(> 3^{20}). f의 5번째 줄 분기 조건이 참이 되는 테스트 케이스를 생성하지 못하기 때문에 덩달아 target_g의 24번째 줄 분기 조건이 참이 되는 테스트 케이스도 생성하지 못한다. target_g 함수 역시 반복문이 반복 실행하면서 문자열 비교를 여러 번 하기 때문에 수행 경로가 매우 많다.

Taint 탐색 전략은 f의 5번째 줄 분기 조건과 target_g의 24번째 줄 분기 조건이 참이 되는 테스트 케이스를 생성한 과정은 다음과 같다. 5번째 줄 분기 조건과 24번째 줄 분기 조건이 참이 되는 테스트 케이스를 생성하기 이전 심볼릭 경로 수식이 실행하는 함수는 f, target_g, SYM_strcmp 함수 3개이고, 커버리지를 높일 타겟 함수로 target_g가 선택되었다. 실행된 각 함수에 대한 target_g의 함수 간 데이터 의존도는 12, 0, 20이고, 12/32의 확률로 함수 f가 target_g에 영향력이 큰 함수로 선택되었다. 심볼릭 경로 수식이 가지고 있는 f의 분기 조건 중에서 무작위 선택된 5번째 줄의 분기 조건 $n->isSocket != True$를 부정하여 $n->isSocket == True$이 되고, 함수 h를 새롭게 실행하게 된다. 함수 h에서 value = “socket”이 되므로 target_g의 24번째 줄 분기 조건이 참이 되는 테스트 케이스를 생성하였다.

Taint-bytes 탐색 전략은 5번째 줄 분기 조건과 24번째 줄 분기 조건이 참이 되는 테스트 케이스를 생성하지 못하였다. 세 함수 f, target_g, SYM_strcmp에 대해서 각 함수가 target_g에 보내는 데이터 타입이 모두 integer 타입이기 때문에 각 함수에 대한 target_g의 함수 간 데이터 의존도는 48, 0, 80으로 함수 f가 target_g에 영향력이 큰 함수로 선택될 확률이 Taint 탐색 전략을 사용했을 때와 동일하다(=48/128). 함수 간 데이터 의존도 측면에서는 Taint 탐색 전략과 크게 다르지 않지만 Taint-bytes의 커버리지가 더 낮은 이유는 탐색 전략 알고리즘의 불확실성 때문이다. 심볼릭 경로 수식이 가지고 있는 f의 분기 조건 중에 부정할 분기 조건을 무작위 선택할 때, 5번째 줄의 분기 조건이 선택되지 않았다. Prog3 외에 다른 프로그램들에 대해서도 탐색 전략 알고리즘의 불확실성 때문에 달성 분기 커버리지에 차이가 나타났다.

제 5 장 관련 연구

5.1 Concolic 탐색 전략

5.1.1. 대표적인 concolic 탐색 전략

가장 일반적으로 많이 사용하는 탐색 전략은 DFS 탐색 전략과 DFS 탐색 전략이다[32]. DFS 탐색 전략은 한 경로를 가능한 가장 깊은 미 탐색 지점으로 확장한 후에 경로 끝에서부터 역추적하는 전략[44, 45]이고, BFS 탐색 전략은 모든 실행 경로를 동시에 확장하는 전략이다. DFS 탐색 전략은 메모리 사용량이 적지만, 대상 프로그램에 반복문과 재귀 호출을 포함하는 실행 경로가 있으면 실행 경로를 많이 탐색해도 분기 커버리지가 더 커지지 않는 문제점이 있다. 반대로 BFS 탐색 전략은 메모리 사용량과 테스팅 시간이 크지만 다양한 경로를 빠르게 탐색할 수 있다. 이론상 DFS와 BFS 탐색 전략 모두 모든 실행 경로를 탐색할 수 있지만, 실제 사용하는 프로그램들은 복잡도가 매우 크기 때문에 경로의 폭발적 증가 문제에 빠지기 쉽다[47, 30, 31].

5.1.2. 휴리스틱 기반의 concolic 탐색 전략

다양한 휴리스틱 기반의 concolic 탐색 전략 관련 연구가 존재한다. Xie et al[49]은 적합성 함수 기반의 경로 탐색 기술을 개발하였다. 이 기술은 실행 경로의 분기의 적합성 함수 값을 계산하여 다음 심볼릭 경로가 특정 분기를 커버하도록 유도한다. 적합성 함수를 사용한다는 점에서 탐색 기반 소프트웨어 테스팅(search-based software testing) 기술[48]과 유사하지만, concolic 테스팅과 결합했다는 점에 의의가 있다. Marinescu et al[40]은 여러 개의 소프트웨어 패치가 있을 때 패치가 일어난 코드 라인을 커버하도록 유도하는 심볼릭 실행 기술인 KATCH를 개발하였다. 이 기술은 주어진 테스트 케이스를 분석해서 가장 좋은 초기 테스트 입력값을 찾고, 여러 휴리스틱 기반의 탐색 전략을 사용해서 소프트웨어 패치들을 테스트한다. 본 논문의 접근 방식은 특정 분기를 커버하는데 초점을 맞춘 것이 아니라 전체적인 분기 커버리지를 높이는데 초점을 맞췄다.

Li et al [41]과 Seo et al[42]는 상대적으로 덜 실행한 경로들을 실행하도록 심볼릭 실행을 유도하는 기술을 개발하였다. 심볼릭 실행은 분기를 만날 때마다 fork해서 분기 조건이 참일 때의 심볼릭 상태와 거짓일 때의 심볼릭 상태를 만든다. 두 심볼릭 상태 중 어떤 상태를 커버할지 결정할 때, 각 심볼릭 상태의 subpath를 고려해서 가장 subpath를 덜 실행한 심볼릭 상태를 커버하도록 심볼릭 실행을 유도한다. KLEE[39]에서는 가장 가까운 커버하지 못한 구문부터 커버하도록 휴리스틱을 정의하였다. 본 논문은 언급한 논문들과는 다르게 함수 간 데이터 의존도를 고려해서 탐색 전략을 설계했다는 차이가 있다.

5.1.3. 여러 concolic 탐색 전략이 결합된 탐색 전략

큰 search space를 효과적으로 탐색하기 위해 여러 탐색 전략을 결합해서 사용하는 연구도 많이 존재한다. 하이브리드 concolic 테스팅[43]은 random 테스팅과 concolic 테스팅을 결합했다. 이 기술은 먼저 random 테스팅을 사용해서 빠르게 프로그램의 깊은 부분까지 탐색하는 테스트 케이스들을 많이 만든다. 그러다가 random 테스팅이 더 이상 커버리지를 높이지 못하는 상태에 도달하면, concolic 테스팅으로 교체해서 random 테스팅으로 커버하지 못한 경로를 탐색한다. 하지만, 논문의 저자들이 언급했듯이, 하이브리드 concolic 테스팅은 주기적으로 입력 값을 받는 반응 프로그램에 적합하지만, 본 논문에서 사용하는 탐색 전략은 고정된 크기의 초기 입력 값을 받는 프로그램에 적합하다. Garg et al. [33]는 피드백 기반의 유닛 테스트 생성 기술과 concolic 테스팅을 결합했다. 이 기술은 Randoop[34]와 유사하게 random 유닛 테스팅을 먼저 시작하고 유닛 테스팅이 일정 커버리지에 도달하면 concolic 시스템 테스팅으로 교체한다. 본 연구에서 제안한 concolic 탐색 전략을 Garg et al.의 concolic 시스템 테스팅에 적용할 수 있다.

5.2 테인트 분석 적용 연구

테인트 분석은 데이터의 흐름을 추적하는 기술이다. 관찰할 변수를 테인트 소스(taint source)로 지정하고 프로그램의 실행 경로를 따라 그 변수 값이 전파되어 최종적으로 테인트 싱크(taint sink)로 지정된 코드에 도달할 경우, 테인트 소스와 테인트 싱크 사이에 데이터 흐름을 추출하고 분석할 수 있다. 테인트 분석을 함수 간 데이터 의존도를 정의하는데 사용하거나 효과적인 concolic 탐색 전략을 개발하는데 사용한 연구는 없지만, 보안 관련 분야에서 매우 다양한 용도로 사용된다[10].

5.2.1 인젝션 공격에 취약한 코드 탐지

인젝션 공격(injection attack)에 취약한 코드를 탐지하는데 사용된다[5,7,9,26]. 인젝션 공격은 SQL 인젝션, 코드 인젝션, 스크립트 인젝션, 커맨드 인잭션, CRLF 인젝션, 사이트 간 스크립팅 공격과 같은 보안 위협의 큰 비중을 차지하는 공격으로, 코드에서 사용하는 데이터를 건드려서 보안 침해를 일으킬 가능성이 있는 함수에 유입되면 발생한다. HTTP 요청과 같이 외부로부터 유입된 입력 값을 테인트 소스로 지정하고, SQL 구문을 실행하는 함수나 시스템 호출처럼 보안 침해를 일으킬 가능성이 있는 코드 라인들을 테인트 싱크로 지정하면 효과적으로 인젝션 공격을 탐지할 수 있다.

5.2.2 데이터 유출 가능성 탐지

데이터 유출 가능성을 탐지하는데 유용하게 사용된다[27]. 대상 프로그램 코드에서 비밀번호 파일 등 민감한 정보를 테인트 소스로 지정하고, send나 write 함수처럼 데이터를 외부로 전송하는 함수들을 테인트 싱크로 지정해서 테인트 분석으로 테인트 소스와 테인트 싱크 간의 데이터 흐름이 존재하는 케이스를 탐지했을 경우, 대상 코드를 민감한 정보를 유출할 수 있는 코드로 간주한다. 이 기술을 휴대전화 등에 적용해서 위치 정보 등 민감한 정보가 네트워크를 통해 전송되는 일을 실시간으로 탐지하는 목적으로 활용할 수 있다[1,2].

5.2.3 악성 코드 분석

악성 코드의 동작을 분석하는데 사용하는 대표적인 기술인 reverse engineering은 동적 테인트 분석을 사용한다[3,4,6]. Ulrich et al[3]은 동적 테인트 분석 및 추가적인 네트워크 분석을 통해 악성 코드의 실행 트레이스를 추출하고 그 중 호출, 의존도, 네트워크 활동과 관련이 있는 실행 트레이스를 분석해서 악성 코드인지 아닌지 분류한다. Heng et al[7]은 동적 테인트 분석을 바탕으로 테인트 그래프를 생성하는 방법을 제시하였다. 이 방법은 기존의 분석들이 오염된 데이터가 전파되고 사용될 때 프로그램의 취약점을 공격할 수 있는지를 예측하는데 한정된 것에 반해서 테인트 그래프를 통해 데이터 흐름이 사용자의 민감한 정보를 훔치거나 이용하려는 행동이 아닌지 분석하고 악성 코드를 탐지하는데 이용된다.

5.2.4 의미상 연관된 코드 그룹화

프로그램의 의미상 연관된 코드만을 그룹화할 때 사용된다. 의미상 연관된 코드들을 그룹화할 때 사용되기도 한다. 연관된 코드를 그룹화하는 기술은 프로그램 특징화(program characterization), 즉 악성코드의 행위 시그니처를 생성하거나 코드 도용 판별을 위한 의미론적 특징점을 찾는 등의 분야에서 노이즈를 제거하는 용도로 사용된다[28,29].

5.2.5 테인트 분석의 성능 향상 연구

바이너리 코드 삽입 기술로 인한 동적 테인트 분석의 성능 오버헤드가 심하기 때문에, 더 효율적인 컴파일러 기반[51, 52], 하드웨어 기반[53,54]의 코드 삽입 기술을 이용한 동적 테인트 분석이 제시되었다. Walter et al [55]은 정적 분석과 결합한 동적 소프트웨어 기반 테인트 분석이 최소한의 오버헤드를 유발하기 때문에 실용적일 수 있었다.

제 6장 맺음말

6.1 요약

이 연구는 기존의 concolic 탐색 전략들(DFS, rev-DFS, CFG, random)은 대상 프로그램에 있는 함수들 간의 데이터 의존도를 고려하지 않고 방문할 분기를 결정해 분기 커버리지가 낮게 나오는 문제를 해결하고자 하였다. 이를 위해 함수 간 데이터 의존도를 활용한 새로운 concolic 탐색 전략 Taint를 설계하였다. Taint는 함수 g의 데이터 의존도가 높은 함수 f가 함수 g로 보낸 변수들의 값이 함수 g의 실행 경로를 결정한다는 가정에서 비롯된 휴리스틱 기반의 탐색 전략이다. 함수 f에 대한 함수 g의 데이터 의존도가 높고 타겟 함수 g에서 커버하지 못하는 분기 p가 존재할 경우, 함수 f에 있는 분기를 부정하는 경우도 고려해서 분기 p를 커버하도록 한다. 함수 간 데이터 의존도는 3개의 design choice들을 고려해서 정의하였고, 이를 측정하기 위해 동적 테인트 분석을 사용하여 함수 간 데이터 흐름을 측정하였다. Taint는 기존의 concolic 탐색 전략들을 사용했을 때보다 1.2%~7.1% 더 높은 분기 커버리지를 달성할 수 있다.

6.2 향후 연구

6.2.1 규모가 큰 프로그램, Event-driven 프로그램에 Taint 탐색 전략 적용

실험에서 사용한 프로그램의 규모가 평균 280.8 LOC로 실제 소프트웨어 분야에서 사용하는 프로그램의 규모보다 작다. 따라서, 향후 연구에는 Taint를 임베디드 C 프로그램이나 SIR 프로그램과 같이 규모가 큰 프로그램에 적용했을 때도 기존의 concolic 탐색 전략보다 높은 분기 커버리지를 달성하는지 분석하고 분기 커버리지를 더욱더 높이 달성하기 위해 Taint 알고리즘과 함수 간 데이터 의존도를 개선할 것이다. 또한, event-driven 프로그램에 대해서는 어떻게 테스팅 환경을 설정해서 함수 간 데이터 의존도를 정의하고 Taint 탐색 전략을 적용할지에 대해 고민해볼 것이다.

6.2.2 함수 간 데이터 의존도를 사용한 실제적인 유닛 컨텍스트 생성

기존의 자동화된 concolic 유닛 테스팅은 타겟 유닛의 부정확한 컨텍스트로 인해 실제 프로그램 실행 과정에서는 탐색할 수 없는 실행 경로를 탐색하여 많은 수의 거짓 경보를 생성하는 문제가 있다. 이를 해결하기 위해 타겟 유닛에 실제적인 유닛 컨텍스트(타겟 유닛과 “밀접하게 연관된” 함수들의 코드)를 합성해서 거짓 경보를 줄이는 연구가 있다[56]. 여기서 “밀접하게 연관된” 함수를 정의할 때 본 연구에서 사용한 함수 간 데이터 의존도를 개선하고 활용하여 concolic 유닛 테스팅의 거짓 경보를 더 줄일 수 있도록 할 것이다.

참 고 문 헌

[1] CABALLERO, Juan, et al. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007. p. 317-329.

[2] WONDRACEK, Gilbert, et al. Automatic Network Protocol Analysis. In: NDSS. 2008. p. 1-14.

[3] BAYER, Ulrich, et al. Scalable, behavior-based malware clustering. In: NDSS. 2009. p. 8-11.

[4] BAYER, Ulrich, et al. Dynamic analysis of malicious code. Journal in Computer Virology, 2006, 2.1: 67-77.

[5] BALZAROTTI, Davide, et al. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In: 2008 IEEE Symposium on Security and Privacy (sp 2008). IEEE, 2008. p. 387-401.

[6] YIN, Heng, et al. Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007. p. 116-127.

[7] SEKAR, R. An Efficient Black-box Technique for Defeating Web Application Attacks. In: NDSS. 2009.

[8] JUNG, Jaeyeon, et al. Sensitive data tracking using dynamic taint analysis. U.S. Patent No 8,893,280, 2014.

[9] SAXENA, Prateek, et al. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In: NDSS. 2010.

[10] SCHWARTZ, Edward J et al., All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE symposium on Security and privacy. IEEE, 2010. p. 317-331.

[11] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009), Introduction to Algorithms (3rd ed.), MIT Press and McGraw-Hill, ISBN 978-0-262-03384-8

[12] NEWSOME, James et al., Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In: NDSS. 2005. p. 3-4.

[13] CLAUSE, James; LI, Wanchun; ORSO, Alessandro. Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 international symposium on Software testing and analysis. ACM, 2007. p. 196-206.

[14] KIM, Moonzoo et al., Industrial application of concolic testing on embedded software: Case studies. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, 2012. p. 390-399.

[15] KIM, Yunho, et al. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012. p. 1143-1152.
[16] KIM, Yunho, et al. Automated unit testing of large industrial embedded software using concolic testing. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013. p. 519-528.
[17] KIM, Taeksu, et al. Concolic testing framework for industrial embedded software. In: 2014 21st Asia-Pacific Software Engineering Conference. IEEE, 2014. p. 7-10.
[18] KIM, Yunho et al. SCORE: a scalable concolic testing tool for reliable embedded software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011. p. 420-423.
[19] KIM, Yunho et al. Scalable distributed concolic testing: a case study on a flash storage platform. In: International Colloquium on Theoretical Aspects of Computing. Springer, Berlin, Heidelberg, 2010. p. 199-213.
[20] KIM, Yunho et al. Concolic testing on embedded software-case studies on mobile platform programs. In: European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE) Industrial Track. 2011. p. 30.
[21] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report EECS-2008-123, Berkeley University, 2008.
[22] Y. Kim, CROWN [Online]. Available: https://github.com/swtv-kaist/CROWN
[23] J. Burnim, CREST [Online]. Available: https://github.com/jburnim/crest
[24] CLARKE, Lori A. A program testing system. In: Proceedings of the 1976 annual conference. ACM, 1976. p. 488-491.
[25] KING, James C. Symbolic execution and program testing. Communications of the ACM, 1976, 19.7: 385-394.
[26] HUANG, Yao-Wen, et al. Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th international conference on World Wide Web. ACM, 2004. p. 40-52.
[27] ENCK, William, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS), 2014, 32.2: 5.
[28] ZHANG, Fangfang, et al. A first step towards algorithm plagiarism detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM, 2012. p. 111-121.
[29] JHI, Yoon-Chan, et al. Value-based program characterization and its application to software plagiarism detection. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011. p. 756-765.

[30] CHEN, Ting et al. State of the art: Dynamic symbolic execution for automated test generation. Future Generation Computer Systems, 2013, 29.7: 1758-1773.
[31] CADAR, Cristian et al., Symbolic execution for software testing: three decades later. Commun. ACM, 2013, 56.2: 82-90.
[32] BALDONI, Roberto et al. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 2018, 51.3: 50.
[33] GARG, Pranav, et al. Feedback-directed unit test generation for C/C++ using concolic execution. In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013. p. 132-141.
[34] PACHECO, Carlos et al., Randoop: feedback-directed random testing for Java. In: OOPSLA Companion. 2007. p. 815-816.
[35] CLAUSE, James et al., Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 international symposium on Software testing and analysis. ACM, 2007. p. 196-206.
[36] HUANG, Shih-Kun, et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In: 2012 IEEE Sixth International Conference on Software Security and Reliability. IEEE, 2012. p. 78-87.
[37] GODEFROID, Patrice, et al. Automated Whitebox Fuzz Testing. In: NDSS. 2008. p. 151-166.
[38] CHA, Sang Kil, et al. Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy. IEEE, 2012. p. 380-394.
[39] CADAR, Cristian, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: OSDI. 2008. p. 209-224.
[40] MARINESCU, Paul Dan et al., KATCH: high-coverage testing of software patches. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013. p. 235-245.
[41] LI, You et al. Steering symbolic execution to less traveled paths. In: ACM SigPlan Notices. ACM, 2013. p. 19-32.
[42] SEO, Hyunmin et al., How we get there: A context-guided search strategy in concolic testing. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014. p. 413-424.
[43] MAJUMDAR, Rupak et al., Hybrid concolic testing. In: 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007. p. 416-426.
[44] GODEFROID, Patrice et al., DART: directed automated random testing. In: ACM Sigplan Notices. ACM, 2005. p. 213-223.
[45] SEN, Koushik et al., CUTE: a concolic unit testing engine for C. In: ACM SIGSOFT Software Engineering Notes. ACM, 2005. p. 263-272.

[46] BURNIM, Jacob et al., Heuristics for scalable dynamic test generation. In: Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering. IEEE Computer Society, 2008. p. 443-446.
[47] ANAND, Saswat, et al. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 2013, 86.8: 1978-2001.
[48] MCMINN, Phil. Search-based software test data generation: a survey. Software testing, Verification and reliability, 2004, 14.2: 105-156.
[49] XIE, Tao et al. Fitness-guided path exploration in dynamic symbolic execution. In: 2009 IEEE/IFIP International Conference on Dependable Systems & Networks. IEEE, 2009. p. 359-368.
[50] WEYUKER, Elaine J. Translatability and decidability questions for restricted classes of program schemas. SIAM Journal on Computing, 1979, 8.4: 587-598.
[51] LAM, Lap Chung et al., A general dynamic information flow tracking framework for security applications. In: 2006 22nd Annual Computer Security Applications Conference (ACSAC'06). IEEE, 2006. p. 463-472.
[52] XU, Wei et al., Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In: USENIX Security Symposium. 2006. p. 121-136.
[53] CRANDALL, Jedidiah R., et al. Minos: Architectural support for software security through control data integrity. In: International Symposium on Microarchitecture. 2004.
[54] DALTON, Michael et al., Raksha: a flexible information flow architecture for software security. ACM SIGARCH Computer Architecture News, 2007, 35.2: 482-493.
[55] CHANG, Walter et al., Efficient and extensible security enforcement using dynamic data flow analysis. In: Proceedings of the 15th ACM conference on Computer and communications security. ACM, 2008. p. 39-50.
[56] KIM, Yunho; CHOI, Yunja; KIM, Moonzoo. Precise concolic unit testing of c programs using extended units and symbolic alarm filtering. In: Proceedings of the 40th International Conference on Software Engineering. ACM, 2018. p. 315-326.
[57] 박건우, 이주현, 송형곤, 조규태, 김윤호, 김문주. (2018). 국방 무기 체계 SW 품질 향상을 위한 Concolic 테스팅 기술. 한국정보과학회 학술발표논문집, (), 429-431.
[58] 박건우, 이주현, 송형곤, 조규태, 김윤호, 김문주. (2019). 국방 무기 체계 SW 품질 향상을 위해 Concolic 테스팅을 통한 테스트 자동 생성. 정보과학회논문지, 46(9), 926-933.

사 사

지난 2 년의 석사 과정 동안 이 논문을 완성할 수 있도록 도움을 주신 모든 분들께 이 지면을 빌어 감사의 말씀을 전하고자 합니다. 먼저 언제나 저를 믿고 격려해 주신 아버지, 어머니, 큰누나 내외, 작은 누나, 그리고 할머니께 감사드립니다. 아버지 어머니 두 분 모두 제가 힘들 때마다 직접 오셔서 아낌 없이 조언과 격려를 주신 덕분에 좌절하지 않고 지금까지 해올 수 있었습니다. 큰누나 내외와 작은 누나의 이해와 배려로 석사 과정을 무사히 지낼 수 있었습니다. 그리고 석사 과정을 잘 마칠 수 있도록 묵묵히 기도해주신 할머니께 감사드립니다.

2 년이라는 짧은 시간 동안 저에게 많은 가르침을 주신 김문주 교수님께 감사드립니다. 많이 부족했던 저를 잘 이끌어주시고, 때때로 흔들리는 저를 잘 바로잡으셨기에 이런 훌륭한 결실을 맺을 수 있게 되었습니다. 연구의 분야에서만 국한되는 것이 아닌, 앞으로도 제 인생에 도움이 될 많은 조언들을 주셔서 제가 크게 성장할 수 있었습니다. 제가 이 연구실에서 연구할 수 있도록 기회를 주신 김문주 교수님께 다시 한번 감사드립니다. 제 연구에 많은 도움을 주신 김윤호 연구조교수님께 감사드립니다. 제가 여러가지 어려움에 부딪힐 때마다 조언과 해결책을 아낌없이 주셨기에 여기까지 올 수 있었습니다. 그리고 실험하는 과정에서 주신 많은 도움에도 감사드립니다. 연구실에서 연구를 함께 한 김현우, 임현수 졸업생에게 감사드립니다. 연구실 식구인 Loc Duy Phan, 이아청, Yang Zidong 학생과 여준호, 이동희 연구원님, 그리고 지준왕 선교사님에게도 감사드립니다. 모두가 각자의 길에서 최고가 되기를 기원합니다.

대학원 생활 동안 항상 힘이 되어준 김유진, 장원준 학생에게도 감사드립니다. 언제나 알게 모르게 저를 지탱해주는 원동력이었고, 덕분에 즐겁게 석사 생활을 할 수 있었습니다. 마지막으로 여기 적지 못한 많은 분들께도 감사드립니다.

여러분의 도움으로 만든 이 작은 결실이 다른 사람에게 조금이나 보탬이 되기를 바랍니다.

약 력

이　　름: 박 건 우

생년월일: 1995년 12월 9일

주　　소: 서울특별시 강남구 압구정로 347, 26동 1006호 (압구정동, 한양아파트)

메일주소: kunwoo1209@gmail.com

학 력

2011. 3. - 2014. 2. 휘문고등학교 (3년 수료)

2014. 3. - 2018. 2. 연세대학교 컴퓨터과학과 (학사)

2018. 3. - 2020. 2. 한국과학기술원 전산학부 (석사)

경 력

2018. 12. 19. 한국정보과학회 2018 한국소프트웨어종합학술대회 최우수논문상

연 구 업 적

[1] 박건우, 이주현, 송형곤, 조규태, 김윤호, 김문주. (2018). 국방 무기 체계 SW 품질 향상을 위한 Concolic 테스팅 기술. 한국정보과학회 학술발표논문집, 429-431.

[2] 박건우, 이주현, 송형곤, 조규태, 김윤호, 김문주. (2019). 국방 무기 체계 SW 품질 향상을 위해 Concolic 테스팅을 통한 테스트 자동 생성. 정보과학회논문지, 46(9), 926-933.