ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 45, No. 4, pp. 342-347, 2018. 4
https://doi.org/10.5626/JOK.2018.45.4.342

효과적인 변이 분석을 위한 C 프로그램 변이 도구 비교: Proteum과 Milu를 사용한 사례 연구

(A Comparative Study of C Program Mutation Tools for Effective Mutation Analysis: A Case Study of Proteum and Milu)

김 윤 호 † 김 현 우 †† 양 웅 규 †† 김 문 주 †††
(Yunho Kim) (Hyunwoo Kim) (Woong-gyu Yang) (Moonzoo Kim)

요 약 변이 분석은 분석 대상 프로그램의 코드를 변형한 프로그램 변이를 실행해 프로그램 변형에 따른 실행 결과 변화를 분석하는 기법이다. 프로그램 변이 분석이 효과적이기 위해선 프로그램 변이 도구가 효과적인 프로그램 변이를 생성할 수 있어야 한다. 예를 들어 생성된 프로그램 변이가 분석 대상 프로그램과 의미가 동일한 동등 변이거나 다른 변이와 의미가 동일한 중복 변이인 경우 변이 분석을 통한 다양한 실행 결과 변화를 볼 수 없기 때문에 변이 분석에 효과적이지 않다. 본 논문에서는 C 프로그램의 효과적인 변이 분석을 위해 변이 도구 Proteum과 Milu를 대상으로 얼마나 효과적인 프로그램 변이를 생성하는지 비교하였다. 효율적으로 효과적인 프로그램 변이를 생성하기 위해 변이된 표현식의 정규형을 계산하고 정규형이 같은 변이를 중복 변이로 제거하였다. SIR 벤치마크의 4개 Linux/Unix 유틸리티 프로그램 프로그램을 대상으로 적용한 결과 Proteum과 Milu가 생성한 변이 중 각각 평균 48.7%, 46.4%가 유용한 변이였다.

키워드: 변이 분석, 프로그램 변형, 변이 테스트, C 프로그램, 소프트웨어 테스팅

Abstract Mutation analysis generates mutants of a target program by applying syntactic changes to the source code and analyzes the difference of execution results of the mutants from those of the original program. For effective mutation analysis, mutant generation tools should be able to generate effective program mutants. For example, a mutant that is semantically equivalent to the original program or another mutant is not an effective mutant, because it does not generate an execution result different from that of the original program or another existing mutant. This paper presents a comparative study of two mutant generation tools for C programs, Proteum and Milu. To generate effective mutants effectively, we generated a canonical form of mutated expressions and removed duplicated mutants that have the same canonical form as that of other mutants. We applied Proteum and Milu to four Linux/Unix utilities in the SIR benchmark and showed that 48.7% and 46.4% of mutants generated by Proteum and Milu were effective mutants on average, respectively.

Keywords: mutation analysis, program mutation, mutation testing, C program, software testing

· 본 연구는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원(NRF-2016R1A2B4008113)을 받아 수행하였음
· 이 논문은 제43회 동계학술발표회에서 '효과적인 변이 분석을 위한 C 프로그램 변이 도구 비교: Proteum과 Milu를 사용한 사례 연구'의 제목으로 발표된 논문을 확장한 것임

† 정 회 원 : KAIST 전산학부 박사후연구원
yunho.kim03@gmail.com
†† 비 회 원 : KAIST 전산학부
hyunooudy@gmail.com
diddndrb12@kaist.ac.kr
††† 종신회원 : KAIST 전산학부 교수(KAIST)
moonzoo@cs.kaist.ac.kr
(Corresponding author임)

논문접수 : 2017년 2월 28일
(Received 28 February 2017)
논문수정 : 2018년 1월 12일
(Revised 12 January 2018)
심사완료 : 2018년 1월 22일
(Accepted 22 January 2018)

정보과학회논문지 제45권 제4호(2018. 4)

1. 서 론

변이 분석(mutation analysis)[1]은 테스트 대상 프로그램의 코드를 다양하게 변형한 다수의 “프로그램 변이”(mutant, 이하 변이)를 생성한 후, 코드 변화에 따른 테스트 실행 결과 변화를 분석하는 소프트웨어 분석 기법이다. 변이 분석에서는 분석대상 프로그램 전반에서 다양한 변이를 생성하기 위해 여러 개의 “변형 연산자”(mutation operator)를 활용하는데, 이 때 각 변형 연산자는 분석대상 프로그램 코드의 특정 패턴마다 특정한 형태의 변형을 발생시킨 변이를 자동으로 생성한다.

변이 분석을 효과적으로 하기 위해선 분석 대상 프로그램과 다른 동작을 수행하는 다양한 변이를 생성하는 것이 중요하다. 예를 들어 생성된 프로그램 변이가 분석 대상 프로그램과 의미가 동일한 동등 변이거나 다른 변이와 의미가 동일한 중복 변이인 경우 변이 분석을 통한 다양한 실행 결과 변화를 볼 수 없기 때문에 변이 분석에 효과적이지 않다. 변이 분석을 수행하려면 생성된 변이 각각에 대해 주어진 테스트 케이스를 실행해야 하기 때문에 효과적이지 않은 변이가 다수 생성되면 변이 분석의 효과가 떨어질 뿐 아니라 변이 분석 시간이 길어지게 된다. 따라서 변이 분석을 수행할 때 효과적인 변이를 다수 생성할 수 있는 변이 도구의 선택이 중요하다.

본 논문에서는 현재 공개되어 사용할 수 있는 오픈 소스 C 프로그램 변이 도구 Proteum[2]과 Milu[3]를 비교하여 각 도구가 얼마나 효과적인 프로그램 변이를 생성하는지 비교하였다. Proteum과 Milu 모두 C 프로그램으로 작성된 소스 코드를 입력으로 받아 변이 프로그램을 생성하는 도구이다. 하지만 Proteum과 Milu가 생성할 수 있는 변이 프로그램의 종류가 서로 다르기 때문에 변이 분석의 효과성과 효율성이 달라지게 된다. 특히 변이 분석의 경우 다수의 프로그램 변이를 대상으로 테스트 케이스를 실행해야 하기 때문에 단순히 많은 수의 변이 프로그램을 생성하는 도구 보다 적은 수의 변이 프로그램을 생성하더라도 효과적인 변이 분석을 수행할 수 있는 도구를 선택해야 한다.

Proteum과 Milu가 생성하는 변이 프로그램의 효과를 비교하기 위해 SIR 벤치마크[4]의 grep과 sed 프로그램을 대상으로 비교 연구를 수행하였다. Proteum과 Milu가 지원하는 변형 연산자를 모두 적용하여 생성한 변이 프로그램을 대상으로 테스트를 수행하여 어느 도구에서 생성한 변이 프로그램이 더 효과적인지 비교하였다. 비교 결과 Proteum의 경우 전체 변이 프로그램 중 48.7%가 유용하였고 Milu는 46.4%가 유용한 변이 프로그램이었다.

논문 구성은 다음과 같다. 2장에서는 변이 분석 기법과 관련 연구에 대해서 소개한다. 3장에서는 효과적이지 못한 변이를 탐지하고 제거하는 방법을 설명한다. 4장에서는 변이 분석 비교를 위한 실험 환경을 설명하고 5장에서는 비교 실험 결과를 설명한다. 끝으로 6장에서는 본 논문의 결론 및 향후 연구 방향을 제시한다.

2. 변이 분석 기법 및 도구

2.1 변이 분석 기법

변이 분석 기법[1]은 원본 프로그램의 특정 요소(구문, 표현식, 연산자, 변수/상수 값 등)를 다른 형태로 변경한 후 프로그램 변이를 생성/실행하고 원본 프로그램과 생성된 프로그램 변이의 실행 결과 차이를 분석하는 기법이다. 프로그램 변이는 원본 프로그램의 특정 요소를 미리 정의된 패턴의 다른 요소로 치환하는 방식을 통해 생성되며 프로그램 변이 도구는 분석 대상 프로그램의 모든 구성 요소를 탐색하여 변이를 생성한다. 이 때 변경할 프로그램 구성 요소의 패턴과 해당 패턴을 어떻게 변경하여 변이를 생성할 지 정의한 것을 변형 연산자(mutation operator)라고 하며 변형 연산자를 적용하여 변형된 프로그램 위치를 변경점(mutation point)이라고 한다. 프로그램 변이가 생성되면 원본 프로그램과 프로그램 변이의 실행 결과 차이를 확인하기 위해 주어진 테스트 케이스를 실행한다. 주어진 테스트 케이스를 원본 프로그램과 변이 프로그램에서 실행했을 때 하나 이상의 테스트 케이스에서 원본 프로그램과 프로그램 변이의 실행 결과가 다른 경우 해당 프로그램 변이를 죽은 변이(killed mutant)라고 하며 주어진 테스트 케이스를 실행했을 때 원본 프로그램과 프로그램 변이의 실행 결과가 똑같다면 해당 변이는 살아있는 변이(live mutant)라고 한다.

2.2 효과적인 프로그램 변이

변이 분석의 효과성은 원본 프로그램으로부터 생성되는 프로그램 변이가 얼마나 효과적인지에 따라 결정된다. 생성된 프로그램 변이가 원본 프로그램과 다른 동작을 수행하고 프로그램 변이 사이의 동작이 다를수록 변이 분석에 효과적이다. 반면 생성된 프로그램 변이가 원본 프로그램과 동작이 동일한 동등 변이(equivalent mutant)거나 다른 변이 프로그램과 동작이 같은 중복 변이(redundant mutant)인 경우 변이 분석의 효과가 떨어진다. 뿐만 아니라 효과적이지 않은 변이를 대상으로 테스트를 수행하느라 테스트 비용이 증가하기 때문에 효과적이지 않은 변이를 적게 생성하는 것이 중요하다.

변이 분석에 효과적이지 못한 변이의 종류는 다음과 같다.

- 죽이기 쉬운 변이 프로그램(Easy-to-kill mutant): 해당 변경점을 실행한 테스트 케이스 중 다수(예: 전체 테스트 중 2/3 이상)의 테스트 케이스가 해당 변이 프로그램을 죽일 수 있는 경우 변경점에 대한 실행 유무만 판단하는 커버리지 조건과 효과가 같으므로 변이 프로그램 실행을 통한 동작 변화를 분석하는 변이 분석에 효과적이지 않다.
- 동등 변이(equivalent mutant): 프로그램 구문 변경을 통해 생성된 변이 프로그램이 원본 프로그램과 의미적으로 동일한 경우이다. 이 경우 모든 입력 값에 대해서 변이 프로그램과 원본 프로그램의 실행 결과가 동일하기 때문에 변이 분석에 효과적이지 않다.
- 중복 변이(redundant mutant): 서로 다른 두 프로그램 변이가 모든 입력 값에 대해 동일한 실행 결과를 갖는 변이이다. 중복 변이의 경우 변이에 따른 동작 변화가 서로 갖기 때문에 변이 분석에 효과적이지 않다.

Willem Visser[5]는 죽이기 쉬운 변이 프로그램이 생성되는 조건을 고려할 때 기존처럼 변형 연산자만 고려하는 것이 아니라 변경점을 고려해야 함을 보였다. 뿐만 아니라 실행 결과 차이를 확인하기 위한 비교 방법이 얼마나 정확한지 여부가 프로그램 변이를 죽이기 쉽게 하는지 결정하는데 얼마나 영향을 미칠 수 있는지 실험적으로 보였다. Schuler와 Zeller[6,7]는 동등 변이를 찾기 위해 테스트 커버리지 기반 분석 방법을 사용하고 약 75%의 확률로 동등 변이를 구분하였다. Jia와 Harman[8]은 프로그램 변이를 생성하기 위해 변형 연산자를 여러 번 적용한 고차 프로그램 변이 생성(higher-order program mutation)[9-11]을 적용하여 동등 변이를 줄이고자 하였다. 기존 연구들과 비교하여 본 논문에서 사용한 중복 변이 제거 방법은 복잡한 프로그램의 의미 분석을 사용한 대신 구문 분석을 사용하여 수 시간 안에 수 십 만 개의 프로그램 변이를 분석할 수 있으며 거짓 양성(중복 변이가 아닌 변이를 중복 변이라고 보고) 결과 없이 중복 변이를 판별해낸다. Baldwin과 Sayward[12]는 컴파일러 최적화 기법을 적용해서 동등 변이를 찾는 방법을 제안하였다. 프로그램 최적화 기법을 적용해서 불필요한 구문 제거 및 정규화를 수행한 후에 원본 프로그램과 변이를 비교해서 동등 변이인지 찾는 방법이다. Offutt과 Craft[13]의 사례 연구 결과 프로그램 최적화 기법을 적용해서 전체 변이의 약 10%가 동등 변이임을 확인할 수 있었다. Offutt과 Pan[14은 제약 조건 해결 기법 (constraint solving)을 사용한 동등 변이 탐지 방법을 제안하였다. 변이 프로그램의 실행 경로 조건과 원본 프로그램의 실행 경로 조건을 분석하여 서로 다른 실행 결과를 갖는 입력을 생성할 수 있는 제약 조건을 만들고 해당 제약 조건이 만족하지 못한 경우를 동등 변이라고 하였다. Voas와 McGraw[15]는 프로그램 슬라이싱 기법을 처음 도입하여 동등 변이를 탐지하였다. 그 후 Harman et al.[16]은 프로그램 슬라이싱 기법을 발전시켜 의존성 분석을 통한 동등 변이 탐지 기법을 제안하였다.

2.3 C 프로그램 변이 분석 도구

본 논문에서는 현재 오픈 소스로 공개되어 활용할 수 있는 C 프로그램 변이 분석 도구인 Proteum과 Milu를 선택하여 비교하였다.

2.3.1 Proteum

Proteum[2]은 브라질 University of São Paulo 대학의 Marcio Eduardo Delamaro와 JoseCarlos Maldonado가 1996년 처음 개발하였으며 2001년 최신 버전인 Proteum 2.0 버전이 발표되었다. Proteum은 Agrawal 외 8인이 정의한 C 프로그램 변형 연산자 중 75개의 변형 연산자를 지원하여 가장 많은 종류의 변형 연산자를 지원한다.

2.3.2 Milu

Milu[3]는 University of College London의 Yue Jia가 개발한 C 프로그램 변이 분석 도구이다. 2008년 처음 개발되었으며 현재 3.2 버전이 가장 최신 버전이다. Milu는 23개의 변형 연산자를 지원한다.

3. 중복 변이 제거 기법

중복 변이를 제거하기 위해 프로그램 구문 분석을 사용하여 변이된 표현식 및 변이 구문의 정형화된 형태를 계산하고 비교하여 중복 변이 및 동등 변이를 제거한다. 중복 변이 및 동등 변이를 제거하는 방법은 크게 3단계로 나뉜다.

3.1 상수 연산 정리

1단계 상수 연산 정리에서는 상수와 상수 사이의 연산을 미리 수행하고 그 결과값으로 정형화된 형태를 나타내는 것이다. 예를 들어 a+(2*4)와 같은 표현식이 있다면 a+8로 정리한다.

3.2 항등원 정리

2단계 항등원 정리에서는 상수와 변수 사이의 연산 중 상수가 항등원의 역할을 하는 경우 항등원을 제거하는 단계이다. 예를 들어 덧셈 연산의 항등원은 0이기 때문에 a+0을 a로 정리하고 a && 3과 같은 경우 3은 C 프로그램에서 항상 참으로 평가하기 때문에 a로 정리한다.

3.3 항진 조건문 정리

3단계 항진 조건문 정리에서는 항상 참인 조건을 갖는 조건문을 정리하여 1로 표현한다. C99 표준에서 _Bool 타입이 새로 신설되었으나 아직 대부분의 C 프로그램에서는 정수 타입으로 참/거짓을 표현한다. C 프로그램에서 0은 거짓, 0 이외의 모든 정수는 참을 나타내므로 만

약 변이된 표현식이 조건문으로 사용되고 정형화된 형태가 0이 아닌 정수로 표현되는 경우 참 값을 나타내는 1로 동일하게 변경하여 중복을 제거한다.

그림 1은 중복 변이를 생성할 원본 코드 예제이다. 그림 2는 효과적이지 않은 변이 제거 기법으로 탐지 및 제거 가능한 중복 변이 예제이다. 원본 코드 5라인의 표현식 1 * 1이 첫번째 변이 프로그램에서는 1 << 1 표현식으로 변이되고 두번째 변이 프로그램에서는 1+1 표현식으로 변이되었다. 하지만 1<<1과 1+1은 모두 C 프로그램에서 2라는 동일한 연산 결과를 나타내기 때문에 두 변이의 변이된 표현식은 같은 정형화된 형태를 갖게 되고 결과적으로 항상 같은 결과를 보여주는 중복 변이가 된다. 그림 3과 그림 4는 첫 번째와 두 번째 변이의 바이너리 파일을 생성한 뒤 동일한 체크섬을 갖는지 보여주고 있다. 그림 3에서는 원본 코드(original.c), 첫 번째 변이(mutant1.c), 두 번째 변이(mutant2.c)를 컴파일하여 각각 바이너리 파일 original, mutant1, mutant2를 생성하는 과정이다. 체크섬 값은 파일명에 영향을 받으므로, 같은 파일명(code.c)으로 변환한 후에 컴파일을 한다. 그림 4는 생성 된 바이너리 파일의 체크섬을 출력한 것이다. 원본 코드의 바이너리만 다른 체크섬을 갖고, 첫 번째 변이 코드와 두 번째 변이 코드는 중복 변이이므로 같은 바이너리 체크섬을 갖는 것을 확인할 수 있다.

본 논문에서 제안하는 중복 변이 제거 기법은 복잡한 C 프로그램의 의미 분석없이 문법 분석만으로 탐지할 수 있고 다양한 변이 연산자에 공통적으로 생성할 수 있는 중복 변이만을 대상으로 하였다. C 프로그램 의미

```
original.c
1    #include <stdio.h>
2    int main() {
3     int a;
4     scanf("%d", &a);
5     if ( a + (1 * 1) == 0)
6     printf("reach!");
7     return 0;
8    }
```

그림 1 중복 변이 예제 - 원본 코드

Fig. 1 An example of duplicate mutant - Original code

```
//Mutant1 (mutant1.c)
1     if ( a + (1 << 1) == 0)
//Mutant2 (mutant2.c)
1     if ( a + (1 + 1) == 0)
```

그림 2 중복 변이 예제 - 변이 코드

Fig. 2 An example of duplicate mutant - Mutated code

표 1 변이 분석 대상 프로그램 정보

Table 1 Information on target programs for mutation analysis

Target Program	LoC	#TCs	Branch Coverage (%)	Line Coverage (%)
grep-2.0	5956	132	46.0	65.4
sed-1.17	4085	73	34.8	62.4
gzip-1.2	6358	213	58.3	75.0
flex-2.4.3	11784	567	80.3	86.7
Average	7045.8	246.3	54.9	72.3

```
:~/Example$ cp original.c code.c && gcc code.c -o original
:~/Example$ cp mutant1.c code.c && gcc code.c -o mutant1
:~/Example$ cp mutant2.c code.c && gcc code.c -o mutant2
```

그림 3 바이너리 파일 생성 과정

Fig. 3 Binary file generation from mutants

```
hyunoo@checker1:~/Example$ md5sum original mutant1 mutant2
c0c2caf5bab41432e30c106b31da42e0  original
b9549a95558436eb762fe8ff88bf9280  mutant1
b9549a95558436eb762fe8ff88bf9280  mutant2
```

그림 4 바이너리 파일 체크섬 출력

Fig. 4 Checksum outputs of binary files

분석은 복잡한 포인터 연산 등으로 인해 부정확한 분석 결과가 나올 수 있으며 이로 인해 중복 변이가 아닌 변이를 중복 변이로 간주하거나 중복 변이를 탐지하지 못할 수 있다. 또한 수 많은 변이를 대상으로 C 프로그램 의미 분석을 수행할 경우 많은 시간이 소요되기 때문에 빠르게 수행할 수 있는 문법 분석만 사용하여 중복 변이를 탐지하였다. 향후 변이 연산의 종류에 따른 중복 변이 생성 가능성을 분석하고 이를 기반으로 각 변이 연산 종류에 맞는 중복 변이 탐지 기법으로 발전시킬 수 있다.

4. 변이 테스트 도구 비교 실험 환경

Proteum과 Milu가 효과적인 프로그램 변이를 생성하는지 비교하기 위해 SIR 벤치마크[4]의 grep과 sed 프로그램을 대상으로 변이 분석을 수행하였다. 표 1은 프로그램의 크기와 사용한 테스트 케이스 수 및 테스트 커버리지를 나타낸다. 사용한 Proteum과 Milu 버전은 각각 최신 버전인 Proteum 2.0과 Milu 3.2 버전을 사용하였으며 Proteum의 75개 변형 연산자와 Milu의 23개 변형 연산자를 사용하여 변이 프로그램을 생성하였다. 실험은 Intel Core i5 4670K (3.4GHz) CPU와 16GB 메모리를 갖고 Ubuntu Linux 64bit 14.04 LTS 버전을 사용하는 시스템에서 수행하였다.

생성된 변이 프로그램의 효과성을 확인하기 위해 생성된 변이 프로그램 중 2.2절에서 설명한 효과적이지 못

표 2 변이 분석 결과[1)]
Table 2 Mutation analysis results

Target Programs	Mutants generated by Proteum						Mutants generated by Milu					
	Non-duplicate				Duplicate	Total	Non-duplicate				Duplicate	Total
	Killed		Alive				Killed		Alive			
	Easy-to-kill	Hard-to-kill	Mutation point reached	Mutation point not reached			Easy-to-kill	Hard-to-kill	Mutation point reached	Mutation point not reached		
grep	33228 (9.9%)	43046 (12.8%)	66454 (19.7%)	83382 (24.8%)	110672 (32.9%)	336782 (100.0%)	4730 (12.0%)	3749 (9.6%)	10439 (26.6%)	7011 (17.9%)	13327 (33.9%)	39256 (100.0%)
sed	21141 (8.3%)	8768 (3.4%)	44052 (17.3%)	77765 (30.5%)	102944 (40.4%)	254670 (100.0%)	3608 (14.6%)	935 (3.8%)	4866 (19.6%)	7732 (31.2%)	7640 (30.8%)	24781 (100.0%)
gzip	79674 (46.1%)	11593 (6.7%)	35210 (20.4%)	26120 (15.1%)	20297 (11.7%)	172889 (100.0%)	6613 (35.3%)	1139 (6.1%)	2677 (14.3%)	2855 (15.2%)	5473 (29.2%)	18757 (100.0%)
flex	77677 (48.4%)	17053 (10.6%)	18335 (11.4%)	19103 (11.9%)	28453 (17.7%)	160621 (100.0%)	9425 (39.8%)	2172 (9.2%)	3560 (15.0%)	2315 (9.8%)	6184 (26.1%)	23656 (100.0%)
Average	52930.0 (22.9%)	20115.0 (8.7%)	41012.75 (17.7%)	51592.5 (22.3%)	65591.5 (28.4%)	231240.5 (100.0%)	6094.0 (22.9%)	1998.8 (7.5%)	5385.5 (20.2%)	4978.3 (18.7%)	8156.0 (30.6%)	26612.5 (100.0%)

한 변이 프로그램의 비율을 비교하였다. 효과적이지 못한 변이 프로그램 중 죽이기 쉬운 변이 프로그램은 해당 변이의 변경점을 실행한 테스트 케이스 중 2/3 이상이 죽인 변이 프로그램으로 정의하였으며 중복 변이는 3장에서 설명한 중복 변이 제거 기법과 컴파일 후 생성된 바이너리의 체크섬 비교를 사용하여 측정하였다. 동등 변이의 경우 생성된 각각의 변이와 원본 프로그램의 의미 구조를 직접 분석해야 하기 때문에 비교하지 못했다.

5. 변이 테스트 도구 비교 실험 결과

표 2는 변이 분석 결과를 나타낸다. Proteum과 Milu는 평균적으로 231450.5개와 266612.5개의 변이를 생성하였다. Proteum이 지원 연산자의 수가 더 많기 때문에 Milu 생성 변이 수의 8.7 배의 변이를 생성할 수 있었다. Proteum이 생성된 변이의 평균 28.4%의 변이가 중복 변이고 22.9%의 변이가 죽이기 쉬운 변이이다. Milu의 경우 30.6%의 변이가 중복 변이고 22.9%의 변이가 죽이기 쉬운 변이였다. 즉 Proteum이 생성한 변이 중에서 48.7%의 변이가 효과적인 변이고 51.3%가 효과적이지 않은 변이였으며, Milu의 경우 46.4%가 효과적인 변이, 53.6%가 효과적이지 않은 변이였다.

표 3은 중복 변이를 제거하는데 사용된 시간을 나타낸다. 바이너리 체크섬만 사용할 경우 평균 2972.5초의 시간이 소요되고 3장에서 제안하는 분석 기법과 바이너리 체크섬 기법을 함께 사용할 경우 866.6초의 시간이 소요되어 약 3.43배 더 빠르게 중복 변이를 탐지할 수 있었다. 바이너리 체크섬의 경우 컴파일을 위해 전체 프로그램을 분석해야 하고 체크섬을 계산하기 위해 바이너리 파일을 다시 읽고 체크섬을 계산해야 하지만 3장

표 3 중복 변이 제거 시간(초)
Table 3 Time for removing duplicate mutants (sec)

Target Programs	Binary checksum only	Static analysis+ Binary checksum
grep	6780.5	1626.4
sed	3921.9	943.0
gzip	2115.1	411.7
flex	2880.5	485.2
Average	2972.5	866.6

에서 제안하는 기법은 실제 프로그램 변이가 발생한 부분의 표현식만 사용하여 비교 가능하기 때문에 중복 변이를 탐지하기 위한 시간이 더 적게 소요되었다.

6. 결론 및 향후 연구

본 논문에서는 C 프로그램의 변이 도구 Proteum과 Milu를 대상으로 효과적인 변이가 얼마나 생성되는지 비교 분석 연구를 수행하였다. SIR 벤치마크의 grep과 sed 프로그램을 대상으로 변이 분석을 수행한 결과 Proteum에서 생성된 변이 중 48.7%의 변이가 효과적인 변이였고 Milu에서 생성된 변이 중 46.4%의 변이가 효과적인 변이로 두 도구가 비슷한 수준의 효과적인 변이를 생성하는 것을 확인하였다. 또한 새로 제안하는 변이된 표현식의 정형화된 형태 비교 방법이 기존에 사용한 바이너리 체크섬 기반의 중복 변이 제거 기법보다 더 빠르게 중복 변이를 찾을 수 있음을 보였다. 향후 연구로는 살아있는 변이 중 동등 변이가 얼마나 존재하는지

1) 2016 한국 정보과학회 동계학술발표회에 발표된 논문의 실험 결과 중 Milu의 중복 변이 수가 실험 오류로 잘못 측정되었으며 해당 오류를 수정한 결과를 사용함

비교하는 연구를 수행할 예정이다. 동등 변이를 정확하게 알아내는 것은 정지 결정 문제(halting problem)이기 때문에 휴리스틱 기법을 사용하여 동등 변이를 탐지하고 비교하고자 한다.

References

[1] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *Proc. of IEEE Transactions on Software Engineering*, Vol. 37, No. 5, pp. 649-678, Jun. 2010.

[2] J. C. Maldonado, M. E. Delamaro, S.C.P.F. Fabbri, A.D.S. Simão, T. Sugeta, A.M.R. Vincenzi, P.C. Masiero, "Proteum: A family of tools to support specification and program testing based on mutation," *Mutation testing for the new century*, Vol. 24, pp. 113-116, Springer, Boston, MA, 2001.

[3] Y. Jia and M. Harman, "A Customizable, MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language," *Proc. of Testing: Academic & Industrial Conference - Practice and Research Techniques*, pp. 94-98, Aug. 2008.

[4] H. Do, S. Elbaum, G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Journal of Empirical Software Engineering*, Vol. 10, No. 4, pp. 405-435, Oct. 2005.

[5] W. Visser, "What makes Killing a Mutant Hard," *Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 39-44, Aug. 2016.

[6] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," *Proc. of Third International Conference on Software Testing, Verification and Validation*, pp. 45-54, Apr. 2010.

[7] D. Schuler and A. Zeller, "Covering and Uncovering Equivalent Mutants," *Journal of Software Testing, Verification and Reliability*, Vol. 23, No. 5, pp. 353-374, Aug. 2013.

[8] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Journal of Information and Software Technology*, Vol. 51, No. 10, pp. 1379-1393, Oct. 2009.

[9] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating Mutation Testing Alternatives: A Collateral Experiment," *Proc. of the 17th Asia Pacific Software Engineering Conference*, pp. 300-309, Nov. 2010.

[10] A.J. Offutt, "Investigations of the Software Testing Coupling Effect," *Journal of ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 1, pp. 5-20, Jan. 1992.

[11] M. Papadakis and N. Malevris, "An Empirical Evaluation of the First and Second Order Mutation Testing Strategies," *Proc. of Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 90-99, Apr. 2010.

[12] D. Baldwin and F. Sayward, "Heuristics for Determining Equivalence of Program Mutations," *Research Report 276*, Yale Univ. Apr. 1979.

[13] J. Offutt and W. Craft, "Using Compiler Optimization Techniques to Detect Equivalent Mutants," *Journal of Software Testing, Verification and Reliability*, Vol. 4, No. 3, 1994.

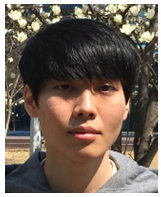
[14] J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and infeasible Paths," *Journal of Software Testing, Verification and Reliability*, Vol. 4, No. 3, pp. 131-154, 1997.

[15] J. Voas and G. McGraw, "Software Fault Injection: Inoculating Programs against Errors," John Wiley & Sons, New York, 1997.

[16] M. Harman, R. Hierons, and S. Danicic, "The Relationship between Program Dependence and Mutation Analysis," *Mutation testing for the new century*, Vol. 24, pp. 5-13, Springer, Boston, MA, 2001.

김 윤 호

2007년 KAIST 전산학과 학사. 2007년~2009년 KAIST 전산학과 석사. 2009년~2017년 KAIST 전산학부 박사. 관심분야는 자동화된 Concolic 유닛 테스팅, 변이 테스팅, 자동 오류 위치 추정, 정형검증



김 현 우

2016년 서강대학교 컴퓨터공학과 학사 2016년~현재 KAIST 전산학부 석사 과정. 관심분야는 자동화된 Concolic 유닛 테스팅, 변이 테스팅

양 웅 규

2016년 KAIST 전산학부 학사. 2018년 KAIST 전산학부 석사. 2018년~현재 TmaxData 연구원. 관심분야는 자동화된 Concolic 유닛 테스팅, 변이 테스팅

김 문 주

1995년 KAIST 전산학과 학사. 2001년 Univ. of Pennsylvania 박사. 2002년~2004년 SECUi.COM 차장. 2004년~2006년 POSTECH 연구원. 2006년~2012년 KAIST 전산학과 조교수. 2012년~현재 KAIST 전산학부 부교수. 관심분야는 Concolic 테스팅, 자동 오류 위치 추정, Concurrency 테스팅, 변이 테스팅, 정형 검증, 내장형 소프트웨어