C 언어 테스트 입력 생성에 대한 CAVM, Austin, CodeScroll 의 비교 및 분석

유병현 O,1, 김준휘 1, 권민혁 2, Phil McMinn3, 유신 1
한국과학기술원 1, 슈어소프트테크(주)2, University of Sheffield3
{byou, junhwi.kim23, shin.yoo}@kaist.ac.kr1, minhyuk@suresofttech.com2,
p.mcminn@sheffield.ac.uk3

Evaluation of CAVM, Austin, and CodeScroll for Test Data Generation for C

Byeonghyeon YouO,1, Junhwi Kim1, Minhyuk Kwon2, Phil McMinn3, Shin Yoo1
Korea Advanced Institute of Science and Technology1, Suresoft Technologies Inc. 2,
University of Sheffield3

요 약

본 연구는 새로운 검색 기반 C 언어 테스트 데이터 생성 도구인 CAVM 을 제시한다. CAVM 은 상용화된 테스트 데이터 도구인 CodeScroll 을 향상하기 위해서 개발되었다. CodeScroll 은 정적 분석과 입력 파티셔닝 기법을 사용하여 테스트 데이터를 생성한다. 현재의 최첨단 C 언어용 자동 테스트 데이터 생성 도구인 Austin 과 다르게 CAVM 은 순수하게 검색 기반 기법 만을 사용하여 동적 자료 구조 데이터를 생성한다. 본 연구는 CAVM 을 49 개의 실험 대상 함수를 이용해 Austin 및 CodeScroll 과 비교하였다. 대상 함수는 소규모의 안티 패턴 케이스 및 실제 프로덕트 레벨의 오픈 소스 코드와 상용 코드로부터 수집하였다. 비교 결과 CAVM 은 CodeScroll 과 Austin 이 커버하지 못하는 분기들을 커버할 수 있었으며, 20 개의 대상 함수에서 가장 높은 분기 커버리지를 달성하였다.

1. 서 론

검색 기반 자동 테스트 데이터 생성은 십 여년 이상 연구자들의 관심을 받았으며 [4,12,16,17,19,21,25] 프로그램 자동 수정 [2,7,26]이나 자동 기능 이식 및 향상 [2, 10], 그리고 결함 위치 추정 [14,27] 등의 자동 분석 기술과 밀접하게 연결되어 연구되고 있다. 하지만 연구의 결과가 본래의 목적을 달성하기 위해 산업계에서 널리 쓰이고 있다고 보기는 힘들다. 또한 C 언어가 여전히 두 번째로 많이 사용되는 언어임에도 불구하고 [1], 기호 분석에 기반한 도구들 [3,24]과 달리 검색 기반 자동 테스트 데이터 생성 기법의 최근 연구는 널리 사용되는 다른 언어들을 제외한 채 Java [4-6]로 작성된 객체 지향 코드의 테스트 데이터 생성에 집중되었다.

본 연구는 C 프로그램의 검색 기반 테스트 데이터 생성을 위해 해결해야 할 기술적 과제, 특히 동적 데이터 구조로 구성된 입력의 생성에 촛점을 맞추었다. 포인터 입력으로 C 함수를 테스트하려면 올바른 “형태(shape)”의 데이터 구조를 찾아야한다. 일반적으로 Java 프로그램의 경우 메소드를 테스트하는 데 필요한 객체의 구조는 테스트

중인 코드베이스에 존재하는 생성자와 메소드 호출을 사용하여 생성 할 수 있기 때문에 이러한 문제가 존재하지 않는다. 반면 C 언어의 경우 포인터 조작을 통해 생성한 다양한 동적 메모리 형태를 테스트 데이터로 사용할 수 있으므로, C 언어를 위한 상용화 단계의 자동 테스트 데이터 생성을 위해서는 해당 문제를 해결하는 것이 매우 중요하다.

현존하는 C 언어로 작성된 함수에 대한 테스트 입력 생성도구로는 상용 툴인 CodeScroll 과 검색 기반 테스트 데이터 생성 도구인 Austin[17] 등이 있다. CodeScroll 은 동적 데이터 구조로 구성된 입력을 생성하는 간단하면서도 효과적인 휴리스틱을 사용한다. 이와 대조적으로 Austin 은 가벼운 기호 분석을 수행하여 동적 데이터 구조의 형태를 생성한 다음, AVM (Alternating Variable Method) [20]을 사용하여 구체적인 데이터를 채운다. 현재 C 언어를 대상으로 하는 최신 검색 기반 테스트 데이터 생성 도구는 Austin 이지만 더 이상 활발히 유지 관리되지 않고있다 [18].

본 연구에서는 C 에 대한 새로운 검색 기반 테스트 데이터 생성 도구 인 CAVM (“ka-boom”이라고 발음)을 제시한다. CAVM 또한 AVM 을 기반으로 하지만 Austin 과는 다르게 동적 데이터 구조의 형태를 순수히 검색 기반 방법으로 생성한다. CAVM 은 동적 데이터 구조의 적절한 형태를 키워가면서 동시에 데이터를 채우는 검색을 함께 진행한다. 또한 코드 재작성을 통해 `strcmp` 라이브러리 함수를 위한 문자열 테스트 데이터, 즉 `char` 배열 생성을 지원한다.

본 연구에서는 동적 데이터 구조를 포함하는 C 코드의 테스트 데이터 생성에 대하여 CAVM 과 CodeScroll 및 Austin 의 유효성을 비교한다. 실증적 비교를 위해 기존에 CodeScroll 이 해결하지 못하는 안티 패턴 사례 뿐만 아니라 오픈 소스 및 상용 코드를 대상으로 실험을 진행하였다. 실험결과를 통해 CAVM 에 구현된 새로운 알고리즘이 CodeScroll 이나 Austin 이 해결하지 못하는 분기를 만족할 수 있음을 확인 할 수 있다.

본 연구의 기술적인 기여 내용은 다음과 같다.

1. 동적 데이터 구조로 구성된 테스트 입력 생성을 위한 새로운 검색 기반 알고리즘 및 구현, CAVM (3 장)

2. 작은 안티 패턴에서부터 상업용 C++ 프론트엔드 모듈까지 49 가지 C 함수를 대상으로 한 실험을 통해 CAVM, CodeScroll 및 Austin 의 동적 데이터 구조를 포함한 테스트 데이터 생성 능력을 비교 (4 장)

3. 기존의 최첨단 도구인 Austin 이나 상용 도구 CodeScroll 이 해결하지 못하지만 CAVM 을 통해 해결할 수 있는 분기 유형에 대한 분석 (5 장).

먼저 CodeScroll 및 Austin 에 대한 설명을 통해 본 연구의 분석을 시작하고자 한다.

2. 배 경

이 장에서는 C 함수에 대한 단위 테스트 데이터를 생성하는 두 가지 도구를 소개한다. 첫째로, CodeScroll 은 정적 분석을 통해 입력을 도출하는 상용 도구이다. 두 번째 도구인 Austin [17] 은 숫자 입력 값을 찾기 위한 동적 검색과 동적 데이터 구조의 “형태” 정보를 찾기 위한 간략화된 기호 실행(Symbolic Execution)을 결합하여 사용한다. Austin 은 현존하는 C 코드에 대한 가장 최첨단의 검색 기반 테스트 데이터 생성 기술이다.

```
1  void foo() { return 10; }
2
3  void testMe(int a, int b, int c) {
4    if(a < 42)
5      if(b == c)
6        if(c == foo())
7          // target
8  }
```

(a) 숫자 자료형 입력 예제

```
1  typedef struct _data {
2    int* internal;
3    int a, b;
4    struct _data* next;
5  } Data;
6
7  void testMe(Data *d) {
8    if (d != NULL) {
9      Data *d_in;
10     d_in = d->next;
11     if (d_in != NULL)
12       if (d_in->a == 2)
13         // target
14   }
15 }
```

(b) 포인터 입력 예제

그림 1 테스트 데이터 자동 생성을 설명하는 코드 예제

2.1 CodeScroll

CodeScroll 은 Suresoft Technology Inc.에서 개발한 C 및 C ++ 용 상용 테스트 데이터 생성 도구이다. Safety critical 소프트웨어 시스템을 테스트하고 구조적 커버리지를 달성하기 위해 자동차, 방위 및 우주 항공 분야의 다양한 업계에서 CodeScroll 을 사용한다[1]. CodeScroll 은 주기적인 값 생성, 랜덤 생성 및 쌍단위 입력 파티셔닝(pairwise input partitioning)을 비롯한 여러 가지 기술을 사용한다. CodeScroll 엔지니어의 경험에 따르면 쌍단위 입력 파티셔닝이 가장 효과적인 방법이었다.

그림 1a 의 예제는 CodeScroll 의 쌍단위 입력 파티셔닝의 동작 과정과 약점을 보여준다. CodeScroll 은 경량 정적 분석을 사용하여 경계 값과 입력 파티션을 식별한다. 그림 1a 의 4 행과 같이 고정된 명확한 경계 값이 있는 분기의 경우 CodeScroll 은 변수 `a` 의 값을 효율적으로 생성 할 수 있다. 예를 들어, `a < 42` 의 조건에 대한 `a` 의 값으로 경계 값과 그 이웃 값인 42, 41, 43 과, 일정 간격 떨어진 값(42 보다 충분히 높거나 낮은 값), 자료형 파티션 (type partitions) (주어진 자료형의 최솟값, 최댓값, 중간값) 들을 시도한다.

결과적으로 CodeScroll 은 조합 상호 작용 테스트 (Combination Interaction Testing, CIT) [23]와 유사한

[1] http://www.suresofttech.com/ko/engineering/dev_vehicle 참조

방식으로 각 변수의 가능한 값들 사이의 쌍단위 상호 작용 커버리지를 달성하고자 한다. 이 프로세스는 대상 함수의 실행을 요구하지 않으며, 따라서 CodeScroll 은 실행 시간의 대부분을 정적 분석에 사용한다. 이 같은 CodeScroll 의 쌍단위 접근은 특정 분기에는 효과적이지만 약점이 있다. 예를 들어 매개 변수 (그림 1a 의 라인 5) 또는 함수 호출 (그림 1a 의 라인 6)에 의해 설정되는 경계 값은 정적 분석에 의해 확정될 수 없으며,낮은 분기 커버리지를 야기한다.

CodeScroll 은 동적 데이터 구조의 테스트 데이터를 생성하는 간단하면서도 효과적인 휴리스틱을 사용한다. 첫째, 포인터는 단일 항목을 포함하는 배열로 간주된다. 항목이 복합 자료형 (예: 구조체) 인 경우 대상 함수에서 사용되는 멤버의 값이 쌍단위 방법을 사용하여 채워진다. 둘째, 복합 자료형에 재귀 포인터 (자신의 자료형을 가리키는 포인터)가 포함된 경우 포인터는 1 의 깊이로 인스턴스화되지만 멤버는 인스턴스화하지 않는다. 예를 들어 그림 1b 의 코드에서 CodeScroll 은 매개 변수 `d` 와 `d-> next` 를 인스턴스화하지만 `d-> next-> a` 변수에는 값을 할당하지 않는다. 이는 쌍단위 조합으로 인해서 테스트 케이스의 가짓수가 지수적으로 늘어나는 것을 피하기 위한 것이다.

2.2 Austin

Austin[17] 은 C 에 대한 검색 기반 테스트 데이터 생성 도구로, 동적 기호 실행과 제약조건 풀이(Constraint Solving)를 사용한다. 수치 입력 값은 AVM 을 사용하여 찾고 포인터 입력은 포인터 등가성 그래프를 CUTE [24]와 유사한 메커니즘으로 "풀이"하여 할당된다.

Austin 은 전술한 접근 방식을 다음과 같이 사용한다. 우선 Austin 은 테스트 대상 C 함수에서 커버되지 않은 분기를 "타겟"으로 선택한다. 임의로 생성 된 초기 입력이 주어지면 Austin 의 계측기는 타겟 분기가 커버 되었는지 아니면 실행 경로가 타겟에서 벗어났는지 여부를 감지한다. 실행 경로가 타겟에서 벗어난 경우 Austin 은 대상 분기가 커버 되지 못하도록 유도한 "중요" 결정 구문(decision statement)의 조건을 평가한다. 이 중요 결정 구문은 함수의 제어 흐름 그래프(Control Flow Graph) 상에 있는 노드 중, 타겟 분기를 제어 종속 (control dependent)하는 노드이다. 구문 a 의 결과가 구문 b 의 실행 여부를 좌우할 때, 구문 b 가 구문 a 에 제어 종속된다고 한다. 그 예시로는 분기가 중첩된 "`if`"문이 있다. 그림 1b 에서 타겟 구문은 8 행, 11 행의 `False` 분기 및 12 행의 `True` 분기에 제어 종속한다.

그 다음 Austin 은 동적 기호 실행 엔진을 호출하여 경로 조건(Path Condition)을 도출한다. 경로 조건은 함수의 입력에 대한 제약 조건으로써 함수가 어느 실행 경로를 따를지를 기술한다. 경로 조건은 현재 입력 값이 따르는 중요 결정 노드까지의 실행 경로로부터 구해진다. 현재 입력값으로 함수를 실행 시켜 실행 경로를 구한 후, 중요 결정 노드에서의 조건을 반전시켜 타겟 분기에 도달하기 위한 올바른 실행 경로를 나타내는 경로 조건을 만든다.

Austin 은 경로 조건을 분석하여 함수의 숫자 자료형 입력 값을 찾기 위해서 AVM 을 호출할지, 포인터 입력을 풀기 위해 기호 실행을 할지 결정한다. 후자의 경우 Austin 은 앞서 언급한 포인터 등가성 그래프를 작성한다. 경로 조건은 포인터에 해당하는 제약 조건을 서술 하기 위해서 $x = y$ 및 $x \neq y$ 의 형태로 단순화 된다. 여기서 x, y 는 `NULL` 이나 포인터 입력에 해당하는 기호 변수(symbolic variable)다. Austin 은 경로 조건으로부터 서로 동등한 포인터를 그룹화 하여 그래프의 노드로 하고 노드 간의 비등가성을 나타내는 엣지를 추가하여 포인터 등가성 그래프를 만든다. 이 그래프는 다음의 과정을 통해 포인터 입력을 생성하기 위해 "풀이"된다. 각 노드 n 에서 `NULL` 포인터를 나타내는 노드로 향하는 엣지가 없으면 Austin 은 노드 n 의 기호 변수가 나타내는 모든 포인터를 `NULL` 로 설정한다. 만약 n 이 다른 기호 변수 s 의 주소를 나타내는 경우 Austin 은 s 가 가리키는 곳으로 구체적인 포인터 입력을 할당한다. 그렇지 않으면, Austin 은 새로운 메모리 위치를 생성하여 n 이 가리키는 구체적인 포인터 입력을 그 위치로 할당한다.

예시인 그림 1b 에서 타겟 분기는 12 행의 `True` 분기이다. 해당 분기의 조건문이 숫자로 된 제약 조건이지만 동적 자료 구조가 그 전에 올바른 모양으로 초기화 되어야한다. Austin 은 세 개의 노드로 구성된 포인터 등가성 그래프를 그리며, 그 노드는 각각 `d` (n1), `d-> next` (n2) 및 `NULL` (n3)이다. 8 행에서 `d` 가 `NULL` 이 아니라는 조건문의, `True` 분기가 실행되어야 하기 때문에 n3 과 n1 사이에는 비등가성을 나타내는 엣지가 형성된다. 기호 분석으로 10 행에서 `d-> next` 가 `d_in` 로 할당 된다는 것을 알 수 있다. `d_in` 은 `d-> next` 와 동등한 포인터이므로 n2 노드로 그룹이 된다. 기호 분석은 이어서 11 행을 `True` 로 실행한다는 것을 알아내고 이는 11 행의 `NULL` 과 `d-> next` 이 같지 않아야 하므로 각 기호 변수에 해당하는 n3 과 n2 노드 간에 엣지가 형성된다. 이 포인터 등가성 그래프를 풀면 `d` 가 새로운 메모리 위치로 초기화되고 `d-> next` 도 별도의 위치로 초기화된다. 그 이후 AVM 검색이 `d_in-> a`(즉, `d-> next-> a`)가 2 로 설정되어야 한다는 라인 12 의 조건을 해결한다.

검색 기반 테스트 데이터 생성 도구 중 C 를 대상으로 하는 가장 최신의 도구이기는 하지만, Austin 은 더 이상 활발하게 관리되지 않으며 오픈 소스 저장소의 마지막 수정은 2011 년에 이루어졌다 [18]. 또한, 제 5 장에서 기술하듯이 Austin 은 커버하지 못하는 특정 분기문 유형들이 존재한다. 이를 동기로 본 연구는 새로운 도구를 제안 및 소개한다.

3. CAVM

CAVM 은 검색 기반 소프트웨어 테스팅 기술을 활용하여 상용 수준의 도구인 CodeScroll 을 보완함으로써 어려운 분기를 보다 효과적으로 처리 하려는 목적의 산학 협력 연구의 오픈 소스 부산물이다.

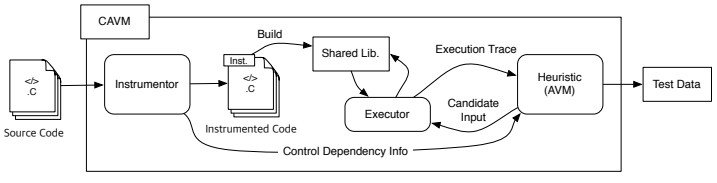

그림 2 CAVM 의 아키텍처

3.1 아키텍처

그림 2 는 CAVM 의 전체 아키텍처를 묘사한다. CAVM 은 먼저 검색에 사용될 분기 커버리지 달성을 위한 적합도(fitness)를 동적으로 측정하기 위해 주어진 소스 코드를 계량화(Instrumentation) 한다. 계량화는 Clang 을 사용하여 C 소스 코드를 직접 수정하는 방식으로 이루어진다. 원본 소스 코드를 계량화된 버전으로 바꾼 후 CAVM 은 대상 프로그램을 공유 라이브러리로 빌드한다. 공유 라이브러리를 통해 CAVM 은 엔트리 포인트를 추가하지 않고도 대상 함수를 직접 가져 오고 호출 할 수 있다. 이후 CAVM 은 타겟 분기를 커버하는 테스트 입력을 찾기 위하여 확장된 버전의 AVM 을 적용한다.

3.2 동적 자료구조에 대한 AVM 의 확장

알고리즘 1 CAVM 의 국소 탐색 알고리즘

LocalSearch($\vec{x}$, *current*)
Input: An input vector, $\vec{x}$, and the current search target, *current*
Output: The minimum fitness value found, *fitness*, and the input vector that corresponds to the value, $\vec{x}$

```
(1)   while budget > 0 and fitness > 0
(2)     e = NEXT(x, current)
(3)     if ISPRIMITIVE(e)
(4)       fitness ← ITERATIVEPATTERNSEARCH(x, e)
(5)     else if ISSTRUCT(e)
(6)       fitness ← LOCALSEARCH(x, e.members)
(7)     else if ISPOINTERTOPRIMITIVES(e)
(8)       foreach i in e
(9)         fitness ← ITERATIVEPATTERNSEARCH(x, i)
(10)      e' ← GROW(e)
(11)      x ← x(e ← e')
(12)      fitness ← EVALUATE(x)
(13)    else if ISPOINTERTOSTRUCT(e)
(14)      if e is NULL
(15)        e' ← INSTANTIATE(e)
(16)        x ← x(e ← e')
(17)        fitness ← EVALUATE(x)
(18)      else
(19)        fitness ← LOCALSEARCH(x, *e)
(20)  return x, fitness
```

알고리즘 1 은 CAVM 에서 사용하는 검색 알고리즘에 대한 의사 코드(pseudo code)이다. 간결함을 위해 의사 코드에는 적합도가 전역 최적 값에 도달 할 때마다 알고리즘이 종료하는 것을 생략하여 표기하였다. 또한 입력 벡터 x 의 모든 요소에 대하여 탐색을 마친 뒤 적합도가 향상되었는 지를 확인하는 부분도 생략하였다. 기본 원시 자료형의 경우 4 행은 반복 패턴 검색 (IPS: Iterative Pattern Search) [15,20]을 호출한다. 나머지 경우는 동적 데이터 구조를 다루는 알고리즘이다. CAVM 은 입력 벡터 x 의 모든 포인터를 `NULL` 로 초기화한다.

5 ~ 6 행은 구조체 입력의 속성 값들에 대해 재귀적으로 검색을 수행한다. 7 ~ 19 행은 포인터 입력을 처리한다. CAVM 은 기본 자료형을 가르키는 포인터를 배열로 간주한다. 7 행부터 12 행에서는 현재 배열의 각 요소에 IPS 를 적용하고 검색이 완료되지 않으면 배열의 크기를 1 만큼 증가시킨다.

마지막으로 13 ~ 19 행은 구조체에 대한 포인터를 처리한다. 현재 값이 `NULL` 인 경우 CAVM 은 메모리를 할당하는 것으로 현재 타겟 분기를 커버 될 수 있는지 여부를 확인한다 (15 행, 구체화 된 구조체의 기본 속성값들이 랜덤으로 초기화 됨). 메모리가 할당된 이후에도 검색이 성공하지 못하면 CAVM 은 구체화된 새로운 구조체의 값, 즉 현재 대상 구조체의 속성값들에 대하여 차례대로 검색을 수행한다 (18 ~ 19 행).

그림 1 의 코드를 예시로 살펴보자. CAVM 은 입력 `d` 를 `NULL` 로 초기화하여 시작한다. 이 입력은 목표 분기에 도달하지 않으므로 CAVM 은 `d` 에 메모리를 할당하게 되고 이는 8 행의 `True` 조건을 만족시킨다. 그런 다음 CAVM 은 `d-> a` 및 `d-> b` 에 IPS 를 적용하나 적합도는 향상되지 않는다. 마지막으로 CAVM 은 `d-> internal` 과 `d-> next` 에 메모리를 할당한다. 다시 IPS 를 속성값들에 적용하는 과정에서 CAVM 은 `d-> next-> a` 의 값을 2 로 만들어 목표 분기에 도달한다.

3.3 문자열 비교

C 언어에서 문자열은 `char` 의 배열로 표현되기 때문에 CAVM 은 `char` 배열을 검색하는 것으로 문자열 입력을 처리한다. CAVM 은 원시 자료형의 배열을 검색 할 수 있지만, 분기에서 `strcmp` 함수가 사용되면 그 리턴 값으로는 검색해야할 방향을 올바르게 구하기 어렵다는 문제가 추가로 발생한다. `strcmp` 는 두 개의 문자열이 동일한지, 아니면 어느 한 문자열이 다른 것보다 알파벳 순으로 더 우선하는지 여부를 나타내는 단일 정수를 리턴한다. 이 리턴 값에는 두 문자열 간의 거리 정보가 없기 때문에 CAVM 이 검색 방향을 결정하기가 어렵다.

문자열 간의 거리 정보를 제공하기 위해 CAVM 은 `strcmp` 의 래퍼 (wrapper)인 `strcmp2` 를 정의하고, 계량화 프로세스 도중에 소스 코드의 조건문이 사용하는 모든 `strcmp` 를 `strcmp2` 로 교체한다. 이것은 Fraser 와 Arcuri [4]가 JAVA 에서 `String.equals()` 메소드를 교체한 것과 동일한 접근법이다. 그러나 C 를 대상으로 하는 기존의 검색 기반 테스트 데이터 생성 도구들 중 이 아이디어를 구현한 것은 없었다.

`strcmp2` 는 부호있는 정수 대신 부호있는 문자열 거리를 반환한다. 문자열 간의 거리는 *lengthDiff* + *norm (charDiff)*로 정의된다. *lengthDiff* 는 두 입력 문자열 사이의 길이의 차이다. *charDiff* 는 두 입력 문자열 내의 문자 간의 요소 단위 거리를 누적한 것이다. 두 문자열의 길이가 다를 때는 더 짧은 문자열의 길이까지 누적한다. 정규화를 위해서는 $norm(x) = 1 - 1.001^{-x}$ 를 사용한다 [11].

4. 실험 설계

4.1 연구 질문

본 연구에서는 아래의 질문들에 대해 살펴보고자 한다.

RQ1. 유효성: CodeScroll, Austin 및 CAVM 중에서 어느 도구가 가장 높은 분기 커버리지를 달성하는가?

표 1 연구 대상 함수

Subjects	Description	Br.	*	Rec. *	`struct`	`strcmp`
AllZeros	Examples from AVMf	6	✓	-	-	-
Calendar		46	-	-	-	-
Line		14	-	-	✓	-
Triangle		16	-	-	-	-
CodeScroll Antipatterns	Set of branches that CodeScroll cannot cover	16	✓	✓	✓	✓
LinkedList	5 utility function for singley linked list	26	✓	✓	✓	-
BinaryTree	7 tree-related functions from a textbook by Horowitz et al.	30	✓	✓	✓	-
busybox-ls	5 functions from ls in Busybox 1.2.0	32	✓	-	-	-
decode.c	22 functions from decode.c	296	✓	-	✓	-
Total	49 C functions	482				

RQ2. 효율성: CodeScroll, Austin 및 CAVM 중에서 분기 커버리지를 가장 효율적으로 달성하는 도구는 무엇인가?

본 연구에서는 각각의 도구를 C 함수에 적용하여 RQ1 과 RQ2 에 대해 살펴본다. Austin 과 CAVM 은 확률적 접근 방식을 채택하기 때문에 RQ1 의 경우 커버리지와 대상 함수 실행 횟수 그리고 RQ2 의 경우 실행 시간을 총 20 번 반복한 실험의 평균으로 측정하였다. CodeScroll 은 확률적이지 않으므로 실행을 반복하지 않는다.

4.2 대상 함수

표 1 은 본 연구에서 대상으로 한 함수의 목록이다. 간단한 함수에서부터 상용 C++ 프론트엔드에 이르기까지 다양한 종류의 함수를 대상으로 실험을 진행하였다. CodeScroll Antipatterns 는 CodeScroll 이 커버하지 못하는 분기 패턴의 유형들로 구성된 예제 함수 컬렉션이다. Line, Calendar, Triangle 및 AllZeros 예제는 McMinn 과 Kapfhammer [20]의 기본 예제를 C 로 구현하였다. LinkedList 는 웹에서 발췌한 C 의 단일 연결 리스트에 대한 유틸리티 함수 구현 모음[2] 이다. BinaryTree 의 7 가지 함수들은 Horowitz 의 책 [13]에서 발췌하였다. busybox-ls 의 5 가지 함수는 busybox 패키지[3] 의 ls 유틸리티 오픈 소스 구현체 에서 선정하였다. 마지막으로, decode.c[4]에는 Edison Design Group 에서 개발 한 C++ 프론트엔드 용 demangler 모듈에서 선택된 24 개의 함수가 포함되어 있다. 총 49 개의 함수, 482 개의 분기를 대상으로 하였다.

"*"로 표시된 열은 포인터 입력이 포함 된 함수인지 여부를 나타내며 "Rec. *"는 재귀적 데이터 구조 (예 : 자신의 자료형을 다시 가리키는 포인터를 포함한 구조체)가 포함되어 있는지 여부를 표시한다. 비슷하게, `struct` 열은 입력 변수가 C 구조체(`struct`) 타입인지 여부를 보여 주며 `strcmp` 열은 입력 변수가 조건문 안에서 `strcmp` 를 사용하여 문자열과 비교되는지 여부를 나타낸다.

이 실험 대상 함수들은 임의의 C 함수를 편향되지 않게 샘플하기보다 오히려 본 연구에서 중점적으로 다루는 주제인 포인터, 배열 및 동적 데이터 구조를 입력으로 하는 C 함수들을 중심으로 선정 되었다. 실험에 사용된 모든 함수들의 소스 코드는 http://coinse.kaist.ac.kr/projects/cavm/ 에서 확인할 수 있다.

4.3 설정

CAVM 에는 대상 함수의 각 입력 변수에 대한 검색 범위를 설정하는 기능이 있지만 Austin 에는 이러한 제어 기능이 없다. 따라서 입력 범위를 좁히지 않고 각 기본 자료형의 전체 범위를 사용하여 두 도구가 같은 공간에서 검색을 수행하였다. Austin 과 CAVM 모두에 대해 최대 실행시간 5 분, 대상 함수 최대 실행 횟수 1,000 번의 제한 조건을 설정하였다. 두 도구 모두 "부수적인" 테스트 입력 [8] (즉, 현재 목표하는 것 이외의 분기를 우연히 커버하는

[2] http://milvus.tistory.com/17 의 예제에서 발췌

[3] Busybox는 작은 단일 실행파일에 유닉스 유틸리티를 모은 컬렉션이다: https://busybox.net .

[4] https://www.edg.com/c

입력[5])을 커버리지 달성으로 인정하고 결과로 수집한다. 실험에서 5 분 이내에 달성된 분기 커버리지는 최종 결과로 포함되나 도구가 5 분 제한 시간 내에 종료되지 않으면 0% 분기 커버리지를 기록하였다.

CodeScroll 은 여러가지 테스트 입력 생성 기술을 지원하지만, 본 실험에서는 2.1 장 에서 논의 된 것처럼 가장 효과적이라고 알려진 쌍단위 입력 파티셔닝 방식을 사용하였다. Austin, CAVM 과 달리 CodeScroll 은 테스트 입력을 생성하기 위해 대상 함수를 실제로 실행하지 않아도 된다 (2.1 장 참조). 또한 CodeScroll 은 GUI 를 통해서만 조작할 수 있기 때문에 다른 도구와 직접적으로 실행 시간을 비교하기는 어렵다. 따라서 CodeScroll 에 대해서는 대표적인 경우에 한하여 대략적인 실행 시간을 측정하였다.

Austin 은 C Intermediate Language (CIL) [22]에 의존하기 때문에 Austin 에서 세는 분기의 개수는 CodeScroll 및 CAVM 에서의 분기의 개수와 다를 수 있다. 예를 들어 Austin 은 복합 논리 연산자를 중첩된 `if` 문으로 나눈다. 따라서 이러한 영향을 받은 Austin 의 결과를 수동으로 조사하고 CodeScroll 및 CAVM 과 같은 분기의 개수를 가지도록 커버리지를 다시 계산하였다.

4.4 실험 환경

CAVM 은 Python 과 C/C++로 작성되었다. 대상 함수에 대한 계량화 탐침 구현은 C/C++로 작성되었으며 Clang 버전 3.9.0 및 GNU GCC 버전 4.9 이상을 요구한다. AVM 검색은 Python3 로 구현되었으며 Python 런타임 버전 3.5 이상을 요구한다. 또한 CFFI[6] 패키지를 필요로 한다.

실험 환경으로는 CAVM 은 Intel 코어 i7-6700K 4.0GHz 및 32GB RAM 이 장착된 컴퓨터에서 실행되며 운영체제는 Ubuntu 14.04 LTS 를 사용하였다. Austin 은 특정 요구사항으로 인해 동일한 사양의 Ubuntu 12.04.5 LTS 컴퓨터 상에서 실험하였다. CodeScroll 은 Intel Core i5-6600 3.9GHz 및 16GB RAM 이 있는 컴퓨터에서 Windows 7 상에서 실행하였다. 4.3 장에서 언급한대로 CodeScroll 실행 시간을 다른 도구와 직접 비교하지 않기 때문에 다른 하드웨어 환경이 문제가 되지 않는다.

5. 결 과

5.1 유효성

표 2 CodeScroll, Austin, CAVM 의 평균 커버리지 및 표준 편차. 가장 높은 커버리지는 굵게 표시, Br. 은 분기 개수, C.S.는 CodeScroll 을 의미함

Function	Br.	C.S.	Austin		CAVM	
			μ	σ	μ	σ
AVMf						
allzeros□	6	0.00	0.00	0.00	**83.33**	0.00
calendar*	46	**100.00**	0.00	0.00	0.00	0.00
line†	14	**100.00**	0.00	0.00	28.57	0.00
triangle‡	16	**93.75**	0.00	0.00	89.06	5.32
Antipatterns						
case1	4	0.00	100.00	0.00	**100.00**	0.00
case2	4	75.00	100.00	0.00	**100.00**	0.00
case3	2	50.00	100.00	0.00	**100.00**	0.00
case4§	2	0.00	0.00	0.00	**100.00**	0.00
case5	2	50.00	**100.00**	0.00	**100.00**	0.00
case6	2	50.00	**100.00**	0.00	**100.00**	0.00
LinkedList						
delete	6	**100.00**	**100.00**	0.00	60.83	23.12
insert	8	87.50	**100.00**	0.00	70.62	14.21
modify	4	75.00	**100.00**	0.00	66.25	20.32
print_list	2	**100.00**	**100.00**	0.00	**100.00**	0.00
search	6	**100.00**	0.00	0.00	**100.00**	0.00
busybox-ls						
bold	2	50.00	**100.00**	0.00	**100.00**	0.00
dnalloc	2	**100.00**	**100.00**	0.00	**100.00**	0.00
fgcolor	2	**100.00**	**100.00**	0.00	**100.00**	0.00
my_stat	10	0.00	0.00	0.00	0.00	0.00
scan_one_dir	16	**6.25**	0.00	0.00	0.00	0.00
BinaryTree						
inorder	2	**100.00**	**100.00**	0.00	**100.00**	0.00
iter_inorder	4	0.00	0.00	0.00	**100.00**	0.00
iter_search	6	**100.00**	0.00	0.00	**100.00**	0.00
level_order	8	62.50	0.00	0.00	**100.00**	0.00
postorder	2	50.00	**100.00**	0.00	**100.00**	0.00
preorder	2	50.00	**100.00**	0.00	**100.00**	0.00
search	6	**100.00**	0.00	0.00	**100.00**	0.00
decode.c						
func1	2	**100.00**	0.00	0.00	**100.00**	0.00
func2	2	**100.00**	0.00	0.00	**100.00**	0.00
func3	48	10.42	0.00	0.00	**29.90**	5.63
func4	14	21.43	0.00	0.00	**71.07**	6.34
func5	14	**21.43**	0.00	0.00	0.00	0.00
func6	16	18.75	0.00	0.00	**27.14**	9.44
func7	30	6.67	0.00	0.00	**11.56**	1.79
func8	6	50.00	0.00	0.00	**75.83**	12.65
func9	44	4.55	0.00	0.00	**69.66**	7.31
func10	28	7.14	0.00	0.00	**62.32**	10.20
func11	2	**100.00**	0.00	0.00	**100.00**	0.00
func12	4	25.00	0.00	0.00	**27.50**	7.69
func13	4	50.00	0.00	0.00	**73.75**	5.59
func14	2	50.00	0.00	0.00	**52.50**	11.18
func15	2	50.00	0.00	0.00	**97.50**	11.18
func16	12	8.33	0.00	0.00	**22.50**	18.56
func17	4	25.00	0.00	0.00	**27.50**	11.18
func18	4	50.00	0.00	0.00	**64.17**	6.11
func19	28	3.57	0.00	0.00	**8.75**	3.57
func20	8	87.50	0.00	0.00	**100.00**	0.00
func21	4	**100.00**	0.00	0.00	**100.00**	0.00
func22	18	**100.00**	0.00	0.00	**100.00**	0.00

5.1 절에서 다음의 이슈를 다룸
□: 간접적인 제어 종속성, *: 큰 검색 공간
†: 낮은 성공률, ‡: 불가능한 분기, §: `strcmp` 의 사용

표 2 는 Austin 과 CAVM 을 20 번 반복 실행하고 CodeScroll 을 단일 실행한 커버리지 결과를 나타낸다. 표에서 decode.c 의 대상 함수는 표의 공간을 절약하기 위해

[5] 부수적인 커버리지는 원래 타겟 분기 이외에도 최종 생성된 테스트 케이스가 커버하는 분기를 의미한다.

[6] C Foreign Function Interface: http://cffi.readthedocs.io

함수 이름을 간략화했다. 대상 함수의 원래 함수 이름과, 소스 코드와 커버리지의 박스 플롯은 CAVM 웹 페이지[7]에서 확인할 수 있다. Austin 과 CAVM 은 반복 실행에서 얻어진 커버리지의 평균(μ)과 표준 편차 (σ)를 표기하였으며, 각 대상 함수에 대해 가장 높은 커버리지를 굵게 표시했다. 총 49 개 함수 중 5 가지의 함수에서 CodeScroll 이 가장 높은 분기 커버리지를 달성하였으며, Austin 은 2 가지 함수에서 가장 높은 분기 커버리지를 보였다. CAVM 은 20 개의 함수에서 가장 높은 분기 커버리지를 유일하게 달성했다. 한편 Austin 은 decode.c 의 어떤 대상 함수도 5 분 제한 시간 내에 커버하지 못했다.

실험에서 커버되기 어려웠던 분기들을 수동으로 분석하여 다음과 같은 공통적인 이슈를 확인했다 (각 이슈의 기호는 표 2 에서 상호 참조 할 수 있음).

간접적인 제어 종속성 (□) : allzeros 함수의 분기 중 하나는 입력 배열 내의 값이 0 인 요소의 개수가 입력의 크기와 같아야 커버된다. 즉 입력 배열 내의 모든 요소의 값이 0 이어야 한다. CAVM 은 이 분기를 효과적으로 처리하지 못한다. 플래그 문제 [9]와 마찬가지로, 대상 분기와 제어 종속 관계가 없는 다른 분기에서 0 의 개수 카운터가 변경되기 때문에 적합도 함수가 탐색에 필요한 정보를 주지 못하며 그 결과 랜덤 재시작을 반복한다.

너무 큰 검색 공간 (*) : Calendar 함수의 `for` 루프는 큰 범위 내에서 초기화 된 인풋을 처리할 때 많은 시간을 소비한다. 반복문은 두 개의 정수 입력 값 사이를 차례로 반복하므로 반복 횟수는 최대 C 의 정수 범위까지 될 수 있다. 큰 범위를 반복하는 함수를 빈번하게 호출하는 것은 제한시간 초과 및 결과적으로 0%의 커버리지를 야기한다. 논문에 실린 결과와 별도로 입력 변수 범위가 [-100, 100]으로 설정되면 CAVM 이 일관되게 100 %의 분기 커버리지를 달성하는 것을 추가로 확인하였다.

낮은 성공률 (†) : Line 함수의 분기는 지정된 제한 시간 및 실행 횟수 제한내에서 커버되기 어려운 조건을 가진다. CAVM 이 모든 분기를 커버하는데 성공하는 경우도 있지만 평균 커버리지는 어려운 분기를 커버하지 못한 실행으로 인해 낮아졌다.

불가능한 분기 (‡) : Triangle 함수는 논리적으로 불가능한 분기를 포함한다. Triangle 함수에는 다음 함수 스니펫과 같은 형태의 구문이 있다.

```
if (a == b) {…} else {if (a == b) {…}}
```

두 번째 `if` 문의 `True` 분기는 첫 번째 `if` 문 때문에 논리적으로 실행 불가능하다. 이를 제외하면 CAVM 과 CodeScroll 은 Triangle 함수의 모든 분기를 커버한다.

`strcmp` 의 사용 (§) : Antipatterns 의 case4 는 `strcmp` 를 호출한다. CodeScroll 이나 Austin 을 `strcmp` 의 호출 결과를 포함하는 조건문을 지원하지 않는다.

마지막으로 Austin 은 다음과 같은 제약사항을 보인다. Austin 이 포인터를 인스턴스화하기 위해서는 대상 함수의 소스 코드 내에 포인터 `NULL` 체크 조건이 명시적으로 쓰여있어야 한다. 소스 코드 내에서 주어진 포인터와 `NULL` 을 명시적으로 비교한 코드가 없으면 기호 실행 단계에서 포인터가 인스턴스화되지 않는다. 이러한 동작이 정상적인 것임을 Austin 의 주 개발자에게 확인한 후 작은 벤치마크(Antipatterns, AVMf, LinkedList 및 BinaryTree)에는 포인터 `NULL` 검사를 명시적으로 삽입한 후 실험하였으나 실제 오픈 소스 및 상용 대상 함수, 즉 ls 및 decode.c 는 수정하지 않았다. 결과적으로 decode.c 의 함수들은 모두 포인터 파라미터를 요구하기 때문에 decode.c 의 모든 대상 함수에서 Austin 은 0% 커버리지를 달성하였다.

표 2 의 결과에 따라 RQ1 에는 다음과 같은 답을 할 수 있다: CAVM 은 CodeScroll 이나 Austin 이 커버하지 못하는 분기를 커버할 수 있으며, 특히 Austin 은 포인터 입력을 인스턴스화 하는 데에 있어 중요한 한계를 가지고 있다.

5.2 효율성

표 3 Austin, CAVM 을 20 번 동작한 평균 실행 시간 및 표준 편차(단위: 초): 가장 짧은 시간은 굵게 표시, 제한 시간 초과는 - 로 표시

Function	Austin		CAVM	
	μ	σ	μ	σ
AVMf				
allzeros□	-	-	**3.62**	0.34
calendar*	-	-	-	6.66
line†	-	-	**43.75**	1.16
triangle‡	-	-	**12.04**	2.28
Antipatterns				
case1	2.01	1.20	**1.30**	0.58
case2	1.85	0.20	**0.43**	0.02
case3	0.76	0.11	**0.74**	0.18
case4	-	-	**0.75**	0.11
case5	**0.72**	0.19	0.79	0.19
case6	**0.99**	0.17	1.18	0.02
LinkedList				
delete	**2.44**	0.79	15.73	15.35
insert	**2.55**	0.87	15.09	13.67
modify	**1.81**	0.26	9.21	0.00
print_list	**0.15**	0.01	0.30	0.14
search	-	-	**150.72**	150.22
busybox-ls				
bold	0.88	0.89	**0.31**	0.15
dnalloc	**32.31**	17.54	62.82	37.86
fgcolor	1.0.1	0.75	**0.36**	0.22
my_stat	-	-	-	-
scan_one_dir	-	-	-	-
BinaryTree				
inorder	**0.14**	0.02	0.19	0.01
iter_inorder	-	-	**71.08**	0.34
iter_search	-	-	**0.86**	0.22
level_order	-	-	**143.82**	0.49
postorder	**0.13**	0.01	0.19	0.00
preorder	**0.14**	0.01	0.19	0.00
search	-	-	**0.83**	0.22
decode.c				
func1	-	-	**0.98**	0.02
func2	-	-	**0.59**	0.01
func3	-	-	**196.48**	17.38
func4	-	-	**16.61**	3.01
func5	-	-	-	-
func6	-	-	**41.46**	5.69

[7] http://coinse.kaist.ac.kr/projects/cavm/

func7	-	-	**150.08**	4.83
func8	-	-	**8.18**	1.31
func9	-	-	**77.69**	15.59
func10	-	-	**54.38**	10.44
func11	-	-	**0.67**	0.06
func12	-	-	**11.21**	1.38
func13	-	-	**5.68**	0.97
func14	-	-	**3.82**	0.81
func15	-	-	**2.05**	0.90
func16	-	-	**37.74**	9.25
func17	-	-	**10.98**	1.66
func18	-	-	**9.99**	1.21
func19	-	-	**123.74**	6.86
func20	-	-	**1.51**	0.06
func21	-	-	**0.73**	0.01
func22	-	-	**12.07**	0.71

5.2 절에서 다음의 이슈를 다룸
□: 간접적인 제어 종속성, *: 큰 검색 공간
†: 낮은 성공률, ‡: 불가능한 분기,

표 3 은 대상 함수에 대한 테스트 데이터를 생성하기 위해 Austin 및 CAVM 이 소모하는 실행 시간을 나타낸다. 대시 (-) 기호는 20 회의 실행이 모두 제한 시간인 300 초 내에 종료되지 않은 완전한 시간 초과를 나타낸다. 두 도구가 모두 제한 시간을 초과하는 경우를 제외하고, 실행 시간이 더 짧은 도구의 평균 실행 시간 값을 굵게 표시하였다. 5.1 절의 분기 커버리지에 영향을 주는 대부분의 요인이 실행 시간에도 영향을 미친다.

간접적인 제어 종속성 (□), 낮은 성공률 (†), 불가능한 분기 (‡) : 이 요인들에 의하여 발생한 반복된 랜덤 재실행 및 제한 시간 초과는 분기 커버리지에는 기여하는 바 없이 실행 시간만을 소모한다.

큰 검색 공간 (*) : 유효성 연구 (5.1 절)와 마찬가지로 입력 변수의 범위가 축소되면 CAVM 의 평균 실행 시간은 약 7 초로 떨어지고 분기 커버리지는 100 %로 유지된다.

비교 결과, decode.c 의 22 개 함수 모두에 대해 CodeScroll 을 실행하면 일반적으로 정적 분석에 5 초, 쌍단위 테스트 데이터 생성에 3 초가 소요된다. 비슷하게 LinkedList 와 BinaryTree 의 함수에서는 각 단계에서 5 초와 2 초가 소요된다.

따라서 표 3 의 결과를 바탕으로 RQ2 에 다음과 같이 답할 수 있다: Austin 과 CAVM 모두 CodeScroll 보다 오래 걸리며 최악의 경우 자릿수 단위의 실행 시간 차이가 있을 수 있다. CodeScroll 은 속도가 빠른 대신 높은 커버리지를 달성하지 못한다. CAVM 은 실생 시간이 오래 걸리지만 더 많은 분기를 커버한다. 마지막으로 Austin 은 CAVM 과 동일한 AVM 을 사용함에도 불구하고 decode.c 파일 내의 함수를 커버할 수 없었으며, 실행 시간 역시 CodeScroll 보다 더 오래 걸릴 수 있다.

6. 고 찰

본 연구를 통하여 향후 테스트 데이터 생성 기법의 개발 방향에 대해 다음과 같은 고찰을 할 수 있었다.

하이브리드화를 통한 이득: CodeScroll 이 사용하는 휴리스틱은 매우 간단하지만 특정 패턴의 분기에는 놀라울 정도로 효과적으로 동작한다. 따라서 이 간단한 휴리스틱을 AVM 같은 보다 자원이 많이 드는 기술과 하이브리드화할 경우 다양한 가능성이 있다고 예상한다. CAVM 의 향후 CodeScroll 통합은 CAVM 이 CodeScroll 이 커버하지 못한 분기에만 동작하는 계단식 모델을 따를 예정이다.

소스 코드를 직접 계량화의 적합성: CAVM 은 의도적으로 개량화 도구로 Clang 을 사용했다. 이는 Clang 이 널리 사용되는 컴파일러의 프론트엔드가 될 만큼 강력하기 때문이다. CAVM 과 비교하였을 때 Austin 이 요구하는 CIL 변환은 종종 빌드 관련 문제를 야기하여 사용성을 저하시킨다.

동적 자료 구조의 순수 검색 기반 처리: CAVM 은 검색 기반으로 동적 자료구조를 인스턴스화 할 수 있다. CAVM 이 현재 채택하고있는 성장 및 검색 접근법이 완전하지는 않지만 (예: 이중 링크드 리스트의 역방향 포인터는 키워서는 안됨), 본 논문은 순수한 검색 기반 기술이 가능함을 보인다.

7. 연구 타당성에 대한 분석

내적 타당도는 사용된 도구의 정확성을 비롯하여 연구의 결과가 주장을 얼마나 지지하는지의 정도를 나타낸다. 소스 코드 계량화에 사용한 Clang 은 널리 사용되는 LLVM 기반의 C 컴파일러 프론트엔드이며 광범위하고 공개적인 정밀 조사가 이루어지고 있다. CodeScroll 은 상용 도구이며 Austin 은 동료 검토 결과를 기반으로 하는 공개 소스 도구이다. CAVM 자체는 추가 검사를 위해 공개 소스로 공개될 예정이다.

외적 타당성 저해 요인으로는 본 연구의 결론을 일반화하는 데에 제약을 미칠 수 있는 요소가 포함된다. 본 논문에서 도출한 결론은 실험을 위해 선택한 대상 프로그램 및 함수의 특성에 귀속되며, 더 넓은 일반화는 광범위한 실증 연구를 통해서만 가능하다. 하지만 현재의 연구는 포인터와 동적 데이터 구조와 같이 C 언어에 특화된 특징을 서로 다른 도구가 어떻게 다루는지를 비교하는 데에 있어 충분한 증거를 제공한다.

8. 결 론

본 연구는 순수한 검색 기반 접근 방식을 사용하여 동적 자료 구조를 처리하는 AVM 기반 테스트 데이터 생성 도구 CAVM 을 제시한다. 기호 분석을 사용하여 필요한 자료 구조의 모양을 결정하는 현재의 최첨단 도구인 Austin 과는 달리, CAVM 은 포인터를 연쇄적으로 인스턴스화하며 동적 데이터 구조를 키워나간다. 본 연구는 CAVM 을 Austin 과 상용 테스트 데이터 생성 도구인 CodeScroll 에 비교하는 실험을 수행하였으며, 실증적 비교의 결과 CAVM 이 다른 도구들이 다루지 못하는 여러 분기를 커버 할 수 있음을 보였다. 본 연구의 향후 작업으로는 CAVM 내부의 제어 의존성 분석 모듈의 개선 및 AVM 알고리즘의 성능 개선 등이 계획되어 있다.

Acknowledgement. 이 논문은 2017 년도 정부 (과학기술정보통신부)의 재원으로 정보통신기술 진흥센터의 지원을 받아 수행된 연구임 (No. R7117-16-0005:

고신뢰 고위험(의료,항공,차량) 소프트웨어 시험 검증을 위한 연동형 클라우드 플랫폼)

참고문헌

[1] TIOBE Software: Tiobe Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 257–269. ISSTA 2015, ACM, New York, NY, USA (2015)

[3] Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008)

[4] Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Softw. Eng. 39(2), 276–291 (Feb 2013)

[5] Fraser, G., Arcuri, A., McMinn, P.: A memetic algorithm for whole test suite generation. Journal of Systems and Software 103, 311 – 327 (2015)

[6] Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013. pp. 291–301 (2013)

[7] Goues, C.L., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering. pp. 3–13 (2012)

[8] Harman, M., Kim, S.G., Lakhotia, K., McMinn, P., Yoo, S.: Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In: Proceedings of the 3rd International Workshop on Search-Based Software Testing (SBST 2010). pp. 182 –191 (apr 2010)

[9] Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Transactions on Software Engineering 30(1), 3–16 (Jan 2004)

[10] Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–14. ASE 2012 (2012)

[11] Harman, M., McMinn, P.: A theoretical and empirical study of search based testing: Local, global and hybrid search. IEEE Transactions on Software Engineering 36(2), 226–247 (2010)

[12] Harman, M., Wegener, J.: Evolutionary testing. In: Genetic and Evolutionary Computation (GECCO). Chicago (Jul 2003)

[13] Horowitz, E., Sahni, S., Anderson-Freed, S.: Fundamentals of Data Structures in C. W. H. Freeman & Co., New York, NY, USA (1992)

[14] Jin, W., Orso, A.: F3: Fault localization for field failures. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 213–223. ISSTA 2013, ACM, New York, NY, USA (2013)

[15] Kempka, J., McMinn, P., Sudholt, D.: Design and analysis of different alternating variable searches for search-based software testing. Theoretical Computer Science 605, 1–20 (2015)

[16] Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering 16(8), 870–879 (1990)

[17] Lakhotia, K., Harman, M., Gross, H.: AUSTIN: A tool for search based software testing for the c language and its evaluation on deployed automotive systems. In: 2nd International Symposium on Search Based Software Engineering. pp. 101–110 (Sept 2010)

[18] Lakhotia, K.: Open source code of Austin, https://github.com/kiranlak/austin-sbst, (Accessed 12/4/2017)

[19] McMinn, P.: IGUANA: Input generation using automated novel algo- rithms. A plug and play research tool. Tech. Rep. CS-07-14, Department of Computer Science, University of Sheffield (2007)

[20] McMinn, P., Kapfhammer, G.M.: AVMf: An open-source framework and implementation of the alternating variable method. In: International Symposium on Search-Based Software Engineering (SSBSE 2016). Lecture Notes in Computer Science, vol. 9962, pp. 259–266. Springer (2016), code and examples available at: http://avmframework.org

[21] Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering 2(3), 223–226 (1976)

[22] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, pp. 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)

[23] Petke, J., Cohen, M.B., Harman, M., Yoo, S.: Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. IEEE Transactions on Software Engineering 41(9), 901–924 (September 2015)

[24] Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 263–272. ESEC/FSE-13, ACM (2005)

[25] Tonella, P.: Evolutionary testing of classes. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 119–128. ISSTA '04, ACM, New York, NY, USA (2004)

[26] Weimer, W., Nguyen, T., Goues, C.L., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09). pp. 364–374 (May 2009)

[27] Yoo, S.: Evolving human competitive spectra-based fault localisation techniques. In: Fraser, G., Teixeira de Souza, J. (eds.) Search Based Software Engineering, Lecture Notes in Computer Science, vol. 7515, pp. 244–258. Springer Berlin Heidelberg (2012)