ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 45, No. 10, pp. 1035-1044, 2018. 10
https://doi.org/10.5626/JOK.2018.45.10.1035

C 프로그램의 동적 및 정적 분석을 통한 시스템 실행에서의 유닛 입력 값 자동 수집 및 재연

(Automated Capturing and Replaying Unit Inputs of C Programs from System Executions through Static and Dynamic Analysis)

임 현 수 † 김 윤 호 †† 김 문 주 †††
(Hyunsu Lim) (Yunho Kim) (Moonzoo Kim)

요 약 유닛 테스트의 높은 오류 검출력에도 불구하고, 시스템 문맥을 고려하지 않고 생성된 유닛의 "거짓 입력" 문제가 (즉, 실제 시스템에서는 불가능한 유닛 동작 생성) 있다. 이를 해결하기 위해, 시스템 테스트 케이스의 실행 과정에서 타겟 함수의 호출 시점의 프로그램 상태를 직렬화하고, 이를 역직렬화하여 유닛 테스트 케이스로 사용하는 Carving & Replay (CR) 기술이 있다. 그러나, Java등의 언어와 달리 C 언어에서는 자체적인 직렬화 방법이 존재하지 않을뿐더러 포인터, 공용체, 구조체 등의 언어적 특성으로 인해 CR에 어려움이 있다. 본 논문에서는 이러한 문제를 프로그램이 사용하는 메모리의 추적, 동적 분석을 통한 런타임의 정보 활용, 정적 타입 분석을 통한 탐침 코드의 삽입 등을 이용해 해결하여 C언어용 CR 도구를 제시한다.

키워드: Carving, Replay, 유닛 테스트, 자동화 테스팅, 직렬화

Abstract Despite the high testing power of unit testing, it has an infeasible input problem, which is an impossible input for a unit in a real system. There is a technique known as Carving and Replay (CR) that serializes the state of the program when a target function is called in system execution and uses it as unit test case by deserializing it, to solve this infeasible input problem. However, unlike programming languages like Java, the C programming language does not provide a serialization method. Also, because of the C programming language's features such as structure, union, and pointer, it has its own challenges for applying the CR technique. In this paper, we examine the challenges and suggest a CR tool for C programs by solving such problems with tracking the memory usage of the program, using run-time information from dynamic analysis, and inserting a probe code by static analysis.

Keywords: carving, replay, unit test, automated testing, serialization

· 이 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원(NRF-2016R1A2B4008113), 정부(과학기술정보통신부)의 재원으로 한국연구재단-차세대 정보 컴퓨팅기술개발사업의 지원(NRF-2017M3C4A7068177), 정부(교육부)의 재원으로 한국연구재단의 지원(NRF-2017R1D1A1B03035851)을 받아 수행한 연구임
· 이 논문은 제44회 한국소프트웨어종합학술대회에서 '시스템 테스트 케이스를 이용한 C 프로그램의 동적 유닛 입력 값 자동 수집 및 재연 기술'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 한국과학기술원 전산학부(KAIST)
hyunsu.lim01@gmail.com
(Corresponding author임)
†† 정 회 원 : 한국과학기술원 전산학부
yunho.kim03@gmail.com
††† 종신회원 : 한국과학기술원 전산학부
moonzoo.kim@gmail.com

논문접수 : 2018년 3월 20일
(Received 20 March 2018)
논문수정 : 2018년 7월 6일
(Revised 6 July 2018)
심사완료 : 2018년 7월 16일
(Accepted 16 July 2018)

정보과학회논문지 제45권 제10호(2018. 10)

1. 서 론

유닛 테스트는 시스템 테스트보다 상태 공간의 크기가 작아 적은 테스팅 비용으로 높은 오류 검출력을 달성하는 효과적인 테스팅 방식이다[1]. 그러나, 유닛 테스트에서는 타겟 유닛만을 테스트하게 되므로 실제로 타겟 유닛이 어떤 시스템 문맥에서 실행되는지를 알 방법이 없다. 그 결과, 유닛 테스트 케이스가 실제 시스템 실행 상에서는 존재할 수 없는 실행 불가능한 테스트 케이스인 경우가 발생할 수 있다.

예를 들어, 그림 1과 같은 소스 코드에서 시스템 문맥을 전혀 고려하지 않은 채 타겟 함수 f에 대한 유닛 테스트 케이스를 생성하면 x = 0이라는 실제 시스템 실행 상에서는 존재할 수 없는 유닛 테스트 케이스를 생성할 수 있다. 그리고 이 실행 불가능한 테스트 케이스는 실제 시스템 실행에서는 발생할 수 없는 유닛의 오류인 division-by-0를 검출하는 문제가 있다.

이러한 실행 불가능한 테스트 케이스를 피해서 테스트 케이스를 생성하는 방법의 하나로 Carving & Replay (CR)라는 기술이 있다[2]. Carving은 시스템 테스트 케이스를 실행하는 과정에서 타겟 유닛이 호출되는 순간에 해당 유닛이 접근하는 프로그램 상태를 직렬화해서 저장하는 기술이며, replay는 이 직렬화된 데이터를 다시 역직렬화 해 유닛의 입력으로 바꾸어 유닛 테스트 케이스를 생성하는 기술이다. 이 CR을 이용하면 시스템 테스트 케이스 실행 과정에서의 유닛의 실행을 모방하는 유닛 테스트 케이스를 만들 수 있다.

유닛 테스팅에 대한 CR의 중요성은 다음과 같다. CR을 통해서 생성된 유닛 테스트 케이스는 모두 실행 가능한 테스트 케이스이므로 이를 시드 유닛 테스트 케이스로 삼아 콘콜릭 테스팅등의 기법을 이용하면 많은 양의 실행 가능한 유닛 테스트 케이스를 찾아낼 수 있다[3]. 또한, 특정 시스템 실행 하에서 유닛의 행동을 디버깅하고자 할 때, 시스템 테스트 케이스를 매번 실행하며 디버깅을 하는 것보다 CR 을 통해 유닛 테스트 케이스를 얻어내고 이를 이용해 유닛의 실행만을 반복하며 디버깅하면 훨씬 적은 비용으로 목적을 달성할 수 있다.

```
// x is input of a target program
int main(int x) {
  if (0 != x) f(x);
  return 0;
}
// Target function
void f(int x) {
  printf("Inverse of x: %f", 1/x);
}
```

그림 1 실행 불가능한 유닛 테스트 케이스 문제
Fig. 1 Infeasible unit test case problem

Java 언어의 경우, 이미 Serializable 인터페이스를 통해 객체를 직렬화하는 방법을 제공하고 있다. 그러나, C 언어의 경우, 언어 자체에서 제공하는 방법도 없을뿐더러, 포인터의 존재로 인해 프로그램 상태를 직렬화하는 데 어려움이 있다[4].

본 논문에서는 C언어에서 CR을 하는 데 있어서 발생하는 문제들에 대해 논하고 이 문제들의 해결 방법을 제시한다. 그리고 이를 통해, C언어용 CR 도구를 만드는 것을 목적으로 한다.

본 논문의 구성은 다음과 같다. 가장 먼저 2장에서는 전체적인 CR 도구의 흐름을 설명하고, 3장에서는 C언어용 CR 도구를 작성하는 데 있어서 발생하는 문제들과 그 해결법에 대해 논한다. 4장에서는 본 논문의 CR 도구에 여전히 존재하는 문제점에 대해 이야기하고, 5장과 6장에서는 CR 도구의 유효성을 검증하는 실험의 설계와 실험 결과를 보인다. 7장에서는 관련 연구와의 비교를 보이며 마지막으로 8장에서는 본 연구의 결론과 향후 연구의 방향에 대해 서술한다.

2. CR 도구의 작동 과정

CR 도구는 두 번의 단계를 통해 시스템 테스트 케이스로부터 유닛 테스트 케이스를 추출해 낸다.

먼저, 대상 C 프로그램 구문 중, 동적으로 타입이 결정되는 구문들에 대한 정보를 추출한다. 이는 시스템 테스트 실행과정에서 carving하는 상태 값을 기록하기 위해 해당 변수의 타입이 중요하기 때문이다. 첫번째 패스에서 CR도구는 유닛 테스트 케이스를 추출해내고자 하는 시스템 테스트 케이스의 실행에서 프로그램 내의 공용체 변수들이 어떠한 필드를 사용하는지의 정보와, void 포인터 형 변수들이 어떤 타입의 포인터 형 변수로 변환되어 사용되는지의 정보를 기록한다. 이를 위해, CR 도구는 첫 번째 패스에서 입력으로 타겟 프로그램의 소스 코드와 타겟 함수를 받아 해당 정보들을 기록해주는 소스 코드를 타겟 프로그램에 추가한 새로운 소스 코드(P')를 출력한다. 사용자는 이를 빌드한 뒤, 빌드한 프로그램에 유닛 테스트 케이스 추출을 원하는 시스템 테스트 케이스를 실행한다. 이 결과로 그림 2에서 보

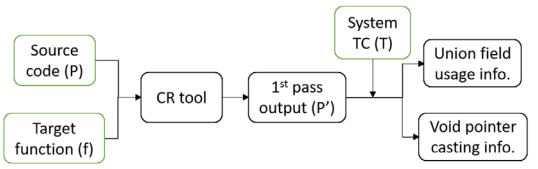


그림 2 CR도구의 첫 번째 패스
Fig. 2 The first pass of the CR tool

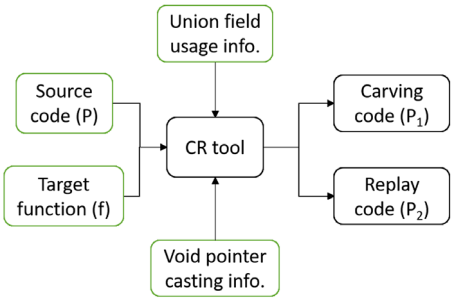


그림 3 CR도구의 두 번째 패스
Fig. 3 The second pass of the CR tool

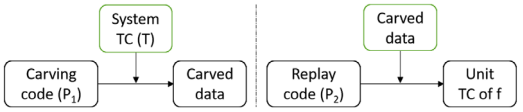


그림 4 Carving and Replay
Fig. 4 Carving and Replay

듯이 공용체 변수들이 각각 어떤 필드를 사용했는지를 기록한 “공용체 필드 사용 정보(Union field usage info.)” 파일과 void 포인터 형 변수들이 어떤 타입의 포인터 형 변수로 변환되어 사용되었는지를 기록한 “void 포인터 변환 정보(Void pointer casting info.)” 파일을 얻게 된다.

두 번째 패스는, 위 첫 번째 단계에서 얻어낸 정보를 토대로 실질적인 Carving & Replay가 이루어지는 단계이다(그림 3). CR 도구는 사용자로부터 타겟 프로그램의 소스 코드와 타겟 함수의 정보를 입력 받고, 첫 번째 패스에서 얻은 “공용체 필드 사용 정보” 파일과 “void 포인터 변환 정보” 파일을 이용하여 두 개의 소스 코드를 출력한다. 출력된 각각의 소스 코드는 모두 타겟 프로그램의 소스 코드를 적당히 변형시켜 생성되며, 하나는 Carving 작업을, 하나는 Replay 작업을 담당하는 소스 코드이다.

사용자는 이 두 개의 소스 코드를 각각 빌드한 뒤, 그림 4의 왼쪽과 같이 유닛 테스트 케이스의 추출을 원하는 시스템 테스트 케이스를 Carving 작업용 프로그램에 실행한다. 실행의 결과로 프로그램이 실행된 디렉토리에 새 디렉토리가 생성되어 그 내부에 Carving된 데이터가 바이너리 덤핑을 통해 파일의 형태로 저장된다. 사용자는 대상 유닛 Replay 작업용 프로그램을 실행하며 인자로 Carving된 데이터가 담긴 디렉토리를 넘기는 것으로 이를 유닛 테스트 케이스로 변환할 수 있다.

3. C언어의 CR에서 발생하는 문제와 해결 방법

3.1 포인터에 할당된 메모리 크기

문제점1. Carving을 위해서는 포인터 변수가 가리키는 공간에 저장된 데이터 모두를 읽어내야 하므로 할당된 메모리 공간의 크기를 알아내는 방법이 필요하다. 그러나, C 언어에서 포인터 변수가 가리키는 메모리 공간의 크기를 알아내는 직접적인 방법은 없다. 간접적인 방법으로 C 프로그램이 사용하는 메모리를 추적하고 관리해주는 AddressSanitizer[5,6]가 있으나, 이 프로그램은 실행 시간에 큰 영향을 미치고, 우리가 필요하지 않은 부분까지 관리해 주어 사용하기에 적절하지 않다.

해결방안1. 타겟 프로그램에 코드를 삽입해 해당 프로그램이 생성하거나 사용하는 모든 메모리 영역에 대한 정보를 추적하는 방법을 사용하였다. 즉, 메모리의 할당과 해제가 일어나는 매 순간마다 현재 프로그램이 할당하여 사용중인 메모리의 정보를 업데이트하였다. 구체적인 방법은 3.1.1에서 자세히 설명한다.

이 사용중인 메모리 영역에 대한 정보를 이용하면, 특정 주소값이 주어졌을 때, 해당 주소가 가리키고 있는 메모리 영역 1바이트가 현재 프로그램에서 사용 중인지, 즉 할당된 메모리인지를 알아낼 수 있게 된다. 이를 이용해서, 주어진 포인터 변수가 가리키는 메모리 공간으로부터 할당되지 않은 메모리 공간을 만날 때까지 순차적으로 인접한 메모리 공간이 할당되었는지를 검색하고 그 영역이 해당 포인터 변수에 할당된 것으로 처리하여 포인터 변수에 할당된 메모리 공간의 크기를 알아내었다.

3.1.1 프로그램에서 사용 중인 메모리 정보의 추적 방법

현재 프로그램이 사용 중인 여러 메모리 영역 정보(즉, 시작주소와 크기)가 M에 포함되어 있다고 하자. 이때, 각 메모리 할당 및 해제 방식에 따라 삽입된 코드에 의한 M의 변동은 다음과 같다.

전역 변수: 프로그램의 처음 시작부터 끝까지 계속해 메모리를 점유하므로, 프로그램의 시작 부분, 즉 main 함수의 진입 시점에서 모든 전역 변수가 사용 중인 메모리 영역을 M에 등록하였다. 전역 변수는 프로그램이 종료되는 시점까지 계속해 메모리 영역을 점유하고 있으므로, M에서 제거하는 연산은 필요하지 않다.

지역 변수: 해당 변수가 선언된 블록 내에서만 유효하므로 블록을 벗어나면 지역 변수가 점유하던 메모리 영역은 해제된다. 따라서, 지역 변수가 선언되는 시점에 해당 변수가 사용하는 메모리 영역을 M에 등록하고, 지역 변수가 선언된 블록이 종료되는 시점, 즉 지역 변수의 유효 범위를 벗어나기 직전에 해당 변수가 사용하는 메모리 영역을 M에서 제거하였다.

동적 할당: 프로그램 내에서 동적으로 malloc 등의 함수를 이용해 메모리를 할당하는 경우, 명시적인 선언부 없이 메모리 할당이 일어나기 때문에 이를 따로 처리해 줄 필요가 있다. 이를 위해서 malloc에 존재하는 hook 기능을 이용하여 동적 할당이 일어나기 직전에 할

당될 메모리 영역을 M에 등록하였다. 동적으로 할당된 메모리의 경우, free등의 함수를 이용해 해제되므로 malloc과 비슷하게 free의 hook 기능을 이용하여 메모리의 해제가 일어나기 직전 해제될 메모리 영역을 M에서 제거하였다.

프로그램 흐름 변경시 처리방법: goto/label, continue, break나 return과 같은 프로그램 흐름 제어문은 프로그램의 흐름을 바꾸어 변수의 유효 범위를 벗어나게 하고, 이는 결국 할당된 메모리 상태에 변화를 일으키기 때문에 따로 처리해 줄 필요가 있다. continue, break, 그리고 return은 해당 구문으로 인해 블록을 탈출하게 되므로 블록 탈출로 인해 유효 범위를 벗어나는 변수들이 사용 중인 메모리 영역을 구문의 실행 직전에 M에서 제거한다. goto/label은 블록을 탈출하여 새로운 블록에 진입하게 되므로 goto 구문의 실행 직전에 현재 범위에 존재하는 모든 변수들이 사용 중인 메모리 영역을 M에서 제거하고, 도착하는 label의 직후에 해당 label의 범위에 존재하는 모든 변수들이 사용 중인 메모리 영역을 M에 등록한다.

3.2 구조체의 carving

문제점 2. 구조체 타입의 변수 값을 carving할 때, 해당 변수에 해당하는 메모리 영역을 그대로 바이너리 덤프하게 되면 크게 두 가지 문제가 발생한다.

첫 번째로, 구조체에 포인터형 필드가 존재할 경우, 이 포인터가 가리키는 메모리 영역의 값 역시 저장해야 한다. 그렇지 않으면, 후에 carving된 데이터를 replay할 때, 실제 유닛의 상태를 그대로 모방할 수 없게 되는 문제가 발생한다.

예를 들어, 그림 5와 같은 구조체형 변수 st_a가 있다고 가정해보자. 이 변수를 carving하는 과정에서 이를 바이너리 형태로 덤프하게 되면, “030000003f235e89”와 같이 int형 멤버 변수 c와 int형 포인터 멤버 변수 k의 값이 저장될 것이다. 다시 말해, int형 포인터 멤버 변수 k가 가리키는 공간의 값, 위의 예제에서는 4가 저장되지 않게 된다.

```
struct st {
  int c;
  int *k;
} st_a;
st_a.c = 3;
st_a.k = (int *)malloc(sizeof(int));
*(st_a.k) = 4;
printf(“%d\n”, *(st_a.k));
```

그림 5 포인터 필드를 포함하는 구조체의 변수

Fig. 5 Variable of structure type that includes a pointer-type field

```
function GenValueSaver(N, T)
input: N: the name of the variable to carve
       T: the type of the variable to carve
output: probe code to carve the target variable
program:
 1 code := emptyString
 2 ...
 3 if T is “struct” then
 4   foreach field of “struct” do
 5     newN := N + “.” + nameOfField
 6     newT := typeOfField
 7     code += GenValueSaver(newN, newT)
 8   endfor
 9 endif
10 ...
11 return code
```

그림 6 Carving을 위한 코드를 생성하는 *GenValueSaver* 함수의 구조체 처리 분기

Fig. 6 Branch in *GenValueSaver* function, which generates code to carve structure type

두 번째로, carving하는 환경과 이를 replay하는 환경이 다를 경우, 비트 채움 등의 존재로 인해 덤프된 바이너리 데이터가 원본과 동일한 구조체 데이터로 변환되리라는 보장이 없다.

해결방안 2. 첫째로 구조체 변수의 각 필드를 따로따로 carving해야 하며, 둘째로 각 필드를 재귀적으로 carving해 포인터형 필드 등을 적절히 처리해주어야 한다. 이 아이디어를 개략적인 알고리즘 의사코드로 작성한 것이 그림 6이다.

그림 6의 함수 *GenValueSaver*는 입력으로 carving할 변수의 이름과 타입을 받는다. 그리고 carving할 변수의 타입에 따라 분기하여 각 타입에 알맞은 탐침 코드를 출력한다. 즉, *GenValueSaver*는 최종적으로 입력받은 변수를 carving 하는 코드를 생성해주고, 이 생성된 코드는 타겟 프로그램의 타겟 함수에 삽입되어 그림 4의 왼쪽에 있는 carving 코드를 생성하게 된다.[1)]

*GenValueSaver*에 구조체 변수가 주어지면 3번째 줄의 조건문에 의해 4번째에서 8번째 줄에 있는 구문이 실행된다. 4번째 줄에 의해 구조체 변수의 각 필드별로 carving하는 코드가 생성되고, 이는 앞서 언급한 구조체 변수의 carving에서 일어날 수 있는 문제를 해결하는 방법의 첫 번째 요소이다. 5번째와 6번째 줄은 구조체의 각 필드를 carving하는 코드를 생성하기 위한 준비 단계이다. 5번째 줄에서는 구조체 변수에 멤버 접근자를

1) 이 절에서는 구조체의 carving에 대해 알아보는 것이 목적이므로 혼동을 피하기 위해 그림 6에서 *GenValueSaver*의 carving할 변수 타입에 따른 분기 중 구조체에 해당하는 분기를 제외한 나머지를 2번째 줄과 10번째 줄의 생략 마크로 생략하였다.

사용해 필드의 값을 가리키는 변수를 얻어내고, 6번째 줄에서는 현재 타겟으로 하는 필드의 타입을 얻어낸다. 마지막으로, 7번째 줄에서는 5번째와 6번째 줄에서 얻어낸 필드의 정보를 이용해 *GenValueSaver*를 호출하고, 그 결과를 carving을 위한 코드에 추가한다. 여기서 다시 *GenValueSaver*가 호출되는 것은 앞서 언급한대로 각 필드를 재귀적으로 carving하기 위해서이다.

이 *GenValueSaver*를 앞서 제시한 그림 5의 예시에 대해 적용하면 다음과 같다. 먼저 *GenValueSaver*가 st_a라는 변수 명과 struct st라는 타입 정보를 이용해 호출된다. struct st는 구조체 타입이므로 그림 6의 3번째 조건문에서 참을 반환하며, 해당 조건문이 실행된다. struct st에는 int형 필드 c와 int형 포인터 필드 k가 존재하고, 이 각각에 대해 5번째에서 7번째 줄의 코드가 실행될 것이다.

우선 int형 필드 c에 대해서, 5번째 줄은 newN을 st_a.c로 설정하고, newT를 int형 타입으로 설정한다. 7번째 줄에 의해 *GenValueSaver*가 st_a.c와 int형 타입을 이용해 호출된다. 다시 말해, st_a의 int형 필드를 carving하는 코드가 생성되고, 이것이 st_a를 carving하기 위한 코드에 추가된다.

마찬가지로 int형 포인터 필드 k에 대해서, 5번째 줄은 newN을 st_a.k로 설정하고, newT를 int형 포인터 타입으로 설정한다. 이후, 7번째 줄에 의해 *GenValueSaver*가 st_a.k와 int형 포인터 타입을 이용해 호출된다. 즉, st_a의 int형 포인터 필드를 carving하는 코드가 생성되고, 이것이 st_a를 carving하기 위한 코드에 추가된다. 여기서 주목할 점은, *GenValueSaver*가 재귀적으로 호출되었기 때문에 st_a.k를 carving하는 코드를 생성하는 데에 포인터 타입을 carving하는 코드를 생성하는 방법을 이용하게 된다는 것이다. 이로 인해 구조체 내부에 포인터 멤버가 있을 때에도, 해당 포인터 멤버가 가리키는 공간의 값을 저장할 수 있게 된다.

3.3 공용체의 carving

문제점 3. 공용체 타입의 변수 값을 carving할 때, 해당 변수를 그냥 바이너리 덤프하게 되면 3.2절과 비슷한 문제가 발생하게 된다. 즉, carving의 대상이 되는 공용체 변수가 포인터 형 필드를 사용하고 있는 경우, 해당 포인터가 가리키고 있는 메모리 영역의 값을 저장하지 않게 되는 문제가 발생한다. 이를 해결하기 위해선, 기본적으로 3.2절의 해결방식과 비슷하게 공용체가 사용하는 필드를 재귀적으로 carving할 필요가 있다. 그러나, 단순히 재귀적으로 carving을 시도할 경우, 멤버 변수들이 같은 메모리 공간에 겹쳐 할당된다는 공용체의 특수성에 의해 문제가 발생할 수 있다.

```
union un {
  int c;
  int *k;
} un_a;
un_a.c = 3;
printf("%d\n", un_a.c);
```

그림 7 공용체 변수
Fig. 7 Union-type variable

```
un_a - c
```

그림 8 그림 7에 대해 생성된 공용체 필드 사용 정보 파일
Fig. 8 Union field usage information file generated from Figure 7

그림 7과 같은 공용체 변수를 생각해보자. un_a는 int형 필드 c와 int형 포인터 필드 k를 갖고 있고, 프로그램에서는 int형 필드 c만을 사용하고 있다. 만약, 3.2절과 비슷하게 단순히 공용체가 사용하는 필드 각각에 대해 재귀적으로 carving을 시도한다면, 본래 int형 값으로 해석되어야 할 값을 특정 메모리 영역에 대한 주소로 해석하여 그 내부에 저장된 값을 carving하려 시도하는 문제가 발생할 것이다.

해결방안 3. 이 문제를 해결하기 위해서는, 3.2.절의 해결 방식에 더해서 실제로 공용체가 프로그램 실행 중에 어떤 필드를 사용했었는지의 정보를 알아내고, 해당 필드만을 재귀적으로 carving해야 한다. 이는 단순히 코드를 정적 분석하여 알아낼 수 없는 동적인 정보이므로 이를 극복하기 위해 CR 도구는 2장에서 언급했던 바와 같이 두 번의 패스를 이용한다. 첫 번째 패스에서 공용체가 시스템 테스트 케이스 실행 상에서 어떤 필드를 실제로 사용하는지를 추적하고, 두 번째 패스에서 이 실제로 사용하는 필드에 대한 재귀적 carving을 한다.

공용체가 실제 실행 중에 어떤 필드를 사용하는지의 정보를 추적하기 위해 첫 번째 패스에서 일어나는 일은 구체적으로 다음과 같다. CR 도구는 타겟 프로그램의 모든 구문을 분석하여 공용체 변수에 대한 접근이 일어나는 구문을 찾아낸다. 그리고, 해당 구문이 접근하는 공용체 변수의 이름과 사용하는 필드의 이름을 파일에 기록하는 코드를 구문 뒤에 삽입해 준다. 이렇게 생성된 프로그램을 원하는 시스템 테스트 케이스를 이용해 실행하면 공용체 변수들이 각각 어떤 필드를 사용했는지의 정보가 적힌 "공용체 필드 사용 정보" 파일이 생성된다. 위 예시에 대해 이 작업을 진행하면, 그림 8과 같은 "공용체 필드 사용 정보" 파일이 생성된다. 기록된 것이 의미하는 것은 '공용체 변수 un_a가 c 필드를 사용하였다'이다.

```
function GenValueSaver(N, T)
input: N: the name of the variable to carve
       T: the type of the variable to carve
output: source code that will be instrumented
to target program to carve the target variable
program:
 1 code := emptyString
 2 ...
 3 if T is "union" then
 4   usedField := GetUnionUsage(N)
 5   foreach field of "union" do
 6     if nameOfField == usedField then
 7       newN := N + "." + nameOfField
 8       newT := typeOfField
 9       code += GenValueSaver(newN, newT)
10     endif
11   endfor
12 endif
13 ...
14 return code
```

그림 9 Carving을 위한 코드를 생성하는 *GenValueSaver* 함수의 공용체 처리 분기

Fig. 9 Branch in *GenValueSaver* function for generating code to carve union type

2장에서 언급했듯이, CR 도구는 이 파일을 이용해 두 번째 패스에서 실제 carving 코드를 생성하게 된다. 두 번째 패스에서 CR 도구는 가장 먼저 "공용체 필드 사용 정보" 파일을 읽고 공용체 변수 이름과 사용하는 필드의 대응 관계를 생성한 뒤 타겟 프로그램에 코드 삽입을 시작한다. 이 코드 삽입 과정은 그림 9에서 보듯이 3.2절의 구조체 carving 코드 삽입 과정과 비슷하게 진행된다.

그림 9는 3.2절에서 설명했던 *GenValueSaver* 함수의 공용체 타입 처리 분기를 나타내고 있다. 대부분의 구조가 3.2절의 그림 6에서 보인 구조체 carving 코드 생성 과정과 비슷하나, 4번째 줄 및 6번째 줄에서 차이를 보인다. 4번째 줄은 carving하고자 하는 공용체 변수가 어떤 필드를 사용했는지의 정보를 얻어오는 부분으로, *GetUnionUsage* 함수는 공용체 변수의 이름을 받아 해당 변수가 어떤 필드를 사용하는지를 돌려준다. 한편, 6번째 줄은 공용체가 사용하는 필드만을 carving하도록 하기 위한 부분으로, 공용체의 필드 중 carving하고자 하는 변수가 사용하는 필드의 경우에만 carving 코드를 생성하도록 한다.

*GenValueSaver*를 그림 7의 예시에 대해 적용하면 다음과 같다. 먼저, CR 도구는 첫 번째 패스에서 생성된 그림 8의 "공용체 필드 사용 정보" 파일을 읽어 un_a 변수가 c 필드를 사용중이란 것을 저장해둔다. 이후, *GenValueSaver*를 un_a라는 변수명과 union un이라는 타입 정보를 이용해 호출한다. 타겟 변수가 공용체 타입이기 때문에 그림 9의 3번째 줄 조건문에서 참을 반환하게 되고, 4번째 줄에서 *GetUnionUsage* 함수는 앞서 얻어둔 공용체 필드 사용 정보를 참조하여 un_a 변수가 c 필드를 사용하고 있음을 반환한다. 결론적으로 7번째에서 9번째 줄은 union un의 int형 필드 c에 대해서만 실행되고, un_a를 carving하기 위한 코드에는un_a.c의 변수명과 int형 타입 정보를 통해 호출한 *GenValueSaver*가 반환한 코드만이 추가될 것이다.

3.4 void 포인터의 carving

문제점 4. C 프로그램에서 void 포인터는 사용 전에 다른 타입의 포인터 형으로 변환되어 쓰이는 일이 많다. 그렇기 때문에 void형 포인터가 가리키는 공간의 값을 단순히 바이너리로 덤프하여 저장하면 문제가 발생할 수 있다. 예를 들어, void형 포인터 변수가 실 사용시에는 어떤 구조체의 포인터 형으로 변환되어 사용되고, 해당 구조체 타입에 포인터형 필드가 존재하게 되면, 이 포인터 형 필드가 가리키는 공간의 값을 저장하지 못하게 되고, 이는 replay 시에 에러를 발생시킨다.

조금 더 구체적인 예시로 3.2절의 예시를 살짝 변형한 그림 10의 프로그램을 살펴보자. 위 프로그램에서 void형 포인터 변수 p가 가리키는 공간의 값, 즉 *p를 단순히 바이너리로 덤프하면 3.2절과 비슷하게 ((struct st *)p)->c와 ((struct st *)p)->k의 값만이 저장될 뿐, ((struct st *)p)->k가 가리키는 공간의 값은 전혀 저장되지 않는다.

```
struct st {
  int c;
  int *k;
} st_a;
st_a.c = 3;
st_a.k = (int *)malloc(sizeof(int));
*(st_a.k) = 4;
void *p = &st_a;
printf("%d\n", *(((struct st *)p)->k));
```

그림 10 void 포인터 변수의 변환과 사용

Fig. 10 Type casting of void pointer variable

해결방안 4. 이 문제를 해결하기 위해서는 void포인터 변수가 실제 시스템 실행 상에서 어떤 타입의 포인터 형으로 변환되어 사용되는지의 정보를 미리 얻어 이 변수를 그 포인터 형으로 간주하여 carving해야 한다. 이 정보를 얻어내는 방법은 3.3절의 공용체 필드 사용 정보를 얻어내는 방식과 비슷하게 CR도구의 첫 번째 패스를 이용한다. 첫 번째 pass에서 CR 도구는 우선 타겟 프로그램의 구문 중 포인터 변환 작업을 수행하는 모든 구문을 찾아낸다. 그리고, 이들 중 void형 포인터 타입을 변환하는 구문에 대해 변환되는 void형 포인터

```
p - struct st *
```

그림 11 그림 10에 대해 생성된 void 포인터 변환 정보 파일
Fig. 11 Void pointer casting information file generated from Figure 10

```
function GenValueSaver(N, T)
input: N: the name of the variable to carve
       T: the type of the variable to carve
output: source code that will be instrumented
to target program to carve the target variable
program:
 1 code := emptyString
 2 ...
 3 if T is voidPointer then
 4   newT := GetVoidPointerUsage(N)
 5   code += GenValueSaver(N, newT)
 6 endif
 7 ...
 8 return code
```

그림 12 Carving을 위한 코드를 생성하는 *GenValueSaver* 함수의 void 포인터 처리 분기
Fig. 12 Branch in *GenValueSaver* function for generating code to carve void pointer type

변수의 이름과 변환될 목적 포인터 타입의 이름을 파일에 기록하는 코드를 해당 구문 뒤에 삽입해준다. 이를 통해 생성된 프로그램을 실행하면, 해당 실행에서 각 void 포인터 변수가 어떤 포인터로 변환되어 사용되는지의 정보가 적힌 “void 포인터 변환 정보” 파일이 생성된다. 그림 11은 그림 10의 예시에 대해 생성한 “void 포인터 변환 정보” 파일로, 파일의 내용이 의미하는 것은 ‘void 포인터 변수 p가 struct st의 포인터로 변환되어 사용된다’이다.

그림 12는 두 번째 패스에서 사용하는 *GenValueSaver* 함수의 void 포인터 처리 분기를 나타낸 것이다. 4번째 줄의 *GetVoidPointerUsage* 함수는 공용체의 경우와 비슷하게 carving하고자 하는 void 포인터 변수가 어떤 타입으로 변환되어 사용되었는지를 돌려주는 함수이다. 이 함수는 CR 도구의 두 번째 패스 가장 처음에서 “void 포인터 변환 정보” 파일을 읽어 알아낸 정보를 이용한다. 4번째 줄에서 carving할 void포인터 변수의 실 사용 타입을 알아낸 뒤, 5번째 줄은 이 변수를 해당 타입으로 간주하여 *GenValueSaver*를 다시 호출한다. 즉, 앞서 보여준 그림 10의 예시에서는 void포인터 변수 p를 struct st의 포인터 타입으로 간주하여 *GenValueSaver*를 호출하는 것이다. 이 방법을 이용하면, CR 도구가 재귀적으로 모든 필드를 carving하는 코드를 생성하게 되므로, 앞서 언급한 문제가 사라지게 된다.

4. 현재 버전 CR 도구의 한계

본 장에서는 현재 버전의 CR 도구가 여전히 가지고 있는 한계점에 대해서 서술한다.

4.1 실행 시간

포인터형 변수가 가리키고 있는 메모리 영역의 크기를 동적으로 알아내는 과정(3.1)에서 할당된 메모리의 크기가 큰 경우 긴 시간이 소요된다(초당 약 400 원소). 이는 메모리 영역의 크기를 알아내는 과정에서 주어진 주소 값으로부터 순차적으로 메모리가 할당되었는지 아닌지를 모두 검사하기 때문이다.

4.2 다중 파일 빌드 시스템

현재 버전의 CR 도구는 단일 소스 코드로 이루어진 타겟 프로그램에 대해서만 적용할 수 있다. 즉, 여러 소스 코드 파일을 컴파일하고 링크하여 만들어지는 타겟 프로그램에 대해서는 정상적으로 적용될 수 없다. 이는 CR도구가 단일 소스 파일을 가정하고 제작되었기 때문으로, CR도구는 현재 여러 파일에 존재하는 같은 이름의 static 전역 변수가 있는 등의 다양한 문제 상황에서 carving & replay를 제대로 수행하지 못한다. 이 부분을 수정하여 CR도구가 다중 파일 빌드 시스템을 지원할 수 있도록 만드는 것이 향후 추가적으로 연구해야 할 부분이다.

4.3 첫 번째 패스에서 생성된 파일의 유효성 문제

함수가 입력으로 공용체/void 포인터 변수의 값을 다른 공용체/void 포인터 변수에 할당한 뒤 사용한다면, 현재 버전의 CR 도구는 이를 정상적으로 carving할 수 없다. 예를 들어, 그림 13과 같은 프로그램이 있다고 가정하자. 여기서 타겟 함수 foo는 입력으로 void 포인터 변수 x를 받는다. 2번째 줄에서는 지역 변수 k를 선언하여 x의 값을 할당하고, 3번째 줄에서는 k를 int형 포인터로 간주하여 포인터 참조를 실행한다. 결국, void 포인터 변수 x는 int형 포인터로 간주되어 사용되는 것이고, 따라서 CR도구의 첫 번째 패스를 수행하여 생성되는 “void 포인터 변환 정보” 파일에는 x가 int형 포인터로 변환된다는 정보가 저장되어야 한다. 그러나, 실제 변환이 되는 변수가 k이므로 “void 포인터 변환 정보” 파일에는 k가 int형 포인터로 변환된다는 정보만이 저장된다. 이 정보만을 가지고 CR도구는 두 번째 패스에서

```
1 int foo(void *x) {
2   void *k = x;
3   printf(“%d\n”, *((int *)k));
4 }
```

그림 13 void 포인터를 지역 변수를 통해 사용하는 경우
Fig. 13 Using void pointer by assigning the value to a local variable

carving & replay를 위한 코드를 생성하게 되는데, 그 과정에서 변수 k와 x가 사실은 같다는 것을 알아낼 방법이 없고, 때문에 void 포인터 x가 어떤 형으로 변환되어 사용되는지 알아낼 수가 없다.

5. 실험 설계

5.1 연구 문제(Research Questions)

CR 도구는 시스템 실행에서의 특정 유닛의 행동을 모방하는 유닛 테스트 케이스를 생성하는 도구이다. 이 도구의 유효성을 검증하기 위해서 먼저 아래의 연구 질문이 주어진다.

RQ1. CR 도구의 유효성 검증: Replay의 실행 결과가, 시스템 테스트 케이스 실행 상에서 타겟 함수가 출력하는 결과와 일치하는가?

한편, 단순히 실행 결과만을 비교하는 것으로는 제대로 CR을 해냈는지를 보장할 수 없다. 때문에 아래의 연구 질문이 추가로 필요하다.

RQ2. Carving의 유효성 검증: Carving된 데이터가 저장된 디렉토리에 타겟 함수가 접근하는 모든 변수의 데이터가 저장되었는가?

Carving하여 저장된 데이터는 각 변수별로 변수의 이름과 같은 이름을 가진 파일이 생성되기 때문에 carving된 데이터가 저장된 디렉토리의 파일명을 살펴보는 것으로 해당 연구 질문에 답할 수 있다.

RQ3. Replay의 유효성 검증: 시스템 테스트 실행에서 나타나는 유닛 내의 구문 실행 순서와 replay 실행에서 나타나는 유닛 내의 구문 실행 순서가 일치하는가?

이 연구 질문은 디버깅 도구인 gdb의 breakpoint 기능을 이용하여 실행 구문을 추적하고, 이를 비교하는 것으로 답할 수 있다.

5.2 실험 설계

CR 도구는 코드의 정적 분석과 코드 삽입을 위해 LLVM/Clang 4.0을 이용해 개발하였다. 데이터의 Carving에서 이루어지는 바이너리 덤핑을 위해서는 C언어용 바이너리 직렬화 라이브러리인 tpl 라이브러리[7]를 이용하였다.

실험의 타겟 프로그램은 Unix 유틸리티인 sed의 1.18 버전을 선택하였고, 타겟 함수는 read_file을 선택하였다. sed는 GNU에서 제공하는 스트림 에디터[8]로 입력으로 주어진 텍스트를 주어진 규칙에 따라 변환하여 출력해주는 프로그램이다. read_file은 sed의 주 작업을 처리하는 함수로, 타겟 파일의 이름을 인자로 받아 해당 파일의 각 라인에 대해 주어진 규칙을 적용하여 그 결과를 출력해주는 함수이다. sed의 read_file을 실험의 대상으로 선택한 이유는 함수의 출력이 표준 출력으로 나타나서 출력의 동일성을 비교하기 쉽기 때문이다. 현재

```
Some r
andom te
xt
```

그림 14 sed에 인자로 넘길 텍스트 파일 text.txt
Fig. 14 A text file text.txt for applying sed

```
s/^.\{3\}//
```

그림 15 sed에 인자로 넘길 규칙 스크립트 파일 script.txt
Fig. 15 A script file script.txt which is a rule that will be applied to text file by sed

```
e r
om te
xt
```

그림 16 그림 14와 그림 15를 이용해 sed를 실행한 결과
Fig. 16 Result of sed using Figures 14 and 15

버전의 CR 도구가 다중 파일 빌드 시스템을 지원하지 않기 때문에, sed 1.18의 모든 소스 c 파일들을 하나의 파일로 합쳐 단일 파일 빌드 시스템으로 고쳐 적용하였다.

sed에 인자로 넘길 텍스트 파일은 그림 14와 같으며, 텍스트 변환 규칙 스크립트 파일은 그림 15와 같다. 그림 15의 규칙은 입력으로 받은 텍스트의 각 라인이 세 글자를 넘어간다면 가장 앞의 세 글자를 제거하는 것이다. 따라서, 그림 14의 텍스트 파일과 그림 15의 스크립트 파일을 이용해 sed를 실행한 결과는 그림 16과 같다.

6. 실험 결과

CR 도구의 첫 번째 패스로부터 “공용체 필드 사용 정보” 파일과 “void 포인터 변환 정보” 파일이 생성되었다. sed의 read_file에서는 void 포인터를 사용하지 않기 때문에, “void 포인터 변환 정보” 파일은 빈 파일로 생성되었다. “공용체 필드 사용 정보” 파일은 그림 17과 같이 생성되었는데, 중복된 내용이 여러번 기록된 것을 확인할 수 있다. 이는 공용체의 필드 접근이 일어날 때마다 해당 정보를 기록하기 때문으로, 공용체의 필드를 접근하는 부분이 반복문에 속해 있다면 발생할 수 있다. 이러한 경우, CR도구는 중복되는 항목들을 무시한다.

CR도구의 두 번째 패스를 통해 carving된 데이터는 그림 18과 같다. 여기서 name.tpl 파일은 read_file이 인자로 받는 char형 포인터 변수의 값을 저장하고 있으며, 나머지 파일은 모두 read_file이 접근하는 전역 변수의 값을 저장하고 있다. append.alloc.tpl 과 같이 파일 명에 둘 이상의 점(.)을 포함하는 파일은 구조체 변수의 각 필드별 carving이 작동한 결과로, append.alloc.tpl은 append라는 구조체 변수의 alloc이라는 필드 값을 저장

```
cur_cmd->x - cmd_regex
cur_cmd->x - cmd_regex
cur_cmd->x - cmd_regex
cur_cmd->x - cmd_regex
cur_cmd->x - cmd_regex
cur_cmd->x - cmd_regex
cur_cmd->x - cmd_regex
cur_cmd->x - cmd_regex
cur_cmd->x - cmd_regex
cur_cmd->x - cmd_regex
```

그림 17 생성된 공용체 필드 사용 정보 파일
Fig. 17 Generated union field usage information file

```
0x7ffd0e3f75c6  append.alloc.tpl        line.text.tpl
0x7ffd0e3f75dd  append.length.tpl       myname.tpl
0xaf4cd0        append.text.tpl         name.tpl
0xaf4d20        bad_input.tpl           no_default_output.tpl
0xaf5c90        hold.alloc.tpl          quit_cmd.tpl
0xaf5d70        hold.length.tpl         regs.end.tpl
0xaf5dd0        hold.text.tpl           regs.num_regs.tpl
0xaf5e30        input_file.tpl          regs.start.tpl
0xaf5ea0        input_line_number.tpl   replaced.tpl
0xaf6410        line.alloc.tpl          the_program.tpl
0xaf6450        line.length.tpl
```

그림 18 Carving된 데이터
Fig. 18 Carved data

```
e r
om te
xt
```

그림 19 Replay 결과
Fig. 19 Result of replay

하고 있다. 0x로 시작하는 이름을 갖는 디렉토리들은 포인터 변수가 가리키는 공간에 저장된 값을 저장하고 있는 디렉토리이다. 예를 들어, read_file의 인자로 주어진 char형 포인터 변수 name은 0x7ffd0e3f75dd의 값을 갖고 있는데, 해당 주소에 저장되어 있는 값은 주소값과 같은 이름을 갖는 디렉토리, 즉 0x7ffd0e3f75dd 디렉토리에 저장되어 있다.

마지막으로 그림 19는 그림 18의 데이터를 이용한 replay의 결과이다.

세 개의 연구 질문에 대한 대답은 다음과 같다.

RQ1: read_file이 출력하는 결과는 곧 sed의 결과와 동일하므로, 그림 16의 결과는 원본 시스템 테스트 케이스에서 read_file이 출력하는 결과이기도 하다. 그리고 그림 19에서 제시된 결과는 원본 시스템 테스트 케이스 실행인 그림 16과 동일하다. 이는 CR이 정상적으로 되었다는 의미이며, CR도구가 유효하다고 할 수 있다.

RQ2: 그림 18의 데이터를 보면 read_file의 인자로 주어지는 name을 비롯해 read_file에서 접근하는 모든 전역 변수들이 저장되어 있음을 알 수 있다. 따라서 CR 도구의 carving이 유효하다

RQ3: gdb를 이용해 구문 실행을 추적한 결과, 시스템 테스트 실행 및 replay 실행에서 유닛 내의 구문이 동일하게 실행되고 있음을 확인할 수 있었다. 따라서 CR도구의 replay가 유효하다.

7. 관련 연구

Carving & Replay와 관련된 몇 가지의 연구들이 이미 존재한다[9-14]. 이 중 일부는 디버깅을 돕거나 오류 상황을 재현하기 위해 개발되었고[9-11], 일부는 스레드 스케줄링 같은 동시성의 재현을 위해 개발되었다[12-14]. ADDA [9]는 사용자가 지정한 외부 이벤트만을 수집해 오류 상황을 재연하는 데에 이용하기 때문에 유닛 테스팅을 직접적으로 지원하지 않는다. 반면, 본 논문에서 제시하는 CR 도구의 경우 유닛 테스팅을 직접적으로 지원한다. ReCrash[10]는 타겟 Java 함수의 인자만을 수집하지만, 본 논문의 도구는 타겟 C 함수의 인자 뿐 아니라 접근하는 전역 변수까지 수집하여 함수의 실행을 정확히 재연해낸다. BugRedux[11]는 조건문에 포함된 전역 변수와 지역 변수의 값을 수집하고, 이를 이용해 입력값을 생성하여 결론적으로 동일한 분기 선택을 유도해낸다. 그러나, BugRedux는 재연을 정확히 못하는 경우가 있는 반면에 (실험에서 사용한 프로그램 실행 중 10%를 재연해내는데 실패했다), 본 논문에서 제시하는 도구는 유닛의 행동을 정확히 재연해낸다. 동시성과 관련된 연구들은 본 논문과 목적으로 하는 부분이 다르므로 직접적으로 비교하기 어렵다.

8. 결론 및 향후 연구

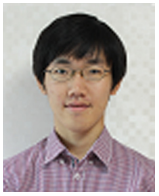
C언어에서 CR을 하는 데에는 C언어의 특성에서 기인하는 여러 가지 문제가 발생한다. 본 논문은 이 문제들을 해결하여 C언어용 CR 도구를 제시하였고, 이를 GNU sed에 대해 적용하여 성공적으로 작동함을 보였다. 그러나, 여전히 제시한 도구에 한계점들이 존재하므로, 추가적인 연구를 통하여 이를 해결해야 한다. 특히, 4.2절에서 제시된 문제의 경우, CR도구의 활용성에 있어서 중요한 문제이기 때문에 빠르게 해결하여야만 한다. 4.3절에 제시된 문제의 경우, 변수들간의 할당 정보를 모두 파일에 기록하여, 이 기록된 정보를 역산해내는 방식으로 해결할 수 있을 것이라 기대하고 있다.

References

[1] Malm and Bach-Sørensen, *Automated Unit Testing*, pp.2, Department of Computer Science, Aalbord University 2008.

[2] Elbaum et al., "Carving and replaying differential unit test cases from system test cases," *TSE*, Vol.

35, No. 1, pp. 29-45, 2009.
[3] Kim et al., "Automated unit testing of large industrial embedded software using concolic testing," *ASE 2013,* pp. 519-528, 2013.
[4] Tauro et al., "Object Serialization: A Study of Techniques of Implementing Binary Serialization in C++, Java and .NET," *International Journal of Computer Applications,* Vol. 45, No. 6, 2012.
[5] GitHub google/sanitizers wiki [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer, 2017.
[6] Serebryany et al., "AddressSanitizer: A Fast Address Sanity Checker," *USENIX Annual Technical Conference*, 2012.
[7] tpl home page [Online]. Available: https://troydhanson.github.io/tpl/, 2017.
[8] GNU sed manual page [Online]. Available: https://www.gnu.org/software/sed/manual/sed.html, 2018.
[9] Clause and Orso, "A technique for enabling and supporting debugging of field failures," *ICSE 2007,* pp. 261-270, 2007.
[10] Artzi et al., "ReCrash: Making software failures reproducible by preserving object states," *ECOOP 2008*, pp. 542-565, 2008.
[11] Jin and Orso, "Bugredux: Reproducing field failures for in-house debugging," *ICSE 2012*, pp. 474-484, 2012.
[12] Liu et al., "Light: Replay via tightly bounded recording," *PLDI 2015*, pp. 55-64, 2015.
[13] O'Callahan et al., "Engineering record and replay for deployability," *USENIX Annual Technical Conference 2017*, pp. 377-389, 2017.
[14] Mashtizadeh et al., "Towards practical default-on multi-core record/replay," *ASPLOS 2017*, pp. 693-708, 2017.



임 현 수
2017년 KAIST 전산학과 졸업(학사). 2017년~현재 KAIST 전산학과 석사과정. 관심분야는 자동화된 Concolic 유닛 테스팅

김 윤 호
정보과학회논문지
제 45 권 제 4 호 참조

김 문 주
정보과학회논문지
제 45 권 제 4 호 참조