# 시스템 테스트 케이스를 이용한 C 프로그램의 동적 유닛 입력 값 자동 수집 및 재연 기술

임현수○, 김윤호, 김문주
한국과학기술원 전산학부
bookman01@kaist.ac.kr, yunho.kim03@gmail.com, moonzoo@cs.kaist.ac.kr

# Automated Capturing & Replaying Dynamic Unit Inputs of C Programs from System Executions

Hyunsu Lim○, Yunho Kim, Moonzoo Kim
School of Computing, KAIST

**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 C언어에서는 자체적인 직렬화 방법이 존재하지 않을뿐더러 포인터 변수를 사용하는 언어 자체의 특성상 CR에 어려움이 있다. 본 논문에서는 C언어의 CR에서 어떠한 문제들이 존재하는지를 살펴보고 그 문제들을 해결하여 C언어용 CR 도구를 제시한다.

## 1. 서론

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 infeasible 테스트 케이스인 경우가 발생할 수 있다.

이러한 infeasible 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 역 직렬화 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 feasible한 테스트 케이스이므로 이를 시드 유닛 테스트 케이스로 삼아 콘콜릭 테스팅등의 기법을 이용하면 많은 양의 feasible 유닛 테스트 케이스를 찾아낼 수 있다[3].

이 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원(NRF-2016R1A2B4008113), 정부(과학기술정보통신부)의 재원으로 한국연구재단-차세대 정보 컴퓨팅기술개발사업의 지원(NRF-2017M3C4A7068177), 과학기술정보통신부 및 정보통신기술진흥센터의 SW중심대학지원사업의 지원(2016-0-00018), 정부(교육부)의 재원으로 한국연구재단의 지원(NRF-2017R1D1A1B03035851), KAIST High Risk High Return Project(HRHRP) 사업의 지원을 받아 수행한 연구임.

Java 언어의 경우, 이미 serializable 인터페이스를 통해 객체를 직렬화하는 방법을 제공하고 있다. 그러나, C언어의 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.

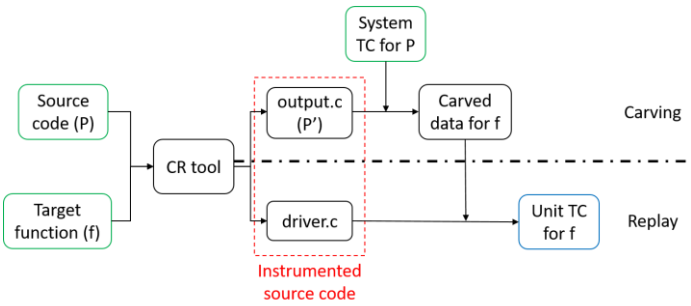


그림 1. CR 도구의 흐름도

## 2. CR 도구의 작동 과정

CR 도구를 이용해 시스템 테스트 케이스로부터 유닛 테스트 케이스를 얻는 과정은 그림 1과 같다. 이는 크게 두 단계인 시스템 테스트 케이스의 실행 과정에서 타겟 함수가 호출되는 순간의 프로그램 상태를 직렬화하여 저장하는 carving 단계와 이 직렬화된 프로그램 상태를 읽어 들여 역 직렬화를 수행해 타겟 함수의 입력 형식으로 바꾸어 주는 replay 단계로 나누어진다.

먼저, CR 도구는 사용자로부터 타겟 함수의 이름과 타겟 프로그램의 소스 코드를 입력받는다. CR 도구는 이 입력으로부터 carving과 replay를 수행하는 두 개의 소스 코드를 생성해낸다.

생성된 소스 코드 중 carving을 수행하는 코드는 `output.c`로, 이는 입력으로 주어진 소스 코드의 타겟 함수에 carving을 위한 코드를 삽입하여 만들어진다. 따라서, 실행하였을 시 carving된 데이터가 저장되는 것 이외에는 입력으로 주어진 소스 코드와 출력 등이 같다.

생성된 소스 코드 중 replay를 수행하는 코드는 `driver.c`로, 이는 위 `output.c`를 통해 carving된 데이터를 읽어들여 타겟 함수의 실제 입력값으로 변환하고 이를 이용해 함수를 실행하는 역할을 수행한다.

두 소스 코드가 생성되고 나면, 사용자는 `output.c`를 컴파일 하고, 생성된 프로그램을 이용하여 시스템 테스트 케이스를 실행한다. `output.c`는 시스템 테스트 케이스가 실행되는 과정에서 타겟 함수가 호출될 때마다 해당 함수가 접근하는 모든 전역 변수와 함수에 넘겨진 인자의 값을 읽고, 이를 binary 형태로 덤프한 뒤 별도의 디렉토리에 저장한다. 저장되는 디렉토리는 함수가 호출될 때마다 새로 생성된다.

`output.c`와 시스템 테스트 케이스를 이용한 carving이 완료되면, `driver.c`를 컴파일하고 생성된 프로그램을 이용하여 carving된 데이터를 유닛 테스트 케이스로 변환할 수 있다.

## 3. C언어의 CR에서 발생하는 문제와 해결 방법

### 3.1. 포인터에 할당된 메모리 크기

Carving을 위해서는 포인터 변수가 가리키는 공간에 저장된 데이터 모두를 읽어내야 하므로 할당된 메모리 공간의 크기를 알아내는 방법이 필요하다. C언어에서 포인터 변수에 할당된 메모리 공간의 크기를 알아내는 간단한 방법은 존재하지 않으므로, 포인터 변수가 가리키는 메모리 공간의 크기를 알아내기 위해 clang 플러그인 AddressSanitizer (ASan)를 사용하였다[5][6]. ASan은 Use-after-free나 Double-free 등 메모리와 관련된 다양한 버그를 감지하는 플러그인으로, ASan의 API에는 특정 주소 값의 메모리 할당 여부를 알아내는 함수가 존재한다. 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.

### 3.2. 동일 주소를 가리키는 포인터

Carving 과 replay를 진행할 때 기존에는 존재하던 포인터 변수 간의 의존 관계가 사라지는 문제가 발생한다. 즉, 서로 다른 포인터 변수가 같은 메모리 공간을 가리키는 경우 각각의 포인터 변수, 그리고 그와 관련된 정보를 단순히 바이너리 형태로 덤프하게 되면, 같은 데이터가 두 번 저장되어 공간 활용이 비효율적이 되는 문제와 replay를 진행할 때 기존에는 존재하던 포인터 변수 간의 의존 관계가 사라지는 문제가 발생한다.

이 의존 관계를 유지하기 위해 포인터 변수를 carving할 때는 다음과 같은 방법을 따른다. 우선 포인터 변수의 값(특정 메모리 공간에 대한 주소 값)을 저장하고 해당 값을 이름으로 하는 디렉토리가 존재하는지 확인한다. 만약 존재하지 않는 경우, 그러한 디렉토리를 새로 생성하고 포인터가 가리키는 공간에 있는 값을 해당 디렉토리 내에 저장한다. 다중 포인터의 경우는 이 방법을 재귀적으로 적용한다.

위의 방식을 이용하면 특정 주소에 저장된 값은 단 한 번만 carving되며, 후에 replay를 진행할 때도 포인터가 가리키는 주소 값이 같다는 정보와 해당 주소 값에 저장된 데이터가 모두 존재하기 때문에 포인터 간의 의존 관계를 유지하면서 replay를 수행해 낼 수 있다.

### 3.3. 구조체의 carving

구조체 타입의 변수 값을 carving할 때, 해당 변수를 그냥 바이너리 형태로 덤프하게 되면 크게 두 가지 문제가 발생한다.

첫 번째로, 구조체에 포인터형 필드가 존재할 경우, 이 포인터가 가리키는 메모리 영역의 값을 저장하지 않게 된다. 이에 따라 후에 carving된 데이터를 replay할 때, 실제 유닛의 상태를 그대로 모방할 수 없게 되는 문제가 발생한다.

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

이 문제를 해결하기 위해 구조체 타입의 경우엔 각 필드를 따로 carving하도록 하였다.

## 4. 한계

### 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. void 포인터의 처리

현재 버전의 CR 도구는 void 포인터형 변수가 존재하고 이것이 후에 다른 타입의 데이터 포인터형으로 캐스팅되어 사용될 때 이를 처리할 수 없다. 이는 void 포인터형인 변수를 carving할 때, 해당 변수가 가리키는 메모리 공간의 값을 단순히 읽어 바이너리로 덤핑하기 때문이다. 예를 들어, void 포인터형 변수이지만 실 사용 시에는 어떤 구조체의 포인터로 변환하여 사용하는 경우, 그 구조체 내부에 포인터형인 필드가 존재한다면 해당 필드가 가리키고 있는 메모리 영역의 값은 carving할 수 없고, 이에 따라 replay시에 에러가 발생하게 된다.

```
//... (include statements)
typedef struct { int i; char *str; } st;
void target_func(int *p1, int *p2, st s) {
  printf("p1 = %p, p2 = %p\n", p1, p2);
  printf("*p1 = %d, *p2 = %d\n\n", *p1, *p2);
  *p1 = 285714;
  printf("p1 = %p, p2 = %p\n", p1, p2);
  printf("*p1 = %d, *p2 = %d\n\n", *p1, *p2);
  printf("s.i = %d, s.str = %s\n", s.i, s.str);
  printf("Address of s.str = %p\n", s.str);
}
int main(int argc, char **argv) {
  st st_a = {1, "A string"};
  int id = 142857, *p1 = &id, *p2 = &id;
  printf("target func call\n");
  target_func(p1, p2, st_a);
  printf("target func called\n");
  return 0;
}
```

그림 2. 타겟 함수 `target_func`를 포함하는 `test.c`

## 5. 실험 설계

CR 도구는 LLVM/Clang 4.0을 기반으로 하여 개발했으며, 포인터가 가리키는 메모리의 크기를 알아내기 위해 clang plugin인 AddressSanitizer를 사용하였다. 또한, 데이터의 바이너리 덤핑을 위해 C용 바이너리 직렬화 라이브러리인 tpl 라이브러리[7]를 사용하였다.

실험 과정은 다음과 같다. 먼저, 3장에서 제시된 문제들이 존재하는 함수를 포함하는 프로그램(그림 2)을 작성한다. 이 프로그램에 존재하는 타겟 함수 `target_func`는 세 인자 `p1`, `p2`, `s`를 받아 이들의 자세한 정보를 출력하여 준다. `p1`과 `p2`는 int 포인터 변수로 같은 주소값이 주어지고, `s`는 char 포인터 필드 `str`과 int 필드 `i`를 포함하는 구조체 변수이다. 이 프로그램에 CR 도구를 적용한 결과를 이용해 답하고자 하는 연구 문제(research questions, RQ)는 다음과 같다.

**RQ1. Carving & Replay가 정상적으로 진행되었는가?** 이는 CR 도구의 유효성을 확인한다. Carving시와 replay시의 타겟 함수의 출력으로부터 유추해낸 타겟 함수 진입 시점의 메모리 그래프가 주소 값을 제외하고 동일하다면 CR이 정상적으로 진행된 것이다.

**RQ2. 포인터 변수가 가리키는 공간에 저장된 값이 모두 carving 되었는가?** 이는 3.1절의 해결법이 유효한지를 확인한다. `s.str`의 값이 carving시와 replay시에 동일하게 출력된다면 char 포인터가 가리키는 공간의 값을 모두 읽어온 것이다.

**RQ3. 포인터 변수 간의 의존 관계가 replay시에 유지되는가?** 이는 3.2절의 해결법이 유효한지를 확인한다. `p1`과 `p2`의 값이 replay시에 동일한 주소값이라면 의존 관계가 유지되는 것이다.

**RQ4. 구조체 변수가 필드별로 carving 되었으며 정상적으로 replay 되는가?** 이는 3.3절의 해결법이 유효한지를 확인한다. 먼저 carving된 데이터의 디렉토리에 `s`의 필드별 파일이 존재한다면 구조체 변수가 필드별로 carving된 것이며, carving시와 replay시의 `s`의 필드값이 동일하다면 replay가 정상적으로 되었다고 할 수 있다.

## 6. 실험 결과

그림 3은 CR 도구로 생성된 `output.c`의 실행 결과와 그 실행을 통해 생성된 디렉토리의 구조이며, 그림 4는 CR 도구로 생성된 `driver.c`의 실행 결과이다. 각각의 RQ에 대한 결론은 다음과 같다.

**RQ1**. 그림 5는 `output`과 `driver`의 출력으로부터 유추해낸 carving 및 replay시 타겟 함수 진입 시점의 메모리 그래프이다. 두 그래프가 주소값을 제외한 부분이 동일하므로 CR이 정상적으로 진행되었음을 알 수 있다.

**RQ2**. `output`과 `driver`의 실행 결과에서 `s.str`이 가리키는 공간에 저장된 값이 **"A string"**으로 동일하다. 따라서 3.1절의 해결법이 유효하다.

**RQ3**. `driver`의 실행 결과에서 `p1`과 `p2`의 주소 값이 같음을 볼 수 있다. 따라서 3.2절의 해결법이 유효하다.

**RQ4**. Carving된 데이터의 디렉토리에 `3param.i.tpl`과 `3param.str.tpl`이 존재하는 것으로 `s`가 필드별로 carving되었음을 알 수 있으며, `output`과 `driver`의 실행 결과에서 `s.str`과 `s.i`의 값이 같음을 확인할 수 있다. 따라서 3.3절의 해결법이 유효하다.

## 7. 결론 및 향후 연구

C언어에서 CR을 하는 데에는 C언어의 특성에서 기인하는 여러 가지 문제가 발생한다. 본 논문은 이 문제들을 해결하여 C언어용 CR 도구를 제시한다. 그러나, 여전히 제시한 도구에 한계점들이 존재하므로, 추가적인 연구를 통하여 이를 해결해야 한다. 예를 들어 4.2절에서 제시된 문제의 경우, 함수 내에서 void 포인터형 변수가 캐스팅되는 포인터 타입을 알아내어 이를 이용해 carving 코드를 생성하는 방식 등의 해결법이 있을 수 있다.

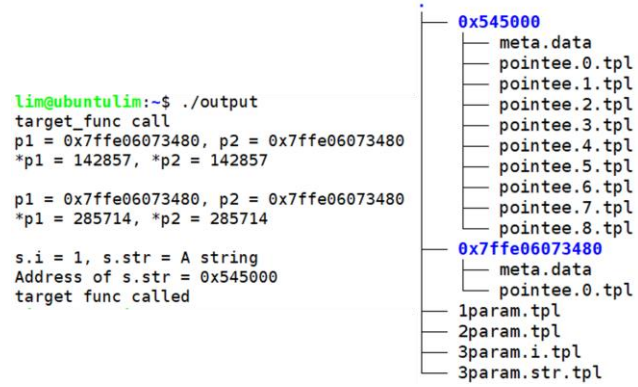


그림 3. output의 실행 결과(좌) 및 carving된 데이터의 디렉토리 구조(우)

```
lim@ubuntulim:~$ ./driver test-0
p1 = 0xe3b010, p2 = 0xe3b010
*p1 = 142857, *p2 = 142857

p1 = 0xe3b010, p2 = 0xe3b010
*p1 = 285714, *p2 = 285714

s.i = 1, s.str = A string
Address of s.str = 0xe3b030
```

그림 4. driver의 실행 결과

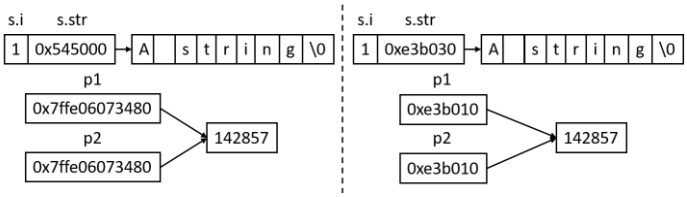


그림 5. Carving시 타겟 함수 진입 시점의 메모리 그래프(좌)와 replay시 타겟 함수 진입 시점의 메모리 그래프(우)

## 참 고 문 헌


[1] Malm and Bach-Sørensen. "Automated Unit Testing." 2008.

[2] Elbaum et al. "Carving and replaying differential unit test cases from system test cases." TSE, 35(1), pp.29-45, 2009

[3] Kim et al. "Automated unit testing of large industrial embedded software using concolic testing." ASE 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", 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", https://troydhanson.github.io/tpl/, 2017.