석 사 학 위 논 문
Master's Thesis

시스템에서 추출한 동적 유닛 컨텍스트를 이용한 효과적인 Concolic 유닛 테스팅

Effective Concolic Unit Testing With Dynamic Unit Contexts Carved from System Tests

2019

임 현 수 (林 炫 秀 Lim, Hyunsu)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

시스템에서 추출한 동적 유닛 컨텍스트를 이용한 효과적인 Concolic 유닛 테스팅

2019

임 현 수

한 국 과 학 기 술 원

전산학부

시스템에서 추출한 동적 유닛 컨텍스트를 이용한 효과적인 Concolic 유닛 테스팅

임 현 수

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2018년 12월 10일

심사위원장 김 문 주 (인)

심 사 위 원 이 흥 규 (인)

심 사 위 원 양 은 호 (인)

Effective Concolic Unit Testing With Dynamic Unit Contexts Carved from System Tests

Hyunsu Lim

Advisor: Moonzoo Kim

A dissertation 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 Computer Science

Daejeon, Korea
December 10, 2018

Approved by

Moonzoo Kim
Professor of Computer Science

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 thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS
20173493

임현수. 시스템에서 추출한 동적 유닛 컨텍스트를 이용한 효과적인 Concolic 유닛 테스팅. 전산학부 . 2019년. 32+iv 쪽. 지도교수: 김문주. (영문 논문)
Hyunsu Lim. Effective Concolic Unit Testing With Dynamic Unit Contexts Carved from System Tests. School of Computing . 2019. 32+iv pages. Advisor: Moonzoo Kim. (Text in English)

초 록

Concolic 테스팅과 같은 자동 유닛 테스팅 기술은 다양한 유닛 테스트 실행을 통해 유닛 테스팅의 이점을 향상시킨다. 그러나, 현재의 자동 유닛 테스팅 기술은 기술적인 한계로 인해 시스템 테스트로부터 동적 유닛 컨텍스트(목적 함수가 읽는 모든 전역 변수 및 함수 인자의 값)를 추출해내지 못하고, 이에 담긴 정보를 활용하지 못하고 있다.

이를 해결하기 위해 복잡한 C 프로그램을 위한 새로운 Concolic 유닛 테스팅 도구인 CUT^2 를 개발했다. 첫번째로, CUT^2 는 시스템 테스트로부터 f의 동적 유닛 컨텍스트를 추출해내는 탐지 코드를 목적 프로그램 P에 삽입한다. 이후, CUT^2 는 탐지 코드가 삽입된 프로그램을 시스템 테스트를 통해 실행하여 f의 동적 유닛 컨텍스트를 추출한다. 여기서 f의 동적 유닛 컨텍스트를 정확하게 추출하기 위해 CUT^2 는 f의 인자 뿐만 아니라 f에 의해 호출되는 함수 및 f에서 읽히는 전역 변수도 추출해낸다. CUT^2 는 추출된 f의 동적 유닛 컨텍스트를 기반으로 하여 심볼릭 테스트 드라이버 및 스텁을 생성하고 f의 동적 유닛 컨텍스트를 최초 입력값으로 하여 Concolic 테스팅을 수행한다.

CoREBench를 사용한 실험에서 CUT^2 는 평균적으로 약 90%의 분기 커버리지를 달성하였으며 이는 기존의 Concolic 유닛 테스팅 기술에 비해 평균적으로 최소 10.9%p의 개선을 이뤄낸 것이다.

핵 심 낱 말 유닛 테스팅, Concolic 테스팅, 동적 유닛 컨텍스트, 실행 캡처 및 추출

Abstract

Automated unit testing techniques like concolic unit testing improve the benefits of unit testing through diverse unit test executions. However, current automated unit testing techniques do not utilize valuable information on *dynamic unit contexts* (DUCs) (i.e., values of all parameters and global variables read by a target function) in system tests due to the technical difficulty to extract them from system tests.

I have developed a new concolic unit testing framework CUT^2 for complex C programs. First, CUT^2 instruments a target program P to insert probes that capture/carve DUCs of f from system tests. Second, CUT^2 carves DUCs of f while executing the instrumented target program with system tests. At this step, to carve DUCs of f accurately, CUT^2 carves not only parameters of f but also global variables updated by f and f's descendant functions. Third, CUT^2 generates a symbolic test driver and stubs that build symbolic search space based on the carved DUCs of f and performs concolic unit testing on f using the carved DUCs as initial test inputs for f.

In the experiments on CoREBench, CUT^2 achieves around 90% branch coverage on average, which is at least 10.9%p higher than the existing concolic unit testing techniques on average.

Keywords Unit testing, Concolic testing, Dynamic unit contexts, Capturing/carving executions

Contents

List of Tables

List of Figures

Chapter 1. Introduction

As software is widely adopted in our daily lives, the reliability of the software has become a critical issue. Software testing is a major method to ensure the reliability of the software. Software testing runs the target software with a set of test inputs and checks if each output of the software meets predefined conditions. Note that for successful testing, it is important to create a sufficient number of test inputs with high quality, since each test input explores a different behavior of the target program.

Unit testing is one of a software testing method that focuses on *unit*, not a whole program (i.e. system-level testing). In unit testing, developers isolate a target function f from a target program by stubbing/mocking out the other functions and test only the function f. Therefore, unit testing costs less than the system-level testing.

However, even though the complexity of unit is smaller than that of the whole system, still it is almost impossible for developers to manually create test inputs which handle every exceptional case in unit testing. Moreover, as the size and complexity of the software grows, that of an unit is also increasing which boosts the cost of unit testing. Thus, to reduce the man-power for testing and test units more exhaustively, automated unit testing techniques such as random unit testing or symbolic unit testing are proposed.

1.1 Previous Approaches

Random testing is the most naïve approach in automated testing technique. As the name itself mentions, random testing randomly generates test inputs for system-unter-test. This technique can generate a large number of test inputs, but the quality of the generated test inputs is not guaranteed.

Search-based testing is a technique to automatically generate test inputs that achieve specific testing goals (i.e., cover specific branch, etc.). It first starts with a random test input and selects the better neighbour according to the fitness or cost function. In other words, the fitness/cost function guides the search-based testing to find test inputs that meet the specific goals. Therefore, it is important to design the fitness/cost function properly in search-based testing.

Symbolic execution is an automated testing technique that aims to cover every branch in system-under-test. It sets the inputs as symbolic values and executes every feasible paths in the system-under-test symbolically. At the end of the execution, path constraint of each path is obtained and test inputs are generated by solving such constraints (i.e., finding the concrete values that satisfy the constraint). This technique is able to automatically generate test inputs that cover every feasible branches in system-under-test. However, symbolic execution suffers from the path explosion problem which is critical for the scalability of the technique.

Concolic (CONCrete + symbOLIC) testing is a technique that combines the concrete execution with symbolic execution. In concolic testing, first, the system-under-test is ran concretely with a test input. During the execution, the technique collects the branch conditions that the execution path goes through to generate path constraint for the input. At the end of the execution, some terms in the path constraint is negated to generate a new path constraint, and a new test input is generated by solving the generated path constraint. This process is repeated until the feasible branchs in the system-under-test is fully covered or time is out. The concolic testing technique is widely adopted in various areas

recently [2, 3, 4, 5], but none of the concolic unit testing techniques focus on utilizing dynamic unit contexts of a target function.

1.2 Thesis Statement and Contributions

1.2.1 Thesis Statement

The thesis statement of this dissertation is as follows:

> Dynamic unit contexts carved from system tests can improve the effectiveness of automated concolic unit testing.

1.2.2 Proposed Approach

To validate the thesis statement, this dissertation suggests CUT^2 (Concolic Unit Testing with Carved dynamic Unit conTexts), which is an automated concolic unit testing technique for complex real-world C programs. For a target function f, CUT^2 first carves the *dynamic unit contexts* (DUCs) of f from system test case executions (Section 2.3). DUC consists of the values of following items, which can affect the execution of f:

- Parameters of f at the entry point of f
- Global variables that are read by f and descendants of f at the entry point of f
- Non-`const` pointer type parameters of direct callees g of f at the exit point of g
- For direct callee g of f, global variables that are read by g and descendants of g at the exit point of g
- Return value of direct callees of f

Based on the carved DUCs, CUT^2 generates symbolic drivers/stubs which declare all variables in DUC as symbolic ones for concolic unit testing of f (Section 2.5). Then, CUT^2 guides concolic unit testing to generate effective test inputs by utilizing a meaningful DUC as an initial test input. In other words, CUT^2 initializes each symbolic variable with the concrete values in DUC.

1.2.3 Technical Challenges and Solutions

There were several challenges when implementing CUT^2 . Mainly, the challenges are caused by the C programming language's characteristics, which tackle the carving of DUCs. The main technical challenges are as follows:

- **Challenge 1:** Obtaining the size of memory region referred by the pointer.

  In order to carve DUCs accurately, the memory region referred by the pointer variable must be carved precisely. Thus, it is necessary to obtain the exact size of memory region referred by the pointer. However, the C programming language does not support direct method to acquire such data.

- **Challenge 2:** Carving `void` pointer type variables.

  Usually, `void` pointers are casted to some other types of pointers before it is used. For example, a `void` pointer can be casted to the pointer of a `struct` type before it is dereferenced. In such case, the memory region referred by the pointer should be carved recursively (i.e., in the aformentioned example case, the `struct` type data in memory region pointed by the `void` pointer should be carved recursively). Therefore, it is necessary to know which type does the `void` pointer is casted to in the system execution. However, this information is not available at the source code level.

- **Challenge 3:** Carving `union` type variables.

  Each field of `union` type variable is placed on the same position of the memory. Therefore, it is necessary to know which field of the `union` type variable is used in the system execution for accurate carving (Note that we cannot just simply dump the memory region of `union` variable because if the `union` variable uses non-primitive type fields such as `struct` type field, then this field should be carved recursively). However, it is not possible to gain such information at the source code level.

CUT^2 solves these challenges as follows:

- **Solution 1:** Obtaining the size of memory region referred by the pointer (Section 2.4).

  CUT^2 instruments a target program to build the information of the memory region that is used by the target program by tracking every memory allocation/deallocation in the target program. This information helps to find the exact size of memory region pointed by the pointer.

- **Solution 2:** Carving `void` pointer type variables (Section 2.3.3).

  A preprocessing step (Step 1 and 2 of Section 2.2) is added to acquire the type that `void` pointers are casted to in the system execution. During the instrumentation step (Step 3 of Section 2.2), this `void` *pointer casting information* is used to generate the code for carving `void` pointer variables.

- **Solution 3:** Carving `union` type variables (Section 2.3.5).

  Analogous to the above solution (i.e., Solution 2), the information about the fields that are used by `union` variables in the system execution is obtained in the preprocessing step. During the instrumentation step, this `union` *field usage information* is used to generate the code for carving `union` variables.

1.2.4 Contributions

The contributions of this dissertation are as follows:

- A new DUC carving tool which handles complex features of the C programming language (Section 2.3).
  - The tool can obtain the size of memory region referred by pointers, which is not directly supported by the C programming language (Section 2.4).
  - The tool properly carves various pointer types (Section 2.3.3).
    - `void` pointer, which could be casted to another type of pointer before it is used, is correctly carved.
    - Opaque `FILE` pointer, which shouldn't be dereferenced directly, is successfully carved.

  - The tool properly carves `union` type, which has multiple fields on the same position of memory (Section 2.3.5).

- Empirical demonstration of the effectiveness of CUT^2 by applying CUT^2 on complex real-world C programs and showing the improved branch coverage compared to the existing techniques (Section 3).
  - In experiments targeting 7 out of 22 faulty revisions of CoREBench, CUT^2 achieves at least 10.9%p higher branch coverage compared to other techniques (i.e., SUT, SUT', OTF, and CUT^{2-} (see the Section 3.1.3)).

1.3 Structure of the Dissertation

The remainder of this dissertation is structured as follows. Chapter 2 presents CUT^2 , the automated concolic unit testing technique for complex real-world C programs in detail. Chapter 3 presents the empirical evaluation of CUT^2 on 7 faulty Coreutil program versions in CoREBench. Chapter 4 presents the related work for automated concolic unit testing and carving/replaying techniques. Finally, Chapter 5 concludes the dissertation with future work.

Chapter 2. Concolic Unit Testing with Carved dynamic Unit conTexts (CUT^2) Technique

2.1 Motivating Example

```
01 ...
02 #define N 100
03 typedef struct _Node{
04   char *str;
05   struct _Node *next;
06 }Node;
07 Node *data=NULL;
08 void input_processing(const char *);
09
10 int main() {
11   char input_str[N];
12   FILE *fp = fopen(...);
13   fgets(input_str, N, fp);
14   parser(input_str);
15   ...}
16
17 size_t my_strlen(const char *str) {
18   const char *s;
19   for (s = str; *s !=NULL ; ++s);
20   return(s - str);}
21
22 // A target function
23 int parser(char *input){
24   // input should be at least 80 characters
25   if (my_strlen(input)< 80){ exit(1);}
26
27   // Checking the input string format
28   if (my_strncmp(input, "HEAD|", 5) != 0){
29     printf("INVALID INPUT\n"); return -1;}
30   if(my_strncmp(input+5,"START0|",7)!=0){
31     printf("INVALID INPUT\n"); return -1;}
32   // 98 more checking conditions
33   ...
34
35   input_processing(input);
36
37   // Checking validity of the processed data
38   for (Node *n=data; n!=NULL; n=n->next){
39     if (strstr(input, n->str)==0){ //not found
40       printf("INVALID DATA\n");return -2;}}
41
42   // Main procedure on data
43   ...
44   ...
45   return 0;}
46
47 void input_processing(const char *const_input){
48   char *token, *input=strdup(const_input);
49   Node *curr;
50
51   token=strtok(input,"|");
52   if (token!=NULL){
53     data=malloc(sizeof(Node))
54     data->str=strdup(token);
55     data->next=NULL;
56     curr=data;}
57   for (token=strtok(NULL,"|"); token!=NULL;
58                       token=strtok(NULL,"|")){
59     Node *new_node=malloc(sizeof(Node));
60     new_node->str=strdup(token);
61     new_node->next=NULL;
62     curr->next=new_node;
63     curr=curr->next;}
64   free(input);}
```

Figure 2.1: A motivating example where concolic unit testing fails to generate test inputs that cover the main procedure of `parser`

2.1.1 Description of the Example

Figure 2.1 shows a string parser example where concolic testing fails to generate useful test cases. Suppose that `parser` (Lines 23–45) is a target function to test. `parser` receives a pointer `input` to an

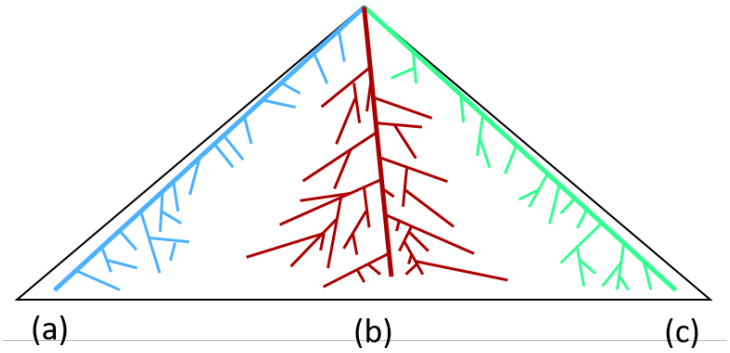

Figure 2.2: Execution paths that are covered by concolic testing according to the initial input. (b) covers larger area than (a) or (c)

input string (Line 23) and performs the following tasks:

1. Checking a length of an input string (Line 25):
   It checks if the length of the input string is greater than or equal to 80 using `my_strlen`.

2. Checking the input string format with 100 rules (Lines 27-33):
   It checks if the input string conforms to the given formats. For example, it checks if the prefix of the input string matches "`HEAD`" (Line 28).

3. Generating a linked list `data` by processing the input string (Line 35):
   It creates a global linked list whose head node is pointed by `data` (declared at Line 7) using its callee function `input_processing` (defined in Lines 47–64). While `input_processing` reads string tokens (separated by the '`|`' character) from `input` one by one (Lines 57–63)[1], it creates corresponding new nodes (Line 59) whose `str` contains a string token read from `input` (Line 60), and links the nodes by setting a `next` pointer to the next node (Line 62).

4. Checking validity of the generated `data` (Lines 37–40):
   It checks if, for every node in the linked list pointed by `data`, a string of `str` appears in `input`. Otherwise (i.e., the correspondence between `data` and `input` is broken), `data` is invalid and `parser` returns -2.

5. Executing the main procedure on `data` (Lines 42-44)
   Finally, it executes the main procedure code to retrieve and update `data`. I assume that this code section is large and complex and, thus, concolic testing should generate test inputs that reach this section for high test coverage.

2.1.2 Limitation of Concolic Unit Testing

Suppose that I apply concolic unit testing to `parser` like conventional concolic unit testing (e.g., CONBOL [2]). Also, let me assume DFS search strategy and symbolic variables initialized with 0s like most concolic execution techniques. Then, concolic unit testing generates hundreds of invalid input

[1] `strtok(str, sep)` reads a string token separated by `sep` characters from a string `str` and returns a pointer to the string token. `strtok` keeps track of a stream position of `str` and `strtok(NULL, sep)` reads from `str` a next string token. `strdup(str)` duplicates a string `str` by allocating memory to contain `str` and returns the allocated memory address.

strings and fails to generate useful test inputs that reach the main procedure of `parser` for the following reasons (Case of (a) or (c) in Figure 2.2):

First, it will generate 80 invalid input strings that do not reach the main procedure of `parser`. Concolic testing keeps generating invalid input strings of increasing lengths (i.e., 0,1,...,79) because of the loop on symbolic input string in `my_strlen` (Line 19). Second, suppose that concolic unit testing reaches Line 27. Still it keeps generating additional several hundred invalid input strings due to the 100 string format checkers in Lines 27-33.

In contrast, if concolic unit testing starts with an initial input string that reaches the main procedure of `parser` (as in (b) of Figure 2.2), it will generate many test inputs that explore the main procedure from the beginning. I can obtain such useful initial input strings from system tests since system tests usually explore normal/main test scenarios.

Note that, to obtain DUCs of a target function f accurately, I have to carve not only parameters of f but also *global variables updated by the descendant functions of* f. For example, I have to carve not only `input` of `parser`, but also global states (i.e., `data`) updated by `input_processing` at the exit point of `input_processing`. If I do not carve and utilize `data` updated by `input_processing`, another validity check at Lines 37–40 will fail and concolic testing will not generate unit test inputs that reach the main procedure of `parser` (Lines 42–44).

2.2 Overview

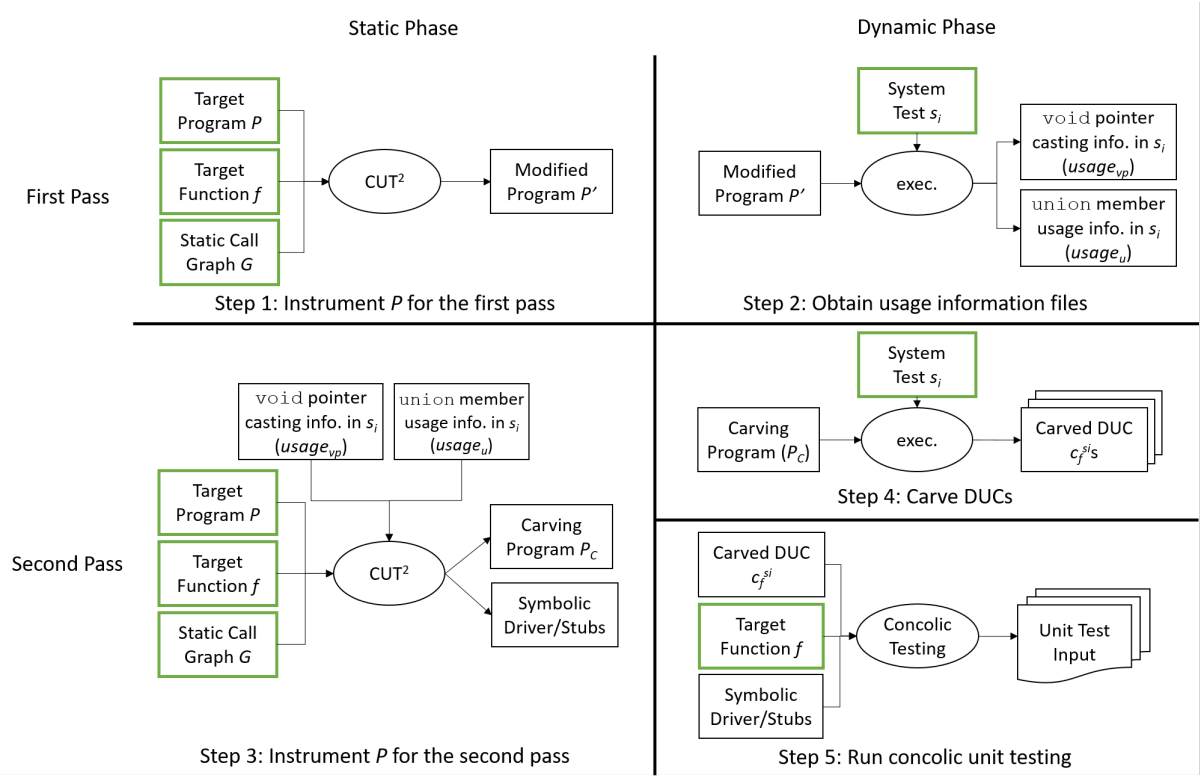

Figure 2.3: Overall process of CUT²

Figure 2.3 shows the overall process of CUT² that carves and utilizes dynamic unit contexts of a target function f in a system test s_i. A *dynamic unit context* (DUC) of f consists of the values of

```
01 union un {
02   int a;
03   char b;
04 }
05
06 int foo(void *p, union un u) {
07   char *c = (char *)p;
08   printf("%s %c\n", c, u.b);
09 }
```

Figure 2.4: Example program for Step 1

parameters and global variables read by f at the entry point of f and at the exit points of the direct callee functions of f. CUT^2 stores all values of variables comprising a DUC of f in files during a system test execution of P. For a pointer variable, the value of the pointer variable and the values in the valid memory region referred by the pointer are stored together.

CUT^2 receives the following inputs and runs in the following two passes:

- source code of a target program P
- a target function f
- a static call graph G of P
- a set of system tests TC

The first pass identifies actual types of data pointed by `void` pointers and which member variable of `union` variable is actually used in a system test s_i. This is because the type of data pointed by a `void` pointer and a member variable accessed by a `union` variable may change at runtime and the type information is necessary to generate and insert probe code to carve target data. In the second pass, CUT^2 carves DUCs of f during executing P with a system test s_i. CUT^2 operates in the following way:

- **Step 1.** CUT^2 instruments P and generates the modified program P' as follows:
  - For every `void` pointer variable vp read by f or updated by the descendant functions of f, CUT^2 identifies actual type of the data pointed by vp by finding casting expression on vp. CUT^2 inserts the probe code that writes the type casting information of a `void` pointer variable to the file $usage_{vp}$ right after the type casting expression.

    For example of Figure 2.4, CUT^2 inserts the probe at the end of Line 7 to report the type of data pointed by a `void` pointer `p` at Line 7 is `char`.
  - For every `union` variable u read by f or updated by the descendant functions of f, CUT^2 identifies the member variable of u that is actually accessed by finding the member accessing expression on u in P. CUT^2 inserts the probe code that writes the accessed member information of a `union` variable to the file $usage_u$ right after the member access of u.

    For example of Figure 2.4, CUT^2 inserts the probe at the end of Line 8 to report that the `char` member b of u is used.

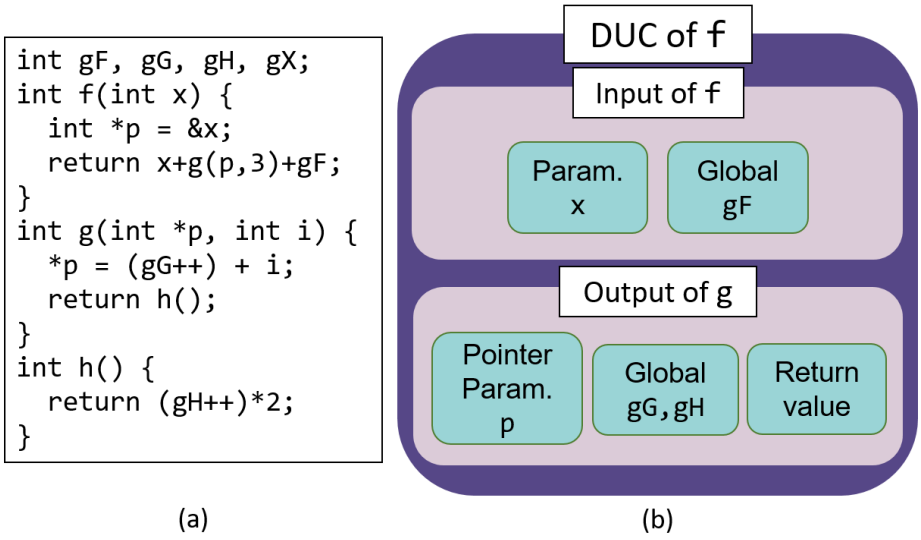

Figure 2.5: Example of DUC. (a) is an example code with target function `f` and (b) is the DUC of `f` of (a).

- **Step 2.** The modified program P' runs with a system test $s_i \in TC$ and generates two files $usage_{vp}$ and $usage_u$. $usage_{vp}$ stores type casting information on `void` pointer variables and $usage_u$ stores member access information of `union` variables.
- **Step 3.** CUT^2 generates the carving program P_C that carves DUCs of f during the system tests of P as follows (Section 2.3):
  - CUT^2 first identifies 1) function parameters of f and 2) global variables read by f. The code for carving variables of 1) and 2) is generated and inserted at the *entry point* of f.
  - For each direct callee g of f, the following code is generated and inserted at the *exit points* of g:
    - The code for carving every global variable updated by g and its descendant functions
    - The code for carving non-`const` pointer parameters
    - The code for carving the return value if g returns a value

    In addition, CUT^2 generates symbolic test drivers/stubs for f which declare all carved variables as symbolic ones and initialize the symbolic variables with the carved variable values.
- **Step 4.** P_C executes with a system test $s_i \in TC$ and generates the carved DUC $c_f^{s_i}$s where $c_f^{s_i}$ consists of the following items (Figure 2.5 shows the example of DUC):
  1. The values of the function parameters of f at the entry point of f.
  2. The values of the global variables at the entry point of f that are read by f and descendatns of f.
  3. For every direct callee g of f, the values of global variables at the exit points of g that are updated by g and the descendants of g.
  4. For every direct callee g of f, the non-`const` pointer parameters at the exit points of g (to carve local data of f that can be updated by g through the pointer parameters of g).

```
01 int len_v = sizeof(v)/sizeof(v[0]);
02 for (int i = 0; i < len_v; i++) {
03   ValueCarver(v[i], "int");
04 }
```

Figure 2.6: Example pseudo-code genreated for an array variable

5. For every direct callee g of f that returns a value, the return value of g.

If a single system test execution calls f several times, multiple DUCs of f are carved.

- **Step 5.** CUT^2 applies concolic unit testing to f (with the symbolic test drivers/stubs generated at Step 3 (see Section 2.5)) with a carved DUC $c_f^{s_i}$ as an initial test input.

2.3 Generation of Carving Program P_C

Algorithm 1 shows how CUT^2 generates the probe code that carves $c_f^{s_i}$ in high level. The generated probe code is inserted at the entry point of f or the exit points of f's direct callee functions. $ValueCarver(v, T_v)$ receives a variable v to carve and a type T_v of v and generates code that carves v according to T_v as follows:

2.3.1 Primitive variable (Lines 3–5)

The algorithm generates the code that simply stores the value of v to a file (i.e., `store_value(`v, T_v`)`).

2.3.2 Array Variable (Lines 6–10)

The algorithm applies itself recursively over every element of an array v. Figure 2.6 shows the pseudo-code generated by the algorithm for an integer array v (the **foreach** iterations of the algorithm are implemented as a loop as shown in Fig. 2.6).

2.3.3 Pointer Variable (Lines 11–28)

Not only the value of v itself but also the values in the memory region referred by the pointer v should be carved together. This is because carved DUCs from a system test s_i should maintain the relationship between a pointer and its pointee (i.e., v and $*v$) as well as the relationship between a pointer with a valid offset k and its pointee (i.e., $v+k$ and $*(v+k)$) as in s_i. Also, carved DUCs should maintain the relationship between pointers (e.g., pointer aliasing) as in s_i. If v is NULL or the pointee of v is already carved, only the value of v without its pointee is stored (Lines 12–13).

Figure 2.7 shows the example pseudo-code to carve a pointer variable `v` for integer(s) which Lines 23–26 of the algorithm generate. First, the code stores the value of a pointer v (i.e., the memory address contained in v). Then, the code recursively applies $ValueCarver$ to carve each element v_i of the *valid memory region* starting from v (the **foreach** iterations of the algorithm are actually implemented as a

```
01 store_value(v, "int *");
02 for (int i = 0; is_valid_memory(v+i); i++) {
03   ValueCarver(*(v+i), "int");
04 }
```

Figure 2.7: Example pseudo-code genreated for a pointer variable

```
01 struct st {
02   int x;
03   char y;
04   int *p;
05 } st_a;
06
07 ValueCarver(st_a.x, "int");
08 ValueCarver(st_a.y, "char");
09 ValueCarver(st_a.p, "int *");
```

Figure 2.8: Example pseudo-code genreated for a `st_a` of `struct st` type

loop as shown in Fig. 2.7). This validity checking of the memory region is one of the main challenges in developing CUT^2 (see Section 2.4).

ValueCarver handles `void` pointer and opague `FILE` pointer variables as follows since I cannot directly dereference such pointers (`FILE` pointer is a predefined system type which should not be dereferenced directly).

- `void` pointer (Lines 14–18): `void` pointer variables are usually casted to pointers of other types before dereference. Thus, CUT^2 first refers to $usage_{vp}$ to know to which type a `void` pointer is casted. Let $\hat{T}_v$ be the type that v is casted to. The algorithm declares a temporary variable v' of the type $\hat{T}_v$ and assigns v to v'. Then it calls $ValueCarver(v', \hat{T}_v)$.
- `FILE` pointer (Lines 19–21): *ValueCarver* stores the name of a file that a `FILE` pointer v refers to and the current stream position on the file. The name of a file that v refers to can be obtained by using a symbolic link under `/proc/self/fd` with v's file descriptor number and the current stream position of the file can be obtained by using `ftell`.

2.3.4 `struct` Variable (Lines 29–33)

ValueCarver carves every member of `struct` variable recursively. Figure 2.8 shows the definition of `struct st` and the example pseudo-code which carves `st_a` of `struct st` type. `struct st` consists of the three members and the algorithm generates code for carving each member.

2.3.5 `union` Variable (Lines 34–37)

Since the member variables of a `union` variable share the same memory, I have to know which member of a `union` variable is actually accessed in a system test execution. The code recursively carves only the accessed member of v after referring $usage_u$.

2.4 Memory Validity Checking

To carve a pointer variable correctly, the entire valid memory region pointed by the pointer should be carved. Thus, I have to identify the range of the valid memory region referred by the pointer. Since the C programming language does not provide a method to obtain the size of a pointed memory region, CUT^2 tracks every allocation/deallocation of the memory region in a target program and uses this information to obtain the size of the valid memory region pointed by a pointer variable to carve.

CUT^2 has a memory manager M which maintains memory allocation information by instrumenting a target program to track memory allocation/deallocation status. For stack and heap areas, it tracks memory allocation/deallocation as follows:

- Memory allocated at stack area has corresponding declaration in a target program. For every declared variable x, CUT^2 inserts the code that adds the memory region information on x to M right after the declaration statement of x. Also, it inserts the code that removes the memory region information on x from M right before the end of the scope of x.
- Memory at heap area is allocated by using `malloc` series library functions (e.g., `malloc`, `calloc`, `realloc`, `free`). I create hooks for the dynamic memory allocation/deallocation functions that insert/delete memory region information to/from M.

2.5 Generation of Symbolic Driver and Stubs

I explain how CUT^2 generates a symbolic driver and stubs to build symbolic search space based on a carved DUC through an example. Figure 2.9 shows a (simplified) symbolic test driver, a stub, and utility functions generated for the example in Fig. 2.1 as follows:

- `parser_test_driver` (Lines 1–6)
- `input_processing_stub` (Lines 8–10)
- `SYM_`T_v`_from_carved_ctx` (v_n) (Lines 12–39) which performs symbolic setting for a variable v (v_n is a name of v) of type T_v based on a carved DUC. For example, Fig. 2.9 has three such functions:
  - `SYM_NodeP_from_carved_ctx` which performs symbolic setting for a variable of `Node` pointer type (Lines 12–21)
  - `SYM_Node_from_carved_ctx` which performs symbolic setting for a variable of `Node` type (Lines 23-30)
  - `SYM_CharP_from_carved_ctx` which performs symbolic setting for a variable of `char` pointer type (Lines 32-39)

```
01 void parser_test_driver(void){
02   // symbolic input setting for data
03   data=SYM_NodeP_from_carved_ctx("0.data");
04   // symbolic input setting for the parameter 'input'
05   char *input=SYM_CharP_from_carved_ctx("input");
06   parser(input);}
07
08 void input_processing_stub(const char *const_input){
09   // symbolic input setting for data
10   data=SYM_NodeP_from_carved_ctx("1.data");}
11
12 Node *SYM_NodeP_from_carved_ctx(const char *name){
13   Node *mem = load_value(name);
14   if (mem!=NULL){
15     int len=get_pointee_size(name);
16     mem=malloc(sizeof(Node)*len);
17     for (int i=0; i<len; i++){
18       char elem_name[256];
19       snprintf(elem_name,256,"%s.%d",name,i);
20       *(mem+i)=SYM_Node_from_carved_ctx(elem_name);}}
21   return mem;}
22
23 Node SYM_Node_from_carved_ctx(const char *name){
24   Node ret;
25   char field_name[256];
26   snprintf(field_name,256,"%s.str",name);
27   ret.str=SYM_CharP_from_carved_ctx(field_name);
28   snprintf(field_name.256,"%s.next",name);
29   ret.next=SYM_NodeP_from_carved_ctx(field_name);
30   return ret;}
31
32 char *SYM_CharP_from_carved_ctx(const char *name){
33   char *mem = load_value(name);
34   if (mem!=NULL){
35     int len=get_pointee_size(name);
36     mem=malloc(sizeof(char)*len);
37     for (int i=0; i<len; i++){
38       SYM_char(mem+i, name);}}
39   return mem;}
```

Figure 2.9: The symbolic test driver/stub for the motivating example in Figure 2.1

Since a target function `parser` reads a global variable `data` and a parameter `input`, `parser_test_driver` declares those variables as symbolic based on a carved DUC. For example, it sets a global variable `data` of a `Node` pointer type symbolically using `SYM_NodeP_from_carved_ctx("0.data")`(Line 3) [2] and a parameter variable `input` of a character pointer type symbolically using `SYM_CharP_from_carved_ctx("input")`(Line 5).

`input` is set symbolically based on a carved DUC of `parser` (saying c_{input}) stored in files as follows.

[2]The prefix "0." of "`0.data`" indicates to use a value of `data` carved at the entry point of a target function `parser`. This is because a global variable can be carved at the entry point of a target function f and at the exit points of the callee functions g_i of f (see Step 4 of Sect. 2.2).

First, `SYM_CharP_from_carved_ctx("input")` reads the pointer value (i.e., a memory address) of `input` from c_{input} (Line 34). If the carved `input` value is not NULL, it obtains the size of the carved memory region pointed by `input` using `get_pointee_size` (Line 35). Then, it allocates memory (Line 36) and applies symbolic setting for each memory element in the allocated memory region using `SYM_char` (lines 37-38). `SYM_char(mem+i, name)` declares a memory location pointed by `mem+i` as a symbolic value of `char` type and initializes the memory location with the carved value of variable whose name is `name` in c_{input} as an initial test input of concolic testing.

`data` is set symbolically using `SYM_NodeP_from_carved_ctx` (Line 3) which is similar to `SYM_CharP_from_carved_ctx` (except creation of member variable names at Lines 18–19).

Algorithm 1: $ValueCarver(v, T_v)$

Input: v: a variable to carve, T_v: a type of v

Output: Source code that will be inserted to the target program to carve v.

```
1  code := empty string
2  switch T_v do
3      case primitive type do
4          code += store_value(v,T_v)
5      end
6      case array type do
7          foreach element v_i of v do
8              code += ValueCarver(v_i, T_{v_i})
9          end
10     end
11     case pointer type do
12         if v is NULL or the pointee of v is already carved then
13             code +=store_value(v,T_v)
14         else if T_v is void pointer type then
15             T̂_v := casted type of v (from usage_vp)
16             code += declaration of a temp. variable v' of type T̂_v
17             code += assign v to v'
18             code += ValueCarver(v', T̂_v)
19         else if T_v is FILE pointer type then
20             code += code that stores the name of a file that v refers to
21             code += code that stores the current stream position of the file
22         else
23             code += store_value(v,T_v)
24             foreach element v_i of the valid memory region pointed by v do
25                 code+= ValueCarver(v_i, T_{v_i})
26             end
27         end
28     end
29     case struct type do
30         foreach member v_i of v do
31             code += ValueCarver(v_i, T_{v_i})
32         end
33     end
34     case union type do
35         m_v := accessed member of v (from usage_u)
36         ValueCarver(m_v, T_{m_v})
37     end
38     return code
39 end
```

Chapter 3. Experiments and Results

3.1 Experiment Setup

I have designed the following research questions to evaluate effectiveness of CUT^2 in terms of branch coverage on the CoREBench C programs.

3.1.1 Research Questions

RQ1. Soundness of carved dynamic unit contexts (DUCs): Does a target function f with a DUC $c_f^{s_i}$ carved from a system test s_i execute in the same way to s_i?

For RQ1, I use `gdb` to extract a line trace of f (i.e., a sequence of the executed lines of f in the execution order) from a system test s_i and a line of f from $c_f^{s_i}$. Then, I check if these two line traces are same.

RQ2. Branch coverage of SUT, SUT', the on-the-fly concolic testing, and CUT^2 : How much does CUT^2 achieve branch coverage, compared to SUT, SUT', and the on-the-fly concolic testing?

For RQ2, I apply CUT^2 to a target function f with a DUC $c_f^{s_i}$ as an initial test input for all system tests s_is that execute f. For fair comparison, I apply SUT, SUT', and the on-the-fly concolic testing (see Section 3.1.3) to f with the same total amount of time spent by CUT^2 .

RQ3. Impact of the DUCs carved from the callee functions of f on branch coverage: How much does CUT^2 achieve branch coverage without the DUCs carved from the direct callee functions of a target function f?

For RQ3, I evaluate the impact of the DUC carved from the direct callee functions of f by comparing CUT^2 and CUT^2 without the DUC carved from the direct callee functions of f (i.e., without carving 3), 4), 5) items in Step 4 in Section 2.2).

3.1.2 Target Program Versions

I target the following 7 out of 22 faulty Coreutil program versions in CoREBench and exclude the following 15 revisions for the following reasons:

- Clang/LLVM cannot handle the following eight versions correctly due to Clang/LLVM file access bug (this bug was reported to the developers): `2e636af1`, `a6a447fc`, `f7f398a1`, `6124a384`, `b8108fd2`, `a860ca32`, `86e4b778`, `61de57cd`, `6fc0ccf7`
- The system tests involve a timing issue, which prevents a target unit with a carved DUC from executing in the same way to the system test: `8f976798`, `d461bfd2`
- The failing system test limits the memory size, which prevents CUT^2 from carving DUCs: `ec48bead`
- The target function is `main` which may be an inappropriate target function for unit testing: `5ee7d8f5`, `2238ab57`
- The prototype implementation of CUT^2 does not handle overlapped memory region correctly yet (which is a very rare case): `76f606a9`

Table 3.1: Target program versions and functions

ID	Target program	Target functions				
		Name	LoC	#Branches	#System tests	Branch Cov (%)
core.62543570	cp	copy_reg	401	86	72	55.8
		copy_internal	1033	212	116	58.5
core.b54b47f9	cut	set_fields	180	92	295	57.6
		cut_fields	117	42	175	66.7
core.be7932e8	cut	set_fields	196	94	287	58.5
		cut_fields	114	44	175	68.2
core.06aeeecb	cut	set_fields	195	94	203	58.5
		cut_fields	115	44	109	68.2
core.a04ddb8d	ls	print_color_indicator	118	68	22	46.3
		quote_name	587	186	64	58.9
core.51a8f707	od	print_char	11	2	1	100.0
		decode_one_format	254	88	5	48.9
core.64d4a280	seq	scan_arg	41	22	36	68.2
		print_numbers	55	20	33	65.0
Average			244.1	78.1	113.8	62.8

For each target program version, I target two functions: the faulty function and the most complex function in terms of cyclomatic complexity. If the most complex function is the faulty function or `main`, I target the next most complex function.

Table 3.1 describes 7 target program versions including a program version ID, a target program name, a target function name, a size of a target function in LoC and a number of branches, a number of system tests that reach the target function, and the branch coverage of the target function achieved by the system tests. For all target program versions, I used all passing test cases provided in the Coreutils package and all failing test cases provided in CoREBench.

3.1.3 Concolic Unit Testing Techniques to Compare

I have compared CUT^2 with the following techniques:

- *Symbolic Unit Testing (SUT)*: It generates a symbolic unit test driver with symbolic arguments to a target function f and symbolic global variables wihtout any constraints on the symbolic values, like CONBOL [2]. Also, SUT uses symbolic stubs that return a unconstrained symbolic value to replace all direct callee functions of f.

- *SUT'*: It is same to SUT but I measure the branch coverage of SUT' using the branches covered by the test cases generated by SUT and the branches covered by the given system tests together (for fair comparison with CUT^2 which utilizes system tests).

- *On-the-fly concolic testing (OTF)*: On-the-fly concolic testing performs concrete execution until a target function f is invoked. When f is first executed, it sets all parameters and all global variables

read by f as symbolic and starts concolic testing from f. Thus, an initial context of f is obtained from the concrete values constructed by the concrete execution starting from `main` until f is first invoked. OTF utilizes given system tests as initial test inputs at system level. OTF represents a hybrid approach of symbolic unit testing and concrete system testing proposed by Păsăreanu et al. [6]

- *CUT^2 without the carved DUCs from the callee functions (CUT^{2-})*: CUT^{2-} is CUT^2 without the DUCs carved from the callee functions of f. CUT^{2-} uses symbolic stubs like SUT.

All concolic unit testing techniques use DFS (depth first search) strategy in the experiments.

3.1.4 Measurement

To answer RQ1, I write a `gdb` script that sets breakpoints at every line of f and prints the executed line whenever `gdb` reaches the breakpoints. I compare the two line traces using `diff`. To answer RQ2 and RQ3, I measure the branch coverage of SUT, SUT', OTF, CUT^2 , and CUT^{2-} . For fair comparison between SUT and CUT^2 which utilizes system tests, I measure the branch coverage of SUT' too.

3.1.5 Testbed Setting

For the experiments, I carve only the first DUC of f from a system test (i.e., one DUC per one system test). For fair comparison, I make the running times of CUT^2 , SUT, SUT', OTF and CUT^{2-} same. For CUT^2 , I set the maximum number of generated test cases as 1,000 for each carved DUC. I assign the same amount of the total time spent by CUT^2 (including the total amount of time spent to carve all DUCs and the total amount of time spent by concolic test generation with all DUCs) to SUT and SUT'. Also, I assign the same amount of the time spent by CUT^2 to CUT^{2-} for each system test.

All experiments were performed on machines equipped with Intel i5 3.6 Ghz CPU and 16 GB of memory running Debian Linux 8 64 bits. I ran one CoREBench docker instance on each machine.

3.1.6 Implementation

The CUT^2 prototype implementation is written in 628 lines of C++ code for the first pass instrumenting tool, and 4,644 lines of C++ code for the second pass instrumenting tool using Clang/LLVM 4.0 [7]. The memory manager library is written in 138 lines of C code.

For concolic testing, I use CROWN [8] which extends CREST [9] to support complex C features such as bitwise operators, `union`, bitfields, and so on.

3.1.7 Threats to Validity

A threat to external validity is the representativeness of the target programs. However, I believe that this threat is limited because Coreutils programs are real-world programs and widely used by many other researchers. Another threat to external validity is the possible bias of the system tests I used for the experiments. To reduce this threat, I utilized all system tests provided by CoREBench and Coreutils.

A threat to construct validity is the use of the line traces to show the soundness of carved DUCs. Two different execution paths might produce the same line trace if two or more branches are located in one line (mainly due to a macro expansion). To reduce this threat, if a target function has the source code line which contain two or more branches, I manually analyzed not only line traces but also execution

Table 3.2: Average size (byte) of DUCs carved at the entry point of target functions and at the exit points of the direct callee functions

ID	Name	Avgerage DUC size		ID	Name	Avgerage DUC size	
		Target func.	Callee func.			Target func.	Callee func.
62543570	copy_reg	2139.4	4.0	a04ddb8d	print_color_indicator	18658.6	1621.9
	copy_internal	5267.7	1651.5		quote_name	344.4	8273.6
b54b47f9	set_fields	1842.8	11.5	51a8f707	print_char	131222.0	8.0
	cut_fields	117.9	830.7		decode_one_format	1121.0	0.0
be7932e8	set_fields	1026.6	306.0	64d4a280	scan_arg	897.1	50.1
	cut_fields	93.5	400.1		print_numbers	115.2	57.3
06aeeecb	set_fields	2066.2	448.4	Average (except `print_char`)		2597.3	1071.2
	cut_fields	74.0	270.4				

Table 3.3: Average carving and concolic testing time of CUT[2] per DUC (seconds)

ID	Name	Carving	TC Gen.	Total	ID	Name	Carving	TC Gen.	Total
62543570	copy_reg	0.15	804.32	804.47	a04ddb8d	print_color_indicator	0.25	778.24	778.49
	copy_internal	0.12	1404.17	1404.29		quote_name	0.15	781.42	781.57
b54b47f9	set_fields	0.04	673.16	673.20	51a8f707	print_char	0.32	231.56	231.88
	cut_fields	0.10	701.03	701.13		decode_one_format	0.02	1102.19	1102.21
be7932e8	set_fields	0.03	701.42	701.45	64d4a280	scan_arg	0.03	405.13	405.16
	cut_fields	0.09	679.07	679.16		print_numbers	0.02	512.03	512.05
06aeeecb	set_fields	0.04	901.24	901.28	Average		0.10	795.00	795.11
	cut_fields	0.08	1455.05	1455.13					

paths using `gdb`. Also, in an extreme case, even if carved parameter or global variable values are different from those in the corresponding system test executions, the two line traces might be same.

A threat to internal validity is possible bugs in the implementations of CUT[2] and concolic unit testing techniques I applied. I extensively tested my implementations to address this threat to internal validity.

3.2 Experiment Result

This section analyzes the experiment results. For all comparison in the experiments in this section, I applied Wilcoxon test with a significance level 0.05 to show the statistical significance. All comparison results in this section are statistically significant unless mentioned otherwise.

3.2.1 Experiment Data

Table 3.2 shows the average size of the carved DUCs. For example, an amount of DUC carved at the entry point of `copy_internal` of `62543570` is 5267.7 bytes long and the one carved at the exit points of all direct callee functions of `copy_internal` of `62543570` is 1651.5 bytes long on average (see the left column of the third row). This table shows that the amount of a DUC carved by the callee functions

Table 3.4: Average length of line traces

ID	Name	Avg. Len. of traces	ID	Name	Avg. Len. of traces
62543570	copy_reg	60.0	a04ddb8d	print_color_indicator	223.2
	copy_internal	83.6		quote_name	127.9
b54b47f9	set_fields	342.0	51a8f707	print_char	1.0
	cut_fields	49.3		decode_one_format	18.0
be7932e8	set_fields	139.5	64d4a280	scan_arg	29
	cut_fields	47.5		print_numbers	35.2
06aeeecb	set_fields	174.5	Average		98.0
	cut_fields	40.9			

```
369 size_t initial = 1;
...
388 if (*fieldstr == '-')
403 else if (*fieldstr==','
404  ||isblank(to_uchar...))
...
540 for (i = 0; i < n_rp; i++)
547   rsi_candidate = ...;
...
553     mark_printable_field(j);
...
563 return field_found;
```

a) A line trace of `set_fields` in `cut.c` (`06aeeecb`) from **a system test execution**

```
369 size_t initial = 1;
...
388 if (*fieldstr == '-')
403 else if (*fieldstr==','
404  ||isblank(to_uchar...))
...
540 for (i=0;i<n_rp;i++)
547   rsi_candidate = ...;
...
553     mark_printable_field(j);
...
563 return field_found;
```

b) A line trace of `set_fields` in `cut.c` (`06aeeecb`) from **a unit test execution with the carved context**

Figure 3.1: Two line traces of `set_fields` in `cut.c` (`06aeeecb`) which are generated from a system test s_i and a unit test execution with DUC carved from s_i

of a target function f is less than the amount of a DUC carved at the entry point of f on average (i.e., 2597.3 bytes vs. 1071.2 bytes on average except an outlier `print_char` of `51a8f707`), but not negligible amount.

Table 3.3 shows the average execution time of CUT[2] including the carving time and the concolic test generation time per a carved DUC. For example, `copy_internal` of `62543570` takes 0.12 seconds to carve a DUC from a system test and 1404.17 seconds to generate 1,000 concolic test executions from the carved DUC as an initial test input on average (see the left column of the third row). On average, the carving time takes less than 0.013% (= 0.10/795.11) of the total concolic unit testing time. Thus, the overhead for carving DUCs is insignificant.

3.2.2 RQ1: Soundness of Carved DUCs

I have confirmed that the carved DUCs are sound because a line trace generated from a system test s_i and one from DUC carved from s_i are same for all 14 target functions with all 113.8 system tests per

Table 3.5: Branch coverage of SUT, SUT', OTF, and CUT^2 (%)

ID	Name	SUT	SUT'	OTF	CUT^2	ID	Name	SUT	SUT'	OTF	CUT^2
62543570	copy_reg	51.2	73.3	63.3	82.6	a04ddb8d	print_color_indicator	61.2	78.8	74.1	89.5
	copy_internal	53.3	71.7	67.1	89.2		quote_name	48.3	68.1	70.7	88.3
b54b47f9	set_fields	58.7	71.7	73.8	93.5	51a8f707	print_char	100.0	100.0	100.0	100.0
	cut_fields	45.2	81.0	84.5	88.1		decode_one_format	46.6	65.9	74.7	80.7
be7932e8	set_fields	60.6	80.9	79.2	93.6	64d4a280	scan_arg	63.6	86.4	86.4	86.4
	cut_fields	50.0	81.8	79.1	88.6		print_numbers	70.0	85.0	85.0	85.0
06aeeecb	set_fields	61.7	69.1	74.5	94.7	Average		58.6	78.3	78.0	89.2
	cut_fields	50.0	81.8	79.1	88.6						

target function on average (see Table 3.1). For example, Figure 3.1 shows a pair of the line traces of `set_fields` in `cut.c` (`06aeeecb`) generated from a system test and DUC carved from the system test, respectively. As shown in the figure, the two line traces in Figure 3.1 are exactly same to each other.

Table 3.4 shows the average length of the line traces of target functions. For example of `set_fields` of `cut` (ID:06aeeecb) (see the left column of the second last row in the table), the average length of the extracted line traces is 174.5 lines.

3.2.3 RQ2: Branch Coverage of SUT, SUT', OTF, and CUT^2

Table 3.5 shows the high branch coverage of CUT^2 (i.e., 89.2% for the target functions on average). Also the table shows that CUT^2 achieves higher branch coverage than SUT, SUT', and OTF. CUT^2 achieves 30.6%p (=89.2%-58.6%) higher branch coverage than SUT on average. When I compare CUT^2 with SUT', still CUT^2 achieves 10.9%p (=89.2%–78.3%) higher branch coverage on average. Comparing to OTF, again CUT^2 achieves 11.2%p (=89.2%-78.0%) higher branch coverage on average. Thus, I have confirmed that DUC based concolic unit testing (i.e., CUT^2) improves concolic unit test coverage in a large degree.

Figure 3.2 shows a concrete example showing the advantage of CUT^2 over the other concolic testing techniques. For a target function `copy_interal`, CUT^2 covers all 24 branches in Lines 1718, 1790, and 1836 while SUT covers none of them and OTF covers only 14 of them (the system tests cover only 10 of them). This is because those branches depend on `src_mode`, which is updated by the callee function `XSTAT` through a pointer parameter `src_sb` (Line 1640). For SUT, the symbolic stub replacing `XSTAT` does not set its arguments symbolic and fails to generate test inputs to reach those 24 branches. For OTF, it spent too much time to explore all callee functions (and their descendant) symbolically and fails to cover all these branches in given time budget. In contrast, CUT^2 covers all these branches by utilizing the carved DUCs from which CUT^2 generates symbolic search space rich enough to cover all those branches but focuses on a target function by not exploring the descendant functions symbolically (i.e., using symbolic stubs).

3.2.4 RQ3: Impact of the DUCs Carved from the Callee Functions of f on Branch Coverage

Table 3.6 shows that DUCs carved from the callee functions of a target function are crucial to achieve high test coverage. CUT^2 achieves 11.4%p (=89.2%-77.8%) higher branch coverage than CUT^{2-} on

```
1609 static bool
1610 copy_internal (char const *src_name, ...
...
1620   struct stat src_sb;
...
1640   if (XSTAT (x, src_name, &src_sb) != 0)
1641   {
1642     error (0, errno, _("cannot stat %s"), ...
1643     return false;
1644   }
1645
1646   src_mode = src_sb.st_mode;
...
1717     if (!S_ISDIR (src_mode) && x->update)
1718     { // 6 branches
...
1788     if (!S_ISDIR (src_mode)
1789         && (x->interactive == I_ALWAYS_NO ...
1790     {     // 10 branches
...
1835     if (!S_ISDIR (src_mode))
1836     {// 8 branches
```

Figure 3.2: Example where CUT^2 achieves higher branch coverage than the other concolic testing techniques

Table 3.6: Branch coverage of CUT^{2-} and CUT^2

ID	Name	Br. Cov. (%)		ID	Name	Br. Cov. (%)	
		CUT^{2-}	CUT^2			CUT^{2-}	CUT^2
62543570	copy_reg	73.3	82.6	a04ddb8d	print_color_indicator	72.5	88.3
	copy_internal	75.9	89.2		quote_name	71.7	89.5
b54b47f9	set_fields	79.3	93.5	51a8f707	print_char	100.0	100.0
	cut_fields	78.6	88.1		decode_one_format	67.0	80.7
be7932e8	set_fields	75.5	93.6	64d4a280	scan_arg	86.4	86.4
	cut_fields	77.3	88.6		print_numbers	75.0	85.0
06aeeecb	set_fields	78.7	94.7	Average		77.8	89.2
	cut_fields	77.3	88.6				

average. Note that the branch coverage achieved by CUT^{2-} is comparable to SUT' and OTF (i.e., CUT^{2-} (77.8%), SUT' (78.3%), OTF (78.0%)).

Thus, the experiment results imply that, for high unit test coverage, it is important to utilize DUC

provided by the descendant functions of a target function (which coincides with the field wisdom that a developer has to develop stubs/mock carefully for high unit test effectiveness).

Chapter 4. Related Works

4.1 Automated Unit Testing Techniques based on System Tests

4.1.1 Generating Unit Tests from System Tests

Elbaum et al. [11] proposed a technique to generate unit tests from system tests; the technique carves Java program states before and after an invocation of a target function f to generate unit test inputs. OCAT [12] captures object instances during system executions and generates unit tests using Randoop with the captured object and the mutated object instances as seed objects. GenUTest [13] automatically generates unit tests and mock objects using captured method sequences during system testing.

A limitation of these techniques is that the executions of the generated unit tests just replay the same behaviors [11, 13] (or similar behaviors [12]) of a target unit in already performed system testing (i.e., they are applicable to only regression testing of evolving software, not to a single version of software). In contrast, CUT^2 automatically generates new tests to explore diverse behaviors and achieve high coverage by utilizing DUCs carved from system tests as initial test inputs of concolic testing. Also, all these related techniques depend on Java serialization [1] so that they cannot handle Java programs with unserializable objects. In contrast, CUT^2 develops its own carving tool applicable to complex C programs without limitation.

4.1.2 Symbolic Unit Testing based on System Tests

Păsăreanu et al. [6] proposed a combined approach of concrete system-level execution and symbolic unit-level execution in Java Pathfinder (JPF). JPF monitors a concrete execution of a target program to check that a concrete state satisfies a user-given condition which indicates a starting point of symbolic execution. When the concrete state satisfies the condition, JPF starts symbolic execution at the location with user-given symbolic input setting. CUT^2 achieves much higher branch coverage than the approach in [6] (see Section 3.2.3), because CUT^2 can reduce symbolic execution space with little loss of useful contexts by replacing the callee functions of a target function with symbolic stubs based on the DUCs carved from system tests.

4.2 Concolic Testing Techniques

Several concolic testing techniques have been propsed. Table 4.1 shows breif comparison between CUT^2 and other concolic testing techniques.

4.2.1 Concolic Testing Engines

EXE [10], CREST [9], and CROWN [8] are concolic testing techniques for C programs. They require a user to write not only symbolic test drivers/stubs but also select which variable/memory region should be symbolic (i.e., which varaible/memory region works as test inputs). PeX [14] (also known as IntelliTest) is a concolic unit testing technique for C# programs. PeX requires a user to

Table 4.1: Related work of concolic testing techniques

Related work	Automatic Generation of		Utilizing
	Symbolic test drivers	Symbolic stubs	DUCs
EXE [10]	X	X	X
CREST [9]	X	X	X
CROWN [8]	X	X	X
PeX [14]	X	X	X
DART [15]	O	X	X
SMART [16]	O	X	X
CUTE [17]	O	X	X
CILpp [18]	O	X	X
CONBOL [2]	O	O	X
UC-KLEE [19]	O	X	X
CUT^2	O	O	O

write symbolic unit test drivers/stubs using C# unit testing frameworks such as NUnit. Contrary to them, CUT^2 automatically generates symbolic unit test drivers/stubs for concolic testing. Moreover, CUT^2 automatically selects which variables to be symbolic according to the carved DUCs.

4.2.2 Concolic Testing Frameworks

DART [15], SMART [16], and CUTE [17] generate symbolic unit test drivers (but not symbolic stubs) and test inputs for C programs. The generated unit test drivers specify only the parameters of a target function f (not the global variables read by f) as symbolic inputs. CILpp [18] generates symbolic unit test drivers and test inputs for C/C++ programs. CILpp modifies the unit test drivers generated from directed-random test generation to symbolic unit test drivers for further enhancement of branch coverage. On the other hand, CUT^2 automatically generates not only test inputs, but also symbolic unit test drivers/stubs utilizing carved DUCs.

CONBOL [2] generates symbolic unit test drivers/stubs and test inputs targeting large-scale embedded C programs. Recently, UC-KLEE [19] generates symbolic unit test drivers using lazy symbolic input initialization. The test driver generated by UC-KLEE directly invokes a target function f. During symbolic execution, whenever f reads un-initialized variables, UC-KLEE specifies the variable as symbolic inputs. UC-KLEE does not replace callees to stubs, but symbolically executes all callee functions. I could not directly compare UC-KLEE with CUT^2 because the UC-KLEE tool is not available and the UC-KLEE paper does not report branch coverage completely.

4.3 Automatic Generation of Mock Objects/Testing Stubs

Several research work [20, 21, 22, 23, 24] use automated mocking to improve test coverage and bug detection capability of automated unit testing. Dsc+Mock [20] generates mock objects for testing the interfaces of Java programs. Dsc+Mock collects type constraints on the interfaces during symbolic execution and generates mock objects using the solution of the type constraints. Galler et al. [21] proposed

a technique to generate mock objects using the design-by-contract specification. Saff et al. [22, 23] proposed a technique to generate mocking objects from system test executions (i.e., mock objects are generated based on the interactions between the target code and its environment captured during system executions). Since these techniques generate only concrete mock objects, not symbolic mocks, they have a limitation in providing various dynamic unit contexts.

4.4 Capture and Replay Techniques

There exist several research techniques to capture and replay program states for testing and debugging purpose [25, 26, 27, 28, 29, 30]. Many of them are developed to help debugging by reproducing field failures [25, 26, 27] or concurrent behaviors [28, 29, 30] (e.g., thread scheduling). Since ADDA[25] captures external events specified by a user to replay field failures, it does not support unit testing directly. ReCrash [26] captures only parameters of a target Java method while CUT^2 carves parameters and global variables of a target C function to obtain DUCs accurately. BugRedux [27] captures values of global and local variables on which conditional statements depend on and tries to generate input values which eventually lead to the same branching decisions as captured. However, Bugredux may fail to reproduce the captured behavior (e.g., 10% of the program runs in the experiments failed to reproduce the target execution paths) while CUT^2 carves and replay DUCs precisely (see Section 3.2.2). In addition, CUT^2 generates symbolic test drivers and stubs which utilize DUCs to build symbolic search space to generate effective test inputs, which is a unique contribution compared to these related techniques [25, 26, 27]. Since the techniques for concurrent behaviors target different domain from CUT^2 , it is not straightforward to compare them with CUT^2 . However, CUT^2 can adopt interesting features of those techniques (e.g., utilizing hardware support and/or parallel records using multi-cores) as future work.

Chapter 5. Conclusion and Future Work

5.1 Conclusion

In this dissertation, I have demonstrated that dynamic unit contexts carved from system tests can improve the effectiveness of automated concolic unit testing. For this purpose, I developed a new concolic unit testing technique CUT^2 , which utilizes dynamic unit contexts (DUCs) of a target function carved from system tests as initial test inputs, for complex C programs and showed that CUT^2 can effectively increase branch coverage by applying CUT^2 to 7 real-world C program versions in CoREBench. In the experiments targeting CoREBench, CUT^2 achieves 90% branch coverage on average, which is at least 10.9%p higher branch coverage than other concolic unit testing techniques (i.e., SUT, SUT', and OTF). In addition, this dissertation emphasizes that utilizing DUCs from the callee functions of a target function f is important for achieving high branch coverage.

5.2 Future Work

As future work, I plan to find another way to utilize the dynamic unit contexts of a target function f carved from system tests.

5.2.1 Dynamic Unit Contexts in Automated Debugging

Dynamic unit contexts of a function f contains the input values of f in the system execution. I expect that DUCs of f from passing test executions and failing test executions have some noticeable difference. I plan to utilize these differences for automated debugging or at least helping the developers to debug the program.

5.2.2 Preconditions of a Function

By analyzing DUCs of a target function from various and diverse valid system executions and finding the common features from those DUCs, I will try to build a tool that finds preconditions of the function automatically. These preconditions can be utilized for many purposes, such as preventing malicious inputs.

5.2.3 Deeper Study on Dynamic Unit Contexts

I plan to compare the DUCs of target functions with different characteristics. For example, faulty functions vs. correct functions and/or simple functions vs. complex functions. If there is a meaningful result, then this could provide a new insight to detect a bug.

Bibliography

[1] "Java Object Serialization Specification," https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html, accessed: 2018-11-27.

[2] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim, "Automated unit testing of large industrial embedded software using concolic testing," in *Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering,* pp. 519-528, Nov 2013.

[3] Y. Kim, Y. Choi, and M. Kim, "Precise Concolic Unit Testing of C Programs with Extended Units and Symbolic Alarm Filtering," *Proceedings of the 40th International Conference on Software Engineering (ICSE),* ACM, pp. 315-326, 2018.

[4] Y. Kim, M. Kim, Y. J. Kim, and Y. Jang, "Industrial Application of Concolic Testing Approach: A Case Study on libexif by Using CREST-BV and KLEE," *2012 34th International Conference on Software Engineering (ICSE),* IEEE, pp. 1143-1152, June 2012.

[5] M. Kim, Y. Kim, and Y. Jang, "Industrial Application of Concolic Testing on Embedded Software: Case Studies," *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST),* IEEE, pp. 390-399, 2012.

[6] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M.Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis,* ser. ISSTA '08, New York, NY, USA: ACM, pp. 15-26, 2008.

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO),* 2004.

[8] "CROWN: Concolic testing for Real-wOrld softWare aNalysis," https://github.com/swtv-kaist/CROWN.

[9] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering,* pp. 443-446, 2008.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *Journal of ACM Transactions on Information and System Security (TISSEC),* Vol. 12, No. 2, Article No. 10, December 2008.

[11] S. Elbaum, H. Chin, M. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Transactions on Software Engineering (TSE),* vol. 35, no. 1, pp. 29-45, Jan 2009.

[12] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: Object capture-based automated testing," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA),* 2010.

[13] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "GenUTest: A unit test and mock aspect generation tool," *Software Tools for Technology Transfer,* vol. 11, no. 4, pp. 273-290, 2009.

[14] N. Tillmann and J. De Halleux, "Pex: White box test generation for .NET," in *Proceedings of the 2nd International Conference on Tests and Proofs,* ser. TAP'08. Berlin, Heidelberg: Springer-Verlag, pp.134-153, 2008.

[15] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* 2005.

[16] P. Godefroid, "Compositional dynamic test generation," *Proceedings of the 34th annual AMC SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* pp. 47-54, 2007.

[17] K. Sen, D. Marinov, and G. Agha, "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 Sotware Engineering,* ser. ESEC/FSE13. New York, NY, USA: ACM, pp. 263-272, 2005.

[18] P. Garg, F. Ivančić, G. Balakrishman, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for C/C++ using concolic execution," *2013 35th International Conference on Software Engineering (ICSE),* pp. 131-141, 2013.

[19] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proceedings of the 24th USENIX Conference on Security Symposium,* ser. SEC'15, Berkeley, CA. USA: USENIX Association, pp. 49-64, 2015.

[20] M. Islam and C. Csallner, "Dsc+Mock: A test case + mock class generator in support of coding against interfaces," in *Proceedings of the Eighth International Workshop on Dynamic Analysis,* ser. WODA'10, New York, NY, USA: ACM, pp. 26-31, 2010.

[21] S. J. Galler, A. Maller, and F. Wotawa, "Automatically extracting mock object behavior from design by contract™ specification for test data generation," in *Proceedings of the 5th Workshop on Automation of Software Test,* ser. AST'10. New York, NY, USA: ACM, pp. 43-50, 2010.

[22] D. Saff and M. D. Ernst, "Mock object creation for test factoring," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering,* ser. PASTE'04. New York, NY, USA: ACM, pp. 49-51, 2004.

[23] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for Java," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering,* ser. ASE'05. New York, NY, USA: ACM, pp. 114-123, 2005.

[24] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," in *Proceedings of the International Conference on Automated Software Engineering (ASE),* 2010.

[25] J. Clause and A. Orso, "A technique for enabling and supporting debugging of field failures," in *29th International Conference on Software Engineering (ICSE'07),* pp. 261-270, May 2007.

[26] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *Proceedings of the 22nd European Conference on Object-Oriented Programming,* ser. ECOOP'08, Berlin, Heidelberg: Springer-Verlag, pp. 542-565. 2008. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70592-5_23

[27] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *2012 34th International Conference on Software Engineering (ICSE),* pp. 474-484, June 2012.

[28] P. Liu, X. Zhang, O. Tripp, and Y. Zheng, "Light: Replay via tightly bounded recording," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation,* ser. PLDI'15, New York, NY, USA: ACM, pp. 55-64, 2015. [Online]. Available: http://doi.acm.org/10.1145/2737924.2738001

[29] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference,* ser. USENIX ATC'17. Berkeley, CA, USA: USENIX Association, pp. 377-389, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3154690.3154727

[30] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum, "Towards practical default-on multi-core record/replay," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Porgramming Languages and Operating Systems,* ser. ASPLOS'17, New York, NY, USA: ACM, pp. 693-708, 2017. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037751

Acknowledgments in Korean

지난 2년의 석사 과정 동안, 제가 이 논문을 완성할 수 있도록 도움을 주신 모든 분들께 이 지면을 빌어 감사의 말씀을 전하고자 합니다. 먼저 언제나 저를 믿고 격려해 준 아버지, 어머니, 그리고 동생에게 감사드립니다. 부모님의 믿음과 격려가 있었기에 지금까지 해내올 수 있었습니다. 동생의 이해와 배려로 석사 과정을 무사히 지낼 수 있었습니다.

저에게 많은 가르침을 주신 김문주 교수님께 감사드립니다. 많이 부족했던 저를 잘 이끌어주시고, 때때로 흔들리는 저를 잘 바로잡아주셨기에 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 한 번 감사드립니다. 제 연구에 많은 도움을 주신 김윤호 연구조교수님께 감사드립니다. 제가 여러가지 어려움에 부딪힐 때마다 토론을 통해 대안과 실마리를 주시며 이끌어 주셨기에 여기까지 올 수 있었습니다. 그리고 실험을 하는 과정에서 주신 많은 도움에도 감사드립니다. 제가 연구실 생활에 잘 적응할 수 있도록 도움을 주신 김현우, 양웅규 졸업생에게 감사드립니다. 연구실에서 함께 공부한 고봉석, 박건우, Loc Duy Phan, 김진솔, 김찬수, 이아청 학생에게도 감사드립니다. 모두가 각자의 길에서 최고가 되기를 기원합니다.

대학원 생활 동안 항상 힘이 되어준 강승모, 임선우, 임한빈, 정정호 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.

Curriculum Vitae in Korean

이 름: 임 현 수

생 년 월 일: 1995년 01월 09일

전 자 주 소: hyunsu.lim01@gmail.com

학 력

2010. 3. – 2012. 2. 고등학교 (2년 수료)

2012. 2. – 2017. 2. 한국과학기술원 전산학부 (학사)

2017. 3. – 2019. 2. 한국과학기술원 전산학부 (석사)

학 회 활 동

1. **임현수**, 김윤호, 김문주, "시스템 테스트 케이스를 이용한 C 프로그램의 동적 유닛 입력 값 자동 수집 및 재연 기술," *Korea Software Congress (KSC),* Dec 20-22, 2017 **(Distinguished best paper award)**

연 구 업 적

1. **임현수**, 김윤호, 김문주, "C 프로그램의 동적 및 정적 분석을 이용한 시스템 실행에서의 유닛 입력 값 자동 수집 및 재연," *Journal of KIISE*, Vol. 45, No. 10, pp. 1035-1044, October. 2018.