석 사 학 위 논 문
Master's Thesis

C 프로그램을 위한 단위 수준 콘콜릭 테스팅에 대한 분석 및 개선

Analysis and Improvement of Unit-level Concolic Testing for Real-world C Programs

2022

양 재 동 (楊 梓 棟 Yang, Zidong)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

C 프로그램을 위한 단위 수준 콘콜릭 테스팅에 대한 분석 및 개선

2022

양 재 동

한 국 과 학 기 술 원

전산학부

C 프로그램을 위한 단위 수준 콘콜릭 테스팅에 대한 분석 및 개선

양 재 동

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2021년 12월 15일

심사위원장 김 문 주 (인)

심 사 위 원 고 인 영 (인)

심 사 위 원 백 종 문 (인)

Analysis and Improvement of Unit-level Concolic Testing for Real-world C Programs

Zidong Yang

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 15, 2021

Approved by

Moonzoo Kim
Associate Professor of School of Computing

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 양재동. C 프로그램을 위한 단위 수준 콘콜릭 테스팅에 대한 분석 및 개선. 전산학부 . 2022년. 62+v 쪽. 지도교수: 김문주. (영문 논문)
Zidong Yang. Analysis and Improvement of Unit-level Concolic Testing for Real-world C Programs. School of Computing . 2022. 62+v pages. Advisor: Moonzoo Kim. (Text in English)

초 록

유닛 레벨 Concolic 테스팅은 짧은 시간 안에 각 함수의 다양한 실행을 하면서 높은 분기 커버리지를 보이는 테스트를 생성할 수 있는 기술이다. 하지만, 현재의 Concolic 테스팅 기법은 여러 문제점을 가지고 있는데, 예를 들어 유닛 레벨 Concolic 테스팅 기법으로는 void 타입의 포인터가 어떤 타입으로 변환 (casting) 되어 사용되는지 알 수 없으며, 외부 라이브러리 함수도 제대로 다룰 수가 없다. 따라서, 실제 프로그램에 유닛 레벨 Concolic 테스팅 기법을 적용하였을 때 충분한 분기 커버리지를 달성하지 못하게 된다.

본 논문에서는 상용 유닛 레벨 Concolic 테스팅 도구인 CROWN2.0 ver 2020을 6개의 실제 프로그램에 적용한 실험을 구성하였다. 해당 실험에서 324개 그룹의 달성하지 못한 분기를 모두 분석하여, 현재 도구가 가지고 있는 13개의 제한점으로 정리하였다. 각 제한점에 대해 왜 해당 분기가 달성되지 못하였는지, 코드 예시를 포함한 설명을 추가하여 차후 Concolic 유닛 테스팅 도구 개발자가 각 제한점을 이해하고, 개선할 수 있도록 하였다. 또한, 각 제한점에 적용할 수 있는 해결법을 제시하고 구현하여, 종합적으로 6개의 테스트 대상 프로그램에서 77%의 분기 커버리지 향상을 보였다.

핵 심 낱 말 소프트웨어 테스팅, 자동화 테스트 케이스 생성, Concolic 테스팅, 단위 테스팅

Abstract

Unit-level concolic testing has become popular as it explores every execution path of each function in a short time with high branch coverage. However, the current concolic unit testing tool suffers a set of limitations. For example, the unit-testing tool cannot determine the actual type of what a void pointer is cast to, and external library functions are not fully supported. Therefore, the branch coverage obtained on real-world programs is not satisfying.

This dissertation conducts comprehensive experiments on six real-world subjects using CROWN2.0 ver 2020, a popular commercial unit-level concolic testing tool. In the experiments, all the 324 groups of unexplored branches of the target subjects are thoroughly analyzed, and this dissertation summarizes the analysis into 13 limitations of the current testing tool. For each limitation, this dissertation provides detailed explanations why the corresponding uncovered branches are not covered with code examples. I expect these explanations can help the developers to understand and improve the concolic unit testing tool in the future. Also, this dissertation proposes and applies solutions the obtained limitations, which gives an average improvement on the branch coverage of 77% for six subjects.

Keywords Software testing, automated test case generation, concolic testing, unit testing

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Software testing is one of the main technical approaches to improve the reliability and safety of software. The software testing procedure provides a set of inputs and checks whether the target software's outputs are consistent with the expected ones. As the size of software increases, the proportion of software testing in the software development cycle is also enlarged. In particular, it is important to choose the method of generating test cases.

The software testing has four granularity levels, i.e., unit testing, integration testing, system testing, and acceptance testing.

- **Unit testing**. Unit testing ensures that a section of an application (known as the "unit/function") meets its design and behaves as intended. Unit testing is the most fine-grained testing approach in the software testing process. Independent units/functions of the software under test will be tested in isolation from other parts of the software.
- **Integration testing**. It is also called assembly testing. Integration testing is based on unit testing to design and assemble all modules into a subsystem or system as required and then check whether the interaction between the functions within each module is normal.
- **System testing**. System testing considers the hardware, software, and operators as a whole architecture and checks whether any part does not follow the system specification.
- **Acceptance testing**. It is the last testing step before deploying the software. The purpose is to ensure that the software is ready and can be used by the users to perform the required functionalities and tasks of the software.

For each granularity of software testing, the first step and an essential part are to generate the test cases. The quality of software depends on the test suites, and the quality of test suites depends on the test case generation techniques that generate them.

There are two ways to generate test cases: manual testing and automatic testing. Manual testing requires a lot of labor costs, and automated testing overcomes this shortcoming. As a result, automated testing has become more and more popular. The existing automated testing techniques can be mainly divided into the following three categories.

1.1 Previous Approaches

(1) **Random Testing.** The random testing technique randomly generates test inputs that meet the requirements of the corresponding sampling functions, e.g., Uniform distribution, Gaussian distribution. The key part of this technique is *randomness*. This technique is simple and easy to implement with a high degree of automation. It is very effective for testing certain types of software and finding the corner case bugs. However, the efficiency of random testing is not satisfying in that it achieves very low branch coverage. Moreover, it generates a lot of redundant test cases, i.e., the test cases with the same execution path. To tackle those limitations, researchers proposed improved random testing. For example, adaptive random testing chooses the next input which is most diverse against (fastest away

from) the already obtained inputs[2, 4, 3, 5]. Fuzzing testing favors the inputs that can cover more not-covered branches[6, 7, 8, 9, 10, 11].

(2) **Search-based Test Case Generation.** The search-based test generation technique uses heuristic search technologies such as a genetic algorithm to generate test cases automatically. Unlike random testing that randomly samples test cases from the entire input space, the search-based technique needs to define a specific fitness function that is used to guide the search process so that an appropriate solution can be found. The efficiency of search-based testing highly depends on the quality of the fitness function, which makes it essential to design a good fitness function[12, 13, 14, 15, 16, 17].

(3) **Constraint-based Test Case Generation.** The constraint-satisfaction problem appears in many fields, especially computer science, e.g., software verification and software testing. One of the representative theories of this problem is Software Modular Theory (SMT), which gives SMT solvers. The main idea of constraint-based test case generation is to provide some conditions to the SMT solver that is used, and then the SMT solver checks the satisfiability of those conditions in the test case generation. There exist many research papers which focus on SMT-based testing. Godefroid et al. developed DART to automatically generate unit test drivers to generate test cases for C programs by using concolic execution[18]. Sen et al. developed CUTE to generate test inputs using concolic testing[19] automatically. Burnim et al. developed CREST[24] which contains a variety of path exploration algorithms, making it more scalable and able to test large-scale software systems. Kim et al. developed CREST-BV[20] which extends CREST by adding the support of bit-vector. Kim et al. developed CONBOL, which automatically generates drivers/stubs and test inputs for large embedded software[21]. Kim et al. developed CONBRIO to automatically generate unit drivers/stubs with an extended unit to achieve high bug detection ability and low false alarm ratio[22].

Concolic testing is one of the most important test case generation technologies based on constraint solving techniques. Unlike manual testing, it tries to explore every execution path of the target subject. However, the system-level concolic testing suffers the path explosion problem when the search space of the target subject is massive.

Unit-level concolic testing reduces the exploration space by focusing on functions rather than the whole system to address this problem. Doing so achieves higher branch coverage and detects bugs more quickly than the system-level testing in principle. However, the experiment results on the real-world program are not desirable due to a set of limitations[1]. For example, concolic unit testing on grep2.0 only achieves 22.6% branch coverage. Thus, to improve the effectiveness of concolic unit testing, we need to thoroughly analyze the results w.r.t branch coverage on real-world programs and discover the limitations. Accordingly, we may propose solutions to improve the branch coverage.

1.2 Thesis Statement and Contributions

1.2.1 Thesis Statement

The thesis statement of this dissertation is as follows:

Concolic testing can be improved to achieve high branch coverage by addressing its limitations identified through systematic coverage analysis on real-world subjects.

[1] I will explain these limitations in chapter 4

1.2.2 Contributions

The contributions of this dissertation are as follows:

- I highlight 13 common problems of CROWN2.0 ver 2020 by extensively analyzing 324 groups of not-covered branches on six different widely-used target subjects. Some common problems might be generalized to be found in other concolic testing engines.
- I propose six ideas and implement these ideas for 10/13 common problems in CROWN2.0 ver 2020, which has successfully improved the coverage of CROWN2.0 ver 2020 by up to 188% on one target subject (77% on average on six target subjects).
- I provide a detailed report to describe the reasons of 324 branch groups to be not-covered. The detailed report could further aid the future of concolic testing research[2].

1.3 Structure of Dissertation

The remainder of this dissertation is structured as follows. Chapter 2 introduces the background of unit-level concolic testing. Chapter 3 performs experiments on six real-world subjects (four from SIR benchmark[27] and two from SPEC 2006 benchmark[28]) and presents the initial testing results. Chapter 4 analyzes each group of unexplored branches and shows 13 limitations of current testing tool. Chapter 5 proposes solutions for these limitations and evaluates the effectiveness of these solutions. Chapter 6 lists the crashes detected by CROWN2.0 and explains how to deduplicate the crashes which have the same execution path. Chapter 7 presents the related work for automated software test case generation techniques. Chapter 8 shows the conclusion and future works.

[2]The report is available at https://bit.ly/not-covered-branches

Chapter 2. Concolic Unit Testing Tool and Its Working Mechanism

This chapter introduces the background of concolic testing techniques and explains how the unit-level concolic testing tool CROWN2.0[23] works.

2.1 Background of Concolic Testing

Concolic (**Conc**rete + Symb**olic**) testing (aka. dynamic symbolic execution) is a technique that tries to explore all the execution paths of a target program. When a target program P is put into test with symbolic input variables, concolic testing first executes the target program with the initial value of the symbolic variables (usually assigned with zero). Then concolic testing collects the symbolic path formula (a sequence of constraints) achieved by current execution. To cover a new path, one constraint of the current execution would be negated, and new test cases would be generated.

There are many searching heuristics to determine which constraint should be altered. For example, DFS (depth-first search) always alters the last constraint of the current execution, Random-negation randomly negates one constraint from the given symbolic path formula[24].

After negating one constraint, the new symbolic path formula would be put into the underlying SMT solver (e.g., Z3[25], STP[26]) to decide whether the new symbolic path formula is satisfiable or not. If the new symbolic path formula can be satisfied, the SMT solver would output the new test inputs and assign them to the symbolic variables.

Then concolic testing repeats the same process until all the paths of the target program are covered or the conditions specified by the users are satisfied (e.g., the given timeout for test case generation is reached). The example in the next section explains how the concolic testing technique works in detail.

2.2 Motivating Example

The function (*triangle_type*) in figure 2.1 accepts three integer inputs (line 7: a,b and c) and checks the type of triangle according to the input values.

For concolic testing, first, we set the three inputs a, b, and c as symbol variables, and execute the first test case (input values are zero). When the first test case is executed, we get the SPF (symbolic path formula) of the execution ($a <= 0$). Then we get a new SPF ($a > 0$) and a new input ($a, b, c =$ 1,0,0) by negating the last constraint ($a <= 0$). Then we execute the second test case to obtain it's SPF ($a > 0 \& b <= 0$), and negate the last constraint of the current SPF to obtain the new SPF ($a > 0 \& b > 0$) and the third test case ($a, b, c =$ 1,1,0) We repeat the procedure iteratively until all paths of the target function are covered.

Table 2.1 shows the detailed process. The first column shows the test input number. The second column shows the value of the current test input. The third and fourth columns show the SPF of the current test input and the SPF that should be satisfied by the next test input. The fifth column shows the value of the next test input.

```c
#include <assert.h>
/*
 * triangle.c
 * This example is taken from Software Testing: A Craftsman's Approach 2/e,
 * by P. C. Jorgensen.
 */
 int triangle_type(int a, int b, int c){
    int i, j, k, match=0;

    int result=1;

    if(a <= 0 || b <= 0 || c <= 0) {
        result = 2;
    } else{
        if(a == b) match = match + 1;
        if(a == c) match = match + 2;
        if(b == c) match = match + 3;
        if(match == 0) {
            if(a + b <= c) result = 2;
            else if(b + c <= a) result = 2;
            else if(a + c <= b) result = 2;
            else result = 3;
        } else {
            if(match == 1) {
                if(a+b <= c) result = 2;
                else result = 1;
            } else {
                if(match == 2) {
                    if(a+c <= b) result = 2;
                    else result = 1;
                } else {
                    if(match == 3) {
                        if(b + c <= a) result = 2;
                        else result = 1;
                    } else result = 0;
                }
            }
        }
    }
    return result;
}
```

Figure 2.1: Triangle Example

2.3 Generation of Symbolic Driver and Stubs

The most important thing is how to declare symbolic variable for different types of inputs (e.g., pointer, structure). For different types of inputs of a target function, CROWN2.0 automatically generate unit driver and stub code to set symbolic inputs. The remaining part of this chapter explains how the driver and stub code are generated.

Table 2.1: Execution Paths of Triangle Program

TC	Input (a,b,c)	Current SPF	Next SPF	Next Input (a,b,c)
1	0,0,0	a<=0	a>0	1,0,0
2	1,0,0	a>0 & b<=0	a>0 & b>0	1,1,0
3	1,1,0	a>0 & b>0 & c<=0	a>0 & b>0 & c>0	1,1,1
4	1,1,1	a>0 & b>0 & c>0 &a==b & a==c & b==c	a>0 & b>0 & c>0 &a==b & a==c & b!=c	UNSAT
			a>0 & b>0 & c>0 &a==b & a!=c	1,1,2
5	1,1,2	a>0 & b>0 & c>0 &a==b & a!=c &b!=c&a+b<=c	a>0 & b>0 & c>0 &a==b & a!=c &b!=c&a+b>c	2,2,3
6	2,2,3	a>0 & b>0 & c>0 &a==b & a!=c &b!=c&a+b>c	a>0 & b>0 & c>0 &a==b & a!=c &b==c	UNSAT
7			a>0 & b>0 & c>0 &a!=b	2,1,2
	2,1,2	a>0 & b>0 & c>0 &a!=b & a==c & b!=c & a+c >b	a>0 & b>0 & c>0 &a!=b & a==c & b!=c & a+c <= b	2,5,2
8	2,5,2	a>0 & b>0 & c>0 &a!=b & a=c & b!=c & a+c <= b	a>0 & b>0 & c>0 &a!=b & a==c & b==c	UNSAT
			a>0 & b>0 & c>0 &a!=b & a!=c	1,2,2
9	1,2,2	a>0 & b>0 & c>0 &a!=b & a!=c & b==c & b+c >a	a>0 & b>0 & c>0 &a!=b & a!=c & b==c & b+c <= a	4,2,2
10	4,2,2	a>0 & b>0 & c>0 &a!=b & a!=c & b==c & b+c <= a	a>0 & b>0 & c>0 &a!=b & a!=c & b!=c	1,2,3
11	1,2,3	a>0 & b>0 & c>0 &a!=b & a!=c & a+b<=c	a>0 & b>0 & c>0 &a!=b & a!=c & a+b>c	3,1,2
12	3,1,2	a>0 & b>0 & c>0 &a!=b & a!=c & a+b>c &b+c<=a	a>0 & b>0 & c>0 &&a!=b & a!=c & a+b>c & b+c>a	1,3,2
13	1,3,2	a>0 & b>0 & c>0 &&a!=b & a!=c & a+b>c & b+c>a & a+c<=b	a>0 & b>0 & c>0 &&a!=b & a!=c & a+b>c & b+c>a & a+c>b	3,4,5
14	3,4,5	a>0 & b>0 & c>0 &&a!=b & a!=c & a+b>c & b+c>a & a+c>b	Finish	Finish

2.3.1 Driver Code Generation

In the driver code, all the global variables used by the target function and all the parameters of the target function are declared as symbolic variables according to their types as follows.

- **Primitive type**.
- **Array type**.
- **Pointer type**.
- **Structure type**.

The first four examples below show how CROWN2.0 handles the four aforementioned types of inputs. In addition, I created a much complex case (the fifth example) that contains two or more different input

```
1  #include "crown.h" // head file used by CROWN2.0
2  /*
3      int primitive_exp(int a){
4         return a;
5      }
6  */
7  int primitive_exp_driver(){
8      int a;
9      SYM_int(a);
10     primitive_exp(a);
11 }
```

Figure 2.2: Set Symbolic Variable of Primitive Type

```
1  #include "crown.h" // head file used by CROWN2.0
2  /*
3      int a[10];
4      int array_exp(){
5         return a[1];
6      }
7  */
8  int array_exp_driver(){
9      for(int i0=0; i0<10; i++){
10         SYM_int(a[i0]);
11     }
12     array_exp();
13 }
```

Figure 2.3: Set Symbolic Variable of Array Type

types to demonstrate that CROWN2.0 can handle the complex case properly.

Primitive Type

For a variable of primitive type, CROWN2.0 specifies that variable as symbolic by using the API $SYM\_ < type >$[1]. Figure 2.2 shows an example.

- Line 3: The function *primitive_exp* has a parameter of integer type
- Lines 8 - 10 : CROWN2.0 declares a symbolic variable a and uses it as the parameter of the target function

Array Type

CROWN2.0 sets each element of that array as symbolic. The detailed code example is listed in Figure 2.3

- Line 3: The global variable a is an array and has 10 elements.
- Line 5: a is used by the target function *array_exp*
- Lines 9 - 12: CROWN uses a for-loop to set each element of the array a as symbolic, then the target function *array_exp* would be called.

[1]The API $SYM\_ < type >$ assigns zero to all input variables for the first TC

```
1  #include "crown.h" // head file used by CROWN2.0
2  /*
3      int pointer_exp(int* a){
4         return a[0];
5      }
6  */
7  int pointer_exp_driver(){
8      int * a = malloc(3*sizeof(int)); // n = 3
9      for(int i0=0; i0< 3; i0++){
10         SYM_int(a[i0]);
11     }
12
13     pointer_exp(a);
14 }
```

Figure 2.4: Set Symbolic Variable of Pointer Type

Pointer Type

CROWN2.0 allocates $< n >$ (decided by the users) elements for the pointee type (CROWN2.0 considers a pointer p points to an array which has $< n >$ elements) and sets each element as symbolic variable. Figure 2.4 shows an example where $< n >$ is assigned with three by default.

- Line 3: The target function *pointer_exp* has a parameter a, which is a pointer and points to an integer.
- Line 8: CROWN2.0 allocate $< n >$ elements for the pointer a. In the example, $< n >$ is three.
- Lines 9 - 11 : CROWN2.0 uses a for-loop to set all $< n >$ elements of a as symbolic.
- Line 13: The target function *pointer_exp* would be called.

Structure Type

CROWN2.0 sets each element of that structure as symbolic according to the element type recursively. Suppose the element is not a primitive type (e.g., pointer type, array type, structure type). In that case, CROWN2.0 sets symbolic input for the element, following how it declares symbolic according to the element type.

Specifically, to prevent the infinite recursive dereferences (e.g., a structure has a pointer that points a variable of the same type with the structure), CROWN2.0 follows a pointer to the structure within a *bound* three by default and assigns NULL to a pointer that is in the *layer* not reachable within the bound. The figure 2.5 explains the bound and layer in detail.

In figure 2.5, the struct Node has a pointer element *next* points to a same type with Node. From layer zero to layer two, *next* is allocated with a struct Node. But at layer three, *next* is assigned with NULL as layer three reaches the given bound three.

Figure 2.6 shows how to declare a symbolic variable of structure type.

- Lines 3 - 7: A structure type A is declared, which has three elements, an integer element, an character element and a pointer element.
- Line 8: The target function *struct_exp* has a parameter a whose type is structure A

```
struct Node{
  struct Node* next;
}
```

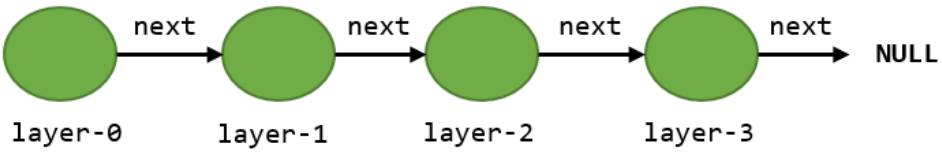


Figure 2.5: Recursive Declaration Example

```
1  #include "crown.h" // head file used by CROWN2.0
2  /*
3      struct A{
4        int ele1;
5        char ele2;
6        int *ele3;
7      };
8      int struct_exp(struct A a){
9          ...
10         return 0;
11     }
12 */
13 int struct_exp_driver(){
14     struct A a;
15     SYM_int(a.ele1);
16     SYM_char(a.ele2);
17     a.ele3 = malloc(3*sizeof(int)); // n = 3
18     for(int i0=0; i0<3; i++){
19         SYM_int(a.ele3[i0]);
20     }
21     struct_exp(a);
22 }
```

Figure 2.6: Set Symbolic Variable of Structure Type

- Lines 14 - 16: CROWN2.0 declares a variable a of type A and sets its elements of primitive type ($ele1$ and $ele2$) as symbolic.
- Lines 17 - 20: For the pointer variable $ele3$, CROWN2.0 allocates n (n is three in the example) elements for $ele3$ and uses a for-loop to declare each element of $ele3$ as symbolic.
- Line 21: The target function *struct_exp* would be called.

Complex Type Example

The figure 2.7 contains a more complex case where a structure contains elements of two or more aforementioned types. I will explain how CROWN2.0 works for the complex case step by step.

- Lines 3 - 7: A structure type A is declared, which has three elements, an integer element, a character element and a pointer element points to a variable of type A.

```
1  #include "crown.h" // head file used by CROWN2.0
2  /*
3      struct A{
4        int ele1;
5        int ele2[10];
6        struct A* next;
7      };
8      int complex_exp(struct A a){
9          ...
10         return 0;
11     }
12 */
13 complex_exp_driver()
14 {
15     struct A a;
16     SYM_int(a.ele1);
17     for(int i0=0;i0<10;i0++){
18       SYM_int(a.ele2[i0]);
19     }
20     a.next = malloc(3 * sizeof(struct A));
21     for(int i0=0;i0<3;i0++){
22       SYM_int(a.next[i0].ele1);
23       for(int i1=0;i1<10;i1++){
24         SYM_int(a.next[i0].ele2[i1]);
25       }
26       a.next[i0].next = malloc(3 * sizeof(struct A));
27       for(int i1=0;i1<3;i1++){
28         SYM_int(a.next[i0].next[i1].ele1);
29         for(int i2=0;i2<10;i2++){
30           SYM_int(a.next[i0].next[i1].ele2[i2]);
31         }
32         a.next[i0].next[i1].next = malloc(3 * sizeof(struct A));
33         for(int i2=0;i2<3;i2++){
34           SYM_int(a.next[i0].next[i1].next[i2].ele12);
35           for(int i3=0;i3<10;i3++){
36             SYM_int(a.next[i0].next[i1].next[i2].ele2[i3]);
37           }
38           a.next[i0].next[i1].next[i2].next = NULL;
39         }
40       }
41     }
42     complex_exp(a);
43 }
```

Figure 2.7: Set Symbolic Variable of Structure Type (complex case)

- Line 8: CROWN2.0 detects that the target function *complex_exp* has a parameter a whose type is structure A. So CROWN2.0 creates a variable a of type A and uses it as the parameter of function *complex_exp* (line 15)
- Line 16: CROWN2.0 sets elements of primitive type of a ($ele1$) as symbolic.
- Lines 17 - 19: For the array variable $ele2$, CROWN2.0 uses a for-loop to declare each element of $ele2$ as symbolic.

```
1. include<string.h>
2. extern int h();
3. int target(char * a){
4.   return h() + strlen(a);
5. }
```

(a) Source Code

```
1. int h(){
2.   int h_ret;
3.   SYM_int(h_ret);
4.   return h_ret;
5. }
```

(b) Stub Code

Figure 2.8: Create Stub for Library Functions

- Lines 20 - 21: For a pointer variable *next*, CROWN2.0 allocates n (three in the example) elements for it. CROWN2.0 detects that the pointer *next* points to a variable of structure type A, so CROWN2.0 recursively sets the symbolic variable for the pointer *next*, and increases the recursion layer by one.
- Lines 22 - 27: Do the same process for layer one.
- Lines 28 - 33: Do the same process for layer two.
- Lines 34 - 38: Do the same process for layer three, but the variable *next* is assigned with NULL (line 38) because the default bound is three.

2.3.2 Stub Code Generation

CROWN2.0 generates symbolic stubs for the library functions, which are called by the target function and do not belong to POSIX[2] library. These symbolic stub functions return symbolic variables according to their return types. The symbolic return variables are created in the same way to create the symbolic inputs of four different types mentioned in Sect. 2.3.1. Finally, CROWN2.0 replaces the original functions with the symbolic stub functions.

Figure 2.8 shows an example on how to create stub for library functions. In the source code of target subject, the target function calls two library functions, i.e., h (L4) and *strlen* (L4). But *strlen* is a POSIX library function, so CROWN2.0 only creates stub for the external library function h (Lines 1-5 of stub code)

[2]CROWN2.0 runs on unix operating system, where the POSIX library is available. Other library functions may not be available and make compilation error happen (e.g., undefined reference to function f)

Chapter 3. Experiment Setup and Baseline Result

This chapter presents the evaluation of the baseline version (i.e., original CROWN2.0 + assign NULL to FILE pointer) of CROWN2.0 ver 2020 (in the rest part of thesis, I will use $CR2^{base}$ to represent the baseline CROWN2.0). To discover the limitations of $CR2^{base}$, I have performed experiments on six real-world open-source C programs (four from SIR benchmark and two from SPEC2006 benchmark). And the branch coverage achieved by $CR2^{base}$ is displayed, which shows that the experimental results are not satisfactory as they are in theory.

3.1 Experiment Setup

3.1.1 Target Subjects

This experiment targets six real-world subjects. Four subjects, which the Linux users frequently use, are selected from the SIR benchmark. And two subjects are selected from the SPEC2006 benchmark[1]. Both SIR and SPEC 2006 are representative benchmarks to measure the effectiveness of software testing. Each subject in these benchmarks has a set of system test cases and the branch coverage (reported by **gcov**) achieved by executing the system-level test cases[2].

Table 3.1 describes the six open-source target subjects, including the size of each subject (reported by script **cloc**), the number of branches of each subject, the number of system-level test cases, the branch coverage of the system-level test cases.

3.1.2 Testbed Settings

Since $CR2^{base}$ has multiple arguments, I will explain each argument and how I set each argument correspondingly.

- *Timeout1*. The maximum execution time of a test case.
- *Timeout2*. The test generation time for each unit/function.
- *Strategy*. The search heuristic of concolic testing (e.g., DFS, reversed DFS)
- *Array Size*. The number of elements N that should be allocated for the pointer p (i.e., $CR2^{base}$ considers p points to an array which has N elements)

For *Timeout1* and *Timeout2*, I set them as 60s and 300s respectively. The reason why I set these arguments is because the exploratory studies with timeout suggests that the increase of *Timeout1* and *Timeout2* beyond 60s and 300s has negligible effects on the overall experimental results. For searching strategy, I use dfs (depth first serach) strategy in the experiments. For array size, I use the default value used by $CR2^{base}$ (three). $CR2^{base}$ uses CROWN[1], to generate test inputs for each function. The initial value of each symbolic variable is assigned with zero following the default behavior of $CR2^{base}$.

[1]The reason why I choose two benchmarks is to avoid making the analysis results overfit to one benchmark.

[2]The SIR benchmark contains many manually added test cases to satisfy the test adequacy criteria (e.g., trigger the artificial faults in the target program)

[1]CROWN is a instrumentation based concolic testing tool to generate test inputs for C programs (website: https://github.com/swtv-kaist/CROWN)

Table 3.1: Detail of Target Subjects

Benchmark	Target Subject	Loc	# of Branch	# of Func	# of Sys. TC	Branch Coverage
SIR	flex2.4.3	11864	2021	144	567	45.7%
	grep2.0	10930	3416	119	809	50.3%
	gzip1.0.7	6358	1446	80	214	55.8%
	sed1.1.7	8060	2395	63	360	47.3%
SPEC2006	libquantum0.2.4	5059	724	111	3	68.5%
	sjeng11.2	18057	6364	164	3	77.9%

Table 3.2: The Experimental Results of CR2^{base}

Target Subject	# of Func	# of Branch	Branch Cov Achieved by CR2^{base}	Avg TCgen Time (s) for Each Function
flex2.4.3	144	2021	22.6%	15.3
grep2.0	119	3416	22.7%	27.3
gzip1.0.7	80	1446	41.4%	52.7
sed1.1.7	63	2395	17.7%	15.8
libquantum0.2.4	111	724	41.6%	5.5
sjeng11.2	164	6364	27.9%	26.7

The experiments are performed on eight machines, each of which is equipped with AMD 8-core Ryzen 7 3800XT (3.9 GHz) CPU and 16 GB RAM, running Ubuntu 18.04 64 bit version.

3.2 Measurement

I measure the branch coverage for each unit (function), then I use **grcov** to collect the coverage of all units and summarize the coverage of each subject.

3.3 Baseline Result

Table 3.2 shows the branch coverage achieved by the TCs, which are generated by the CR2^{base}. The first to third columns show the name, the number of functions, and the number of branches for each subject. The fourth column shows the branch coverage achieved by CR2^{base}. The fifth column shows the average test case generation time of all functions in each subject.

Figure 3.1 shows the distribution of TC generation time for six target subjects (x-axis is the # of function and y-axis is the TC generation time). The average branch coverage is 29.0% only, which shows that CR2^{base} may have its limitations or weaknesses for being applied to test the real-world programs. Therefore, we need to find out the probable limitations and thus improve the effectiveness of CR2^{base}.

In figure 3.1, we can find the number of functions whose test case generation time reaches 300s only account for about 10% of the total number of functions. The branch coverage for the remaining 90% functions is supposed to be 100% in principle. Therefore, I collected information about these functions and showed the detail in table 3.3.

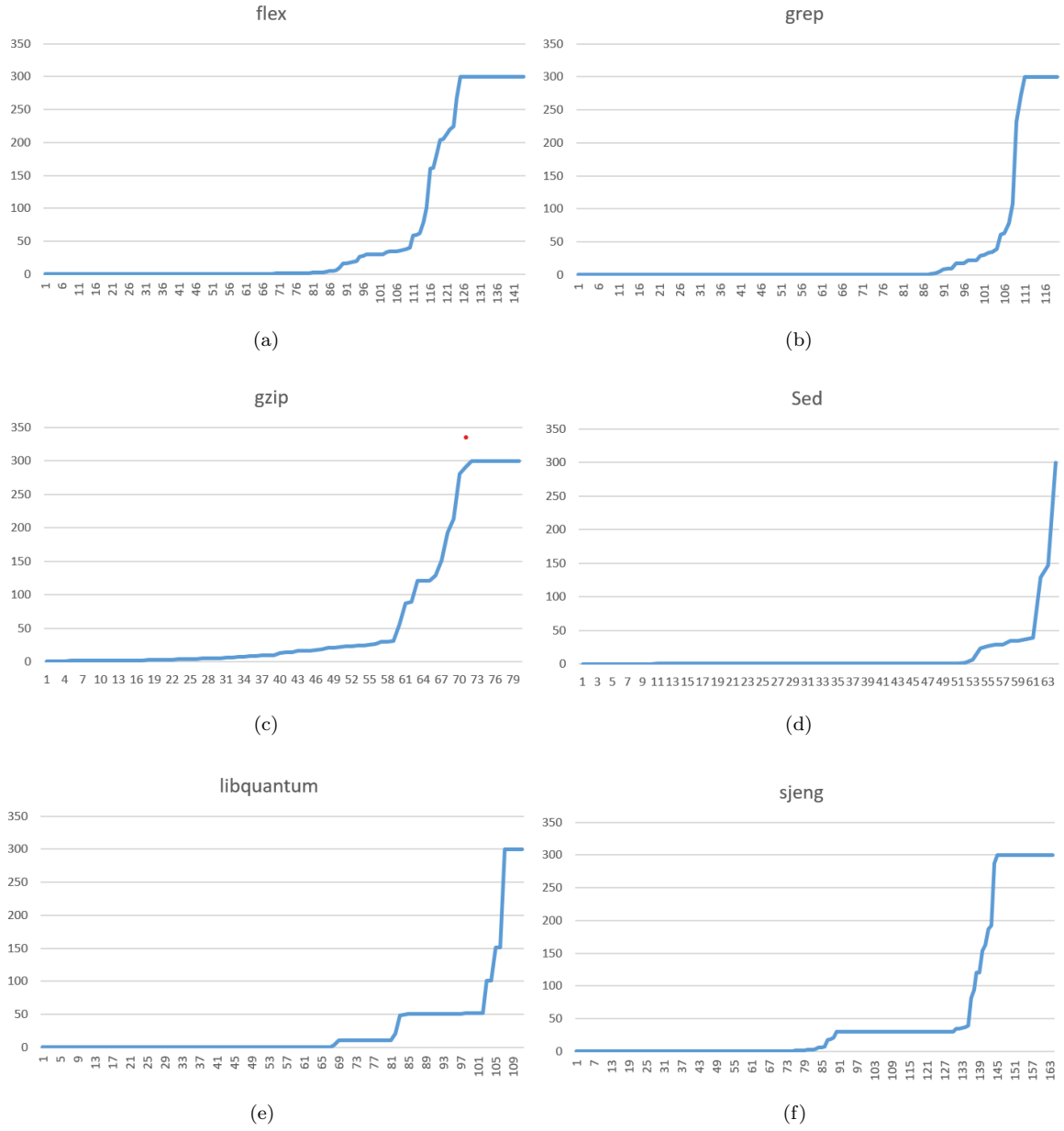


Figure 3.1: TC generation time distribution for 6 target subjects

Table 3.3: Branch Cov for Functions Not Reach Timeout2

Subject	# Function Not Reach Timeout2	# Branch	# Covered Branch
flex2.4.3	124	1855	348
grep2.0	110	3153	655
gzip1.0.7	71	1306	498
sed1.1.7	62	1858	330
libquantum0.2.4	111	677	236
sjeng11.2	144	5956	1535

The third column of Table 3.3 lists the total number of branches of functions whose test case generation time did not reach 300s, and the fourth column indicates the branch coverage achieved by $CR2^{base}$ for these functions.

It is worth noting that although $CR2^{base}$ finished the test case generation process for these functions within a given time (300s), the branch coverage achieved by $CR2^{base}$ is still not satisfactory (should be 100% in principle). So there may exist some weaknesses which prevent $CR2^{base}$ from generating more test cases. I will explain these weaknesses in detail in the next chapter.

Chapter 4. Limitations of The Current Concolic Testing Tool

As explained in Chapter 3, the branch coverage achieved by CR2^{base} is only 29.0% on average. There may exist some limitations in CR2^{base}. For this reason, a detailed analysis of each subject is necessary to discover the limitations. This chapter explains the limitations found while analyzing the source code of the target subjects.

4.1 Analysis Details

To discover the limitations of CR2^{base}, I first open the **lcov** report of each subject and then collect the groups of branches that CR2^{base} does not cover from the first line until the last line of the subject. Then 324 groups of not-covered branches are discovered, and each group contains 31 branches on average. For each group of not-covered branches, I utilize the test case generation log and **gdb** to analyze why these branches cannot be covered. Finally, I discovered 13 limitations of CR2^{base}. The short descriptions of all 13 limitations are listed in Table 4.1.

4.2 Limitations

4.2.1 FTCC: The Execution of First TC Crashes

CR2^{base} uses CROWN engine to run concolic testing. The effectiveness of CROWN engine depends on the execution result of the first TC. Specifically, suppose the execution of the first TC crashes. In that case, CROWN engine cannot generate more than 1 test case because the execution path of the first test case cannot be obtained, which is essential to generate the next test input.

Therefore, it is the user's responsibility to choose a good seed test case to avoid the crash of the first TC. If the seed test case is not specified, CROWN assigns zero to all symbolic variables for the first TC and executes the first TC by default. CR2^{base} cannot determine the seed test case for each unit. In this case, the default behavior of CROWN engine is adopted.

Figure 4.1 shows one example that causes the crash of the first TC. In the example, for the first TC of the function *example1*, CR2^{base} assigns the parameter a of *example1* with zero. At line 15, the division by zero error happens (the value of a is zero), which prevents CR2^{base} from generating more than one test cases.

4.2.2 NSEF: No Support for External C Library Functions (Except for C File-Handling Functions)

The CROWN engine tries to obtain the symbolic path formula (SPF) from the source code of the target program and the execution of the current test case. Then CROWN negates a constraint in the SPF to generate the next test case.

In principle, all the branch conditions that are executed by the current test case should be included into the SPF. However, there may exists branch conditions that use external functions whose source code are not available (e.g., library function). In this case, CROWN cannot obtain the constraints for these

Table 4.1: 13 Limitations

Number	Abbreviation	Description
1	FTCC	The Execution of First TC Crashes
2	NSEF	No Support for External C Library Functions (Except for C File-Handling Functions)
3	NSFF	No Support for External File-handling Functions
4	NSFP	No Support for Function Pointers
5	NSGP	No Support for Global Void Pointer
6	NSLS	No Support for Local Static Variable
7	NSSF	No Support For Symbolic File
8	NSSP	No Support For Symbolic Pointers
9	RT1	Timeout1 Is Reached
10	TIWC	Target Program's Illegal Write to The Structure of $CR2^{base}$
11	UB	Existence of Unreachable Branches in Unit-testing
12	USV	Uncovered Branch Caused by The Unrealistic Symbolic Input Values
13	WDFS	Weakness of Dfs

```
/*
example1_driver(){
    int a;
    // a is assigned with 0 for the 1st TC
    SYM_int(a);
    example1(a)
}
*/
int example1(int a)
{
    int b = 0;
    for(int i=0; i<a; i++){
        b = b + a;
    }
    int c = b / a;
    return c;
}
```

Figure 4.1: The example of FTCC

branch conditions so that the "then" or "else" branch of these branch conditions cannot be covered. Figure 4.2 explains this kind of limitation.

- At line 11, s is a parameter of function *example*2 and it points to an array which has three (array size) symbolic element.
- At line 14, an external function *strcmp* is called, and this function returns **-1**. The "then" branch of the "if" statement at line 15 cannot be covered because of the following two reasons:

  Reason 1: *strcmp* is an external function, so CROWN engine cannot write the constraint ($strcmp(s, \text{"}aaa\text{"}) != 0$) into the SPF. Not to mention negating this constraint to cover the "then" branch.

```
/*
example2_driver(){
    char *s;
    s = malloc(3*sizeof(char));
    for(int i=0; i<3; i++){
        SYM_char(s[i]);
    }
    example2(s)
}
*/
int example2(char* s)
{
    ...  //s is not altered
    if(strcmp(s, "aaa") == 0){
        ... // not covered
    }
    else{
        ...
    }
    ...  // s is not used by other conditions
    return 0;
}
```

Figure 4.2: The example of NSEF

Reason 2: s is not used by other branch conditions (i.e., the value of s would not be altered)

4.2.3 NSFF: No Support for External File-handling Functions

Similar to section 4.2.2[1]. If a branch condition uses the variable whose value is assigned by the file-handling functions, the "then" or "else" branch of this branch condition would not be covered. Figure 4.3 explains this problem.

At line 4, *var* is assigned with the return value of *getchar* (i.e., get a character from stdin). The value of *var* becomes concrete, so five branches (i.e., line 6, line 9, line 12, line 15, and line 19) cannot be guaranteed to be fully covered by CR2$^{\text{base}}$.

4.2.4 NSFP: No Support for Function Pointers

For a variable fp of function pointer type, CR2$^{\text{base}}$ cannot determine which function should be assigned to fp. Hence, the NULL value is assigned to the function pointer fp, which may cause crash in the unit-testing. Figure 4.4 shows the crash caused by this limitation.

- At line 1, fp is a function pointer. CR2$^{\text{base}}$ is not intelligent enough to obtain all the functions that may be assigned to fp. Hence, fp is assigned with **NULL**
- At line 22, crash happens (dereference on NULL pointer) when executing the target function *example*4 , which prevents CR2$^{\text{base}}$ from generating more than one test cases.

[1]The reason why I separate file-handling functions from external functions is because all the 6 subjects uses file-handling functions.

```
int example3()
{
    ...
    char var = getchar(); // get a character from stdin
    if(var == 'a'){
        ...
    }
    else if (var == 'b'){
        ...
    }
    else if (var == 'c'){
        ...
    }
    else if (var == 'd'){
        ...
    }
    ...
    else{
        ...
    }
    return 0;
}
```

Figure 4.3: The example of NSFF

```
void (*fp)(); // function pointer
void cand1();
void cand2();
void tmp_func1(){
    ...
    fp = cand1;
    ...
}
void tmp_func2(){
    ...
    fp = cand2;
    ...
}
/*
example4_driver(){
    ...
    fp = 0; // fp is assigned with NULL
    example4();
}
*/
void example4(){  // fp
  fp();
  ...  // not covered
}
```

Figure 4.4: The example of NSFP

```
1  void* glob; // function pointer
2
3  /*
4  example5_driver(){
5      ...
6      glob = 0; // glob is assigned with NULL
7      example4();
8  }
9  */
10 void example5(){  // glob = NULL
11     int * a = glob;
12     if(a == NULL){
13       ...
14     }
15     else{
16       ...  // not covered
17     }
18     ...
19 }
```

Figure 4.5: The example of NSGP

4.2.5 NSGP: No Support for Global Void Pointers

For a variable of void pointer type, CR2^{base} cannot determine which concrete pointer type (e.g., int* , char*) the void pointer should be casted to, so NULL value is assigned naively. If a branch condition depends on the value of the void pointer or the value of the variable that the void pointer points to, the branch may not be covered. Figure 4.5 shows an example of this limitation.

- At line 1, *glob* is a global void pointer.
- At line 6, *glob* is assigned with NULL because CR2^{base} cannot determine the type that *glob* should be casted to.
- At line 11, variable a is assigned with the value of *glob* (NULL).
- At line 12, only the "then" branch of the "if" statement can be covered by CR2^{base}. The "else" branch (line 15- line 17) cannot be covered because the value of a is NULL.

4.2.6 NSLS: No Support for Local Static Variables

CR2^{base} cannot set local static variables as symbolic. And for the execution of each test case, CR2^{base} calls the target function only once. So if the target function consists of local static variables and a branch condition in the target function uses the these variables, there may exist not-covered branches. The detailed example in Figure 4.6 displays this problem.

- At line 9, the local static variable i is assigned with zero.
- At line 10, the "then" branch of "if" statement cannot be covered because the driver code only calls the target function once, which makes the value of i always be zero.

```
/*
example6_driver(){
    ...
    example6(); // example6 is called only once for each execution
}

*/
void example6(){
    static int i = 0;
    if(i == 1){
      ...  // not covered
    }
    else{
      ...  // example6() is not called
    }
    i = i + 1;
}
```

Figure 4.6: The example of NSLS

```
FILE * f; // FILE pointer
...
/*
example7_driver(){
    ...
    f=NULL;  // FILE pointer f is assigned with NULL
    example7();
}
*/
void example7(){
    if(!f){
      ...  // not coverd
    }
    else{
      ...
    }
    ...
}
```

Figure 4.7: The example of NSSF

4.2.7 NSSF: No Support for Symbolic File

In the CR2^{base}, the symbolic file is not supported (i.e, NULL is assigned to the FILE pointer by the driver function). Hence, some branches,e.g., a branch condition checks the value of the FILE pointer, may not be covered by CR2^{base}. The example in Figure 4.7 shows the problem.

- At line 1, a FILE pointer f is declared
- At line 6, f is assigned with NULL by the driver function.
- At line 11, only the "else" branch of "if" statement (i.e., lines 14 - 16) can be covered by CR2^{base} because the value of f is NULL.

```
1  /*
2  example8_driver(){
3      ...
4      char * buf = malloc(3*sizeof(char));
5      for(int i=0; i<3; i++){
6          SYM_char(buf[i]);
7      }
8      int size;
9      SYN_int(size);  // the initial value of size is 0;
10     example8(buf, size);
11 }
12 */
13 void example8(char *buf, int size){
14     char * pb = buf;
15     char * pe = pb +  size;
16     ... // size is not used by other branch conditions
17     if(pb != pe){
18         ... // not covered
19     }
20     ... // size is not used by other branch conditions
21
22 }
```

Figure 4.8: The example of NSSP

4.2.8 NSSP: No Support for Symbolic Pointer

CR2$^{\text{base}}$ cannot declare the pointer variable as symbolic variable because the pointer's value (address) should not be symbolic. So if a branch condition uses pointer variables, CR2$^{\text{base}}$ may not cover this branch condition because the pointer variables are not symbolic variables, and CROWN engine cannot write the constraint of this branch condition into SPF. The toy program in figure 4.8 explains this kind of limitation.

- At line 9, *size* is assigned with zero (by the marco *SYM_int*). The value of *size* is never altered since it is not used by any branch conditions.
- At line 15, *pb* and *pe* point to the same address for the value of *size* is 0.
- At line 17, the "then" branch of "if" statement cannot be covered because the condition $pb == pe$ is always satisfied and CR2$^{\text{base}}$ cannot support the symbolic pointer.

4.2.9 RT1: Timeout1 Is Reached

If the execution of current test cases reaches *timeout1*, CR2$^{\text{base}}$ will terminate the execution. In this case, the coverage of current test case cannot be recorded by CR2$^{\text{base}}$. The infinite loop is one of the reasons that cause the execution to reach *timeout1*, and figure 4.9 explains this problem.

- At line 5 and line 7, a and b are assigned with zero by the driver function.
- At line 14, the "while" loop will never stop because the value of a and b are always 0 and $0*2 == 0$, so *timeout1* is reached, and the coverage of the current test case cannot be recorded.

```
/*
example9_driver(){
    ...
    int a;
    SYM_int(a);
    int b;
    SYM_int(b);
    example9(a,b);

}
*/
void example9(int a, int b){
    ... // a and b are not altered
    while(a <= b){
        a = a * 2;
    }
    ... // a and b are not altered
}
```

Figure 4.9: The example of RT1

```
#0  0x00000000005ad477 in crown::SymbolicMemoryWriter::MemElem::read (
    this=0x808480, addr=8355808, n=8, val=..., next_elem=0x7e6cb0)
    at libcrown/symbolic_memory_writer.cc:85
#1  0x00000000005acdc7 in crown::SymbolicMemoryWriter::read (this=0x7dccb8,
    addr=8355808, val=...) at libcrown/symbolic_memory_writer.cc:224
#2  0x0000000000595aa4 in crown::SymbolicInterpreter::Load (this=0x7dcc20,
    id=10025, addr=8355808, value=...) at libcrown/symbolic_interpreter.cc:77
#3  0x00000000005898be in __CrownLoad (id=10025, addr=8355808, ty=6, val=0,
    fp_val=0) at libcrown/crown.cc:360
#4  0x00000000004870a3 in re_match_2 (bufp=0x7f7fd0, string1=0x7e71b0 "",
    size1=0, string2=0x7f2210 "", size2=0, pos=0, regs=0x7f28d0, stop=0)
    at grep.c:4837
#5  0x0000000000486436 in re_search_2 (bufp=0x7f7fd0, string1=0x7e71b0 "",
    size1=0, string2=0x7f2210 "", size2=0, startpos=0, range=0,
    regs=0x7f28d0, stop=0) at grep.c:4618
```

Figure 4.10: Example Crash Trace of TIWC

4.2.10 TIWC: Target Program's Illegal Write to The Structure of CR2$^{\text{base}}$

CR2$^{\text{base}}$ uses memory to record the SPF obtained from the execution of the current test case. In principle, the memory space used by CR2$^{\text{base}}$ and the one used by the subject should not conflict with each other. However, the collision may happen (e.g., illegal memory access) and CR2$^{\text{base}}$ engine cannot continue the test case generation process. Figure 4.10 shows an example in function *re_search_2* of file "grep.c" of subject "grep".

As crash site #0, the function $crown :: SymbolicMemoryWriter :: MemElem :: read$, which is an internal function of CROWN engine, crashes due to target program's illegal write to the memory used by CR2$^{\text{base}}$. The crash of CR2$^{\text{base}}$ terminates the test case generation process.

```
1  #define INC(a) if(a < 0) b+=1; else return;
2  /*
3  example10_driver(){
4      ...
5      int a;
6      SYM_int(a);
7      example10(a);
8  }
9  */
10 void example10(int a){
11     int b = 0;
12     INC(a);
13     ...// a is not altered
14     INC(a);
15 }
```

Figure 4.11: The example of UB

4.2.11 UB: Existence of Unreachable Branches in Unit-testing

There may exist an unreachable branch in the target program. The SPF of the unreachable branch is UNSAT, so that CR2$^{\text{base}}$ cannot generate a test case to reach the branch. Figure 4.11 shows the example. Both line 12 and line 14 use the macro INC declared at line 1.

- At line 12, if the "then" branch is covered, the value of a should be less than zero, otherwise the function $example10$ will be terminated (by executing "return" statement)
- At line 14, the value of a is always less than zero, and the "else" branch cannot be covered.

4.2.12 USV: Uncovered Branch Caused by The Unrealistic Symbolic Input Values

The CR2$^{\text{base}}$ uses a black-box solver to generate values for all symbolic variables according to the given SPF. The values obtained from the solver may not appear in the system-level execution of the target function. Then crash may happen, which makes CR2$^{\text{base}}$ unable to record the crash test case coverage. The example in figure 4.12 shows how the unrealistic values cause crash.

At line 12, to cover the "then" branch of "if" statement, the SMT solver outputs 4296947295 and assigns it to a. In such case, line 16 will access $buf[10000000]$ which causes crash because the array buf only contains 10 elements. Due to the crash, the coverage for the "then" branch of line 11 cannot be recorded.

4.2.13 WDFS: Weakness of Dfs

The dfs (default strategy used by CR2$^{\text{base}}$) tries to explore each execution path in a depth-first-search way. This strategy has a limitation on exploring loop statement. Because the loop statements contain more execution paths than other statements. If dfs strategy is applied, all the $timeout2$ may be consumed on exploring loop statement only.

The example in Figure 4.13 and its execution paths in Figure 4.14 explains this problem.

```
int buf[10];
/*
example11_driver(){\
    ...
    unsigned int a;
    SYM_unsigned_int(a);
    example11(a);
}
*/
void example11(unsigned int a){
    int lc;
    if(a != 0){
        int ind = 0;
        do{
            ...
            lc = buf[a++];
        }while(ind < a)
    }else{
        ...
    }

}
```

Figure 4.12: The example of USV

```
/*
example12_driver(){
    ...
    int a;
    SYM_int(a);
    example12(a);
}
*/
long example12(int a){
    long b = 0;
    if(a < 0) {
        ... // not covered
    }
    for(int i=0; i< a; i++){
        b+=i;
    }
    return b;
}
```

Figure 4.13: The example of WDFS

- At line 9, CR2$^{\text{base}}$ detects that function *example*12 has an parameter a of integer type, so CR2$^{\text{base}}$ declares a symbolic variable and uses it as the parameter of function *example*12 (lines 4 - 6)
- At line 11, a is a symbolic variable (assigned with 0), so the "for" loop at line 14 are explored first.
- At line 14, each non-negative value of a produces an unique execution path. For example, if a is assigned with 2, line 14 would be executed three times, if a is assigned with 10, line 14 would be executed 11 times. Hence, the "for" loop contains 2^{31} paths, and CR2$^{\text{base}}$ has to explore all the

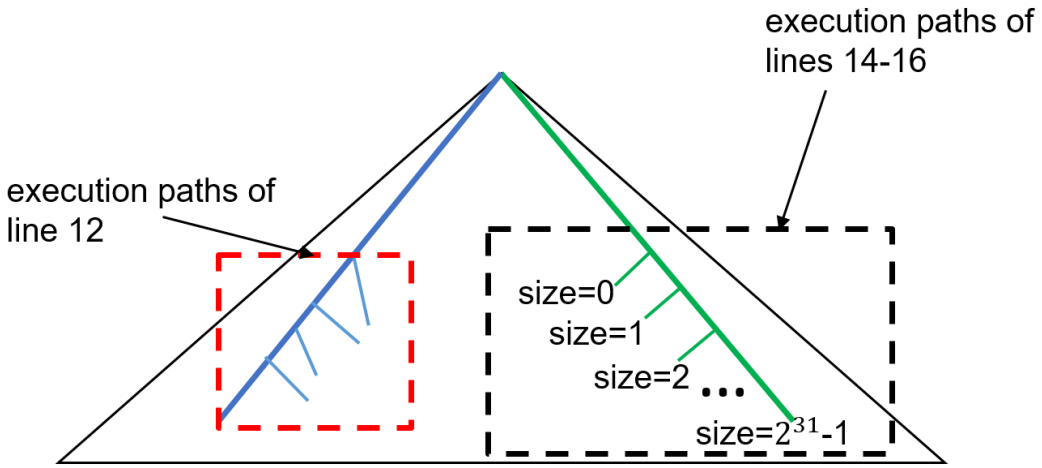


Figure 4.14: Execution paths of figure 4.13

paths before exploring line 12, i.e., generating 2^{31} test cases before reaching line 12.

Figure 4.14 shows the execution paths of figure 4.13. As we can see, CR2$^{\text{base}}$ has to cover all the paths of the right side before covering the left side. To avoid such case, CR2$^{\text{base}}$ should not be limited to using one search strategy, but utilize the advantages of multiple search strategies and adjust the search strategy adaptively to achieve higher coverage.

For example, we can create a strategy that gives priority to input values that can cover more branches instead of more paths, in this way, even though different input values may cover different paths, they may have the same branch coverage. For example, the value of a in the current test case is 2, and there are two candidate values of a (3 and 0) for the next test case. These two values have different execution paths, but if a is assigned with 3, the branch coverage is the same as the one when a is 2 (both cover the "if" and "else" branches of the "for" statement at line 14), so we choose 0 as the value of a for the next test case.

Chapter 5. Proposed Solutions

In chapter 4, I discovered 13 limitations of the baseline CR2^{base}. In this chapter, I will explain the six solutions for the limitations mentioned in chapter 4.

5.1 Apply Combined Strategy (to solve WDFS)

As explained in figure 4.13, the dfs strategy may make CR2^{base} spend all its running time on a complex statement, e.g., loop statement. So a single search strategy may not cover as many execution paths as possible within the specified time (given by the user). To avoid such a case, I propose a combined strategy that utilizes all four CROWN search strategies (i.e., dfs, rev-dfs, random, and cfg). Algorithm 1 shows how the combined strategy works.

- Line 1: The combined strategy first divides T (timeout2) into four time slots, each slot has $\frac{T}{4}$.
- Line 2: In the given time slot, the algorithm tries to generate test cases using the dfs strategy
- Lines 3 - 4: Check whether CROWN engine finishes its execution in the given time slot or not. If the dfs strategy ends in a slot, we assume all the paths of the target program are explored and then stop the test case generation process.
- Lines 6 - 9: In the opposite case, we assume that the complex statements (e.g., loop statement) make dfs strategy reach the given time slot. To solve this problem, I utilize different search strategies by stopping the dfs strategy and running the remaining three strategies (i.e., rev-dfs, random, cfg) in order (line 6 - line 8); each strategy uses a time slot. Then we accumulate all the test cases generated by the four strategies and output all of them (line 9).

5.2 Create Stub for All File-handling Functions (to solve NSFF)

I observed that all six subjects use at least one file-handling function (i.e., functions in "stdio.h"). The file-handling functions play an important role in the Linux system (operating system used by CR2^{base}). So I collected all the functions declared in "stdio.h" and replaced them with their corresponding stub functions.

Regarding stub function, I make it return symbolic variable (whose type is the same as the original function). Table 5.1 shows the detained information. The first and second columns show the return types in detail. The third to seventh columns show the names of all functions of "stdio.h".

The return types can be classified into three categories,i.e., primitive integer type (including char, int, long, and unsigned long), character pointer type, and FILE pointer type. The remaining part of this section will explain how to handle these three categories, respectively.

- **Primitive Integer Type**. As explained in figure 5.1 ,for functions that return integer type, the original function is replaced by stub function using macro (line 1: #define). The stub function will declare and return a symbolic variable according to the return type of the original function (lines 4 - 9).

Algorithm 1 Working Mechanism of Combined Strategy

Input: T: timeout2 which is giver by the user

Output: a sequence of TCs

1: $t \leftarrow \frac{T}{4}$
2: $TC1 \leftarrow Run(dfs, t)$ ▷ Run(x, y): run strategy x in y seconds, return test cases
3: **if** CROWN finishes in t **then**
4: $\quad TC \leftarrow TC1$
5: **else**
6: $\quad TC2 \leftarrow Run(revdfs, t)$
7: $\quad TC3 \leftarrow Run(random, t)$
8: $\quad TC4 \leftarrow Run(cfg, t)$
9: $\quad TC \leftarrow TC1 + TC2 + TC3 + TC4$
10: **end if**
11: **return** TC

Table 5.1: Return Types of File-Handling Functions

Return Type		Function Name				
Primitive Type	void	clearerr	perror	rewind	setbuf	
	int	fclose	feof	ferror	fflush	fgetc
		fgetpos	fprintf	fputc	fputs	fscanf
		fseek	fsetpos	getc	getchar	printf
		putc	putchar	puts	remove	rename
		scanf	setvbuf	sprintf	sscanf	ungetc
		vfprintf	vprintf	vsprintf		
	long	ftell				
	unsigned long	fread	fwrite			
Pointer Type	char *	fgets	gets	tmpnam		
	FILE *	fopen	freopen	tmpfile		

```
#define getc(x,...) stub_getc();
...
// getc returns integer
int stub_getc(){
    int res;
    SYM_int(res);
    return res;
}
```

Figure 5.1: Stub Function that Returns Integer

- **Character Pointer Type**. For functions that return character pointer type, their stub functions will declare a pointer that points to a symbolic array, and the length of the symbolic array is the same with $< n >$ (array size, the argument of CROWN2.0). Figure 5.2 shows an example.
- **FILE Pointer Type**. Like the character pointer type, the stub functions would use a symbolic variable to return either a NULL pointer or a FILE pointer that points to a real file. Figure 5.3

```
#define gets(x,...) stub_gets();
...
char stub_gets(){
    char * res = malloc(N*sizeof(char)); // N is array size
    for(int i=0; i< N; i++){
        SYM_char(res[i]);
    }
    return res;
}
```

Figure 5.2: Stub Function that Returns Character Pointer

```
#define freopen(x,...) stub_freopen();
...
char stub_freopen(){
    char tmp;
    SYM_char(tmp);
    FILE * res = fopen("dummy.txt", "a+");
    return tmp>=0?res:NULL;
}
```

Figure 5.3: Stub Function that Returns FILE Pointer

shows an example.

5.3 Utilize Other Concolic Testing Tool (to solve NSSP, NSEF)

As it is shown in figure 4.2, CR2^{base} has its weakness in exploring the branch whose branch condition includes external library functions and symbolic pointers.

It is difficult to overcome this limitation because I have to check the source code of all the library functions, which is a time-consuming task. But I found a concolic testing tool KLEE[1], which has two very useful features[1], i.e., supporting symbol pointers and some library functions. Hence, I use KLEE as an auxiliary tool of CROWN2.0 to overcome the limitation on external library functions. The following two parts will explain how KLEE supports the above two features.

5.3.1 Support Symbolic Pointer

If a conditional statement contains only symbolic pointers, KLEE will try to find a statement with equivalent syntax that can help generate symbolic path formula to replace the original conditional statement. Figure 5.4 shows how KLEE solves the NSSP examples mentioned at Figure 4.8.

Line 17: KLEE detects that statement $pb! = pe$ is equivalent with $size! = 0$ (Ref: L15), so the new statement $size! = 0$ will replace the original statement to generate test cases to cover the branches at line 18.

[1]In the future, I will add these two features into the CROWN engine

```
13  void example8(char *buf, int size){
14      char * pb = buf;
15      char * pe = pb +  size;
16      ... // size is not used by other branch conditions
17      if(pb != pe){                    size!=0
18          ... // not covered
19      }
20      ... // size is not used by other branch conditions
21
22  }
```

Figure 5.4: Example of Solving NSSP

```
11  int example2(char* s)
12  {
13      ...  //s is not altered
14      if(strcmp(s, "aaa") == 0){
15          ... // not covered
16      }
17      else{
18          ...
19      }
20      ...  // s is not used by other conditions
21      return 0;
22  }
```

(a) Code example in Figure4.2

```
int strcmp(register const Wchar *s1, register const Wchar *s2)
{
    int r;
    while (((r = ((int)(*((Wuchar *)s1))) - *((Wuchar *)s2++))
            == 0) && *s1++);
    return r;
}
```

(b) KLEE's implementation of *strcmp*

Figure 5.5: Example of Solving NSEF

5.3.2 Support External Library Function

Suppose the target project executes a library function. In that case, KLEE will first check whether the library function is in the list of library functions implemented separately by KLEE, e.g., all the functions defined at "string.h". If so, KLEE will use its implementation of the library function and continue the process of test case generation. Figure 5.5 shows how KLEE solves the NSSP example mentioned at Figure 4.2.

Line 14: KLEE detects that statement *strcmp* is a library function defined at "string.h" (belongs to POSIX library), so KLEE would replace *strcmp* with the one implemented by KLEE to generate test cases to cover line 15, e.g., assign symbolic variable s with "aaa".

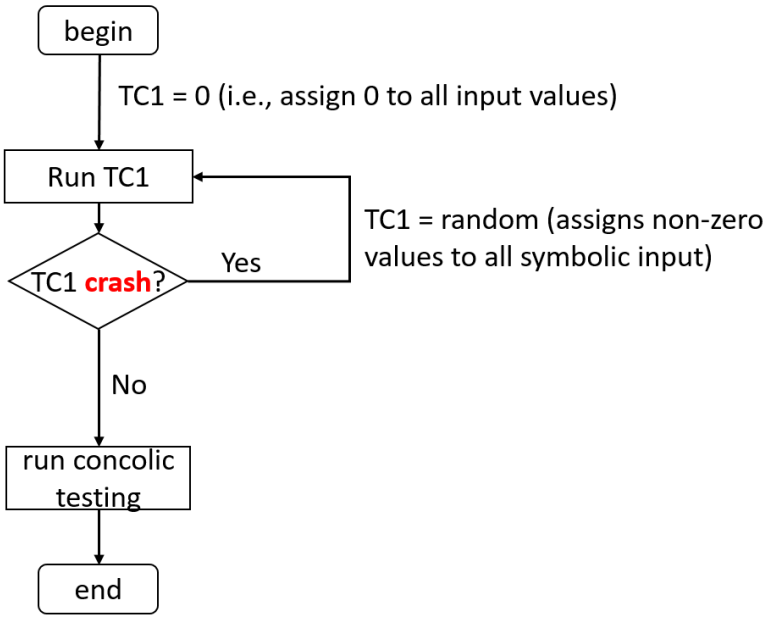


Figure 5.6: Mechanism for Handling The Crash of First TC

5.4 Execute The First TC with Random Inputs (to solve FTCC)

The CR2^{base} will terminate its test case generation process if the execution of the first test case crashes. The default behavior of CROWN engine may cause the crash, i.e., all inputs of the first test case are zero. I made CROWN2.0 generate a random input if the first test case crashes to handle such a case. The detail is shown in Figure 5.6.

The revised version of CROWN2.0 would check the execution result of the first test case. If the first test case crashes, CROWN2.0 would assign all symbolic input variables with non-zero values and execute the new test case again until finding a non-crash test case. Then CROWN2.0 can continue its test case generation process.

5.5 Execute The Target Function Multiple Times (to solve NSLS)

As explained in section 4.2.6, the driver only calls the target function once. If the target function has a branch condition that uses static local variables, CR2^{base} cannot guarantee to cover all the branches of the target function. For this case, I slightly modified the driver and made it call the target function M times, where the users decide the value of M. Figure 5.7 shows an updated version of the driver code. The original driver code calls the target function only once (Lines 2 - 6).

The updated driver of $func$ is displayed (lines 9 - 15), where the driver would execute the target function M times (line 13). In this case, the "then" branch (line 24) can be covered if M is greater or equal to 2.

```
/* original driver
func_driver(){
    int a;
    SYM_int(a);
    func(a);
}
*/
// updated driver
func_driver(){
    int a;
    SYM_int(a);
    for(int i=0; i<M; i++){  // M is decided by the users
        func();
    }
}
// target function
void func(int a){
    static int i = 0;
    if(i == 1){
      ...  // not covered
    }
    else{
      ...  // func is not called
    }
    i = i + 1;
}
```

Figure 5.7: Driver that Calls Target Function Multiple Times

5.6 Static Analysis for Pointer Types (to solve NSGP, NSFP)

CR2$^{\text{base}}$ cannot determine the actual type of what a global void pointer is cast to, and the function pointer is not supported by CR2$^{\text{base}}$. To solve these two limitations, I make CROWN2.0 analyze all **assignment statements** (e.g., "$a = b$") to determine the type that a void pointer should be casted to and which function should be assigned to function pointers.

- **Global Void Pointer**

  CROWN2.0 identifies the type that the void is **first casted to** and then allocates the corresponding space for this void pointer. Then CROWN2.0 will set the symbolic environment for the given space. Figure 5.8 shows an example of how this approach works. At line 1, *glob* is a global void pointer. At line 3, *glob* is casted to a pointer to integer (int *). Hence, CROWN2.0 regards *glob* as "int *" type and makes it points to an integer array (Line 10). Then, CROWN2.0 sets each element of the integer array as symbolic variable (Line 12).

- **Function Pointer** For a function pointer fp, CROWN2.0 statically examines all the assignment statements and identifies all the functions that are directly assigned[2] to fp. Then CROWN2.0 uses a symbolic variable *choice* and a "switch-case" statement to determine which function should be assigned to fp.

  For example, figure 5.9 has function pointer fp. After CROWN2.0 identifies that fp may point to

[2] I will add support for indirect (i.e., $b = cand1; fp = b;$) assignment in the future.

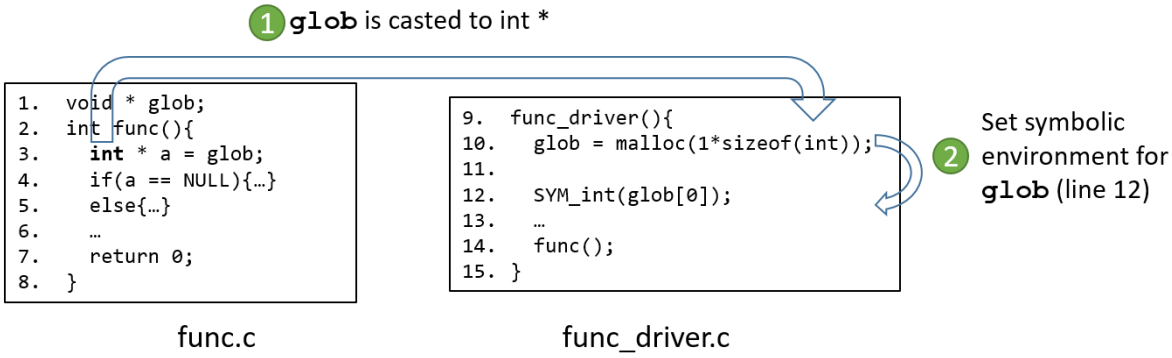


Figure 5.8: Solutions for Global Void Pointer

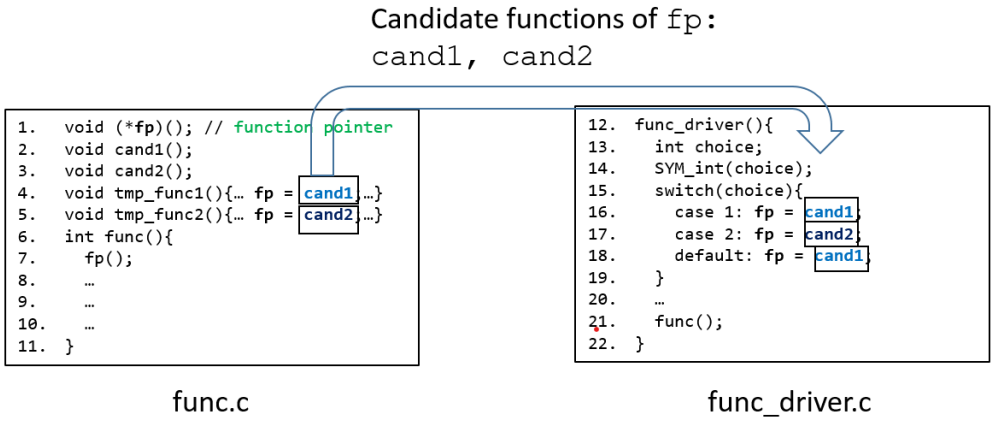


Figure 5.9: Example of Symbolic Setting for Function Pointer *fp*

cand1 or *cand2* at line 4 or line 5 respectively, it assigns *cand1* or *cand2* to *fp* depending on the value of a symbolic variable *choice* (lines 14-19), as shown in the right part of figure 5.9.

Chapter 6. Coverage Report and Crash Analysis

To evaluate the effectiveness of the proposed techniques, both branch coverage and bug detection ability are used as metrics.

6.1 Coverage Achieved after Applying Solutions

To improve branch coverage, I developed CROWN^{imp} that extends the CR2^{base} by implementing all the six features aforementioned in Chapter5. The arguments of CROWN^{imp} are set as follows.

- For *Timeout1* and *Timeout2*, I set them as 60s and 300s respectively (same with the one used by CR2^{base}).
- For searching strategy, I use combined strategy in the experiments. The combined strategy runs five sub-strategies in order (dfs - rev_dfs - random - cfg - KLEE). Each sub-strategy has a running time of 60s (=300s/5)
- For array size, I use the default value used by CROWN2.0 (three).
- For the additional argument M (# of times a target function is called), I use its default value 2.

The experiments of CROWN^{imp} are performed on eight machines, each of which is equipped with an AMD 8-core Ryzen 7 3800XT (3.9 GHz) CPU and 16 GB RAM, running Ubuntu 18.04 64 bit version.

Table 6.1 shows the experimental results of CR2^{base} and CROWN^{imp}. The first column shows subject names. The second column shows the number of branches for each subject. The third and fourth columns show branch coverage achieved by CR2^{base} and CROWN^{imp}.

The experimental results in Table 6.1 show that CROWN^{imp} outperforms CR2^{base} on all 6 target subjects in terms of branch coverage. The improvement is 77% on average for all six target subjects (up to 188% for a subject).

In addition, to demonstrate that my solutions can improve branch coverage, I have provided eight coverage improvement examples from the target projects. For each example, the lines with the blue background are covered lines, and the lines within the red boxes are the not-covered lines.

Table 6.2 shows the number of covered branches after solving the eight limitations. The first column shows the subject name. The second to tenth columns show the number of newly covered branches after solving each limitation[1].

6.1.1 Improvement by Solving FTCC

FTCC stands for "the execution of the First TC Crashes". If the first test case crashes, then CR2^{base} cannot generate more than one test case because the execution path of the first test case is not available, which is essential to generate the next test inputs. Hence, the coverage of the target function is 0%.

To tackle the above problem, CROWN^{imp} assigns all input symbolic variables with random non-zero values and executes the new test case again until finding a non-crash test case. Figure 6.3 shows how CROWN^{imp} solves FTCC in function *add_current* of file "newbook.c" of subject "sjeng".

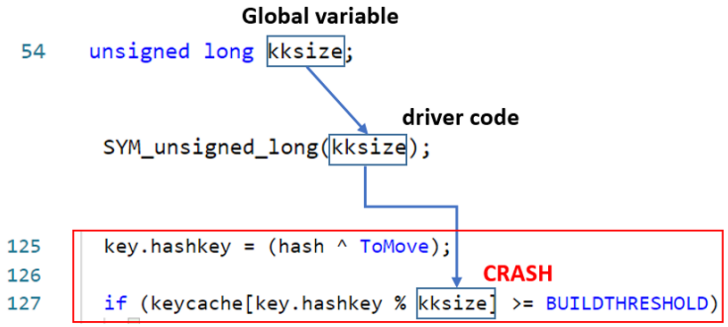
[1] I am not sure the number of branches in the last column is covered by solving which limitation. I will analyze the detailed reason in the future.

Table 6.1: Experimental Results of CR2^{base} and CROWN^{imp}

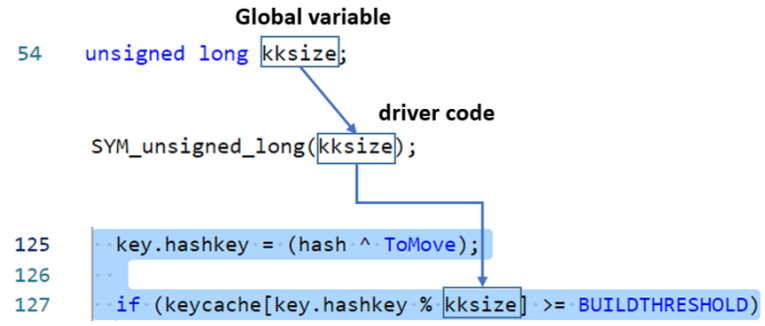
Subject Name	# of Branch	Branch Cov of CR2^{base}	Branch Cov of CROWN^{imp}	Average TC Generation Time
flex 2.4.3	2021	22.6%	53.2%	122.7
grep 2.0	3416	22.7%	52.7%	56.7
gzip 1.0.7	1446	41.4%	50.1%	98.9
sed 1.17	2395	17.7%	51.1%	54.9
sjeng 11.2	6364	27.9%	50.4%	93.7
libquantum 0.2.4	724	41.6%	50.1%	9.8

Table 6.2: Coverage Improvement by Solving Eight Limitations

Subject Name	# Covered Branches								
	FTCC	NSEF	NSFF	NSFP	NSLS	NSSP	USV	WDFS	Unknown
flex 2.4.3	77	3	356	0	0	22	104	0	57
grep 2.0	29	98	1	0	4	537	223	75	57
gzip 1.0.7	18	20	0	1	0	0	85	0	1
sed 1.17	3	30	35	0	4	597	101	0	34



(a) Coverage of CR2^{base}



(b) Coverage of CROWN^{imp}

Figure 6.1: Function *add_current* of file "newbook.c" of subject "sjeng"

Why CR2^{base} Cannot Cover

Line 127: Symbolic global variable *kksize* is assigned with **0** (by the driver code) for the first test case[2]. And division-by-zero crash happens, which prevents CR2^{base} from generating more than one test

[2]SYM_type(var) assigns var with 0 for the first test case

cases.

Why CROWN^{imp} Can Cover

Line 127: CROWN^{imp} detects that the first TC crashes and tries to assign *kksize* with random non-zero value ,e.g., one. Then the crash at line 127 can be avoided.

6.1.2 Improvement by Solving NSEF

NSEF stands for "No Support for External C Library Functions". A branch condition **B** may use POSIX library functions whose source code is not available for CROWN2.0. Besides, CR2^{base} does not replace POSIX library functions with stub functions. Therefore, CR2^{base} may fail to cover both the "then" and "else" branches of the branch condition **B** because CR2^{base} cannot obtain the symbolic path formula of **B** that cannot be negated to generate the next test case.

To solve this limitation, CROWN^{imp} utilizes KLEE to replace some POSIX library functions, e.g., all functions declared in "string.h", with the ones that are implemented by KLEE. By doing so, CROWN^{imp} obtains the symbolic path formula of library functions for generating the next test case. Figure 6.2 shows how CROWN^{imp} solves NSEF in function *regerror* of file "sed.c" of subject "sed".

Why CR2^{base} Cannot Cover

Line 7102: *msg* is an array which has three elements and each of its element is assigned with zero, which makes the external function *strlen* return zero. So *msg_size* is assigned with one.

Line 7104: *errbuf_size* is an symbolic unsigned integer, no value of *errbuf_size* can satisfy both condition $errbuf\_size! = 0$ (L7104) and $errbuf\_size < 1$ (L7106). Therefore, lines 7108-7109 cannot be covered.

Why CROWN^{imp} Can Cover

CROWN^{imp} uses KLEE which detects that *strlen* (L7102) is a POSIX library function and uses KLEE's own implementation of *strlen*. So *msg_size* (L7102) can be assigned with two ,e.g., *msg*="aa". Then lines 7108-7109 can be covered, CROWN can output the value one to satisfy the condition $errbuf\_size! = 0$ (L7104) and $errbuf\_size < 2$ (L7106) .

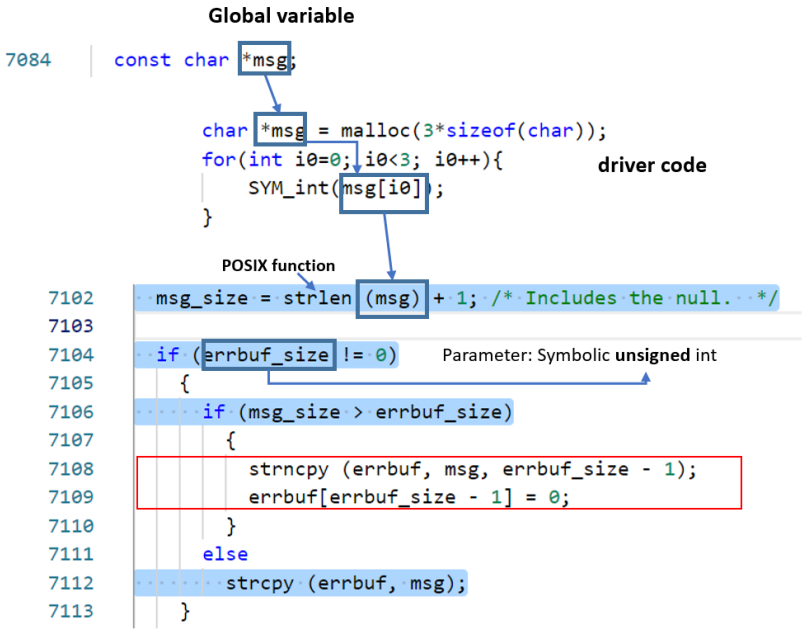
6.1.3 Improvement by Solving NSFF

NSFF stands for "No Support for External File-Handling Functions". Similar to NSEF, if a branch condition uses a variable whose value is assigned by the file-handling functions, then one of the "then" and "else" branches of this branch condition may not be covered because file-handling functions cannot return symbolic variables.

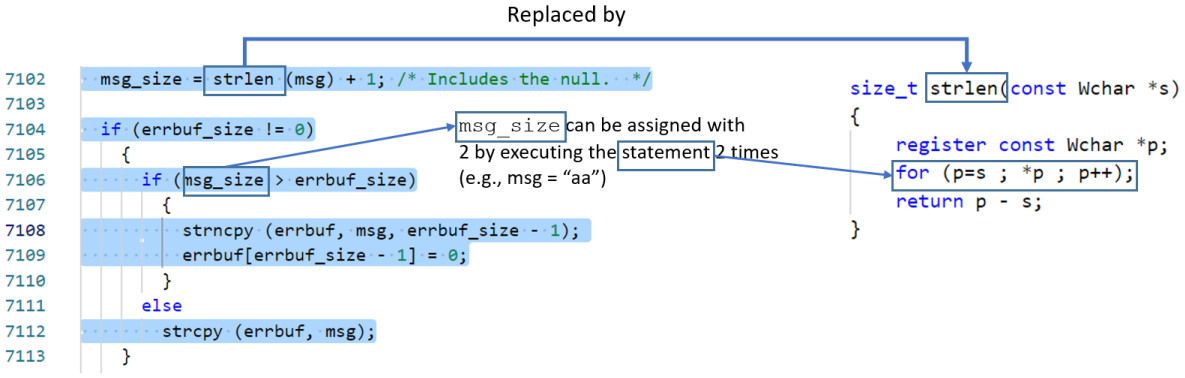
To solve this problem, CROWN^{imp} replaces all file-handling functions with the stub functions. The stub functions can return symbolic variables so that the not-covered branches can be covered. Figure 6.3 shows how CROWN^{imp} solves NSFF in function *read_pattern_space* of file "sed.c" of subject "sed".

Why CR2^{base} Cannot Cover

The CR2^{base} cannot make the external file-handling function $feof$ returns non-zero value, which makes line 1739 not-covered.

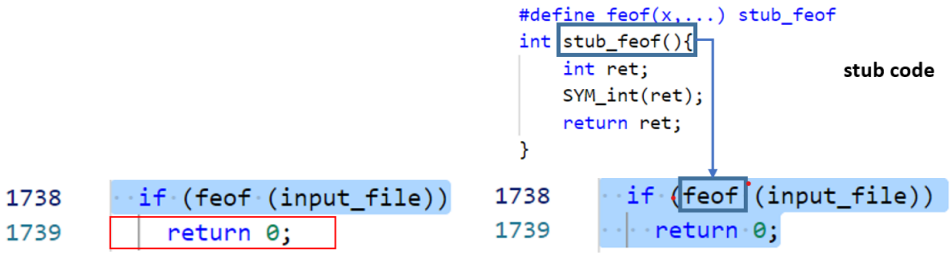


(a) Coverage of CR2$^{\text{base}}$



(b) Coverage of CROWN$^{\text{imp}}$

Figure 6.2: Function *regerror* of file "sed.c" of subject "sed"

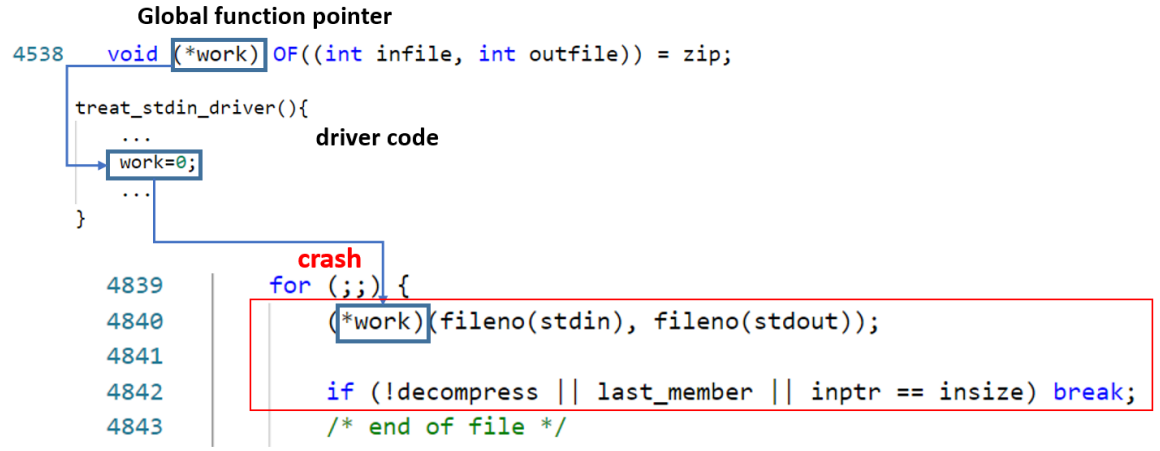


(a) Coverage of CR2$^{\text{base}}$ (b) Coverage of CROWN$^{\text{imp}}$

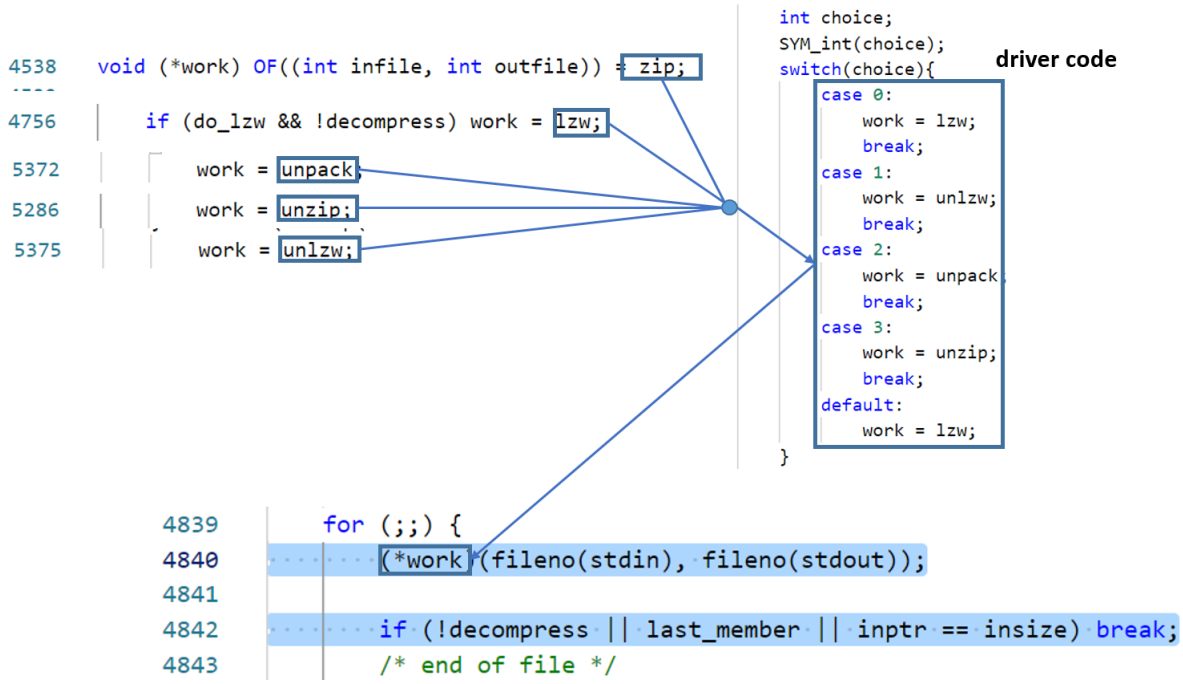
Figure 6.3: Function *read_pattern_space* of file "sed.c" of subject "sed"

Why CROWN$^{\text{imp}}$ Can Cover

The CROWN$^{\text{imp}}$ replaces $feof$ with stub function $stub\_feof$ that returns a symbolic integer variable, in this case, line 1739 can be covered.



(a) Coverage of CR2$^{\text{base}}$



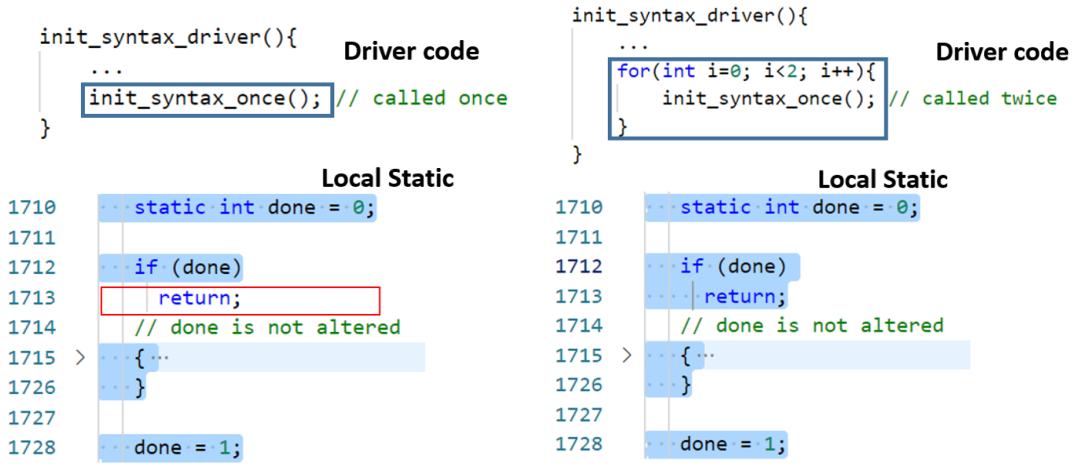
(b) Coverage of CROWN$^{\text{imp}}$

Figure 6.4: Function *treat_stdin* of file allfile.c of subject gzip

6.1.4 Improvement by Solving NSFP

NSFP stands for "No Support for Function Pointers". The driver code generated by CR2$^{\text{base}}$ cannot determine which function should be assigned to function pointer fp. Instead, a NULL pointer is assigned to fp, which can cause null-dereference crash.

CROWN$^{\text{imp}}$ solves this limitation by examining all the functions that are assigned to fp and using a symbolic variable and a "switch-case" statement to decide which function should be assigned to fp. Figure 6.4 shows an example of function *treat_stdin* of file "allfile.c" of subject "gzip" to demonstrate that this problem is solved by CROWN$^{\text{imp}}$.



(a) Coverage of baseline CR2^{base}

(b) Coverage of CROWN^{imp}

Figure 6.5: Function *init_syntax_once* of file "grep.c" of Subject "grep"

Why CR2^{base} Cannot Cover

The driver generates by CR2^{base} just assigns the global function pointer *work* with NULL value, which causes NULL-dereference crash at line 4840.

Why CROWN^{imp} Can Cover

The CROWN^{imp} detects that *work* has five candidate functions (L4538, L4756, L5372, L5286, L5375), so it generates a driver which uses a symbolic variable to decide which candidate functions should be assigned to *work*. In such case, *work* will never be NULL and line 4840 can be covered.

6.1.5 Improvement by Solving NSLS

NSLS stands for "No Support for Local Static Variables". The CR2^{base} cannot set local static variables as symbolic, and the driver code generated by CR2^{base} only calls the target function *once*. So if a branch condition of a target function uses a local static variable, then CR2^{base} cannot guarantee to cover all branches of this branch condition because the local static variable is not a symbolic variable.

CROWN^{imp} solves this limitation by calling the target function multiple times (twice by default). Figure 6.5 shows an example of function *init_syntax_once* of file "grep.c" of subject "grep" to demonstrate that the problem NSLS is solved by CROWN^{imp}.

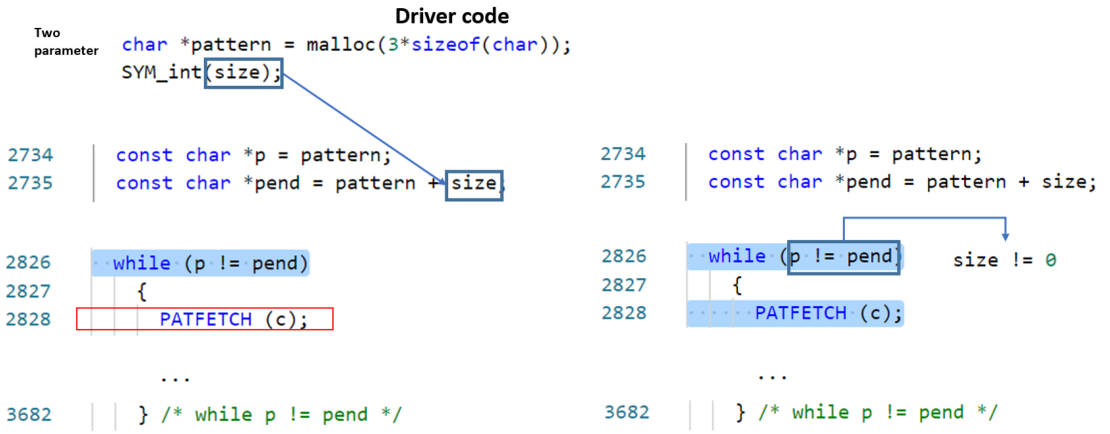
Why CR2^{base} Cannot Cover

The driver generated by CR2^{base} just calls the target function once, in such case, line 1713 cannot be covered because the value of **local static variable** *done* is always zero (L1710: initialized with zero) at line 1712.

Why CROWN^{imp} Can Cover

The driver generated by CROWN^{imp} calls the target function twice (by default)[3]. So for the first execution of target function, *done* becomes one (L1728), and for the second execution, the not-covered

[3]The users can decide how many times a target function should be called



(a) Coverage of CR2$^{\text{base}}$ (b) Coverage of CROWN$^{\text{imp}}$

Figure 6.6: Function *regex_compile* of file "grep.c" of subject "grep"

line (L1713) can be covered.

6.1.6 Improvement by Solving NSSP

NSSP stands for "No Support for Symbolic Pointers". CR2$^{\text{base}}$ cannot write symbolic path formula for a branch condition that contains symbolic pointers only, not to mention negating the branch condition to generate the next test case. Hence, the "then" or "else" branch cannot be covered by CR2$^{\text{base}}$.

CROWN$^{\text{imp}}$ solves this weakness by using KLEE as a supporting tool. KLEE tracks the value of each symbolic pointer in a branch condition and tries to replace the original branch condition with a new one that contains symbolic primitive values while not changing the syntax. By doing so, the new branch condition can be negated to generate the next test case. A concrete example is shown in Figure 6.6

Why CR2$^{\text{base}}$ Cannot Cover

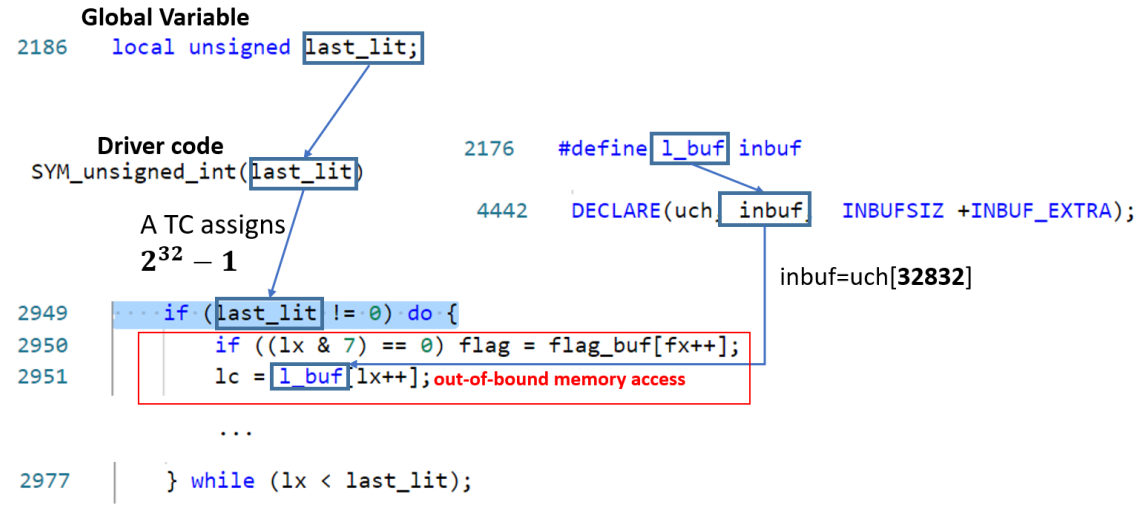
In lines 2734-2735, symbolic parameter variable *size* is initialized with 0, so *p* and *pend* points to the same address. *p* and *pend* are not altered by lines 2736-2826. The CR2$^{\text{base}}$ cannot generate symbolic path formula for the condition $p! = pend$ as both of them are symbolic pointer, not to mention negate that condition to generate new test cases. Thus, line 2828 cannot be covered.

Why CROWN$^{\text{imp}}$ Can Cover

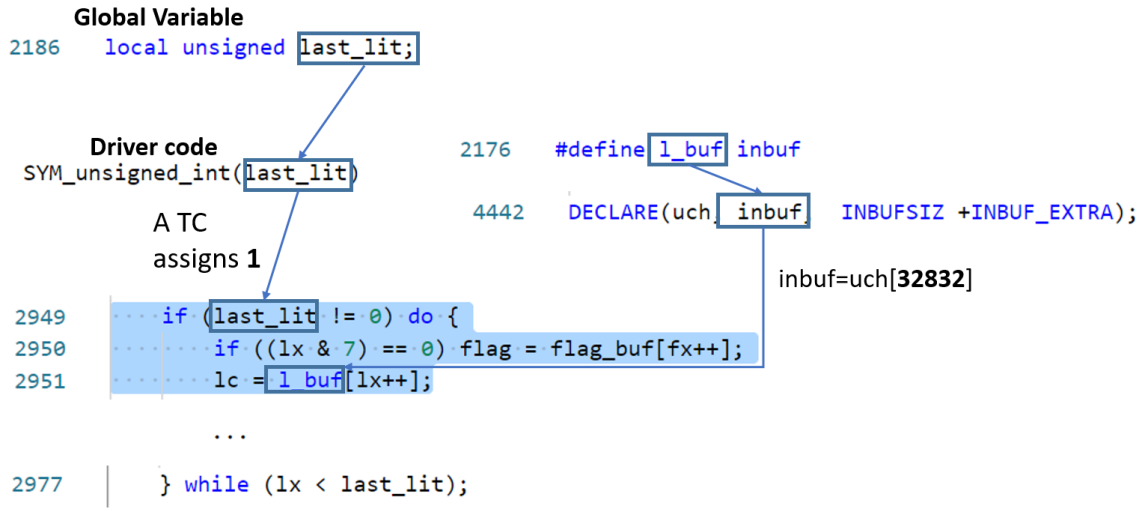
CROWN$^{\text{imp}}$ uses KLEE to support symbolic pointer. KLEE detects that *p* and *pend* are never altered by lines 2736-2826, then it deduces that $size == pend - p$ (L2826). After that, KLEE would replace the statement $p! = pend$ with a new one with the same syntax , i.e., $size! = 0$. Finally, KLEE generates a new test case by negating the new condition $size! = 0$ to cover line 2828.

6.1.7 Improvement by Partially Solving USV

USV stands for "Uncovered Branch Caused by Unrealistic Symbolic Values". The CR2$^{\text{base}}$ possibly outputs an unrealistic value for a symbolic variable and thus creates an execution crash. Therefore, the



(a) Coverage of CR2$^{\text{base}}$



(b) Coverage of CROWN$^{\text{imp}}$

Figure 6.7: Function *compress_block* of file "allfile.c" of subject "gzip"

coverage of such test cases are lost when using CR2$^{\text{base}}$. Figure 6.7 shows an example from function *compress_block* of file "allfile.c" of subject "gzip" and explains this limitation and its solution.

Why CR2$^{\text{base}}$ Cannot Cover

CR2$^{\text{base}}$ generates a very large value for a symbolic variable. The variable is used to access an array, and its assigned value is larger than the actual size of the array, which causes an out-of-bound memory access error. The following content explains how the crash happens in detail.

Line 2949: *last_lit* is a symbolic variable, to cover the "then" branch of the "if" statement of this line, CR2$^{\text{base}}$ outputs a very large value ,i.e., $2^{32}-1$, and assigns it to *last_lit*.

Line 2951: *l_buf* is an array with 32832 elements, and variable *last_lit* indicates how many elements of array *l_buf* will be accessed. Thus, an out-of-bound memory access happens at this line (because $32832 << 2^{32}-1$).

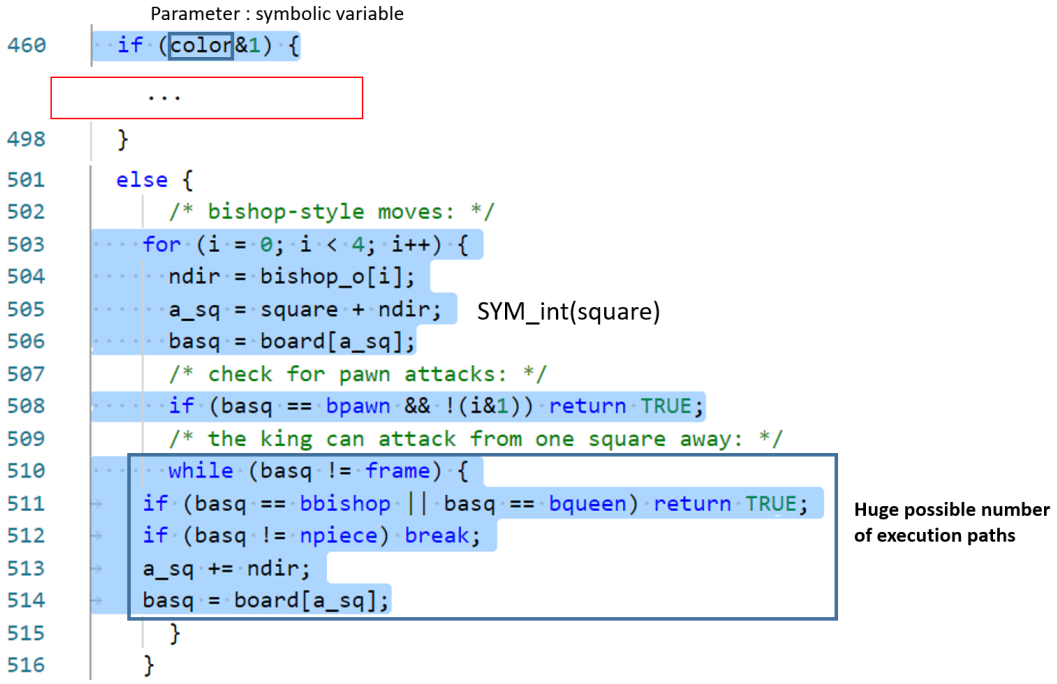


Figure 6.8: Function *nk_attacked* of file "attacks.c" of subject "sjeng"

Why CROWN^{imp} Can Cover

The KLEE used by CROWN^{imp} outputs 1 and assigns this value to *last_lit*, in such case, the crash at line 2951 can be avoided. The reason why CROWN engine and KLEE output different values even though given the same input constraints is that they use different SMT solvers (Z3[25] and STP[26] respectively). Thus, one of these two solvers may output a value that can avoid some crashes. So the limitation of USV can be partially solved.

6.1.8 Improvement by Solving WDFS

WDFS stands for "Weakness of DFS". CR2^{base} uses DFS as its path exploration strategy, which may spend all the *timeout2* (test case generation time for each function) on exploring the loop statement. As a result, the other statements are not covered. Figure 6.4 shows an example of function *nk_attacked* of file "attacks.c" of subject "sjeng" to demonstrate that the problem WDFS can be solved by CROWN^{imp}.

Why CR2^{base} Cannot Cover

Line 460: variable *color* is a symbolic integer (initialized with zero), the "else" branch (L501) is explored first. Inside the "else" branch, these is a "while" loop statement (lines 510-515), which has huge possible execution paths. The default dfs strategy (used by CR2^{base}) has to explore all those execution paths before covering lines 461-498 and timeout2 is reached. Thus, lines 461-498 remains not-covered.

Table 6.3: Crashes reported by address-sanitizer

Target Subject	# of Branch	# of Function	Branch Coverage	# of Crash TC
flex2.4.3	2021	144	53.20%	4341
grep2.0	3416	119	52.70%	9172
gzip1.0.7	1446	80	50.10%	10359
sed1.1.7	2395	64	51.10%	10315
libquantum0.2.4	724	111	50.10%	3392
sjeng11.2	6364	164	50.40%	19477

Why CROWN^{imp} Can Cover

CROWN^{imp} used combined strategy which utilizes different sub-strategies to cover lines 461-498, e.g., cfg and random can cover these lines.

6.2 Crash Deduplication and Analysis

To know the bug detection ability of CROWN^{imp},I collected all the test cases that are generated by CROWN^{imp} for four SIR subjects, recompiled the driver code for each function with the address-sanitizer[4], and then collected all the crashes that are reported by the address-sanitizer.

Table 6.3 shows crash information of each SIR subject. The first to third columns show the name, the number of functions, and the number of branches for each subject. The fourth column shows the branch coverage achieved by CROWN^{imp}. The fifth column shows the number of crashes reported by CROWN^{imp}.

6.2.1 Crash Deduplication Approach

The address-sanitizer reported more than 30-thousand crashes. Manually analyzing all the crashes is time-consuming and ineffective, so how to reduce the human effort to analyze the crash is very important. It is worth noting that the search strategy of CROWN2.0 includes cfg and random, which may generate many duplicate test cases (the test cases that have the same execution path). With this feature, many crashes may be redundant. So It is worthwhile to reduce the # of crashes and then analyze the unique crashes only.

There exist many approaches to remove duplicate test cases. For example, the stack hashing technique accumulates the last N function calls leading to the crash cite, hashing these traces to distinguish unique crashes[29]. The Clustering-based approach uses a clustering algorithm to put the crashes that have similar crash traces together in the same group[30]. Both approaches are based on an observation: if two crashes have exactly the same crash lines, then the two crashes are the same. Otherwise, they are different.

Figure 6.9 shows the example of two different crashes. The left part shows the crash stacks and function names in the call stacks. The right part shows the line numbers of the target subject in the crash stack. The first test case crashes at line 2551, and the second test case crashes at line 2589. So two crashes are regarded as different crashes. As it is shown in figure 6.9, we can distinguish two different

[4]I use address-sanitizer because it can detect more memory errors (e.g., out of bound access) than the default compilation options of gcc

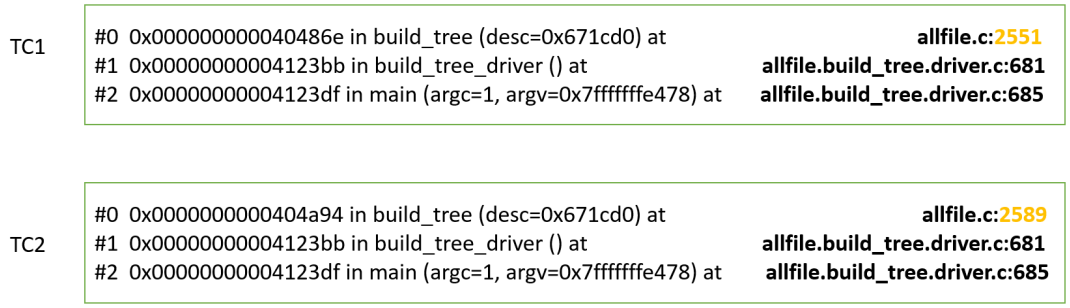


Figure 6.9: Example of two different crashes

Algorithm 2 Working Mechanism of Crash Deduplication

Input: : C. lists of crashes of a subject, L. function list of a subject

Input: : K. # first crash lines to compare

Output: : S. a set of unique crashes

1: **for each** $f \in L$ **do**
2: $U \leftarrow \{\}$
3: $V \leftarrow \{\}$
4: **for each** $c \in C[f]$ **do**
5: **if** first K lines of c not in V **then**
6: $U \leftarrow U + c$
7: $V \leftarrow V+$ (first K lines of c)
8: **end if**
9: **end for**
10: $S \leftarrow S + U$
11: **end for**
12: **return** S

crashes by comparing the first crash line only. From this observation, we may need to compare the first K lines (rather than all crash lines) to determine whether the two crashes are the same or not.

Algorithm 2 describes how to deduplicate crashes for each target subject. The algorithm accepts lists of crashes and a function list of a target subject, and outputs a set of unique crashes S.

- At line 1, a function and its crashes are chosen.
- At line 2 and line 3, two sets (U and V) are created. U contains the unique crashes for the function chosen at line 1. V contains the first K lines of crashes in U.
- From line 4 to line 9, for each crash, if its first K lines are not in V, it is regarded as a unique crash. The crash will be added into set U and the first K lines of this crash will be added into set V.
- At line 10: Add all the unique crashes of the chosen function into set S. Then select another function from the function list L until all the functions are chosen.
- At line 12: Return S that contains all the unique crashes.

Table 6.4: Crash Deduplication Result

Target Subject	# of Crash TC	# of Unique Crash TC (K=3)	# of Unique Crash TC (K=5)
flex2.4.3	4341	123	124
grep2.0	9172	241	248
gzip1.0.7	10359	56	56
sed1.1.7	10315	166	167
libquantum0.2.4	3392	289	289
sjeng11.2	19477	581	579

6.2.2 Crash Deduplication Result

After applying the crash deduplication approach using different values of K (number of crash lines being compared), I obtained the results in table 6.4. The first column shows the subject name. The second column shows the number of crashes reported by CROWN^{imp}. The third and fourth columns show the number of unique crashes obtained by using K as three and five respectively. We can note that different values of K do have not much difference on the crash deduplication result.

6.2.3 False Alarm Example

After the crash deduplication process, there are still more than 500 crashes. And most of them are false alarms, i.e., the crash cannot be reproduced by the system-level test cases. Figure 6.10 gives an example of one false alarm reported by the unit-level test case.

The target function f has a parameter x which is used to get the xth element of an array (line 5). Before the execution line 5, the assertion statement would be executed to check whether x is in a valid range or not (between zero and six). And crash would happens if x is not in the valid range.

In unit-level concolic testing of function f, the parameter x would be replaced by a symbolic variable. And the concolic testing engine would try to generate a value of x to violate the assertion at line 4, e.g., x = -1 . Hence, a crash would be reported by concolic testing tool.

It is worth noting that function f would be invoked by function *main* only, and function *main* has a "sanity check" before calling function f to ensure that the value of the parameter of f is in a valid range. Therefore, the error reported in the unit testing cannot be reproduced.

6.2.4 Crash Analysis

For the remaining crashes after the crash deduplication process, I manually analyzed their crash traces one by one to discover true positive. I focus on analyzing the crashes on the four SIR subjects because the two subjects from SPEC2006 benchmark generated too many crash test cases to analyze. I will analyze the crashes of two SPEC2006 subjects in the future.

The criteria to whether a crash is true positive or not are to see if any system-level test cases can reproduce the crash. If the system-level test cases can reproduce the crash, the crash is a true positive (i.e., an existing bug makes the crash happen)

Figure 6.11 and Figure 6.12 show the system-level crash trace and the unit-level crash trace. The first two lines show the crash address in the memory. The remaining lines show the crash stack, which contains the function name, file name, line number, etc.

```c
// Target Function
int f(int x){
    int arr[7] = {1,2,3,4,5,6,7};
    assert(0<=x && x <= 6);
    if(arr[x] == 1){
        return 0;
    }else{
        ... // do other things
    }
}
int main(){
    int x;
    scanf("%d", &x);
    if(x < 0 || x >6){
        return -1;
    }else{
        return f(x);
    }
}
/*
f_driver(){
    int x;
    SYM_int(x);
    f(x);
}
*/
```

Figure 6.10: Example of False Alarm

```
==7078==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000030
(pc 0x00000040ebe6 bp 0x7ffdea840c60 sp 0x7ffdea840bd0 T0)
    #0 0x40ebe5 in execute_program ./sed.c:1274
    #1 0x410167 in read_file ./sed.c:1205
    #2 0x412529 in main ./sed.c:477
    #3 0x7fc20a2ab82f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
    #4 0x401878 in _start (./sed.exe+0x401878)
```

Figure 6.11: System-level crash trace

```
==21012==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000030
(pc 0x000000411b2f bp 0x7ffea7185ed0 sp 0x7ffea7185e40 T0)
    #0 0x411b2e in execute_program ./sed.c:1274
    #1 0x413047 in read_file ./sed.c:1205
    #2 0x41e5dc in read_file_driver ./sed.read_file.driver.c:773
    #3 0x41e5f9 in main ./sed.read_file.driver.c:777
    #4 0x7f4ddd9eb82f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
    #5 0x401ac8 in _start (./sed.read_file.driver_debug+0x401ac8)
```

Figure 6.12: Unit-level crash trace

Both the system-level and unit-level test cases have the same crash trace (lines in the call stack: 1205->1274), so this crash (detected by CROWN^{imp}) is a true positive.

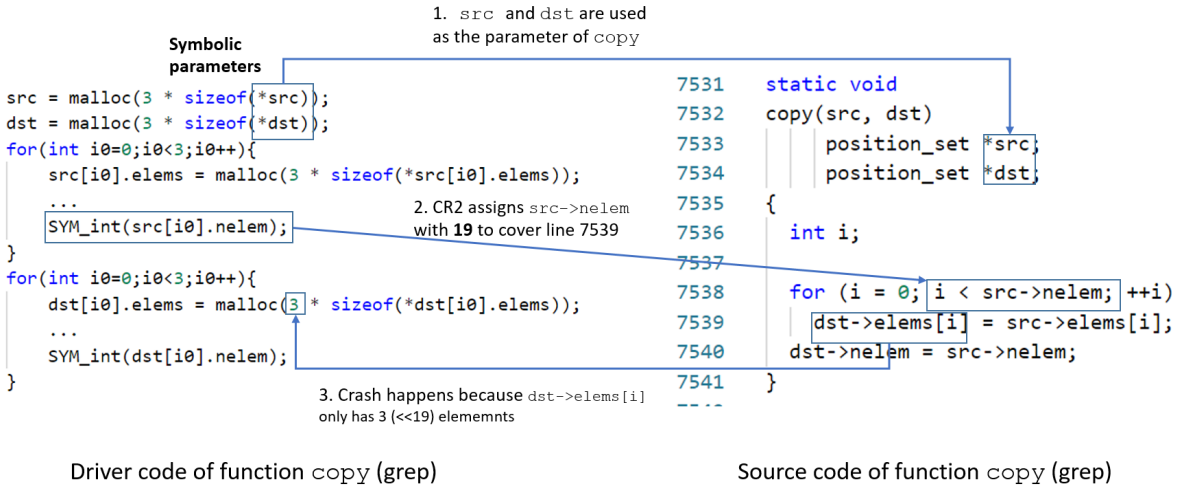


Figure 6.13: Example of USIV

6.2.5 Three Causes of False Alarms

I have found three causes of false alarms, and the following content explains these three causes with concrete examples.

USIV. Unrealistic Symbolic Input Value

The value of symbolic input may not be in the valid range, thus causing the crash. For example, a symbolic *size* represents the length (saying 10) of an array, and CROWN may output a value for *size* which is larger than the actual length of the array. Then an out-of-bound memory access crash will happen. The concrete example in Figure 6.13 shows this kind of false alarm found at function *copy* of file "grep.c" of subject "grep".

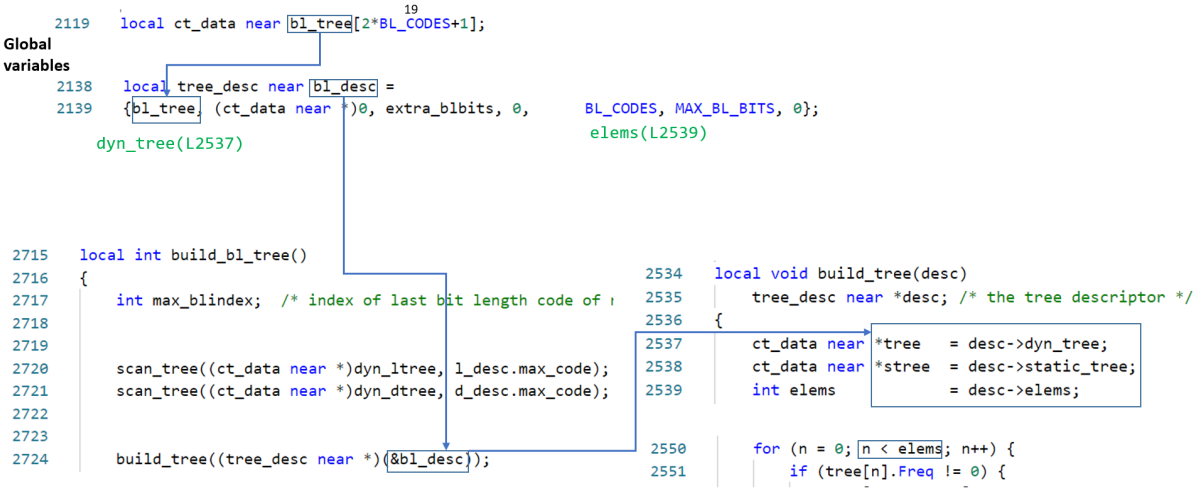
Line 7538: $src- > nelem$ is a symbolic variable, to satisfy the condition $i < (src- > nelem)$, CROWN assigns 19 to this variable. Then crash happens at line 7539 because the two arrays at this line only contains three ($<<$ 19) elements.

USVI.Unrealistic Symbolic Variable Initialization

The driver code will reinitialize all the global variables, whether initialized by the target subject or not. This feature may cause false alarms. Figure 6.14 explains a false alarm example in function *build_tree* of file "allfile.c" of subject gzip.

Sub-figure (a) of Figure 6.14 shows why crash will not happen at line 2551 during the system-level execution.

- Line 2139: The *dyn_tree* field of structure *bl_desc* is initialized with an array of 39 elements (*bl_tree*). The *elems* field of structure *bl_desc* is initialized with 19 (=macro BL_CODES).
- Lines 2537-2539: *dyn_tree* field is assigned to variable *tree* (L2537), *elems* field (19) is assigned to variable *elems* (L2539). Now *tree* points to an array which contains 39 elements, and the value of *elems* is 19.



(a) Execution Path

```
bl_desc.dyn_tree = malloc(3 * sizeof(*bl_desc.dyn_tree));
...
SYM_int(bl_desc.elems);
```

(b) Driver Code

Figure 6.14: Example of USVI

- Line 2551: Crash would never happen at this line because n is less than *elem*, which is less than the # of elements (=39) of the array that *tree* points to.

Sub-figure (b) of Figure 6.14 shows how the crash is caused by unrealistic variable initialization in the driver code.

The driver code reinitialized the *dyn_tree* field of structure *bl_desc* by making it points to an array with only three elements (the actual length should be 39). To cover line 2551, CROWN assigns four to variable *elem*, and the out-of-bound memory access happens at line 2551 (because $3 < 4$)

ICIO.Invalid Comparison Caused by Integer Overflow

An integer overflow may cause some branches to be covered under unexpected conditions, causing crashes when executing that branch. Figure 6.15 shows this kind of false alarm found at function *inflate_codes* of file "allfile.c" of subject "gzip".

- Line 1518: both w, d and e are unsigned variable, so (unsigned)(0-9136) =4294958160 ¿ 23632 and the "then" branch is executed.
- Line 1520: memory overlap crash happens when executing *memcpy*.

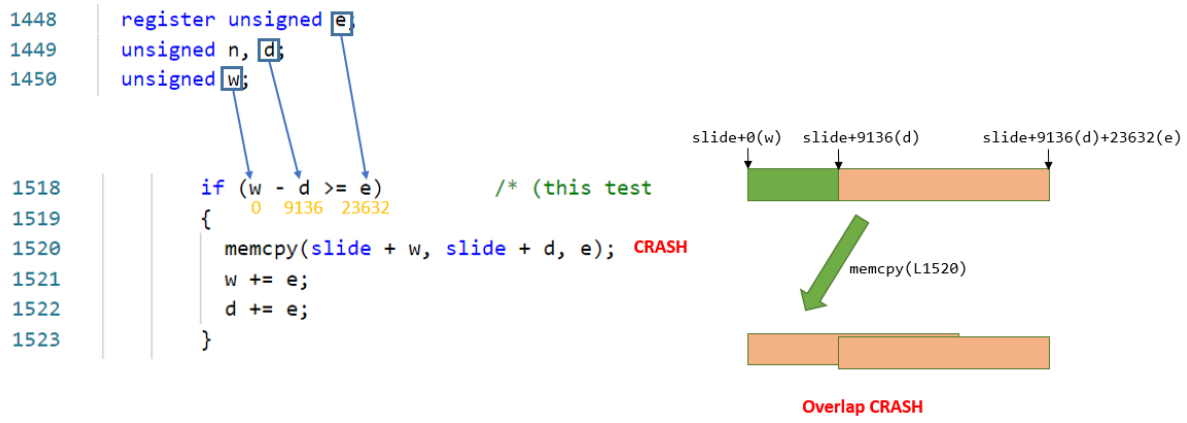


Figure 6.15: Example of ICIO

Chapter 7. Related Works

7.1 Automated Test Case Generation

The automatic test case generation technique refers to the automatic construction of test cases that meet certain conditions (e.g., assertions, constraints that are given by the users) based on the software product and the related documents. The automatic test case generation technique can greatly reduce the cost of software testing and shorten the software development cycle. At present, the automated test case generation technique is one of the most important research areas of software testing.

7.1.1 Automated Test Case Generation in Research Community

Since people began to study software testing in the 1960s, automatic test case generation has become an important research topic. Random testing is one of the earliest test case generation techniques. It randomly generates test cases and then monitors the program execution process. It is a software testing technology with a high degree of automation. In 1962, Remfer published a paper introducing automated testing and believed that random testing was an important automated testing technique[31]. Although random testing is simple, easy to implement, and highly automated, it has disadvantages such as low code coverage and excessive redundant test cases. Later, people proposed improved random testing techniques such as adaptive random testing, random testing with guidance information, and fuzzy testing.

Recently, there have been many papers on automated software testing. Among them, Stucki et al. developed a software testing system named PET, which can record information about the statement execution such as the number of executions, maximum value and minimum value of a variable, etc[32]. Ramamoorthy et al. developed a system called ACES, which can detect unreliable structures of programs[33]. Boyer et al. developed a system that generates test cases and verifies the program assertions according to the paths of the target program[34]. This is the first software testing system that has the ability to automatically generate test cases. It uses symbolic execution as its test case generation technique. The EFFIGY that is developed by IBM also adopts the idea of symbolic execution, which replaces the specific values of variables with symbolic variables to simulate the execution of the target program[35]. These tools are the earliest software testing or software verification tools. Clarke systematically introduced the approach of using symbolic execution to generate test cases based on the execution path[36]. This method generates an output corresponding to symbolic input according to the path constraints and the constraints provided by the user.

7.1.2 Automated Test Case Generation in Industry

At present, the industry has issued multiple test case generation tools. The functional test generation tool SoftTest developed by Bender & Associates company can perform unit testing, integration testing, system testing, and acceptance testing. The Panorama C/C++ tool developed by Internation Software Automation company has many functionalities such as test case generation, test case replaying. The CROWN2.0[23] developed by Vpluslab company can automatically generate driver/ stubs for each function and perform automated test case generation using concolic testing. In addition, there are some open-source test generation tools such as CUTE[19], CREST[24], KLEE[1] and JPathFinder[53] etc.

These tools are based on symbolic execution and constraint solving technology. However, the symbolic execution technique still faces some problems, such as path explosion problem and the solving nonlinear path constraint condition problem, so the symbolic execution technology needs to be improved.

7.2 Concolic Testing/Dynamic Symbolic Execution

Recently, concolic testing technique has attracted lots of researchers again. The current research on concolic testing mainly focuses on three directions:

- How to improve the concolic testing technique
- How to expand the application range of concolic testing
- Case studies applications based on concolic testing

To solve the path explosion problem of concolic testing technique, researchers mainly improve concolic testing from three perspectives:

(1)Revising the traditional concolic testing technique by utilizing different kinds of technology e.g., using distributed, parallel technology or cloud computing technology to improve the efficiency of concolic testing.

Yu et al. proposed a method to reduce the time cost of automated test generation by using parallel technology and improve the scalability of concolic testing[37]. The SCORE system is a dynamic symbolic execution tool that uses distributed technology and can be deployed on a large number of nodes. The SCORE system also introduces the idea of cloud computing[38].

(2) Adopting an active path exploration strategy, focusing on how to quickly and directly reach the target branch. Sun et al. used program slicing technology to find functions that affect path conditions, avoid searching for irrelevant path spaces, and reduce analysis costs[39]. Prabhu et al. used a two-stage learning and conflict avoidance strategy to improve the branch coverage of concolic testing[40]. In the first stage, a conflict-driven learning mechanism was used to skip paths that may encounter conflicts. In the second stage, a more intelligent conflict learning mechanism was used to avoid searching for unreachable branches. Do et al. believe that many heuristic strategies have a large computational cost, which makes the efficiency of concolic testing technology worse, so they proposed a method of alternately using heuristic search and random search to reduce testing cost[41]. In addition, many researchers have proposed a variety of methods to utilize the static structure of the target program to guide the searching process, which can also improve the efficiency of concolic testing[42].

(3) Negative search strategy, it emphasizes how to avoid searching infeasible searching spaces and how to quickly utilize backtrack. Krishnamoorthy et al. proposed a fallback strategy based on the reachability graph and conflict analysis[43]. The reachability graph was used to determine the path reachability based on the depth-first search, and only the paths related to the critical path were explored.

Extending the application range of concolic testing is a new research topic. Canini et al. applied concolic testing technology to distributed systems to automatically detect whether the system is abnormal and improve the reliability of the distributed system[44]. Taneja et al. applied concolic testing technology to regression testing, using the PIE model to track which part of the target program is modified part is tracked, and then use the concolic testing technology to generate test cases[45]. Papadakis et al. combined concolic testing with mutation testing, inserted the constraints that can trigger mutations into

the source program, and then used concolic testing technology to automatically generate test cases[46]. Kim et al. combined concolic testing and fuzzing technique and utilized these two techniques to generate test cases in an adaptive way[52]. Artzi et al. applied concolic testing technology to fault localization, using symbolic execution and concrete execution information to locate the fault location[47].

At present, there are not many research reports on the case study of concolic testing technology, and the main case studies focus on software testing. Kim et al. used concolic testing technology to test the large storage system that was developed by Samsung[48]. This approach overcomes the weakness of traditional testing technology that cannot generate effective test cases, i.e., test cases with high coverage achievement, and has a smaller time cost. In addition, they had also reported multiple detected vulnerabilities when using concolic testing technology to test the document management system developed by Samsung, which proved the high efficiency of concolic testing[49]. Jang et al. reported a case study of using concolic testing technology to detect malicious software and found that in combination with reverse tracking technology, many hidden paths could be discovered[50].

7.3 Automated Unit-level Test Case Generation

System-level test case generation often faces the problems of low coverage and path explosion. In order to solve the above problems, unit testing approach was proposed. The target of unit testing is the unit/function rather than the entire project, so the search space for each unit is relatively small, and the coverage achieved by unit testing is higher than the one of system-level testing.

Automated unit test generation techniques analyze a given function under test and generate input values of the target function, i.e., value of parameters, global primitive variables, and variables with structure/union type, automatically.

In order to generate test input for each function individually, a testing environment is essential, so the testing tool needs to do the following two things:

- **Driver Generation**. Generate a test driver to build a test environment for each target function, e.g., how to set the input values for the parameters of the target function and global variables and how many times the target function should be called for each test case execution.
- **Stub Generation**. Generate test stubs to simulate the behavior of some library functions whose source code is invisible to the users.

Table shows a list of realated work on authmated unit-level test case generation techniques.

CUTE[19] generates unit test drivers (stubs are not generated) to generate test input for the target C programs. The unit test drivers set all the parameters used by the target function as symbolic variables. CONBOL[21] generates test drivers, stubs, and test inputs for large embedded C programs developed by Samsung. UC-KLEE[54] generates test drivers for the target function using lazy symbolic input initialization, i.e., whenever an uninitialized variable is read during execution, UC-KLEE sets that variable as symbolic variables and generates test inputs. Like CUTE, UC-KLEE cannot replace the library functions with stub functions. CONBRIO generates both the driver and stubs for each target function, moreover, the driver generated by CONBRIO contains extended unit, i.e., other functions that have a direct or indirect callee-caller relationship with the target fictions to help improve the test coverage and filter out the false alarms (crashes which are not infeasible in the system level test cases). Intelligen[55] selects only the most potentially vulnerable functions, i.e., the function that contains more pointer dereference statements and memory-related function calls, from the target subject, synthesizes

Table 7.1: Related Works of Unit Testing

Tool	Technique	Driver	Stub	Local Static	Comb. SS	Random Value if 1st crash	FP	Stub for File-handling	Utilize Other Engine
CUTE[19]	Concolic Testing	O	X	X	X	X	X	X	X
UC-KLEE[54]	Symbolic Execution	O	X	X	O	X	X	X	X
CONBOL[21]	Concolic Testing	O	O	X	X	X	X	O	X
CONBRIO[22]	Concolic Testing	O	O	X	X	X	X	O	X
MAESTRO[52]	Concolic Testing, Fuzzing	O	O	O	O	X	O	X	O
$CR2^{imp}$	Concolic Testing	O	O	O	O	O	O	O	O

driver for the selected functions similar to the way CROWN2.0 does and generates test inputs using the fuzzing technique. MAIST[51] generates both driver and stubs for each target function, in addition, it adds support for function pointer type by obtaining the target functions that are assigned to function pointer fp. MAESTRO[52] generates driver, stubs, and test inputs for each target function by utilizing both the fuzzing and concolic testing techniques in an adaptive way. EvoSuite[56] automatically generates unit-level test inputs for java language programs.

Table 7.1 shows a brief comparison of different unit testing techniques.

Although the unit-level test case generation technique can improve the coverage, it often generates a large number of invalid test cases, i.e., the test case whose execution path cannot be reproduced from the system level. Therefore, unit testing needs to study how to avoid generating these invalid test cases.

7.4 Obstacles of Automated Test Case Generation

The test cases generation plays an important role in the software testing process. The quality of the test suite generated by test case generation tool directly affects the effectiveness of software testing. In the past, many test case generation tools required the users to set the input variables, that is, the users were responsible for being knowledgeable of the code of the software under test, which reduced the efficiency of test generation. The current software structure is becoming more and more complex, making it time-consuming to understand the software and set the input variables manually, further increasing the difficulty of test case generation. Therefore, the automatic test case generation tool is getting more and more attention. It does not require the tester to perform complicated operations, nor does it require the tester to have an in-depth understanding of the code, which greatly improves the degree of automation of the testing process. But the automatic test case generation tool is not perfect yet. There are mainly three,i.e., random testing, search-based testing, constraint-solving based testing, types of automated test case generation methods, each of which has its own weaknesses.

The random test case generation tool has a high degree of automation, but its test case generation process is blind, resulting in a large number of invalid or redundant test cases, making it achieve low code coverage and low test efficiency. The researchers have proposed approaches such as adaptive random testing to improve random testing. But such techniques have a huge time cost and memory cost to provide feedback to further test case generation process, making it hard to be put into industrial applications.

Search-based test case generation can overcome the shortcomings of low random test code coverage. It is a highly potential test generation technology, but it also faces some problems. Search-based test generation tools cannot obtain and process the execution environment of the software under test, for

example, they cannot process the interactive information between the software under test and the underlying operating system, network access, database access, etc. Moreover, the fitness function directly affects the test case generation process, and sometimes the fitness function cannot provide sufficient guidance for the search process if the designed fitness function is not effective enough.

Constraint-solving-based test case generation also has some problems. First of all, techniques such as symbolic execution and static analysis cannot track the library functions and the dynamic data structures in the program, e.g., pointers, classes in object-oriented programming languages, which severely limits the capabilities of the test case generation technique based on constraint solving. Secondly, due to the problem of path explosion, if an unreasonable strategy is adopted, a large number of redundant test cases may be generated, which reduces the efficiency of testing. Finally, due to the limited ability of the SMT solver, it cannot handle nonlinear constraints, e.g., $10 = x^2 + e^x$. These above-mentioned characteristics limit the development of constraint-solving-based test case generation.

Chapter 8. Conclusion and Future Works

8.1 Conclusion

In this dissertation, I have studied a commercial automated testing software tool, CROWN2.0 ver 2020, and used this tool to perform experiments on six real-world projects. In the experiment, I found that the branch coverage achieved by CROWN2.0 ver 2020 in the six real-world projects (four from SIR and two from SPEC2006) was relatively low at the beginning, i.e., 28.9% on average for each subject. In order to find the reason that causes low coverage achievement, I read each branch or even each line of the source code of the six target projects. Then I got 324 groups of not-covered branches. After extensively analyzing all these 324 groups of not-covered branches, I found that the CROWN2.0 ver 2020 has 13 common problems.

To solve the above 13 problems, I proposed six ideas for these 13 problems, applied the six ideas to CROWN2.0 ver 2020, and re-performed experiments on the six target projects. The experimental results show that the ideas give a 77% branch coverage improvement (up to 188%) on average for six subjects. In addition, I wrote documents for each group of unexplored branches. The document contains the code example and reason why each group cannot be covered by CROWN.

8.2 Future Work

In future work, I plan to improve the effectiveness of CROWN2.0 (branch coverage, bug detection ability) through the following aspects.

8.2.1 More Analysis on The Detected Crashes

I analyzed the 595 crashes from four subjects of SIR benchmark. In the future, I will analyze the remaining two subjects of SPEC2006 benchmark to check if the detected 868 crashes of two SPEC2006 subjects are true positive or not.

8.2.2 Performing Experiment on More Subjects

In this dissertation, I performed experiments on six target subjects and proposed five approaches to improve the coverage achieved by CROWN2.0 on these six subjects. These approaches may be over-fitting on these six subjects, i.e., the solutions may not effectively improve the coverage on other projects. In the future, I plan to apply CROWN 2.0 to more large-scale projects and analyze the not-covered branches in large projects to find more ways to improve the effectiveness of CROWN2.0.

8.2.3 Integrating Other Testing Tools

At present, CROWN2.0 uses two test case generation tools, i.e., CROWN and KLEE. These two testing tools are developed based on symbolic execution, so they inevitably face path explosion problems. So in the future, I want to integrate fuzzing tools into CROWN2.0. The fuzzing technique (e.g., AFL) can help to detect some corner-case bugs in a short period of time.I want to make CROWN2.0 observe

the coverage achieved by the concolic testing dynamically. When the coverage has not improved for a long time, CROWN2.0 can switch to fuzzing to improve its coverage and bug detection ability.

8.2.4 Obtaining Seed TC from System Level TC to Guide Concolic Testing

A good test case can make concolic testing achieve a higher branch coverage in a short time. In the future, I plan to use system-level test cases to automatically generate seed test cases to guide the process of concolic testing. The basic idea is as follows.

First, observe the execution path of the system-level test cases, obtain a system-level test case with the longest execution path , generate a unit-level test case according to the execution path of the system-level test case , and then use the unit-level test case as a seed.

Bibliography

[1] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. *Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.* OSDI. Vol. 8. 2008.

[2] Chen, Tsong Yueh, Hing Leung, and I. K. Mak. *Adaptive random testing." Annual Asian Computing Science Conference.* Springer, Berlin, Heidelberg, 2004.

[3] Chen, Tsong Yueh, Fei-Ching Kuo, and Huai Liu. *Adaptive random testing based on distribution metrics.* Journal of Systems and Software 82.9 (2009): 1419-1433.

[4] Chen, Tsong Yueh, R. Merkel, P. K. Wong, and G. Eddy. *Adaptive random testing through dynamic partitioning.* Fourth International Conference onQuality Software, 2004. QSIC 2004. Proceedings.. IEEE, 2004.

[5] Ciupa, Ilinca, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. *ARTOO: adaptive random testing for object-oriented software.* Proceedings of the 30th international conference on Software engineering. 2008.

[6] *American fuzzy lop.* URL: http://lcamtuf.coredump.cx/afl/. Accessed: 2021-12-25

[7] Böhme, Marcel, Van-Thuan Pham, and Abhik Roychoudhury. *Coverage-based greybox fuzzing as markov chain.* IEEE Transactions on Software Engineering 45.5 (2017): 489-506.

[8] Fioraldi, Andrea, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. *AFL++: Combining incremental steps of fuzzing research.* 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020.

[9] Rawat, Sanjay, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. *VUzzer: Application-aware Evolutionary Fuzzing.* NDSS. Vol. 17. 2017.

[10] Chen, Peng, and Hao Chen. *Angora: Efficient fuzzing by principled search.* 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018.

[11] *Libfuzzer – a library for coverage-guided fuzz testing* URL: https://llvm.org/docs/LibFuzzer.html. Accessed: 2021-12-25

[12] Ramirez, Aurora, José Raúl Romero, and Sebastian Ventura. *A survey of many-objective optimisation in search-based software engineering.* Journal of Systems and Software 149 (2019): 382-395.

[13] Afzal, Wasif, Richard Torkar, and Robert Feldt. *A systematic review of search-based testing for non-functional system properties.* Information and Software Technology 51.6 (2009): 957-976.

[14] Ali, Shaukat, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. *A systematic review of the application and empirical investigation of search-based test case generation.* IEEE Transactions on Software Engineering 36.6 (2009): 742-762.

[15] McMinn, Phil, and Gregory M. Kapfhammer. *AVMf: An open-source framework and implementation of the alternating variable method.* International Symposium on Search Based Software Engineering. Springer, Cham, 2016.

[16] Harman, Mark, and Phil McMinn. *A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation.* Proceedings of the 2007 international symposium on Software testing and analysis. 2007.

[17] Harman, Mark, and Phil McMinn. *A theoretical and empirical study of search-based testing: Local, global, and hybrid search.* IEEE Transactions on Software Engineering 36.2 (2009): 226-247.

[18] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. *DART: Directed automated random testing.* Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. 2005.

[19] Sen, Koushik, Darko Marinov, and Gul Agha. *CUTE: A concolic unit testing engine for C.* ACM SIGSOFT Software Engineering Notes 30.5 (2005): 263-272.

[20] Kim, Yunho, Moonzoo Kim, Young Joo Kim, and Yoonkyu 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, 2012.

[21] Kim, Yunho, Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang, and Moonzoo Kim. *Automated unit testing of large industrial embedded software using concolic testing.* 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013.

[22] Kim, Yunho, Yunja Choi, and Moonzoo Kim. *Precise concolic unit testing of C programs using extended units and symbolic alarm filtering.* Proceedings of the 40th International Conference on Software Engineering. 2018.

[23] *CROWN2.0* https://www.vpluslab.kr/crown2

[24] Burnim, Jacob, and Koushik Sen, *Heuristics for Dynamic Test Generation*, Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2008.

[25] De Moura, Leonardo, and Nikolaj Bjørner. *Z3: An efficient SMT solver.* International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008.

[26] *STP* Ganesh, Vijay, and David L. Dill. *A decision procedure for bitvectors and arrays.* In Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007).

[27] Do, Hyunsook, Sebastian Elbaum, and Gregg Rothermel. *Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.* Empirical Software Engineering 10.4 (2005): 405-435.

[28] *SPEC CPU2006 Documentation*, https://www.spec.org/cpu2006/Docs/, accessed:2021-09-03

[29] Robert Swiecki. 2016. honggfuzz. http://honggfuzz.com

[30] Jiang, Zhiyuan, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. *Igor: Crash Deduplication Through Root-Cause Clustering.* (2021).

[31] Remfer G. *Automated Program Testing* Proceedings of The 3rd Computing and Data Processing, 1962

[32] Stucki, Leon G. *Automatic generation of self-metric software.* Proceedings 1973 IEEE Symposium on Conputer Software Reliability. 1973.

[33] Ramamoorthy, C. V., R. E. Meeker, and J. Turner. *Design and construction of an automated software evaluation system.* Proc. 1973 IEEE Symp. Computer Software Reliability. 1973.

[34] Boyer, Robert S., Bernard Elspas, and Karl N. Levitt. *SELECT—a formal system for testing and debugging programs by symbolic execution.* ACM SigPlan Notices 10.6 (1975): 234-245.

[35] King, James C. *A new approach to program testing.* ACM Sigplan Notices 10.6 (1975): 228-233.

[36] Clarke, Lori A. *A system to generate test data and symbolically execute programs.* IEEE Transactions on software engineering 3 (1976): 215-222.

[37] Yu, Xiao, Shuai Sun, Geguang Pu, Siyuan Jiang, and Zheng Wang. *A parallel approach to concolic testing with low-cost synchronization.* Electronic Notes in Theoretical Computer Science 274 (2011): 83-96.

[38] Kim, Yunho, and Moonzoo Kim. *SCORE: a scalable concolic testing tool for reliable embedded software.* Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011.

[39] Sun, Tao, Zheng Wang, Geguang Pu, Xiao Yu, Zongyan Qiu, and Bin Gu. *Towards scalable compositional test generation.* 2009 Ninth International Conference on Quality Software. IEEE, 2009.

[40] Prabhu, Sarvesh, Michael S. Hsiao, Saparya Krishnamoorthy, Loganathan Lingappan, Vijay Gangaram, and Jim Grundy. *An efficient 2-phase strategy to achieve high branch coverage.* 2011 Asian Test Symposium. IEEE, 2011.

[41] Do, TheAnh, Alvis CM Fong, and Russel Pears. *Scalable automated test generation using coverage guidance and random search.* 2012 7th International Workshop on Automation of Software Test (AST). IEEE, 2012.

[42] Dong, Yu, Mengxiang Lin, Kai Yu, Yi Zhou, and Yinli Chen. *Achieving high branch coverage with fewer paths.* 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops. IEEE, 2011.

[43] Krishnamoorthy, Saparya, Michael S. Hsiao, and Loganathan Lingappan. *Strategies for scalable symbolic execution-driven test generation for programs.* Science China Information Sciences 54.9 (2011): 1797.

[44] Canini, Marco, Vojin Jovanovic, Daniele Venzano, Boris Spasojevic, Olivier Crameri, and Dejan Kostic. *Toward online testing of federated and heterogeneous distributed systems.* Proceedings of The 2011 USENIX Annual Technical Conference. No. CONF. 2011.

[45] Taneja, Kunal, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. *Guided path exploration for regression test generation.* 2009 31st International Conference on Software Engineering-Companion Volume. IEEE, 2009.

[46] Papadakis, Mike, and Nicos Malevris. *Automatic mutation test case generation via dynamic symbolic execution.* 2010 IEEE 21st International Symposium on Software Reliability Engineering. IEEE, 2010.

[47] Artzi, Shay, Julian Dolby, Frank Tip, and Marco Pistoia. *Fault localization for dynamic web applications.* IEEE Transactions on Software Engineering 38.2 (2011): 314-335.

[48] Kim, Yunho, Moonzoo Kim, and Nam Dang. *Scalable distributed concolic testing: a case study on a flash storage platform.* International Colloquium on Theoretical Aspects of Computing. Springer, Berlin, Heidelberg, 2010.

[49] Kim, Yunho, Moonzoo Kim, Young Joo Kim, and Yoonkyu 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, 2012.

[50] Jang, Seongsoo, Ho-Yeon Kim, Young-Hyun Choi, and Tai-Myoung Chung. *A Study of Advanced Hybrid Execution Using Reverse Traversal.* 2011 International Conference on Information Management, Innovation Management and Industrial Engineering. Vol. 2. IEEE, 2011.

[51] Kim, Yunho, Dongju Lee, Junki Baek, and Moonzoo Kim. *Concolic testing for high test coverage and reduced human effort in automotive industry.* 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2019.

[52] Kim, Yunho, Dongju Lee, Junki Baek, and Moonzoo Kim. *MAESTRO: Automated test generation framework for high test coverage and reduced human effort in automotive industry.* Information and Software Technology 123 (2020): 106221.

[53] Păsăreanu, Corina S., Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. *Combining unit-level symbolic execution and system-level concrete execution for testing NASA software.* Proceedings of the 2008 international symposium on Software testing and analysis. 2008.

[54] Ramos, David A., and Dawson Engler. *Under-constrained symbolic execution: Correctness checking for real code.* 24th USENIX Security Symposium (USENIX Security 15). 2015.

[55] Zhang, Mingrui, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. *IntelliGen: Automatic Driver Synthesis for Fuzz Testing.* 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2021.

[56] Fraser, Gordon, and Andrea Arcuri. *Evosuite: automatic test suite generation for object-oriented software.* Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011.

Acknowledgment

Two and a half years of graduate life in KAIST is coming to an end. Here, I met my advisor and many labmates. They showed me great concern in life and study. I believe that two and half years at KAIST will be a meaningful experience in my life.

First and foremost, I would like to express my deepest appreciation to my advisor Prof. Moonzoo Kim. He allows me to study and do research at SWTV lab. Every time I have questions regarding my thesis, I can always receive the answer from him the first time. I am deeply impressed by his good personality, rigorous academic attitude, and patience. It's my honor to be his student. Besides my advisor, I would like to thank the committee members, Prof. Inyoung Ko and Prof. Jongmoon Baik, for reviewing my paper and giving me insightful comments.

I also wish to thank Prof. Yunho Kim for giving me kindful help regarding slide writing, research attitude, etc. He plays an important role in my graduate life. Without his help, I may not successfully make defense and graduate at KAIST.

I am also grateful to my labmates. They are Ahcheong Lee, Irfan Ariq, Robert Sebastian Herlim, Kunwoo Park, and Loc Duy Phan. Thank you so much for helping me when I am in trouble. Besides, I gratefully acknowledge the assistance of Bokyung Kim and Joonwang Ji. Thank you for giving me guidance on life in Korea and life in the laboratory

Special thanks to my roommate Fanchen Bu and my friends Jiansong Wan, Guoyuan An, and Lin Wang. Whenever I am not in a good mood due to research or life problems, they always listen to my complaints and give me many suggestions. In addition, I also thank my friend Tang Tang for eating with me every weekend and discussing some interesting things with me.

It would not be easy to find adequate words to convey the appreciation to all. Lots of love and thank all of you.

Curriculum Vitae

Name : Yang Zidong

Date of Birth : October 11, 1997

Educations

2019. 9. – 2022. 2. Korea Advanced Institute of Science and Technology
Master of Science in School of Computing

2015. 9. – 2019. 6. Wuhan University
Bachelor of Engineering in School of Computer Science

2012. 9. – 2015. 6. Hebei Wuyi Middle School