國立交通大學

資訊科學與工程研究所

碩 士 論 文

藉由正規限制式之擬真運算處理符號檔案輸入

Dealing with Symbolic File Input by Regular Constraint Symbolic Execution

研 究 生：黃佑鈞

指導教授：黃世昆 教授

中 華 民 國 九 十 九 年 九 月

藉由正規限制式之擬真運算處理符號檔案輸入

Dealing with Symbolic File Input by Regular Constraint Symbolic Execution

研　究　生：黃佑鈞　　　　Student：Yu-Chun Huang

指導教授：黃世昆　　　　Advisor：Shih-Kun Huang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Department of Computer and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

September 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年九月

藉由正規限制式之擬眞運算處理符號檔案輸入

學生：黃佑鈞　　　　　　　　　　　　　　　　指導教授：黃世昆　教授

國立交通大學資訊科學與工程學系研究所碩士班

摘要

軟體測試是軟體發展中一個重要的程序，而符號測試則是現今軟體測試中重要的技術之一。若符號測試範圍能涵蓋到檔案輸入的話，則我們將不需要修改程式碼來標記哪些變數有可能是符號變數，而可以透過標準輸入直接進行符號測試，符號測試將更爲方便且全面。近年來，拜電腦的運算能力大幅提升及關於減少描述程式之限制式的改善方法不斷地被提出，符號測試涵蓋於檔案等級已經不再是難以達成的事情。在眾多的符號測試工具之中，KLEE 著眼於 LLVM(Low Level Virtual Machine) 所具備的中介語言對各程式語言及各硬體架構的良好獨立性，所以架構於 LLVM 之上來進行符號測試，是目前發展較爲成熟且較受到矚目的符號測試工具之一。然而在 KLEE 中，由於符號檔案測試涵蓋了C語言標準函式庫，所以產生了過多的執行路徑，造成測試一個不到百行的小程式就得花上不少時間及記憶體，而所產的涵蓋新執行路徑的測試資料更是大部分只跟C語言標準函式庫的程式碼有關，只有一小部分是關於受測程式的新執行路徑。我們希望透過簡單的實作正規表示式限制符號檔案於 KLEE 之方式，專注於產生程式預期接收的測試輸入，以便在短時間內能更快的進入新的執行路徑。

關鍵字： 軟體測試、符號測試、正規表示式

Dealing with Symbolic File Input by

Regular Constrained Symbolic Execution

Student: Yu-Chun Huang Advisors: Prof. Shih-Kun Huang

Institute of Computer Science and Engineering

National Chiao Tung University

Abstract

Software testing is an important procedure of software development process, and symbolic testing is one of the most important techniques in the domain of software testing. If we can handle symbolic testing with file level, we can process testing directly via standard input rather than modifying source code and marking the symbolic variables. Nowadays, symbolic testing with symbolic files is feasible because of the powerful computing ability and the techniques about reducing constraints. In various symbolic testing tools, KLEE based on the top of LLVM(Low Level Virtual Machine) because that LLVM's assembly level instruction set is independent of programming languages and architectures. KLEE is one of well-developed and famous symbolic tools. However, KLEE does symbolic files testing still have some problems: symbolic files testing cover the C standard library and generate too many execution paths. The result is: (1) Waste a large amount of time and run out of memory when test a source code which is less than 100 lines. (2) The most ratio of test cases which can cover new paths relative testing source code to all test cases is too small. In other words, we spend too much time on generating new paths which cover C standard library. We want to improve the efficiency of generating execution paths by using regular expressions which describes a given format. In this way, we can explore useful execution paths which is covered the new part of source code in short period.

Keywords: Software Testing, Symbolic Testing, Regular Expression

誌謝

首先，我得感謝我的父母，在我人生中不斷的爲我加油打氣，尤其是寫論文這難熬的期間，更是給予我無微不至的關心，這份恩情難以回報。

也謝謝交大資工系計中的助教們，除了在工作上幫了我許多忙外，平常相處的歡樂氣氛也讓我解了我不少生活上的煩悶。

接著感謝清大橋藝社的大家，在大學及碩士生涯上都給與我不少的關心及鼓勵。

謝謝淵耀、宏坤、培郁、維農、信翰、旻璟、存澤等在我從大學到研究所求學路上的室友們，除了平常互相討論生活的甘苦外，在日常生活上也幫了我很多忙。

感謝軟體品質實驗室的各位，如世欣、士瑜、孟緯、博彥和立文、昌憲、琨瀚、友祥、文健、彥廷學長們，在研究上及學業上都給了我不少幫助。尤其是世欣，感謝你在我熬夜在 Lab 寫論文時陪我討論關於 KLEE 的許多問題。

再來也謝謝軟體安全實驗室這兩屆的專題生們，不論是承憲、(王民)誠、紘誌，或者是正宇及宏麒，你們承先啓後所架構出的 Wargame 系統，在我擔任程式安全課程助教時著實幫了我不少忙。

另外也要感謝兩位口試委員，孔崇旭老師及宋定懿老師，花了許多心血對這篇論文提出了極爲受用指導及建議，實在受益良多。

特別要謝謝黃世昆老師，不只是提供了實驗室的所有資源，還在研究上給予我許多指導及在生活上給了我許多幫助外，也熱心指導我許關於做事的態度，讓我感激在心。

眞的，要感謝的人實在是太多了，提都提不完，最後謝謝一路上所有幫助過我的人，謹以此論文，獻給你們。

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem Descriptions and Motivation

Software testing is a very important stage in procedures of software development [1]. If we develop software without testing, we can not prove that software can satisfy the specifications completely. If the implementation does not adhere to the specifications, there may be bugs (do not implement the requirements) and security problems (implement some function which does not be requested), as Figure 1-1 shows.

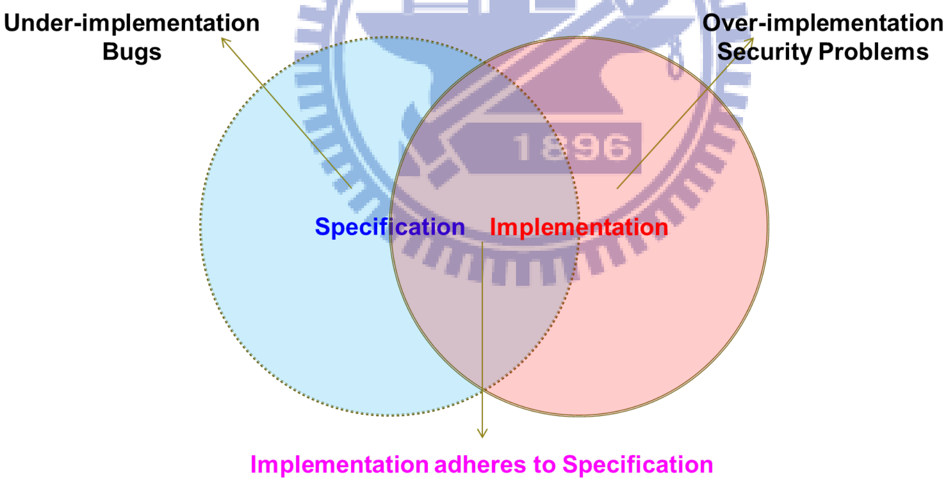

Figure 1-1: The relation about specification and implementation

Symbolic testing is one of mainstream techniques in the domain of software testing. Symbolic testing is more precise than other testing method, but it also spent more time and computing resource. In generally, before doing symbolic testing, we must modify source codes to provide information that which variable is symbolic. But if symbolic testing can support file level testing, we can do symbolic testing easily by making standard input symbolic.

We choose KLEE [2] as our main symbolic testing tool because KLEE has following advantages:

- It is a famous and well-developed symbolic testing tool
- It can do symbolic testing with basic file-level supported
- It built on top of the LLVM(Low Level Virtual Machine) [3] [4] [5] compiler framework. Via LLVM framework, we can validate and check many programs independent of programming languages and CPU architectures (if LLVM supported the programming language and CPU architecture). In the other words, we can effectively extend the testing range of programming language.

KLEE doing symbolic files testing with C standard library to provide detailed path and instruction information about standard library functions. The advantage is the bug which is dependent on library can be found easily. But the disadvantage is that the amount of execution paths is growing rapidly. For example, if we call fscanf function, fscanf function will call vfscanf function next. After calling the vfscanf function, vfscanf function call __scan_getc function. Because the input is unconstrainted, when we meet the branch instructions like if, switch, for and while, all conditions of branch instructions is always satisfiable. Because no execution path is discarded by unsatisfiablity of constraints, the rate of execution paths growing is too rapid to handle.

For example : the Figure 1-2 showed above is a source code on NCTU On-line Judge System [6]. Execute the program via KLEE [2] with symbolic files and files' size upper bound is 20 bytes, the result is: 330368 execution states, only 10225 test cases can exit program normally. Then only 13 test

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char* argv[]) {
  char buf[1024];
  unsigned long long i,j,k;
  while (fgets(buf,1024,stdin) != NULL) {
    if (strcmp(buf,".\n")==0) break;
    if (strcmp(buf,".")==0) break;
    sscanf(buf,"%llu %llu",&i,&j);
    if(i == 0)
        printf("%u\n",j);
    else
        printf("%llu\n",  (i << 32) + j);
  }
  return 0;
}
```

Figure 1-2: source code of test case 1

cases can produce new execution paths. And according our observation (The relative data will show in evaluations section), the 13 test cases is produced at early period. In other words, the efficiency of generating new execution paths is not good enough.

By previous example, we can find out a critical problem: How do we reduce the execution states efficiently? The main causes of a large number of execution states are:

1. To arrive the goal about general using, standard library to handle all possible situation. So the code of standard library is significant complexity.

2. No constraints describes and limits standard input, so all conditions about branches in standard library are almost possible.

In this thesis, we proposed a method called regular constraint to limit constraints with regular expression and resolve the above problem. By this method, we can scale the level of tested program.

1.2 Objective

In this thesis, we observe the problem that KLEE does symbolic files testing with uClibc [7] standard library and generates most execution states which is only covering paths in standard library code. To resolve the problem, we do symbolic files testing with focusing on the program's core functionality. In the other word, we want to test the "specification" part which we mentioned early. If we want to describe the input data format, use regular expression is a normal and reasonable method.

So we propose one method to resolve the above problem and implement the method on KLEE:

- Descript standard input format by regular expression, and combine regular expression and constraints, to arrive the effect we want.

Because we aim at "Test the part which programmers expect", the rules of on-line judge adhere to our requirement, we use some source code on NCTU On-line Judge to evaluate.

2 Background

2.1 Execution Paths and Constraints

Execution Paths is the set of executed instructions in order. If two executing paths's executed instructions or their order is not the same, we say the two execution paths are different. Constraints describe the reservations about an execution path. By way of converting instructions information to constraints, we can obtain the description about the limitation of data value , life time of variable , the relation about variables and the information about stack and heap. Constraint can be devided into two category:

- Constant constraints: The all information constraints provided is independent of user input.
- Symbolic constraints: The information is dependent on user input or execution environment.

For example, in Figure 2-1, n is an integer user entered. The value of x depends on n, so line 2 is symbolic constraints. And if we do not consider the value of x, this function has three execution paths because of if-else in line 4 to 9.

```
1  void test(unsigned int n){
2          unsigned int x = 3 * n + 1;
3          unsigned int y = 3 * x - 2;
4          if (x % 3)
5                  puts("Path1\n");
6          else if (y % 5)
7                  puts("Path2\n");
8          else
9                  puts("Path3\n");
10 }
```

Figure 2-1: An example for explaining execution path

2.2 Constraints Solver

The formal name of Constraints Solver is Satisfiability Modulo Theories Solver. In theory, if we can transform a problem to a satisfiability problems, we can use SMT(Satisfiability Modulo Theories) solver to resolve our problems. As known to all, satisfiability problems is NP-complete problems, if your constraints amount reach the significant degree, the SMT solver may not resolve the problem certainly. For enhancing the motive to improve the SMT solver, SMT-LIB arises a competition called SMT-COMP (Satisfiability Modulo Theories Competition) [8]. every year

CVC3 [9](advenced work from CVC Lite [10]), STP [11] and Yices [12] are some famous SMT solvers using by symbolic testing tools.

2.3 Symbolic Execution

Symbolic testing is one of the most important techniques in the domain of software testing. The concept of symbolic testing is proposed in 1970s [13] [14].

In symbolic executions, we use symbols to represent the program variables and execute the program with these symbols. This method is more powerful than concrete execution because one symbolic execution can be corresponding to a set of concrete executions.

In Figure 2-1, the function read an integer, does some arithmetic computing, and output depends on symbolic variables. In line 3, the symbolic variable x will be represented as $3 * n + 1$, symbolic variable y will be represented as $9 * n + 1$. During symbolic execution, we can obtain a execution tree which is the control flow of this program. In Figure 2-1 line 6, you will choice one of paths depending on symbolic variable. Usually, the execution will always enter line 7 because symbolic x is not a multiple of 3. But when we use correct constraints like $x = (3 * n + 1)\%2^{32}$ and $y = (3 * x - 2)\%2^{32}$, we can resolve a counter example that n is 2000000001. If n is 2000000001, the execution will enter line 7 or 9. Using symbolic execution, we can collect constraints and use constraints solver to get the test case to solve the execution paths. So find the same bug type of integer overflow easily.

2.4 Concolic Execution

The term "concolic" is combined "concrete" with "symbolic" . As the word "concolic" is originated from, concolic execution is a testing technique that use properly concrete values to replace the symbolic variable. At a specific time (usually when detecting a branch), we gather all constraints and give SMT solver the information. The solver will find a counterexample to try exploring other execution paths.

This method is better for these reasons:

- Avoid generating redundant test cases.
- Usually has better path coverage than other testing method.

EXE [15], DART [16], CUTE [17]and Crest [18] perform concolic testing. EXE can address bit-level acccurate modeling of memory , but byte-level memory model has more efficiency and the effect is only a little deterioration. Crest is advanced work for CUTE, it use Yices [12] as constraints solver and use CIL insert instrumentation code.

2.5 Fuzz Testing

Fuzz testing is also called "Random Testing" or "Brute Force Testing" because that the test data is generated randomly. Peach [19] and zzuf [20] is some famous fuzzing tools.The basis of fuzz testing's theory is described as following:

- Although the efficiency of exploring executing paths per test case is poorer than other testing tools, the efficiency of generating test cases is better.

$$Efficiency = \frac{Found\ bugs}{Test\ cases} * \frac{Generating\ test\ case}{Time}$$

- The difficulty of designing a fuzz testing tools is easy than other testing tools.

But in real world, pure fuzz testing usually only finds simple faults of programs. So it usually combine with other testing method to help other methods obtain more improvement. For example, Automated White Fuzz [21] Testing is a work combined with symbolic testing, and Hybird Concolic Testing [22] is a work combined with concolic testing.

3 Related Work

3.1 LLVM Compiler Framework

LLVM (Low Level Virtual Machine) provides a virtual instruction set, which provides rich, language-independent, type and SSA(Static Single Assignment) information about instruction operands. Nowadays, it supports many programming languages and back-ends of machine architecture.

3.2 KLEE Symbolic Virtual Machine

KLEE is a redesigned work from EXE. The biggest different of KLEE and EXE is that KLEE is a symbolic testing tool which bases on top of LLVM compiler infrastructure, which has language independent instruction set and type system. Klee use bitcode, which a middle-end instructions llvm-gcc or clang produced, as input to do symbolic testing. KLEE execute bitcode by instruction and instruction, and then convert instruction information to constraints. KLEE use ExecutionState object to manage the path and constraint information.KLEE use some state search algorithm to choose which state should be executed first. It also cache the queries and their results to speed up the speed of query.

KLEE use STP as its default constraints solver. But it also design another constraints solver: Kleaver, which can be presented by KQuery language. The KQuery language is closely related to the C++ API for KLEE's Exprs (which is LLVM instruction appending some information KLEE maintains).

3.3 Catchconv

Catchconv [23] is based on Valgrind [24] framework. Catchconv combines symbolic execution and run-time type inference from a sample program run to generate test cases. And it mentions how to convert program and symbolic memory address to constrains of STP in detail. The architecture and implementation of Catchconv can execute query parallel easily, but the amount of constraints Catchconv generating still too much.

3.4 Alert

Alert [25], [26] is abbreviated from "Automatic Logic Evaluation for Random Testing". Alert is a testing tools using concolic execution technique and refering EXE and Cute testing framework to implement. It use CIL [27] (C Intermediate Language) to instrument the source code and produce an instrumented code. Alert executes with instrumented code and collects constraints by instrumented part. And it use CVC3 as constraints solver. In 2009, Alert support symbolic testing with files input [28], and use STP as constraints solver. The method Alert used is "List & Pick", list all possible constraints about files input and pick some of them to union (operation of "or"). Alert implement symbolic files testing by instrumented some file functions of C standard library like fopen, fgets, fseek and fscanf.

CAST [29] is a improvement and modified work from ALERT. It simplify CIL instrumented part and implement a parser which is written in Ocaml let user can define universal check self. The most important contribution of this work is implement of symbolic pointer read/write with symbolic value. This implement is a significant improvement to the auto generating of exploit because that if you can control a symbolic pointer and symbolic value, you almost can access any memory address.

Alert implements many useful features and functions, but its disadvantage is the level of tested sources can not scale still now.

3.5 zzuf

As early mentioned, zzuf is a fuzz testing tools with standard input and network fuzz supported. Zzuf's primary testing target is media players (like mplayer [30]), image viewers and web browsers. It has many options to control generating test cases. For example, it can control the ratio of difference between original data and generating data. It also can assign the byte offset range to fuzz. It use the regular expression to match file name, and decide fuzz the file or not.

3.6 HAMPI

HAMPI [31] is a SMT solver based on top of STP and it is for string constraints over fixed-size string variables. HAMPI do some implements to make constraints expressing membership in regular languages and fixed-size context-free languages. Given set of constraints, HAMPI outputs a string if constraints are satisfiable. The implement steps of HAMPI are following :

1. Normalize constraints to HAMPI's core string constraints
2. Translate core string constraints to a quantifier-free logic of bit-vectors
3. Call STP to judge the bit-vector constraints are satisfiable or unsatisfiable.
4. If constraints are satisfied, compute and output the string solution.

HAMPI does evaluations of symbolic testing using KLEE. First, HAMPI conpile the input grammar file into STP bit-vector constraints. Then it converts STP bit-vector constraints into C language code that expresses by API of KLEE. Finally, it inserts these code to test source code and runs KLEE on target program. The line coverage result of experiments is better than original KLEE.

	KLEE	Alert(linyt)	catchconv	zzuf	Klee with HAMPI
Testing Method	Symbolic	Symbolic	Symbolic	Fuzz	Symbolic
Symbolic testing with file	Yes	Yes	Yes	N/A	No
Regular expression supported	No	No	No	Only filter Args	Yes
Advantage	Can do library level symbolic testing	Implement symbolic index	Easy to proceed in parallel Run-time type inference	Many options to control generating data. Good for blackbox testing.	Regular expression supported
Disadvantage	The paths of doing symbolic file testing is too significantly.	Can not scale	The amount of constraints too significantly. Not fit the loop structure.	Usually only test easy fault	Need to modify source files

Figure 3-1: The Comparison of Some Relative Works

4 Methods

In this thesis, we implement the part of converting regular expression to constraints and resolving these constraints with original constraints. In this section, we will describe our method to handle the standard input constraints with user-defined regular expression information. We introduce that how does KLEE test with symbolic files first, and explain the reasons that why do we choose the method. Finally, we display the overview of our method. The details about implementation will be described at next section.

4.1 The architecture of KLEE with symbol files

As figure 4-1 show, the source code of C language is compiled in LLVM bitcode by llvm-gcc (or clang) at first. At the next step, KLEE reads the bitcode and initializes the testing environment. To implement symbolic files testing, KLEE creates a symbolic buffer and construct the relation between standard input and symbolic buffer, then KLEE select a state from the set of states to collect constraints information instrution by instrution. If KLEE meets the branch instruction (the definition is: according some conditions like value of a variable, you has two or more basic blocks to choose. For example, if, case, while and for) or alloc instruction, it will use solver to check the possible paths, then let forked state to wait until it is selected by searcher. If there are no execution states waiting to be selected, KLEE resolves the execution constraints per path to generate the test case which can explore the corresponding execution path.

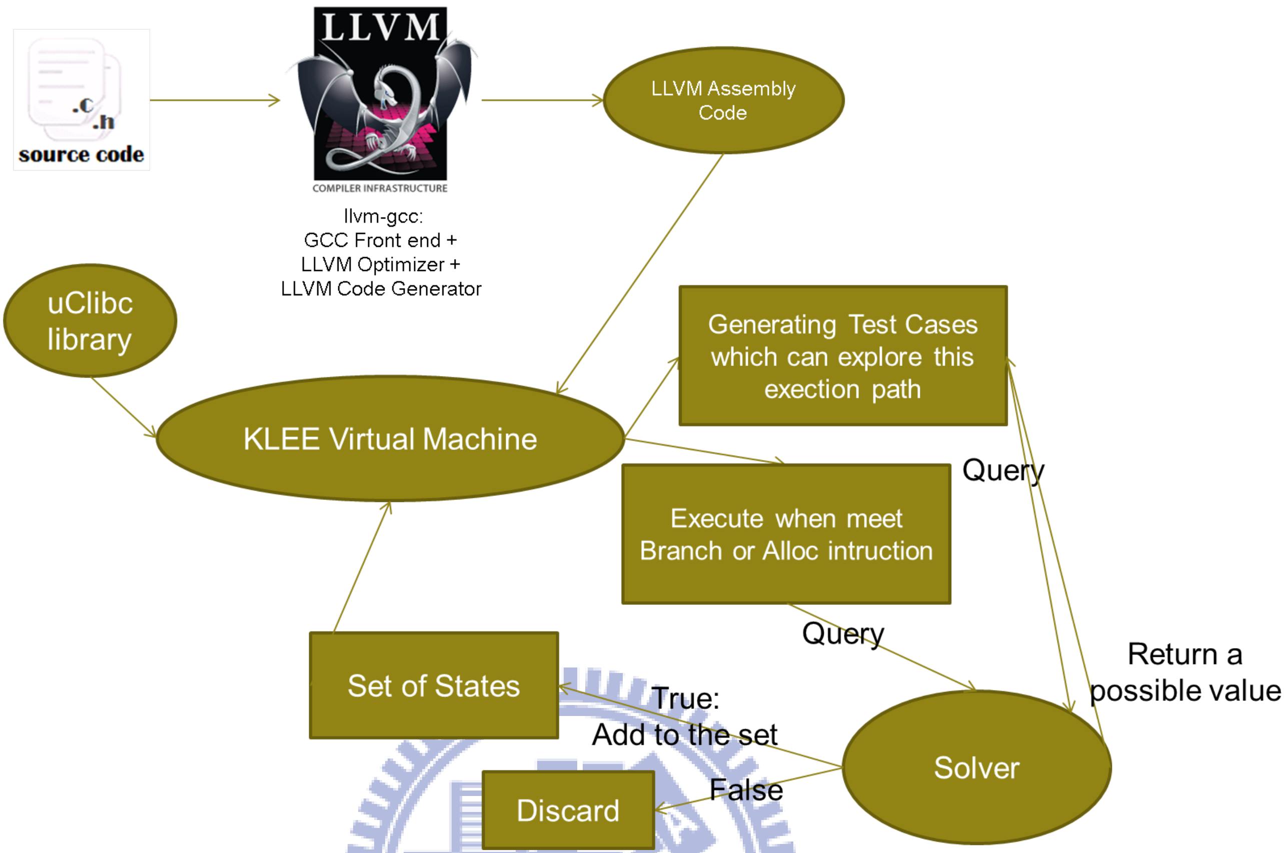

Figure 4-1: Klee Architecture

4.2 The architecture of KLEE with regular expression symbolic files support

4.2.1 The intuitive idea

The most important key of this thesis is: how do we use regular information to limit constraints. Usually, the method we using regular expression is that let strings to match regular expression. But the situation of handling constraints is different from handling strings. The character value of a string is fixed, but the constraints may contains many possible values. Consider the difference, we consider two methods to handle this problem at first. The first method is : At KLEE selects a execution state, we use SMT solver to resolve constraints and get the value which is not contradictory to constraints. If the value can not adhere to regular expressions, we discard the execution state. The problem of this method is: The resolved value can not represent constraints, it is only a possible value about constraints. So if a execution state is discarded, that does not mean the execution state can not satisfy regular expression. In some cases, this method filter more than 99 percentage of execution paths that is not contradictory to user-defined regular expression.

The second method is: We use regular expression to generate fixed value inputs, then convert these inputs to constant constraints. For every input, we create execution state and add corresponding constant constraints to execution state. This method also has some problems:

- The number of possible cases which can satisfy regular expression are significant.
- Be the same with the problem of fuzz testing, most of generated values are corresponding the same execution path.

4.2.2 The improved idea

According the analysis above mentioned, the two methods have some critical problems. The main reason is that we use fixed values to replace constraints or regular expression in some procedures, so some information is missing and inaccurate. We consider other methods which does not loss any information. The third method is "regular expression expansion" method. The "expansion" means that expanding all possible value in non-fixed times expression range,then we convert these expanded regular expressions to fixed-length constraints. And we append every constraints to forked states, the advantage of this method is needless to use SMT solver (There is no failed result by add constraints because the initial execution state has no constraint.) The disadvantage is that the number of possible expanding states may be significant.

The fourth method is ”checking every time” method. When KLEE’s searcher select a execution state, we compare the constraints to regular expression byte by byte. The advantage is we can append no constraints to states (because of the information of non-fixed times expression can not convert to constraints, so if we use third method, there still is some error), the disadvantage is that the number of solver query requests is significant. The number of solver query requests usually are bottleneck of symbolic testing, so this method is not efficient.

Finally, consider the size of standard input buffer usually has some limit, the number of possible expanding states can be reasonably limited, we use third method to implement.

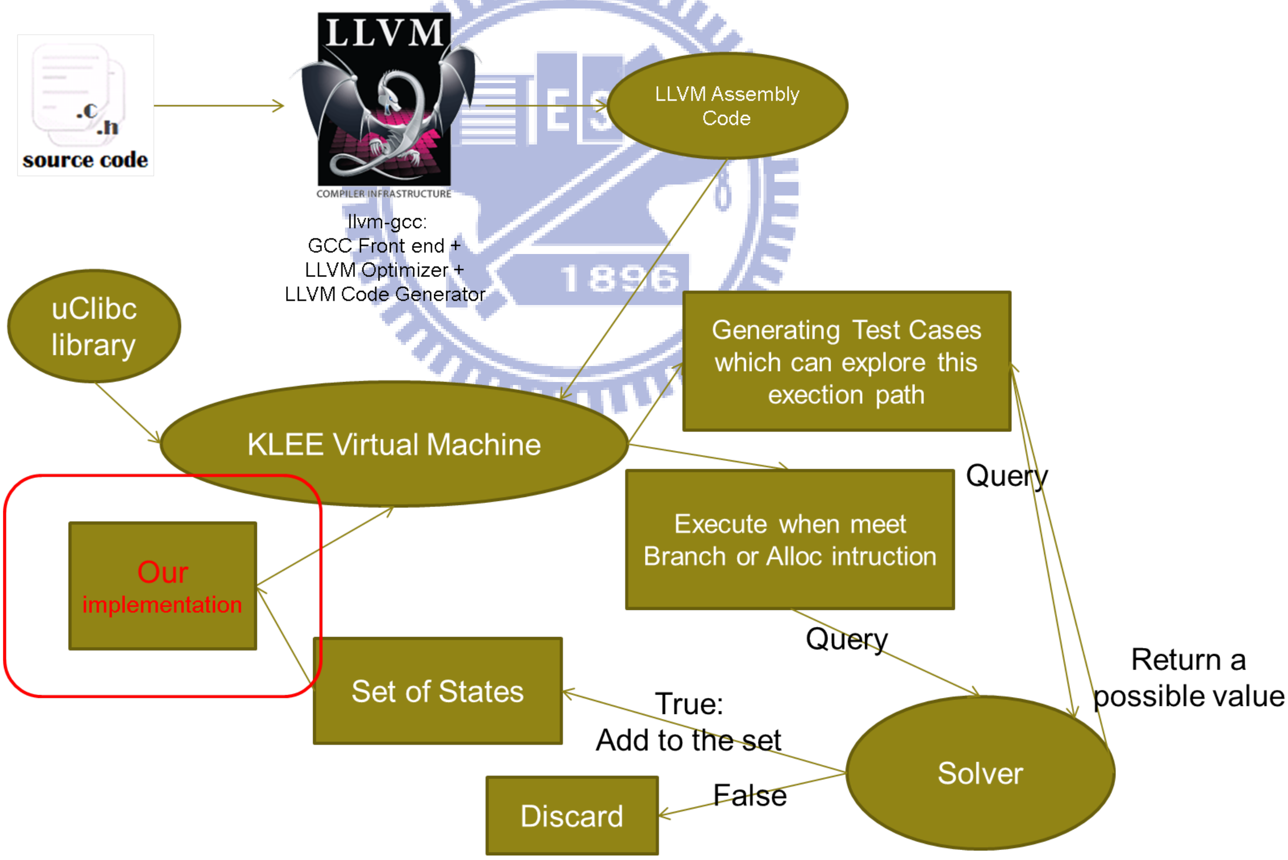

Figure 4-2: Klee Modified Architecture

4.2.3 The Overview of Flowchart

Our method is described as following: First, at the moment of KLEE's searcher selects a state to continue exploring paths, match the path constraints to regular expression's information, see Figure 4-2. If it can be resolved after adding regular expression's information, still execute. If it can be resolved, then discard this execution path.

Because the smallest unit of regular expression is character and the size of character is byte, we operate adding constraints of regular expression information in byte level. The flow of our method is showed as Figure 4-3.

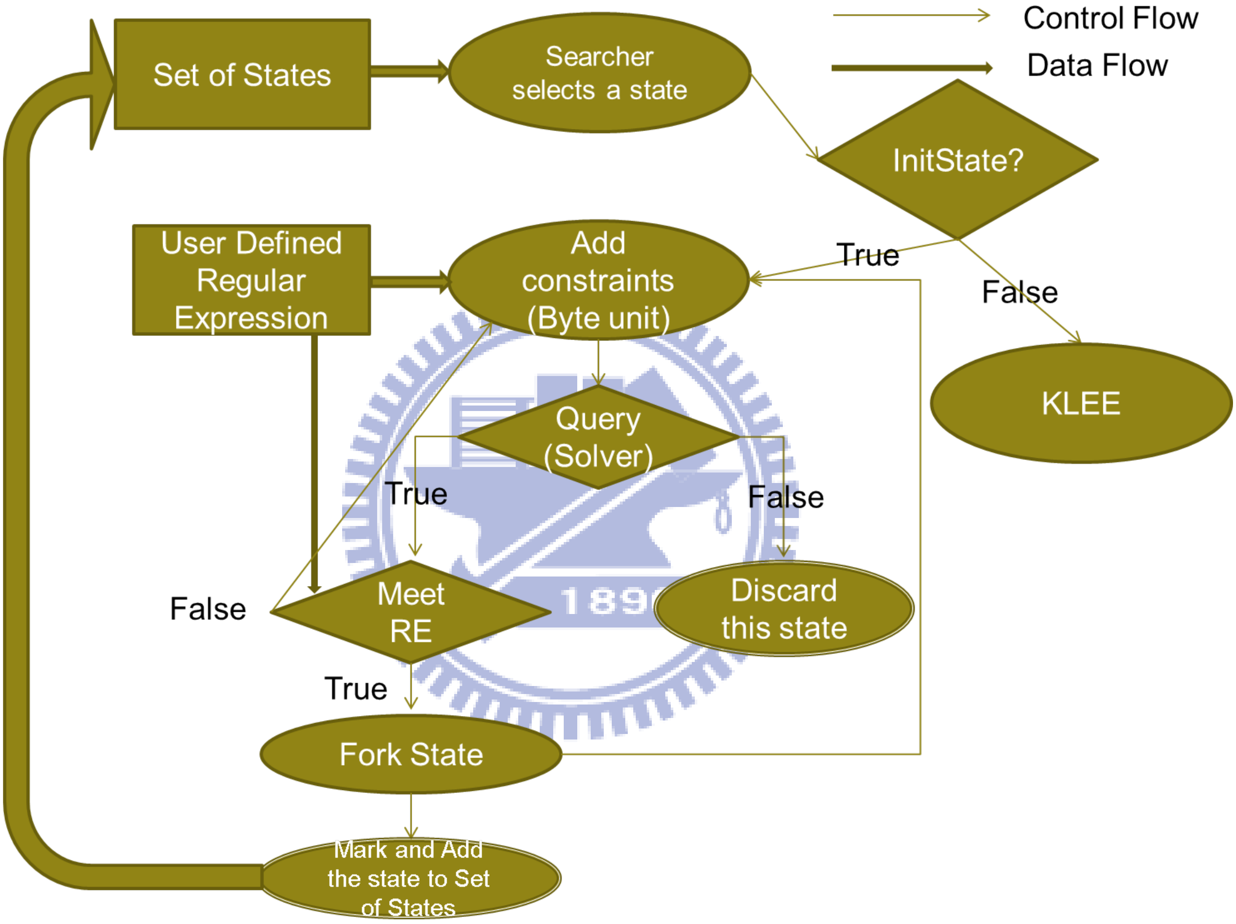

Figure 4-3: KLEE with Regular Expression Matching Logic

5 Implement

5.1 Data Struct

A regular Expression includes two important type of information. First type of information is contents of comparison, second type of information is the number range of match times. So we define two kinds of structure to record and describe the information of regular expressions.

5.1.1 ByteUnit

Structure ByteUnit consists of the type of contents of comparison. It consists of two attributes:

- type: We only implement following type about contents of regular expressions:
  - .: all, all possible character.
  - \d: digit, from character '0' to character '9'.
  - \w: word, same as $[a-zA-Z0-9\_]$.
  - \s: space, \t, \n, \f, \r and \v.
  - [...]: bracket, all list characters are possible values.
  - [ˆ...]: nbracket , all list characters are impossible values.
- contents: A vector of characters. If the type is bracket, nbracket(not including beginning ˆ) or other, this attribute contains the character(s). We handle the contents of bracket or nbracket type at patch function.

5.1.2 TimesExpr

Structure TimesExpr records the match times information (like {?,?}, ?, * and +) about some elements of regular expression. It consists of four attributes:

- min: An unsigned integer records the minimum number of match times.
- max: An unsigned integer records the maximum number of match times. If the match times is a fixed value rather than a range, the value of max is zero.
- nextLevelTEs: A vector contains of TimesExpr objects, and these TimesExpr are sequencing by order. Because a regular expression usually contains more than one match times information, and some match times information usually contains other match times information. For example (\d{2,4}){3}, the {3} contains the match times information {2,4} .
- unit: It contains a ByteUnit object.

To explain easily, in regular expression, parentheses represent a TimesExpr. If a TimesExpr contains no nextLevelTEs, it must contain a ByteUnit object.

5.2 Parse Regular Expression to Structure

In this section, we introduce the functions to parse regular expression and the flowchart of parse procedure.

5.2.1 preparse Function

$TimeExpr *\ preparse(void)$

To avoid some regular expression like (\w{2,3})(\d{3,4}) has more than two TimesExpr after parsing, so we add parentheses at begin and end, then call parse function to parse. Finally, this function return a TimesExpr to repersent all regular expression .

5.2.2 parse Function

$void\ parse(TimesExpr\ \&te,\ std :: string\ re)$

Parse() is the most important function to handle regular expression to our defined structures. If there is a right parenthesis, find the corresponding left parenthesis and create a TimesExpr to represent this parentheses (Notice that if before the parenthesis has continuous odd "\"s, the parenthesis only is a common character). Then call parse function recursively to parse the part of regular expression in parentheses. In this situation, the TimesExpr contains many TimesExpr.

If there is not a right parenthesis, we still create a TimesExpr, but this TimesExpr only contains a ByteUnit. We also assign the corresponding type according the character parsed.

After parse a TimesExpr, we call times function to check the minimum and maximum value of this TimesExpr.

In there is no parentheses, we parse ByteUnit and still create a TimesExpr to append ByteUnit to TimesExpr.

5.2.3 times Function

$unsigned\ int\ times(std :: string\ partnre,\ unsigned\ int\ \&min,\ unsigned\ int\ \&max)$

Function to handle times expression. Set up the value of minimum and maximum in TimesExpr and return the string length of time expression.

5.2.4 reparse Function

$void\ reparse(TimeExpr\ \&te)$

Function to handle times expression which is redundant: a TimesExpr is a redundant TimesExpr if its contains no unit and its minimum is 1 and its maximum is 0. We will erase the redundant TimesExpr to improve efficiency.

5.2.5 Example

We use the regular expression (\d{2,4})+ to explain the procedures.

1. First, call preparse function to add a pair of parenthesis to (\d{2,4})+: ((\d{2,4})+).

2. Create a TimesExpr object to represent all regular expression, and set up the minimum and maximum value to 1 and 0.

3. Call parse function, find out the right parenthesis. So find the corresponding left parenthesis and call parse function recursively: \d{2,4}.

4. Find out a ByteUnit: \d, create TimesExpr to contains the ByteUnit and set the ByteUnit type to digit. Then call times function to set up the minimum value and maximum value of TimesExpr. Finally, append this TimesExpr to nextLeverTEs of upper level TimesExpr.

5. Find out a ByteUnit: the space character. Create TimesExpr to contains the ByteUnit and set the ByteUnit type to other. Then call times function to set up the minimum value and maximum value of TimesExpr. Finally, append this TimesExpr to nextLeverTEs of upper level TimesExpr.

6. Finish parsing the \d{2,4}, call times function and found the '+' symbol, so we set up minimum to 1 and maximum to a defined number.

7. The parsing result is showed in Figure 5-1.

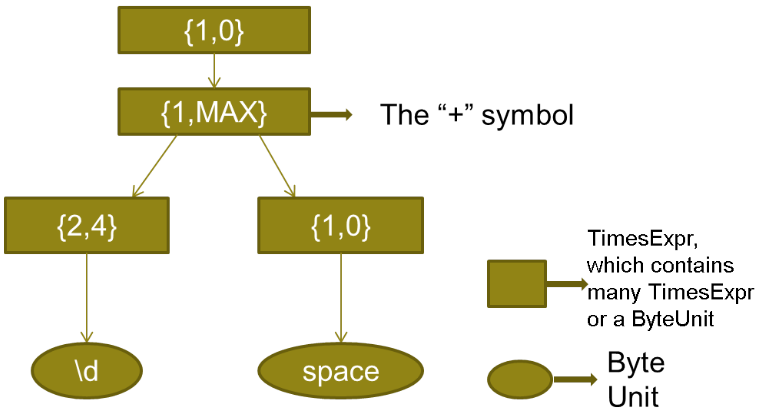

Figure 5-1: The example we handle regular expression (before reparse)

8. Call reparse, the result is showed in Figure 5-2.

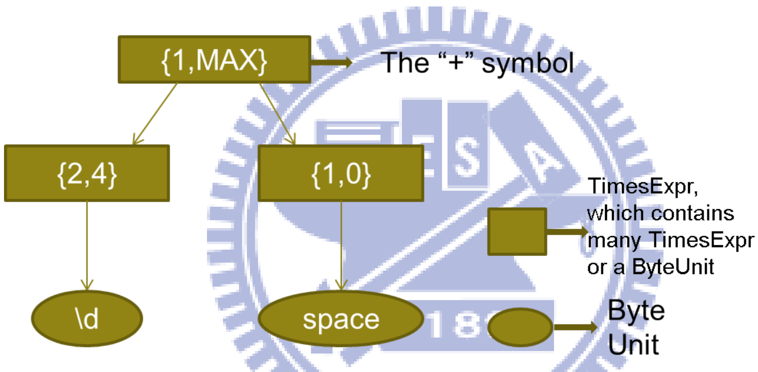

Figure 5-2: The example we handle regular expression (after reparse)

5.3 Patch Constraints by Parsed Regular Expression

5.3.1 patch function

$void\, patch(TimeExpr\ \&te,\ std :: vector <\ std :: pair < ExecutionState*,\ unsigned\, int > > \ \&patchstates,\ bool\ top)$

The parameters patchstates contains the states waiting to be patched and the patched byte size of the corresponding state. This function to handle these part:

1. Add constraints to all execution states in patchstates according to regular expression and the corresponding byte size. After adding constraints, increase 1 to the byte size.

2. Fork execution state and maintain the relationship between original state and fork state (KLEE's PTree structure).

At beginning, patchstates only contains initial state, and its patched standard input size is 0. If TimesExpr contains nextLevelTEs, call patch function recursively with per TimesExpr in nextLevelTEs. If TimesExpr contains ByteUnit, we patch patchstates according type of ByteUnit and patched byte size.

```

std::vector< std::pair<ExecutionState*, unsigned> >::iterator si = patchstates.begin(),
        se = patchstates.end();
std::vector< std::pair<ExecutionState*, unsigned> > addStates;
while(si!=se){
         unsigned int count = 0; // record the times
         for(;count<te.min;count++){
                ExecutionState * tmp = si->first;
                if(si->second == tmp->symbolics[0].first->size)
                        break;

                // generating constraints and add to si->second byte
                (si->second)++;
        }

        for(;count<te.max;count++){
          ExecutionState * tmp = si->first;
          if(si->second == tmp->symbolics[0].first->size)
            break;

          ExecutionState *forkState = tmp->fork();
          addStates.push_back(std::make_pair(forkState,si->second));

                 // generating constraints and add to si->second byte
                (si->second)++;
        }
        si++;
}
patchstates.insert(patchstates.end(), addStates.begin(), addStates.end());
addStates.clear();
```

Figure 5-3: The pseudo code about times expression patch

The patching method is described in Figure 5-3 simply. First, we patch states according the TimesExpr minimum value, then if we adhere to the requirement of minimum value, we try to patch states according the maximum value. Because all times between minimum and maximum adhere to requirement, we want to list all possible situation. So we fork state, add one to a temp vector: addStates, and process with another state.

We explain how we implement some feature in Figure 5-4, Figure 5-5, Figure 5-6, Figure 5-7 and Figure 5-8.

```
ref<Expr> Space_L = ConstantExpr::alloc(8, Expr::Int8);
ref<Expr> Space_R = ConstantExpr::alloc(14, Expr::Int8);
ref<Expr> CharSpace_L = ConstantExpr::alloc(31, Expr::Int8);
ref<Expr> CharSpace_R = ConstantExpr::alloc(33, Expr::Int8);
ref<Expr> INT_L = ConstantExpr::alloc(47 Expr::Int8);
ref<Expr> INT_R= ConstantExpr::alloc(58, Expr::Int8);
ref<Expr> Upper_L= ConstantExpr::alloc(64, Expr::Int8);
ref<Expr> Upper_R= ConstantExpr::alloc(91, Expr::Int8);
ref<Expr> Lower_L= ConstantExpr::alloc(96, Expr::Int8);
ref<Expr> Lower_R= ConstantExpr::alloc(123, Expr::Int8);
ref<Expr> CHARUD_L= ConstantExpr::alloc(94, Expr::Int8);
ref<Expr> CHARUD_R= ConstantExpr::alloc(96, Expr::Int8);

void range_AND(state, L, cur, R){
        ref<Expr> result1 = SltExpr::create(L, cur); // L <  cur
        ref<Expr> result2 = SltExpr::create(cur, R); // cur < R
        ref<Expr> result3 = AndExpr::create(result1, result2); // L < cur < R

        solver->mustBeFalse(state, result3, failed);

        if(failed) terminateStateOnError(state);
        else addConstraint(state, result3);
 }

 void range_OR(state, L, cur, R){
        ref<Expr> result1 = SltExpr::create(cur, L); // cur < L
        ref<Expr> result2 = SltExpr::create(R, cur);// cur > R
        ref<Expr> result3 = OrExpr::create(result1, result2); // cur < L || cur > R
        solver->mustBeFalse(state, result3, failed);

        if(failed) terminateStateOnError(state);
        else addConstraint(state, result3);
 }

void range_EQ(state, C, cur){
        ref<Expr> result1 =  EqExpr::create(cur, C);
        solver->mustBeFalse(state, result1, failed);

        if(failed) terminateStateOnError(state);
        else addConstraint(state, result1);
 }
```

Figure 5-4: The pre-defined instruction and function

```
ref<Expr> C= ConstantExpr::alloc(char1, Expr::Int8);

ref<Expr> cur = (state.addressSpace.findObject(state.symbolics[0].first))->read8(i);
func_EQ(state, C, cur);
```

Figure 5-5: The pseudo code about handling any char

```

unsigned int pos;
std::vector< ref<Expr> > OrExprs;
ref<Expr> cur = (tmp->addressSpace.findObject(state.symbolics[0].first))->read8(i);

for(pos=0;pos<te.unit->contents.size();pos++){
        if(te.unit->contents[pos] == '\\'){
                ref<Expr> target, result1;
                switch(te.unit->contents[++pos]){
                        case '-':
                        case '\\':
                        case '.':
                                target = ConstantExpr::alloc(te.unit->contents[pos], Expr::Int8);
                                result1 = EqExpr::create(cur, target);
                                OrExprs.push_back(result1);
                                break;
                        default:
                                break;
                }
        }
        else if(te.unit->contents[pos+1] != '-'){
                ref<Expr> target = ConstantExpr::alloc(te.unit->contents[pos], Expr::Int8);
                ref<Expr> result1 = EqExpr::create(cur, target);
                OrExprs.push_back(result1);
        }
        else{
                ref<Expr> limitL = ConstantExpr::alloc(te.unit->contents[pos]-1, Expr::Int8);
                ref<Expr> limitR = ConstantExpr::alloc(te.unit->contents[pos+2]+1, Expr::Int8);
                ref<Expr> result1 = SltExpr::create(limitL, cur);
                ref<Expr> result2 = SltExpr::create(cur, limitR);
                ref<Expr> result3 = AndExpr::create(result1, result2);
                OrExprs.push_back(result3);
                pos+=2;
        }
}
if(OrExprs.size() == 1){
        if(te.unit->type & nbracket)
                OrExprs[0] = NotExpr::create(OrExprs[0]);
                addConstraint(state, OrExprs[0]);
}
else{
        ref<Expr> result = OrExpr::create(OrExprs[0], OrExprs[1]);
        for(pos=2;pos<OrExprs.size();pos++)
                result = OrExpr::create(OrExprs[pos], result);

        if(te.unit->type & nbracket)
                result = NotExpr::create(result);
        addConstraint(state, result);
}
```

Figure 5-6: The pseudo code about handling bracket and nbracket

```
ref<Expr> cur = (state.addressSpace.findObject(state.symbolics[0].first))->read8(i);

range_AND(state  ,INT_L, cur, INT_R);
```

Figure 5-7: The pseudo code about handling IsDigit

```
ref<Expr> cur = (state.addressSpace.findObject(state.symbolics[0].first))->read8(i);

range_AND(state, INT_L, cur, Lower_R);
range_OR(state, INT_R cur, Upper_L);
range_OR(state, Upper_R, cur, CHAR_UD_L);
range_OR(state, CHAR_UD_R, cur, Lower_L);
```

Figure 5-8: The pseudo code about handling IsWord

5.3.2 Example

We use the regular expression (\d{2,4})+ and sample code like Figure 5-9 to explain the details of our implementation. First, we separate the part of times expression and part of comparison from regular expression. We can get a structure like Figure5-2. If the times is big than the min value of times expression and small than the max value of times expression, We can fork the execution state, one of execution state try to adhere the max value of times expression, another execution state is continuing to do next comparison with other byte constraints. If match the all regular expression, we will mark and add the forked execution state to set of execution states. According this flow, we will get the result like Figure 5-10.

```
#include<stdio.h>
#include<string.h>

int main(){
        char buf[12];
        int i, j;
        fgets(buf,12,stdin);
        if (buf[6] == 'a') print("branch!\n");
        while(sscanf(buf," %d %d",&i,&j)==2){
                printf("%d %d\n",i, j);
        }
}
```

Figure 5-9: source code of example using to explain implementation

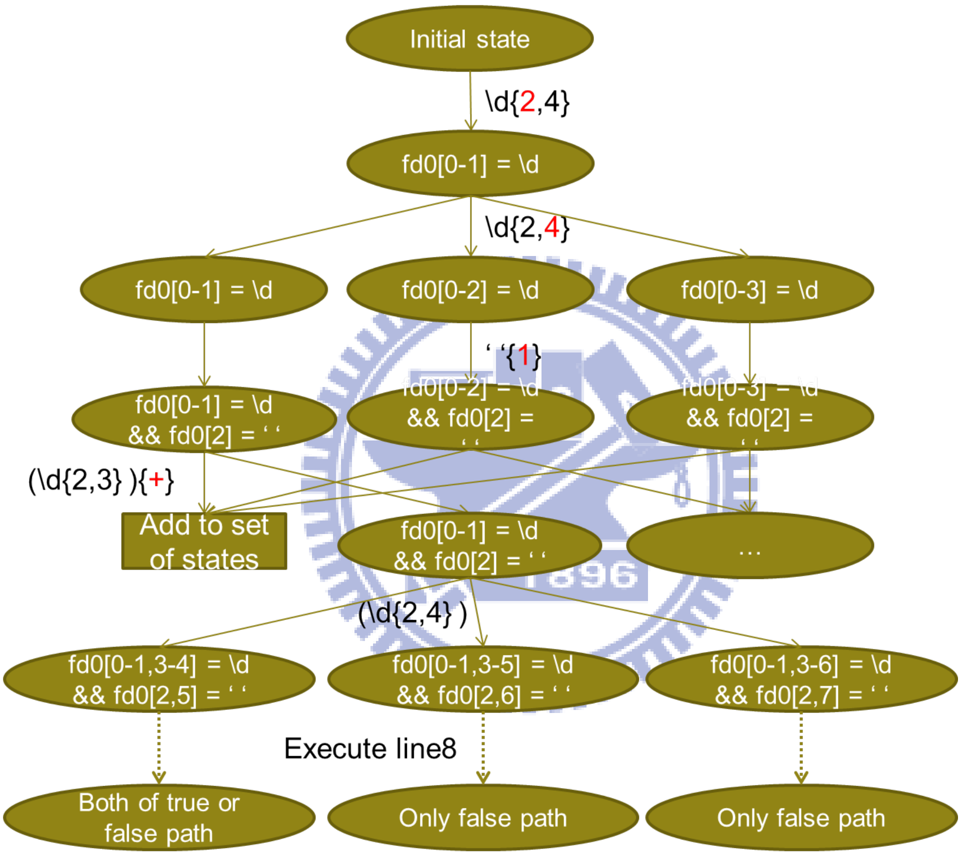

Figure 5-10: The example we handle regular expression

6 Evaluations

6.1 Compare with Original KLEE

In the section, we use some examples to demonstrate the functionality of our implementation.

6.1.1 Test case 1

We use regular expression (\d{3} \d{2}.)+ to do symbolic file testing with our implement, and the source code is Figure 1-2. The produced test cases and the cover paths show in Table 6-1 when buffer size is 8 .

Test Case #	Input Contents	Cover Line
1	32323220 3232002E	6 9 11 15
2	30303020 3232002E	6 9 13 15
3	30303020 30300a2E	6 9 13 6 8 15

Table 6-1: The test data produced by our method about test case 1

If want to fulfill the regular expression, the execution path including line 7 is impossible, so all the possible paths we explored. Then we can see the comparison with original KLEE in Table 6.1.1:

Test data 1	Test data 2	Test data 3
out implement 16 seconds	klee 8 hours	klee 17 seconds
3 new paths	14 new paths	12 new paths

Table 6-2: Comparison with KLEE

According the compare of last two test data, we can observe obviously the efficiency of generating new execution paths at early period of executing.

Compare with first and third test data, At first glance, original KLEE seems to have more efficient, but analyse the test data carefully, we will found that most of these execution paths is relative with uClibc [7] standard library, the new execution paths which is relative with Figure 1-2 is two test cases only, see Table 6-3.

Test Case #	Input Contents	New Path?
1	2E000000 00000000	v
2	2E010000 00000000	v
3	2E010101 01010102	
4	00000000 00000000	
5	10000000 00000000	
6	00100000 00000000	
7	01010000 00000000	
8	30000000 00000000	
9	20000000 00000000	
10	2D000000 00000000	
11	2B000000 00000000	
12	41000000 00000000	

Table 6-3: The test cases original KLEE produced about test case 1

6.1.2 Test case 2

We use regular expression (\d{2}\n)+ to do symbolic file testing with our implement, and the source code is Figure 6-1. We set the size of stdin to 10, and our generating test cases is showed Table 6-4:

Test Case #	Input Contents
1	30311032 32103232 1032
2	31361032 32103232 1032
3	30371032 32103232 1032
4	30321032 32103232 1032
5	30341032 32103232 1032
6	30361032 32103232 1032
7	30351032 32103232 1032
8	30331032 32103232 1032

Table 6-4: The test cases our method produced about example 2

Except the path of test case 2 is generating because of new execution path about uClibc standard library, other test cases is obviously relative with line 20-61 of Figure 6-1 . The paths generated is cover all paths we can observe. The time spending to generating the test cases is 2.3 seconds in our method. But original KLEE spends 49.3 seconds and can not resolve any test case. Our method use the memory size of 13.7 megabytes , show as Figure 6-2, but KLEE use the memory size of 2 gigabytes, show as Figure 6-3. According we observe, Klee can not resolve any execution paths because that the memory size is not enough. In this test case, the advantage about adding regular constraints is obvious.

```
#include<stdio.h>
#include<math.h>
unsigned int table[31];
void func(int n);
int main(int argc, char* argv[]){
    int ca = 1, num, tmp, i, j;
    table[0] = 1;
    for(i=1;i<31;i++)
        table[i] = 2 * table[i-1];
    while(scanf("%s",&num)){
        printf("case %d:\n",ca++);
        func(num);
        printf("\n");
        if (ca > 10) break;
    }
    return 0;
}

void func(int n){
     unsigned int tmp1, tmp2, flag, i, j;
     if (n == 1){
        printf("{}\n{1}\n"); return;
     }
     else if (n == 2){
        func(1);
        printf("{2}\n{1,2}\n"); return;
     }
     else if (n == 3){
        func(2);
        printf("{3}\n{1,3}\n{2,3}\n{1,2,3}\n"); return;
     }
     else if (n == 4){
        func(3);
        printf( the answer n == 4 ); return; //change some code
     }
     else if (n == 5){
        func(4);
        printf( the answer n == 5 ); return; //change some code
     }
     else if (n == 6){
        func(5);
        printf( the answer n == 6 ); return; //change some code
     }
     else{
        func(n-1);
        tmp1 = pow(2,n); tmp2 = pow(2,n-1);
        for(i = tmp2; i < tmp1 ;i++){
              flag = 0;
              printf("{");
              for (j = 0;j<n;j++){
                  if(i & table[j]){
                       if (flag)
                          printf(",%d",j+1);
                       else{
                           flag = 1; printf("%d",j+1);
                       }
                  }
              }
              printf("}\n");
        }
        return;
     }
}
```

Figure 6-1: source code of example 2

```
sqlab@klee:~/klee-test$ ./show-testcase 197
klee-out-197/test000001.ktest
        object    0: data: '01\n22\n22\n2'
klee-out-197/test000002.ktest
        object    0: data: '16\n22\n22\n2'
klee-out-197/test000003.ktest
        object    0: data: '07\n22\n22\n2'
klee-out-197/test000004.ktest
        object    0: data: '02\n22\n22\n2'
klee-out-197/test000005.ktest
        object    0: data: '04\n22\n22\n2'
klee-out-197/test000006.ktest
        object    0: data: '06\n22\n22\n2'
klee-out-197/test000007.ktest
        object    0: data: '05\n22\n22\n2'
klee-out-197/test000008.ktest
        object    0: data: '03\n22\n22\n2'
sqlab@klee:~/klee-test$ klee-stats --print-all klee-out-197
-------------------------------------------------------------------------------------------------------------------------------------
| Path         | Instrs | Time(s) | ICov(%) | BCov(%) | ICount | Solver(%) | States | Mem(MB) | Queries | AvgQC | Tcex(%) | Tfork(%) |
-------------------------------------------------------------------------------------------------------------------------------------
| klee-out-197 |  14307 |    2.33 |   21.98 |   12.51 |  12080 |     74.86 |      0 |   13.69 |      73 |   123 |   51.50 |     0.33 |
-------------------------------------------------------------------------------------------------------------------------------------
```

Figure 6-2: The result and information about our method in example 2

```
sqlab@klee:~/klee-test$ ./show-testcase 193
sqlab@klee:~/klee-test$ klee-stats --print-all klee-out-193
---------------------------------------------------------------------------------------------------------------------------------------
| Path         |   Instrs | Time(s) | ICov(%) | BCov(%) | ICount | Solver(%) | States | Mem(MB) | Queries | AvgQC | Tcex(%) | Tfork(%) |
---------------------------------------------------------------------------------------------------------------------------------------
| klee-out-193 | 26593420 |   49.29 |   21.76 |   12.02 |  12080 |      1.68 |      5 | 2006.49 |      15 |   120 |    1.61 |     0.00 |
---------------------------------------------------------------------------------------------------------------------------------------
```

Figure 6-3: The result and information about klee in example 2

6.2 Compare with Other Work

6.2.1 Compare with Catchconv

We use testcase 1, 2 to do test with catchconv. Catchconv also can explore paths as well as our method, so we compare the time and memory usage. Figure 6-4 is the comparison result between our work and catchconv, we can find out that catchconv spends more times and memory than our method to explore path, and it also need to modify many scripts to do a testing.

	Our method	catchconv
Testcase 1	Time: 33 sec Memory: 14.9MB	Time: 5 min Memory: 60MB
Testcase 2	Time: 2.3 sec Memory: 13.7MB	Time: more than 30 min Memory: more than 1GB

Figure 6-4: The Comparison between our work and catchconv

6.2.2 Compare with HAPMI

We use program bc and logictree to compare with HAPMI. logictree is a solver for propositional logic formulas, and bc is a command-line calculator and simple programming language.

Because HAMPI does not release the script which can convert it constraints to KLEE API (like *klee_assert*()), we construct the almost same environment: one core of 2.66 GHz i7-920 CPU and 1GB of RAM. We also run these program 1 hour using our defined regular expression and use same buffer size. Finally, we use the gcov tool to measure the line coverage. The result is shown as Figure 6-5. Because the difference of tested program version, the ELOC (which means "Executable Lines of Code") also has some different. The upper ELOC is measured by us, and lower ELOC is measured by HAMPI.

Program	ELOC	Input size	Compare target	Our method (ori/regexp)	Hapmi (ori/grammer)
bc	1771 /1669	6	total line coverage	22.2%/33.1%	27.1%/43.0%
			parser file line coverage(324/332)	32.1%	39.5%
logictree	1447 /1492	7	total line coverage	18.3%/53.0%	31.2%/63.3%
			parser file line coverage(17/17)	53.0%	64.7%

Figure 6-5: The Comparison between our work and HAMPI

There are two reason to cause the result:

1. HAMPI does not use symbolic file testing. The affect can be observed by coverage of original KLEE testing (22.2% VS 27.1%). So HAMPI

does not need to challenge uclibc library which is more complex than KLEE original libc library.

2. The language level of HAMPI implemented is Context-free language, so is can cover more possible cases. For example, HAMPI can handle $a^n b^n$ problem easily(same number of right and left parenthesis). This case does not be handled well by regular language.

6.2.3 Compare with zzuf

We use zzuf to do fuzz bc program, the input buffer size is also 6. We control some parameter like fuzzing ratio to observe the affect. To do fuzz testing smoothly, we add "quit\n" to the end of fuzzing file and do not fuzz the part. The fuzzing ratio is 0.1, 0.2, 0.3 and 0.4, the every fuzzing ratio test with the number of test data: 1000, 2000, 3000, 4000, 5000 and 6000. The result is shown as Table 6-5 and Figure 6-6. The "all" means combining the fuzzing ratio is 0.1, 0.2, 0.3 and 0.4.

num \ r	0.1	0.2	0.3	0.4	all
1000	36.0%	38.2%	38.6%	39.5%	40.3%
2000	38.7%	39.6%	39.7%	40.0%	41.4%
3000	39.5%	40.9%	40.3%	40.2%	42.5%
4000	39.8%	41.1%	40.6%	40.2%	42.5%
5000	39.8%	41.5%	41.9%	40.2%	44.0%
6000	39.8%	41.6%	41.9%	40.3%	44.0%

Table 6-5: The line coverage of program bc using zzuf

To observe and analyse the figure, we can find out that:

- If the fuzzing ratio is too small or too big, the result of coverage is not good.

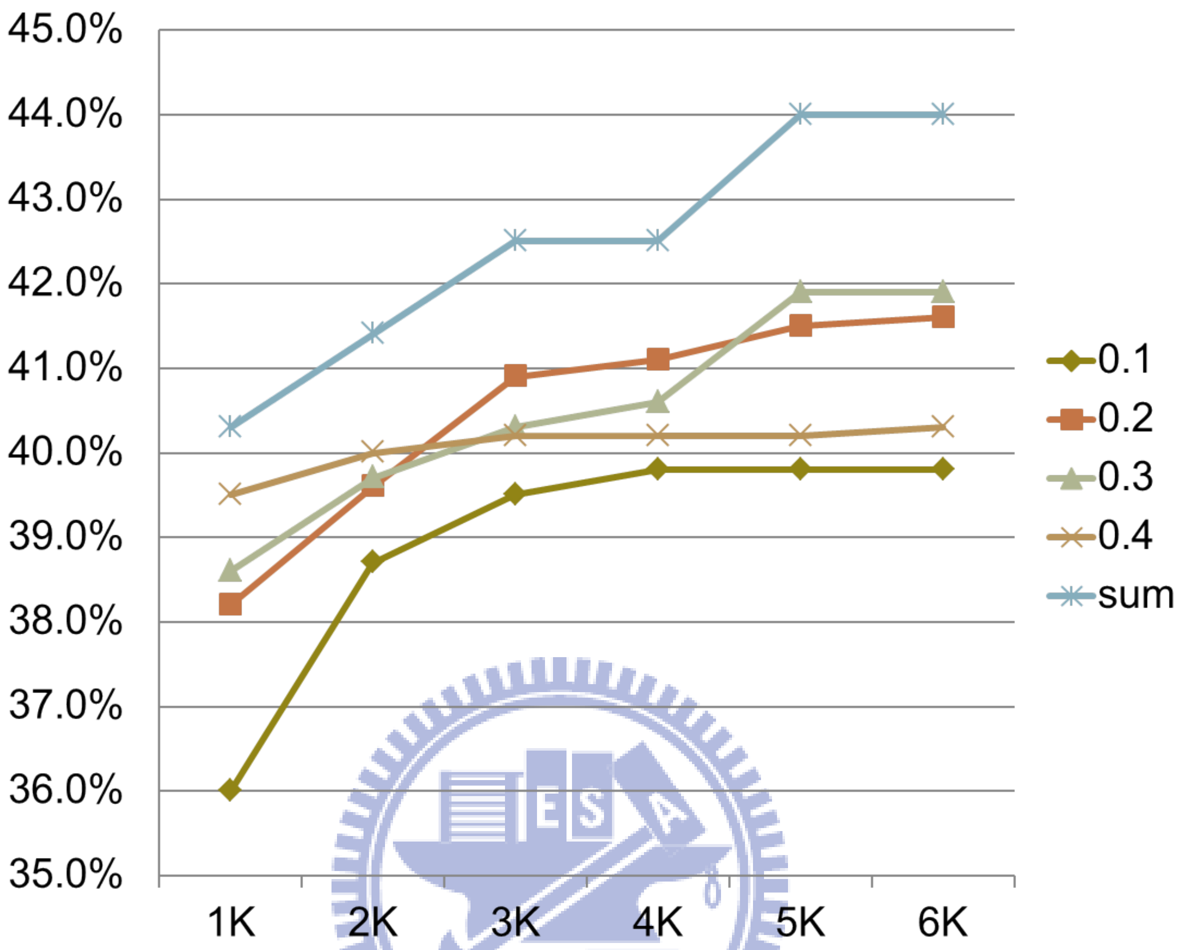

Figure 6-6: The figure of coverage comparison using zzuf

- Fuzz testing can gain the high coverage at beginning. But when we use more test data to do fuzz testing, the improvement of the coverage is small.

7 Conclusions

Scalability is an important issue in symbolic file testing. In KLEE, the complexity of C standard library is a main reason to bother the scalability. To address this problem we proposed regular expression constraints to limit the execution paths and focus on testing part of specifications. Description of input format is flexible and easily by using regular expression, . Finally we implement our method on KLEE to reduce the usage memory and enhance the line and path coverage. We hope this work can help programmers to test their program easily.

8 Future Work

As mentioned at result of HAMPI comparison, context-free language can handle more situation, so we can add context-free grammar supporting in KLEE. We can use Lex and Yacc tools to implement this feature.

But describing a context-free grammar usually difficult than describing regular expression, we can still maintain regular language part for some small programs testing. We can improve this part by using PCRE [32], which is short from Perl-compatible regular expressions, to support our work. PCRE provides some APIs, the important two function is "pcre_compile()" and "pcre_exec()". The function pcre_compile() can be used to build a data structure containing regular expression information, and the function pcre_exec() can be used to match a given string and given compiled regular expression. We can use pcre_compile() first, then refer pcre_exec() to implement a function which can match constraints and expression.

References

[1] G.J. Myers. *The art of Software Testing.* Wiley, 2004.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[3] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[4] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*, San Diego, California, Dec 2003.

[5] LLVM, http://llvm.org/.

[6] NCTU Online Jduge System, http://progexam.cs.nctu.edu.tw/.

[7] uClibc, http://www.uclibc.org/.

[8] Satisfiability Modulo Theories Competition (SMT-COMP), http://www.smtcomp.org/.

[9] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[10] Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[11] Vijay Ganesh and David Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer Berlin / Heidelberg, 2007.

[12] The Yices SMT solver, http://yices.csl.sri.com/.

[13] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In

Proceedings of the international conference on Reliable software, pages 234–245, New York, NY, USA, 1975. ACM.

[14] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[15] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):1–38, 2008.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[17] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *In CAV*, pages 419–423. Springer, 2006.

[18] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[19] Peach Fuzzing Platform, http://peachfuzzer.com/.

[20] zzuf, http://caca.zoy.org/zzu/.

[21] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[22] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software engineering*, 2007.

[23] David Alexander Molnar, David Molnar, David Wagner, and David Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical report, UC Berkeley EECS, 2007.

[24] Valgrind, http://valgrind.org/.

[25] Li-Wen Hsu. Resolving unspecified software features by directed random testing. Master's thesis, NCTU, 2007.

[26] C. F. Yang. Resolving constraints from cots/binary components for concolic random testing. Master's thesis, NCTU, 2007.

[27] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[28] Yan-Ting Lin. Non-primitive type symbolic input for concolic testing. Master's thesis, NCTU, 2009.

[29] You-Siang Lin. Cast: Automatic and dynamic software verification tool. Master's thesis, NCTU, 2009.

[30] MPlayer, http://www.mplayerhq.hu/.

[31] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[32] PCRE, http://www.pcre.org/.