國立交通大學

資訊科學與工程研究所

碩 士 論 文

應用目標導引與隨機測試之特定程式狀態產生器

Target Directed Random Testing for Feasible State Generation

研 究 生：劉彥佑

指導教授：黃世昆　教授

中 華 民 國 九 十 六 年 六 月

Target Directed Random Testing for Feasible State Generation
應用目標導引與隨機測試之特定程式狀態產生器

研 究 生: 劉彥佑　　　　　　Student: Yen-Yo Liu

指導教授: 黃世昆　　　　　　Advisor: Prof. 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 Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirement
for the degree of
Master
in

Computer Science
June 2007
Hsinchu, Taiwan, Republic of China

中華民國九十六年六月

應用目標導引與隨機測試之特定程式狀態產生器

研 究 生：劉 彥 佑　　　　指 導 教 授：黃 世 昆

摘要

為了增強軟體的強健度，找出軟體的錯誤行為一直是軟體工程領域裡面一個相當重要的課題。在過去已有許多這方面的研究使用靜態或動態程式分析的技術。然而，有時靜態分析回報的錯誤不一定在執行時期會發生，而動態分析通常無法找出全部潛在的錯誤。在本論文中，我們實做一個測試平台，此平台在執行被測程式的過程中，可以同時蒐集目前執行路徑上的條件限制，然後利用這些條件限制來自動地產生測試資料，以覆蓋不同的執行路徑。由於我們的測試平台可以自動地產生高覆蓋率的測試資料，因此我們的方法理論上可以找出全部潛在的錯誤。另一方面，因為我們有實際執行被測程式，所以回報的錯誤都是在執行時期真實會發生的。在本論文中，我們使用此平台來檢測一個由靜態分析工具回報的程式狀態所隱含的臭蟲，是否可能在執行時期發生。此平台會自動地嘗試產生一組測試資料來觸發目標的程式狀態，或是回報目標的程式狀態不可能在執行時期觸發。

關鍵字：

軟體測試、測試資料生成

Target Directed Random Testing for Feasible State Generation

Student : Yen-Yo Liu　　　Advisor : Shih-Kun Huang

Abstract

Locating software bugs is an important topic in software engineering for enhancing software robustness. Research topics in these areas with static analysis or dynamic analysis have been proposed. However, the diagnosis of static analysis usually has false positive, and dynamic analysis usually has false negative. We implement a testing framework which runs a tested program and collects symbolic constraints along its execution path. It can automatically generate test cases to cover different execution paths. In theory the diagnosis of our testing framework has no false negative because it can automatically produce test cases with high coverage. On the other hands, the diagnosis of our testing framework has no false positive because we concretely run the tested program. In this thesis, we use this tool to check whether or not a program state with potential bug reported by a static analysis tool is feasible in run time. The tool automatically tries to find a test case which can trigger the target program state, or report the target program state is infeasible.

Keywords:

Software Testing, Test Case Generation

誌謝

這篇論文的完成，首先要感謝我的指導教授黃世昆老師，這兩年來費心的指導，並且給予我許多的鼓勵，讓我受益良多。再來要感謝實驗室的昌憲學長，不論在研究或論文寫作上，都提供我許多的寶貴的建議。最後當然要感謝所有的實驗室成員，包括揚杰、立文、泳毅、友祥，我永遠會記得與你們在實驗室打拼的日子。

Table of Contents

List of Figures

List of Tables

1. Introduction

Automatic software testing is an important technique for finding software bugs. How to design an efficiently testing mechanism to find bugs is the key problem. Traditional random testing is easy to use without any user specification. But it maybe never covers some important execution paths because of the corresponding narrow input space[1]. So there are some path directed random testing techniques which are introduced for reducing redundant testing[2-6]. DART (Directed Automatic Random Testing)[2] combines symbolic execution and concrete run. It concretely runs the program and collects the constraints along with the execution path. The objective of DART is fully covering all execution paths. However, full coverage is infeasible to achieve for a large software system, because the number of all execution paths is too huge to completely enumerate.

Let us define program state as: *(L, P)*, where *L* is a program control location, when an execution on *L*, it satisfies a predicate *P*. Sometimes we only want to reach some program state in order to validate the correctness or check some security holes rather than full coverage. So we need an efficient and smart testing framework which can guide the testing to our target state rather than full coverage. In other words, we are interesting in the feasibility of some special program state *(L, P)*.

1.1 Motivation

Many software security issues can be modeled by state reachability problems. For example, we can model the bound checking of array access to a program state (*L*, *P*), which *L* is an array dereference statement, and *P* is the constraint of array index which is out of bound. In another example, when invoking a library we want to check whether the passed parameter violates the precondition. In the case of *malloc()*, we can check whether the parameter of *malloc()* can be zero or a negative value. If we can check whether these program states can be triggered in run time, the reliability or security of the program will be improved.

1.2 Objective

Once a security problem is modeled as a property state, the feasibility of that state can be checked by software testing technique. We implement a target directed random testing framework which combines symbolic execution and concrete run similar to DART. In addition, we introduce CFG information to avoid the testing taking execution paths which are not reachable to the target state. It can let the testing find a test case triggering the target state faster if there is such a test case.

However, there are often too many execution paths to enumerate from program entry point to target state. So we add several restrictions about the control paths between entry point and target state. In other words, our testing framework checks the feasibility of an incomplete execution path. This path contains a sequence of branch decisions. Our implementation then checks whether the specified decisions can be feasible in run time.

In our work, such an incomplete execution path is produced by CQual[7], a type based analysis tool. It can generate a data flow from input parameter to one statement with a potential security problem. The data flow can be easily transformed to an incomplete execution path. The detail will be described in Section 4.

1.3 Example

Here we give an example demonstrating how our testing framework improves directed random testing to accelerate the path searching:

```
1   void testme(int i, int j)
2   {
3           int k;
4           if(i>=0){
5                   if(i<10){
6                           k = -10;
7                   }
8                   else{
9                           k = -20;
10                  }
11          }
12          else{
13                  k = 10;
14          }
15
16          if(k>0){
17                  if(j>=0)
18                          puts("This is our target state!");
19          }
20          else{
21                  while(j<10000){
22                          j++;
23                  }
24          }
25  }
```

Figure 1: A Motivation Example

The target state is at line 18. Our testing framework will automatically search a feasible execution path from the entry to line 18. If we use DFS (Depth First Searching) path searching for full coverage, at beginning it is possible that our testing enters the loop at line 21. In this case the testing will continue searching the execution paths in the loop, which is not related to our target state at line 18. However if our path searching is directed by the target at line 18, once the execution enters a statement such as the loop at line 20, which is unreachable to the target location, it will generate next test case to take another branch right away. In this example, the next test case will execute the statement at line 17. So our testing framework can avoid exploring the execution paths which is unreachable to our target state.

2. Related Work

2.1 Static Analysis

Traditional static program analysis for bug finding can check the correctness of the source code without any concrete run. The diagnosis of static analysis is often complete. So it can often detect some tricky bugs which are hard to be found for humans. However, static analysis usually has false positive. The warnings generated by static analysis must be verified by testers whether that bug will really happen.

ESC/Java[8] is a compile-time checking tool that attempts to find errors in JML-annotated Java programs by static analysis of the program code and its formal annotations. Users can add annotations to provide some information such as pre-condition and post-condition and allow ESC/Java to verify these properties.

2.2 Dynamic Analysis

Unlike static analysis, dynamic analysis concretely runs the program and finds bugs only when it is really happened. The bug found by dynamic analysis is often sound. However, users must provide test cases for dynamic analysis to cover more program states.

STOBO (Buffer overflow detection by dynamic analysis)[9] is a dynamic buffer overflow detector. It checks the preconditions of each memory copy operation. If a precondition is conflicted, then buffer overflow will possibly happen. The drawback of STOBO is it needs users to provide test cases.

2.3 Concolic Execution

DART[2] combines concrete run and symbolic execution. It can collect the program constraints in runtime and solve it to get the test cast for next run. CUTE[3] is the follow-up work of DART. It can handle pointer operations, and get approximated pointer constraints.

EGT[4] and its following work EXE[5, 6] are similar to CUTE, but they use bit-vector constraint solver. So it can detect the behavior of arithmetic overflow, or view a memory block as untyped bytes. In addition, EXE use the memory model of CRED[10]. So it can provide full pointer constraints rather than approximation.

2.4 Model Checking

Similar to our work, given a model (a finite state machine), a model checker checks whether this model satisfies a specified property. The checker systematically explores state space, and tries to find a counter-example which violates the specification. For software model checking, the model is based on CFG (Control Flow Graph). And some property checking can be reduced to program state reachability problem. However, the reachability problem is un-decidable because of state explosion. The most detailed model for software includes the memory configuration. Obviously such a model will have state explosion problem.

Many researches on this area use abstraction to reduce the number of states. Given an abstraction function, the concrete states will be grouped and mapped to abstract states. However, if the abstraction is too coarse, the abstract model will have not enough information to model the real behavior of software about the specified property. So there is some work using the counterexample found by model checker to refine the abstract model. The abstract model will be refined until the model checker finds a counterexample which is not spurious. Or the model checker cannot find a counterexample any more. In the latter case we can conclude the specified property will never happen. Such a refinement scenario is called CEGAR (Counterexample Guided Abstraction Refinement) loop[11].

BLAST[12] implements CEGAR loop to check reachability of a specified label in the program. In BLAST the C programs are represented as CFA (Control Flow Automata), which is a CFG with operators on edges. It builds a reachability tree from CFA. Each node of the tree is labeled by a vertex of CFA and a set of predicates which are called reachable region. If we find there is a path to the specified error label in the tree, the path will be checked using symbolic execution. If the path is infeasible, the tree will be refined.

CBMC[13, 14] is a bounded model checker for ANSI C. It takes a strictly-conforming C program, and unwinds the transition relation with fix number of transitions to obtain a Boolean formula. The formula is passed to SAT solver. If the formula is satisfiable, a counterexample is extracted from the output of the SAT solver. If the formula is not satisfiable, the program can be unwound further to search a longer counterexample.

3. Concolic Execution Implementation

In this section, we describe our implementation of concolic execution testing framework in detail. “Concolic execution” mixes concrete run and symbolic execution. Similar to DART, a program performs symbolic execution in run time, so symbolic execution can easily obtain all run time information. Symbolic execution is used to convert a concrete run to a constraint system. After a concrete run is finished, the testing framework modifies the constraint system to represent another execution path which we want to cover in the next testing. Finally our testing framework will query the theorem prover with the resultant constraint system to get a test case for next run.

3.1 Testing Framework Overview

Our testing framework contains three components: CIL[15] (instrumentation tool), symbolic execution library, and CVCL[16] (a decision procedure). Before testing, we use CIL to transform a program to a relatively simplified form and instrument some codes for symbolic execution. The instrumented codes call the symbolic execution library. In addition to instrumentation, we also use CIL to compute CFG (control flow graph) information for path searching. Symbolic execution library collects the symbolic constraints during run time. It uses a store to record accumulated symbolic constraints, and also communicates with CVCL. CVCL is an automatically validity checker for first-order logic. It can check the validity of a given logic context. In our implementation, CVCL is used to be queried whether a given constraint system which represent an execution path has any solution satisfying it or not. If there is a solution, then we get the next test case.

Comparing the activating time of these three components, CIL is at static time, on the other hand, symbolic execution library and CVCL is at run time. Figure 2 shows the architecture of our testing framework:

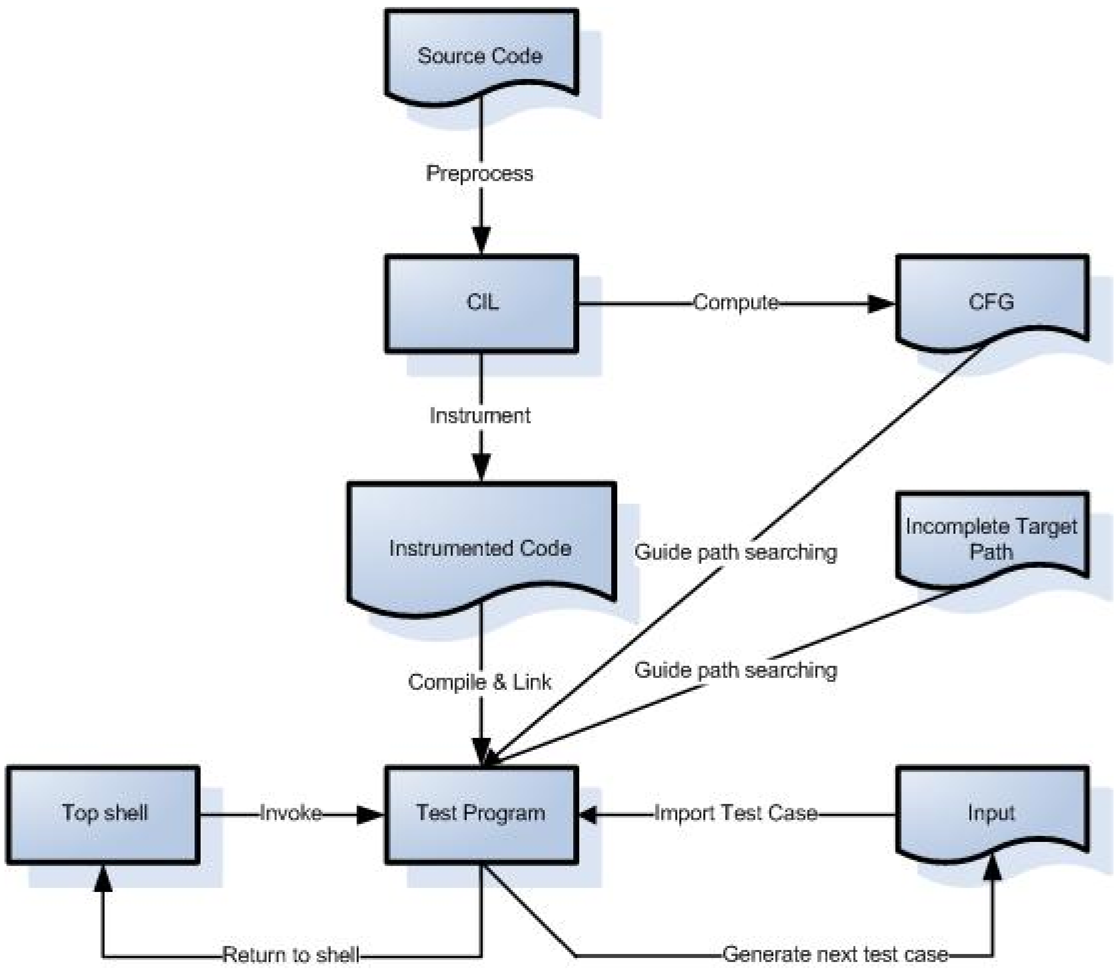

Figure 2: Testing Framework Architecture

The input to the testing framework is the file "Incomplete Target Path". This file is generated by CQual and the detail will be described in Section 4. It specifies some statements which must be executed. It can be seen as *L* in program state *(L, P)*. In addition, we can manually add some additional codes before the target location to check the constraints of the variable in target state, then specify the target path must pass along in these additional codes. It can be seen as *P* in program state *(L, P)*. By the additional path information, the testing not only tries to find an execution path to the target location, but also checks whether the variables can satisfy specified constraints.

After CIL preprocessing, we compile and link the instrumented program with our symbolic execution library and CVCL. The objective program is a “self-testing” program. In other words, it is both a tester and testee. It does symbolic execution and generates next test case by itself. As Figure 3 shows:

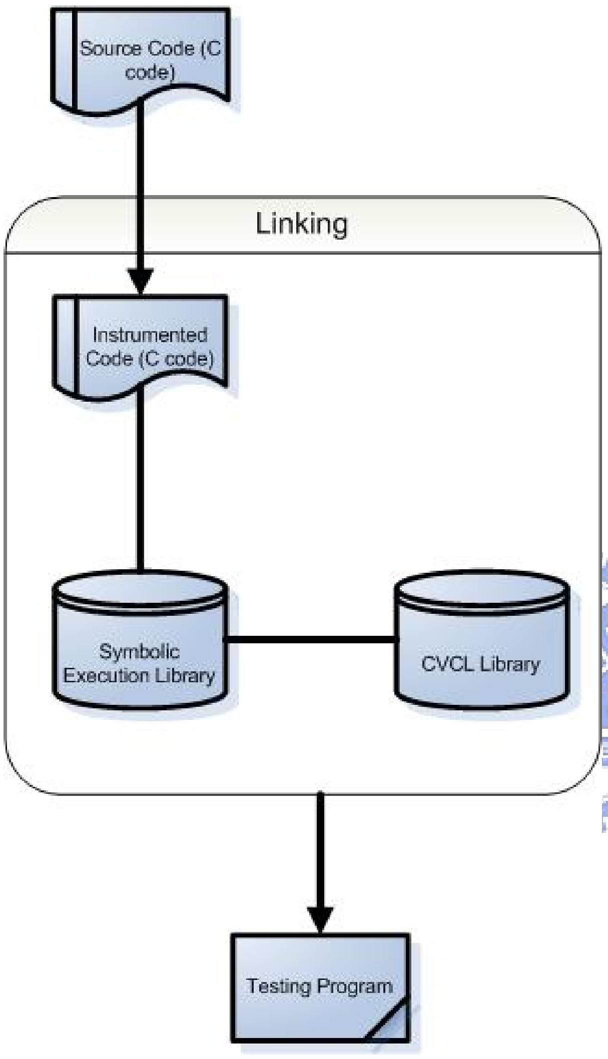

Figure 3: Tested program links with symbolic execution and CVCL library

After instrumentation, the user must supply some additional codes which are used as testing driver for initializing the testing and calling the entry function. After compiling the preprocessed code, it then links with symbolic execution library and CVCL. Finally, the testing program is standalone and executable.
The life cycle of testing program is:
Initialize and load the input which is generated by the last testing.
Execute the testing code, and accumulate the symbolic constraint system.
Tune the symbolic constraint system to represent the next execution path it wants to explore.

Solve the constraint system by CVCL to get the next test case.
There is a top shell repeatedly invoking the testing program. The testing program will be invoked many times. Each time the program terminates, it will generate a file containing input data which will be loaded as next test case for next execution, until the testing find a test case following the specified path, or the testing determines that there is no execution which can pass along this path. Instead of the above two cases, it is also possible that the top shell invokes the testing program and never stops. It may happen because of too complex constraint for CVCL to solve, or the library call losses completeness for the constraints.
The detail of Step 2 will be covered in this section. And Step 3 and 4 will be covered in Section 4.

3.2 Preprocessing

In order to instrument the program with symbolic execution library call, we use CIL[15] (C Intermediate Language) as our instrumentation tool. CIL is an OCaml application which provides source-to-source program transformation. We transform a source program by CIL in two phases. In the first phase, CIL simplifies the source program to a relatively simplified form. The semantic of simplified program is the same as the original program. In the second phase, CIL adds some codes on specific program location which performs symbolic execution. After the above two preprocessing, user must manually add a testing driver which loads test case from a file to its inputs and calls the entry function for testing. The whole preprocessing is shown in Figure 4:

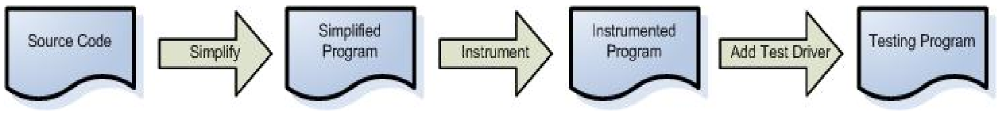

Figure 4: Preprocessing by CIL

After simplified in the first phase, the program will be instrumented with CIL. The simplified program has no complex grammar of C, with a relatively simple form.

In C, each statement has many writing styles or syntax. CIL can transform each kind of C statement to its unique syntax so that the instrumentation can just consider limited cases to decide what code should be added on some location. On the other hand, we can also easily design our symbolic execution library API, because every statement will be transformed to a unique form, and the API applied on each statement can just serve for that special form.

The simplified program has the following property:

1. All control statements, like ***if-statement*** and ***switch***, are transformed to ***if-else*** form.

2. All predicates in ***if-statement*** are transformed to a binary relation with atomic variables.

3. All loop statements are transformed to ***while(1)***, ***if***, and ***goto*** statements.

4. Reduce all complex program expressions to three address forms.

The following is an example which demonstrates how we use CIL to do the above mentioned simplification:

```
1    void testme(int i)
2    {
3            int j;
4            for(j=0;j<10;j++){
5                    i = 2*j + i;
6            }
7            if(i==10){
8                    printf("if block\n");
9            }
10           else if(i == 20){
11                   printf("else if block\n");
12           }
13           else{
14                   printf("else block\n");
15           }
16   }
```

```
1    void testme(int i )
2    { int j ;
3      int __cil_tmp3 ;
4      int __cil_tmp4 ;
5      int __cil_tmp5 ;
6
7      j = 0;
8      while (1) {
9        __cil_tmp3 = j < 10;
10       __cil_tmp5 = ! __cil_tmp3;
11       if (__cil_tmp5 != 0) {
12         goto while_0_break;
13       }
14       __cil_tmp4 = 2 * j;
15       i = __cil_tmp4 + i;
16       j ++;
17     }
18     while_0_break: ;
19     if (i == 10) {
20       printf("if block\n");
21     } else {
22       if (i == 20) {
23         printf("else if block\n");
24       } else {
25         printf("else block\n");
26       }
27     }
28     return;
29   }
```

Figure 5: Original Program and Simplified Program

The left of Figure 5 is the source code, and the right of Figure 5 is the simplified code. In this example, CIL transforms a ***loop*** and an ***if-statement*** to a fixed and simple form. In the loop of source code at line 4, CIL transforms this ***for-loop*** to a ***while-loop*** by using ***if*** and ***goto*** statement. Next, CIL transforms the ***if-statement*** with ***else-if*** block at line 7 to a form without ***else-if*** block. We can observe that the simplified code preserves the same control flow and semantics with the original source code.

3.3 Symbolic Execution

When a program executes, variables and the program counter can represent current execution state. However, symbolic execution can be seen as stateless execution. In symbolic execution, each executed statement and instruction will be transformed to a symbolic logical formula. After the symbolic execution finished, these symbolic logic formulas will be combined to a conjunction. The conjunction can be seen as the constraint of this execution path.

We call the conjunction ***C*** a symbolic constraint system. In ***C***, each expression ***e*** is built from variables, constants, and function symbols. The function symbols include plus, minus, multiply, etc. For example, "***a+3***" is an expression. Each atomic formula ***f*** is built from expressions and relation symbols. The relation symbols include equal, less than, larger than, etc. For example, "***a+3<b***" is an atomic formula. Finally each predicate ***p*** is composed of atomic formulas using negation, conjunction, and disjunction. For example, "***a+3<b*** $\wedge$ ***b=10***" is a predicate. Composed of predicates, ***C*** is a predicate built by conjunctions.

For a program ***P***, we must build a mapping from its code to symbolic constraints in order to transform an execution path to a symbolic constraint system ***C***. When the program ***P*** executes a statement, it will use this mapping to transform the current statement to a predicate in ***C***. Symbolic execution can be seen as a process to transform an execution path to a symbolic constraint system ***C***.

For ***if-statement***, let ***p*** be the predicate of its conditional expression. After simplification, the ***if-statement*** will include exactly two blocks: ***then-block*** and

else-block. If an execution path enters ***then-block***, symbolic execution adds ***p*** to its constraint system. If an execution path enters the ***else-block***, symbolic execution adds $\neg \boldsymbol{p}$ to its constraint system.

For an assignment statement, such as "***a = b + c;***", after simplification with the form of "***l-var = r-expression;***", symbolic execution just adds "***l-val is equal to r-expression***" to its constraint system. Unfortunately, symbolic execution is stateless and therefore after assignment the ***l-val*** is no longer the original one in symbolic constraint system. We thus need to rename the ***l-val*** after an assignment. We will refer this ***l-val*** to its new name in the subsequent symbolic execution. The name of each variable in ***C*** is the name of variable in program ***P*** concatenated with a reference count. For example, consider the following code:

```
if(x == y){
        x = z;
        if(x<0)
                x = -x;
}
```

Figure 6: An example for symbolic execution

Assume an execution path takes both ***then***-branches of the two ***if-statements***. After symbolic execution, the symbolic constraint system will be:

$$(x0 = y0) \wedge (x1 = z0) \wedge (x1 < 0) \wedge (x2 = -x1)$$

Note that after the variable ***x*** being assigned, its name in symbolic constraint system differs from its original name.

For procedure invocation and returning, we expand symbolic execution of the calling statement by symbolically executing the statements contained in the called procedure. Just like the effect of assignment, when doing inter-procedure symbolic execution, we must rename the variable in symbolic constraint system. If the name of a local variable in ***proc1*** is the same as a local variable in ***proc2***, we must make these variable names be distinct in constraint system ***C***. In principle, let each local variable name in each call frame be unique. The detail of inter-procedure symbolic execution can be referred to Section 3.4.

An execution path can be seen as a sequence of ***if-statements***, each with a sequence of instructions. It can be transformed to a conjunction of predicates by symbolic execution, where each predicate is corresponding to each ***if-statement*** or instruction along this execution path.

3.4 Combination of symbolic execution and concrete run

In this section, we describe the detail of how the testing program performs symbolic execution during runtime. And we will describe how the symbolic execution library collects symbolic constraints for each kind of statements in C programming language.

3.4.1 Concolic Execution

While testing program is invoked, its execution will pass through four phases as described in Section 3.1. The testing program initially loads the test case from the "Input" file. This file specifies all input data the testing program reads. The order of each data in the input file is the same as the order of these data been accessed in the testing program. If the testing program is invoked in first time, the input file is empty and the testing program will generate random value for each input data. Otherwise, the input file will contains input values which are determined by last invocation of testing program.

After the input data is read, the testing program starts to enter the entry testing function. While the program executes a branch node, in simplified program which is a ***if-statement***, it will call symbolic execution library ***check_branch()*** to check whether the current taken branch diverges from the target path. If it diverges, it terminates current execution and generates next test case to avoid this divergence. The detail of path searching algorithm will be described in Section 4.

When the testing program executes a statement, it will call a symbolic execution library ***addPredicate()*** for ***if-statement*** or ***symbolicExec()*** for other statements. Recalled in Section 3.1, these symbolic execution library calls are instrumented by CIL. Following is the skeleton of instrumented program:

```
instrumented_program(parameters)
{
        if(p){
                addPredicate(bID, 1, p);
                ….
                a = b + 10;
                symbolicExec(a, PLUS, b, 10);
        }
        else{
                addPredicate(bID, 0, ¬p);
                …
        }
}
```

Figure 7: Skeleton of instrumented program

Both ***addPredicate()*** and ***symbolicExec()*** are called to perform symbolic execution, except that ***addPredicate()*** also maintains a data structure which records the branch history taken by current execution. The branch history is used for path searching which will be described in Section 4. Figure 8 is the algorithm of symbolic execution in ***symbolicExec()*** and ***addPredicate()***.

```
symExec(lhs, operator, op1, op2)
    if( isSymbolic(op1) || isSymbolic(op2) ){
        setSymbolic(lhs)
        c ← GenerateConstraint(lhs, operator, op1, op2)
        addConstraint(c)
    }
else{
    setConcrete(lhs)
}
```

Figure 8: Algorithm of *symExec()*

This algorithm has four inputs: ***lhs***, ***operator***, ***op1***, ***op2***. These inputs represent a statement: "***lhs = op1 operator op2;*** ". If there is no ***op2*** in the statement, in other words, ***operator*** is unary, ***op2*** would be empty. Each of ***lhs***, ***op1*** and ***op2*** has their variable name ***n***, type ***t***, and concrete value ***v***. Given ***n*** and ***t***, we can define or find its corresponding variable in symbolic constraint system by the naming rule described in Section 3.2. Concrete value ***v*** is used when this variable is not affected by input, i.e., not symbolic.

For each variable, we maintain a flag recording whether it is a symbolic variable. Each symbolic variable in the concrete program has a corresponding variable in constraint system. Once a variable becomes symbolic by ***setSymbolic()***, CVCL then creates a new variable corresponding to that variable in its logical context. The naming rule of this new CVCL variable follows the rule described in Section 3.3. On the other hand, when a variable is not symbolic it will be replaced by its concrete value in symbolic constraint system.

Initially, all input variables are symbolic. CVCL creates corresponding variables in its context. The symbolic property of these input variables then propagates to other variables by assignment. As above algorithm shows, when a variable is assigned by an expression, we will check whether there is any symbolic variable included in the expression. If this expression is symbolic, we set the variable ***lhs*** to be symbolic, and perform symbolic execution for this statement. Otherwise we set ***lhs*** to be non-symbolic (concrete) and do nothing. In other words, we just execute the statement.

At line 4 of Figure 8 we call ***GenerateConstraint()*** to produce a symbolic constraint ***c***. ***GenerateConstraint()*** will check whether ***op1*** or ***op2*** is symbolic. If one of them is not symbolic, we will place this variable's concrete value on the constraint ***c***. If not, we will place its corresponding CVCL variable on the constraint ***c***. For example, consider the following statements:

x = 10;   z = x + y;

Assume ***y*** is symbolic and ***x*** is obviously concrete because of the assignment with a concrete value. The generated constraint ***c*** for the second statement will be:

z1 = 10 + y1

At line 5 of Figure 8, we add ***c*** to our constraint store. Then ***c*** is parsed by our constraint solver. Note that ***GenerateConstraint()*** can produce each kind of constraint for various operators in C.

3.4.2 Bit-Vector based constraints

We hope the constraints generated by symbolic execution can exactly interpret the executed statement. Fortunately, our constraint solver CVCL can validate bit-vector variables. In our constraint system, each constraint is composed of bit-vector variables and bit-vector operators so that we can use bit-vector variables to simulate real program behavior better. For example, consider the following program:

```
1    if(i>10){
2         i = i + 10;
3         if(i>=0)
4              malloc(i*sizeof(int));
5    }
```

Figure 9: An example demonstrating integer overflow

Assume the variable ***i*** is symbolic. If there is an execution with ***i*** greater than 10 at line 1, generally ***i*** will be naturally assumed to be a positive integer. So the programmer will expect the execution to execute line 3 and 4, and use ***malloc()*** with a positive integer parameter. Is it possible there is an execution path with <1,2,3,5>? If we use bit-vector constraint in our constraint system and assume the size of ***i*** is 32 bit, we can query following constraint:

```
BVGT(i1, 0bin00000000000000000000000000001010)     and
(i2 = BVPLUS(32, i1, 0bin00000000000000000000000000001010))  and
SBVLT(i2, 0bin00000000000000000000000000000000)
```

Figure 10: Bit-vector constraint system for path <1,2,3,5> in Figure 9

The above constraint represents the execution path of <1,2,3,5>. This path has ***i*** greater than 10 at line 1, but it is a negative value at line 3. After above constraint being solved by CVCL, we get following solution:

i1 = 0bin01111111111111111111111111111111

If let ***i*** be 2147483647 (the largest integer of 32-bit), after the addition at line 2, ***i*** will be overflowed and has a negative value. Note this kind of behavior can only be simulated by bit-vector constraint system.

Because real computer uses bit-level arithmetic, our bit-level constraint system can model the program behavior more precisely. Fortunately, CVCL can support many bit-vector operators, and each can easily be used to simulate some corresponding operations in C programs. Besides basic bit-vector addition, minus, multiplication, 2’s complement, and bit-wise operator including right shift and left shift, it also support bit-wise extraction and concatenation operator. These two operators can be used to simulate the operation of extracting a portion of bytes on a data structure. Or it can be used in symbolic pointer dereferencing as in EXE[5].

In the next section we will discuss how we handle each special operation in C where there is no direct transformation from it to symbolic constraints.

3.4.3 Divide and Modulo

Consider the following C statement:

a = b / c;

Because CVCL currently does not support division operator, in order to handle this operator instead of using bit-wise shifting, we use some trick to replace the division operator with multiplication operator. In other words, we translate this equation into:

$$\boldsymbol{a * c = b - r} \qquad \textbf{(1)}$$

Here r is an arbitrary symbolic variable and the only constraint for r is:

$$\boldsymbol{0 \leqq r < c} \qquad \textbf{(2)}$$

Note that the constraints added to symbolic constraint system contain no division operator. CVCL can correctly parse and solve this constraint.

Consider the case which ***c*** is larger than ***b***. Instead of using equation (1) as our constraint, we use the following constraint:

$$(\boldsymbol{a} = \boldsymbol{0}) \wedge$$

$$(\boldsymbol{r} = \boldsymbol{b})$$

Since we have the remainder variable in above equations, we can also handle modulo operator in the similar manner with division.

3.4.4 Casting

Since CVCL supports bit-vector variable and bit-wise operation, we can easily perform symbolic execution on casting operation. In C programming language, casting operation can be classified to explicit casting and implicit casting. However, after simplification by CIL, all implicit casting will be transformed to explicit casting. Additionally, the casting operation only happens in following form:

a = (type) b;

In other words, we can treat casting as an independent operator. It does not appear together with other operators like plus and minus. The fact of one statement containing no more than one operator eases the design of ***symbolicExec()*** API. For example, the left hand side of Figure 11 will be transferred to the right hand side of Figure 11:

```
int i;
char j;
long k = i + j;
```

```
int i ;
char j;
long k;
int __cil_tmp1;
int __cil_tmp2;

__cil_tmp1 = (int) j;
__cil_tmp2 = i + __cil_tmp1;
k = (long) __cil_tmp2;
```

Figure 11: The source code and simplified code for casing operations

Because we only support integer type in our symbolic constraint system, different variable types are distinguished by their size. No matter its type is signed or unsigned, each variable in the constraint system is represented by a bit-vector with associated number of bits, which is determined by its size of type. For example, assume the ***int*** type has 32-bits, any variable of ***int*** type will be represented by a bit-vector of 32 bits in constraint system.

In ***symbolicExec()***, we classify casting to up casting and down casting. Up casting casts a bit-vector to another bit-vector of superior size. On the contrary, down casting casts a bit-vector to another bit-vector of smaller size. Figure 12 is the algorithm for performing symbolic execution on casting operations:

```
1   Cast (var, orgType, castType)
2       if(orgLen > castLen)
3           return EXTRACT(var, castLen-1, 0)
4       else if(orgLen < castLen){
5           if(isSignedType(orgType))
6               return SX(var, castLen)
7           else
8               return EXTENSION(var, castLen)
9       }
10      else
11          return var
```

Figure 12: Algorithm of Symbolic Casting

For down-casting (line 2), we only extract the lower bits of the casted bit-vector. For up casting (line 3), we consider whether the type of casted variable is signed or unsigned. If it is a signed (line 5), we do sign extension on the casted bit-vector. If it is unsigned (line 7), we do unsigned extension (extended by all 0) on the casted bit-vector. Here is an example of casting operation:

```
int i;
char c = i;
unsigned short j = c;
int k = j;
```

Figure 13: Example Program for Symbolic Casting

After symbolic execution, following constraints are generated:

$$(c1 = (i1)[7:0]) \land$$

$$(j1 = SX(c1,16)) \land$$

$$(k1 = BVPLUS(32, j1, 0bin00000000))$$

The first constraint is got by extracting lower 8 bits in bit-vector ***i1***, and let ***c1*** be equal to the extracted bit-vector. The second constraint let ***j1*** be equal to sign extension of ***c1***. The third constraint let ***k1*** be equal to unsigned extension of ***j1***. Above symbolic execution of casting operations can exactly simulate the real behavior of integer casting operators in ANSI C.

3.4.5 Inter-Procedure

The symbolic execution for inter-procedure needs some trick to enable the symbolic constraints to propagate between procedures without variable naming collision in the constraint system. In other words, we need a mechanism to pass the symbolic parameter from caller to callee, and pass the symbolic return value from callee back to caller. Our solution is to maintain a stack, and pushes the symbolic parameters to this stack when calling a procedure. When one execution enters the called procedure, it pops all the symbolic variables in the stack. And lets the symbolic parameters of the called procedure be equal to the popped symbolic variables. For example, consider the following piece of code:

```
int called_proc(int m, int n)
{
        return m+n;
}
…..
int r = called_proc(i,j);
…..
```

```
int called_proc(int m, int n)
{
        _popParameter (“n”, T_INT);
        _popParameter (“m”, T_INT);
        temp = m + n;
        …
        _pushReturnValue(“temp”,temp
,T_INT,);
        return temp;
}
…
_clearStack();
_pushParameter(“i”, i, T_INT);
_pushParameter(“j”, j, T_INT);
r = called_proc(i, j);
_popReturnValue(“tmp”, T_INT);
…
```

Figure 14: Symbolic parameter and return value propagation

When calling a procedure, the caller will first clear the stack by ***_clearStack()***, then push all its parameters into the stack by ***_pushParameter()***. The three parameters of ***_push()*** is the variable name ***n***, type ***t***, and concrete value ***v***. These information is used by CVCL for constructing constraints. In the called procedure ***called_proc()***, it will pop the symbolic variables stored in the stack and let each popped variable be equal to the corresponding local variable in the constraint system by ***_popParameter()***. When the called procedure returns to the caller, it pushes the return variable into the stack by ***_pushReturnValue()***. After returning from callee, caller pops the variable stored in the stack and let it be equal to a variable in the constraint system by ***_popReturnValue()***. After executing above operations, symbolic execution produces the following constraints for symbolic propagation of parameter and return variable:

```
(n1 = j1)  ∧   (m1 = i1) ∧
(temp1 = BVPLUS(32, m1, n1))   ∧
(__returnVal1 = temp1) ∧
(tmp1 = __returnVal1)   ∧
(r1 = tmp1)
```

Figure 15: The generated constraint system of the program in Figure 14

Sometimes we have no source code of callee. For this condition, when we returned from callee, we must check whether the callee is instrumented or not. If it is instrumented, let ***_popReturnValue()*** act as above described. If it is un-instrumented, the return value can be thought as a constant value in symbolic constraint system because we cannot perform symbolic execution in that procedure. Hence the variable assigned by the return value becomes concrete (non-symbolic).

On the other hand, it is possible that a local variable's name is the same as another local variable's name in another procedure. Because our symbolic constraint system has no call frame, each local variable in each call frame must be different and hence have different variable name in constraint system. In other words, we use naming to make an abstract call frame in symbolic execution. We therefore introduce a flag called context number to distinguish variables in different call frames. Context number is a global variable. Initially, it is set to zero. When an execution enters a procedure, context number is increased by one. On the other hands, when an execution leaves a procedure, the context number is decreased by one. We use context number to rename the variable in symbolic constraint system. So each variable name in symbolic constraint system has the following format:

variableName	_ContextNumber	_ReferenceCount

We first rename the variable by the context number. Each "vairableNume+ContextNumber" has its own reference count introduced in Section

3.3. And it also has a flag recording whether it is symbolic. Note that this new naming rule will only be applied on local variables. Consider following program:

```
int A(int i)
{       return B(i);      }
int B(int i)
{       return i;         }
int C(int i)
{       return i;         }

void testme(int i)
{
        A(i);
        C(i);
}
```

Figure 16: Inter-Procedure Program

All the four procedures have a local variable ***i***. After symbolic execution, the generated constraints are:

$$
\begin{array}{l}
(i\_2\_1 = i\_1\_1) \wedge \\
(i\_3\_1 = i\_2\_1) \wedge \\
(\_\_returnVal\_3\_1 = i\_3\_1) \wedge \\
(\_\_returnVal\_2\_1 = \_\_returnVal\_3\_1) \wedge \\
(i\_2\_2 = i\_1\_1) \wedge \\
(\_\_returnVal\_2\_2 = i\_2\_2)
\end{array}
$$

Figure 17: Generated constraints for the program in Figure 16

We can observe above constraint system to understand how the context number naming rule distinguishes each ***i*** of different call frame in symbolic constraint system.

Note that at line 10 and line 11 of Figure 16, the context number in ***A()*** is equal to the context number in ***C()***. In this condition symbolic constraint system cannot distinguish variables in different call frame with the same context number. Following is our solution to this problem: When a procedure finished, we set all local variable to be concrete (non-symbolic). This is just like when a program leaves a procedure all local variables will be dropped and never be used. Because a non-symbolic variable will be replaced with its concrete value during symbolic execution, setting all local variables to be concrete will make them never appear again in the constraint system.

Note that the above described trick for inter-procedure symbolic execution can also be applied on recursive procedure invocation.

4. Target Directed Random Testing

In Section 3 we have described how our testing framework combines concrete execution with symbolic execution. In this section we describe how our testing framework triggers the target program state. We will discuss how to transform a target program state to a target path by CQual, and how our testing framework finds a test case which can pass along the target path.

4.1 Incomplete Target Execution Path

Our target state must be specified as: ***one*** variable satisfies some constraints in a given program location. For example, we can specify the parameter of some ***malloc()*** must be greater than zero. We first use CQual to check such a state. CQual is a type-based tool for finding potential bugs in C programs. CQual extends the type system of C with extra user defined type qualifiers. We use two new types, ***tainted*** and ***untainted***. The ***untainted*** type is the subtype of ***tainted*** type. Let all input data be ***tainted***, and the variable of the target state be ***untainted***. CQual can try to find out a data flow which propagates the tainted data to the untainted variable. Consider the following piece of program:

```
fread($tainted buffer)
int n = buffer[0];
malloc($untainted n);
```

Figure 18: An example program with tainted and untainted type

CQual will report the following data flow:

```
Preclude.cq:41      $tainted <= *fread_arg1
test.c:1                     <= buffer[]
test.c:2                     <= malloc_arg1
preclude.cq:33               <= $untainted
```

Figure 19: The data flow generated by Cqual for the example in Figure 18

Above reported data flow shows that the parameter of the ***malloc()*** can be affected by the input data ***buffer***. However, the reported data flow has false positive because CQual is a static analysis tool. We therefore use our testing framework to verify whether the reported data flow is really feasible in run time.

The data flow can be transformed to an incomplete execution path. The incomplete execution path specifies a list of branches the data flow should take. We call each branch a path node. Between each path node, it maybe contains some un-specified execution path. Our testing framework tries to find a test case which passes along all specified path nodes.

4.2 Path Searching Algorithm

After one run of concolic execution described in Section 3, it will generate a symbolic constraint system which can represent current execution path. The testing program can easily modify this constraint system to represent another execution path. Then let CVCL solve it and get a test case passing along this execution path. Here we have an algorithm which searches feasible paths in the execution tree. Consider the execution tree of Figure 20:

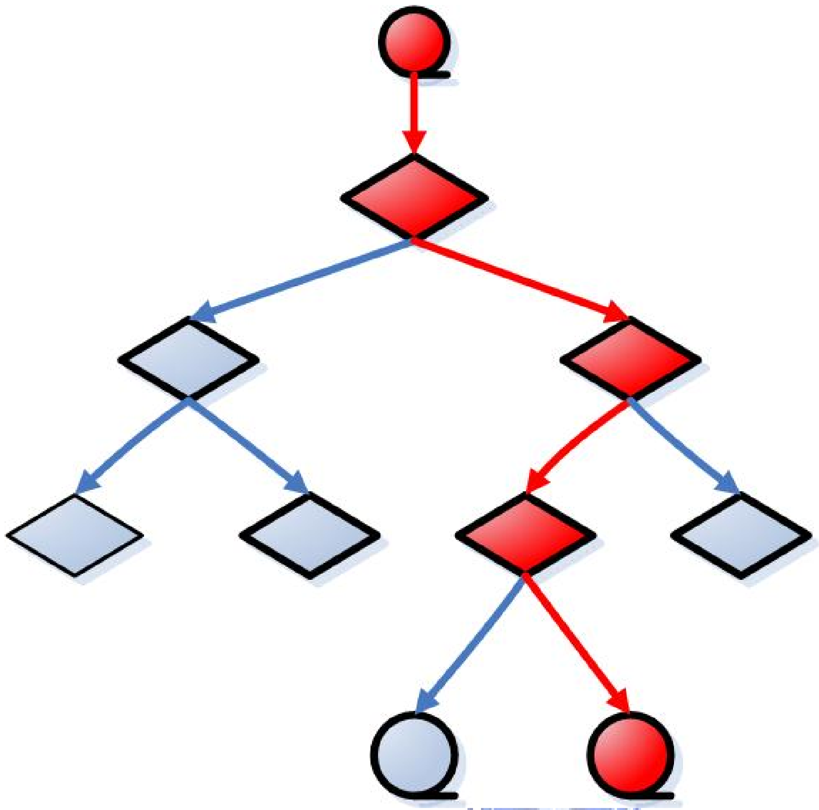
Figure 20: Execution Tree of a program

Each diamond in above diagram is a branch node, or ***if-statement***. Each circle is other statement which has only one control path to next statement. Each red diamond is the specified path node in the target path. Note that between each red node, there may be some unspecified branches. Our path searching algorithm will try to find a test case which execution path passes along from root to these red nodes.

If we use DFS path searching, each time searching next execution path we can easily negate the last added constraint in constraint system. However, DFS path searching causes our testing to be stuck with some location, for example, the loop statement. So we introduce CFG (Control Flow Graph) to guide our path searching. CFG is statically produced by CIL, it represents the control flow of a program. Each time a program enters an ***if-statement***, it will ask CFG whether current execution is impossible to reach next target path node. If so, it terminates its execution and negates the constraint of last taken branch. Then it queries CVCL to get next test case. Path searching approaches target path node faster, because we eliminate many execution paths which are statically non-reachable to the target path node.

Now we describe our path searching algorithm in detail. Before execution, the testing program will initialize some data structure:

```
cfGraph ← loadCFGandComputeReachability()
path_record, path_len ← target_path
branch_hist, hist_len ← current_path
depth ← 0
isComplete ← 0
parameters ← input()
setSymbolic(parameters)
```

Figure 21: Initialization of some data structure before testing

cfGraph contains the CFG generated by CIL and all-pair reachability. ***path_record*** records the path nodes specified by the file "Incomplete Target Path". This data structure also records whether a prefix of these specified nodes have been explored. ***branch_hist*** records all symbolic branch nodes that should be taken. It is specified by last execution of the testing program. ***depth*** indicates the number of symbolic branches that the current execution takes. ***isComplete*** indicates whether the path searching has covered all execution paths which are reachable to the target path in CFG. ***Input()*** will read the test case generated by last execution. Finally, it will set all input data be symbolic.

When an execution takes a branch of ***if-statement***, it will call the symbolic execution library ***addPredicate()*** as described in Section 3.4. In ***addPredice()***, it will use the following ***check_path()*** algorithm to check whether the taken branch is unreachable to the target path node in CFG:

```
1   check_path( branchID, branch, isBranchSymbolic)
2       checkWhetherDone(branchID, target_path)
3       if(depth<hist_len){
4           if(branch_hist[depth].branch != branch)
5               //Failed caused by incomplete constraints
6               exit(1)
7           else if(depth = hist_len -1)
8               branch_hist[depth].done ← true
9       }
10      else{
11          if(isBranchSymbolic)
12              recordBranch(branch_hist)
13          nextNode ← cfGraph[branchID].next(branch)
14          nextTarget ← target_path
15          if(!isReachable(cfGraph, nextNode, nextTarget)){
16              if(isBranchSymbolic) depth++
17              solveConstraints()
18              if(!isComplete) exit(0) //There is another path not been explored
19              else exit(1)        //The target path is infeasible
20          }
21      }
```

Figure 22: Algorithm of check_path(), used in addPredice()

isBranchSymbolic is used to specify whether current branch node is symbolic. At line 2, ***checkWhetherDone()*** checks whether the current execution has passed along all path nodes in the specified target path. If so, then our testing finishes. At line 3, if the current symbolic branch depth is less than the length of ***branch_hist***, the taken branch should follow current ***branch_hist***. At line 10, if the depth is larger than the length of ***branch_hist***, in other words, if there is no specified branch in ***branch_hist***, it adds current taken branch to ***branch_hist***, then uses CFG to check whether the next executing statement is not reachable to first unvisited path node in target path. If it is not reachable, it then calls ***solveConstraints()*** to tune current constraint system and queries CVCL to get next test case. After solving, if the flag ***isComplete*** is set, then we can claim that all the execution paths which are reachable in CFG for the target path are explored. We can conclude that the given target path is infeasible. Otherwise, the testing framework will continue to invoke the testing program. Next we describe the algorithm of ***solveConstraints()***:

```
solveConstraints
    j ← depth – 1
    while(j>=0){
        if(branch_hist[j].done = false) {
            branch_hist[j].branch ← !branch_hist[j].branch
            negateLastConstraints()
            r ← queryCVCL()
            if(r){
                writeSolutionToInputFile()
                return
            }
            else{
    //This path is infeasible, back trace to parent branch node
                j ← j – 1
                popConstraints()
            }
        }
        else{
//All descents are fully traversed, back trace to parent branch node
            j ← j - 1
            popConstraints()
        }
    }
    if(j<0) isComplete ← true
    return
```

Figure 23: Algorithm of solveConstraints(), used in check_path()

The while loop in ***solveConstraints()*** repeatedly pops out constraints from symbolic constraint system by ***popConstraints()***. It pops out the constraints generated by last if-statement. When all constraints are popped out, the while loop terminates. In this case, the flag ***isComplete*** will be set. In the while loop, it negates the last constraint in the constraint system and queries CVCL. If CVCL can resolve a solution for current constraint system, it then records the solution to the file “Input” for next test case. If CVCL determines there is no solution on current constraint system, it then calls ***popConstraints()*** and back traces to parent branch node. On the other hand, at line 17 the flag ***done*** of the branch node is set. It shows that all descent CFG nodes have been explored and will back trace to the parent CFG node such as line 13.

Note our path searching algorithm combines DFS searching with CFG guiding. DFS searching can automatically try to find out an execution path passing along the target path. On the other hand, CFG guiding can improve searching more effectively by only searching paths which are reachable in CFG to the target state.

1. Experimental Results

To evaluate the functionality of our tool, we test with Antiword[17]. Antiword is a free MS Word reader for Linux which converts the binary files from Word to plain text and to PostScript. We test the Antiword-0.37, which is the latest version.

First we use CQual to analyze Antiword. As described in Section 4, we set the buffer of ***fread()*** be tainted, and the parameter of ***malloc()*** be untainted. Following is one of CQual reported data flow:

```
preclude.cq:41       $tainted  <=  *fread_arg1
misc.c:193                     <=  *aucBytes
wordwin.c:182                  <=  aucHeader[]
wordwin.c:200                  <=  *aucHeader
properties.c:119               <=  *aucHeader
stylesheet.c:615               <=  cast
stylesheet.c:615               <=  cast
stylesheet.c:615               <=  tStshInfoLen
stylesheet.c:636               <=  xmalloc_arg1
preclude.cq:105                <=  $untainted
```

Figure 24: The tainted data flow generated by CQual

We take ***vGet8Stylesheet()*** as our entry function. And we manually add the following codes just before ***xmalloc()*** at line 636 of ***stylesheet.c***:

```
int xmalloc_arg1 = (int)tStshInfoLen;

if(xmalloc_arg1<0) printf("Target state is triggered!\n");
```

The additional codes are used for checking whether the parameter of ***xmalloc()*** is less than zero. We then set the file "TargetPath" according to the dataflow in Figure

24. The first run executes following execution path in ***vGet8Stylesheet()***:

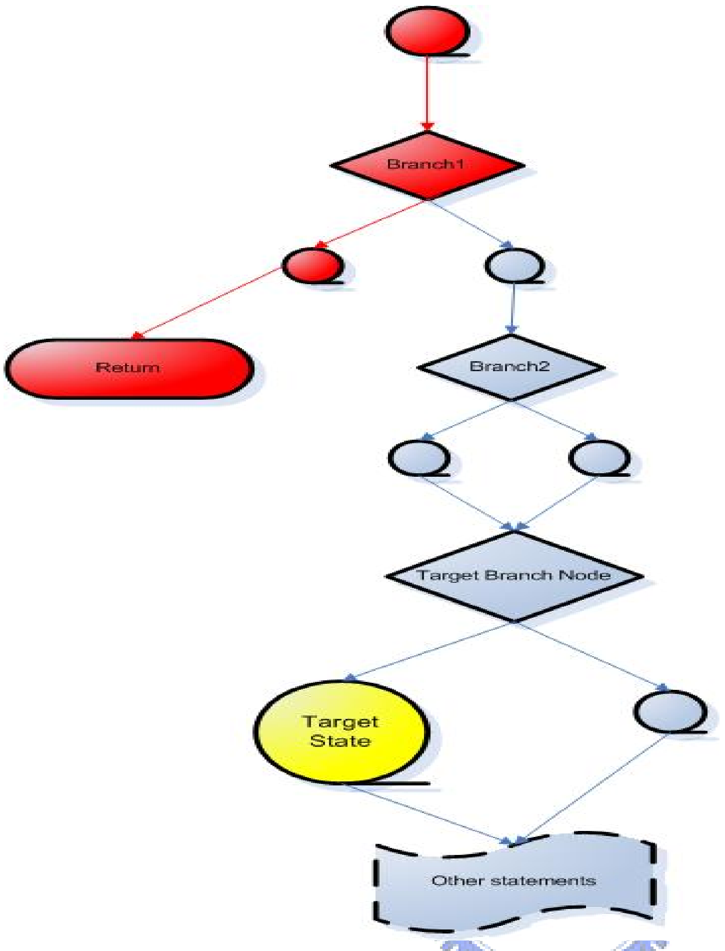

Figure 25: Execution path of testing run 1

After negating ***branch1***, the second run executes following path:

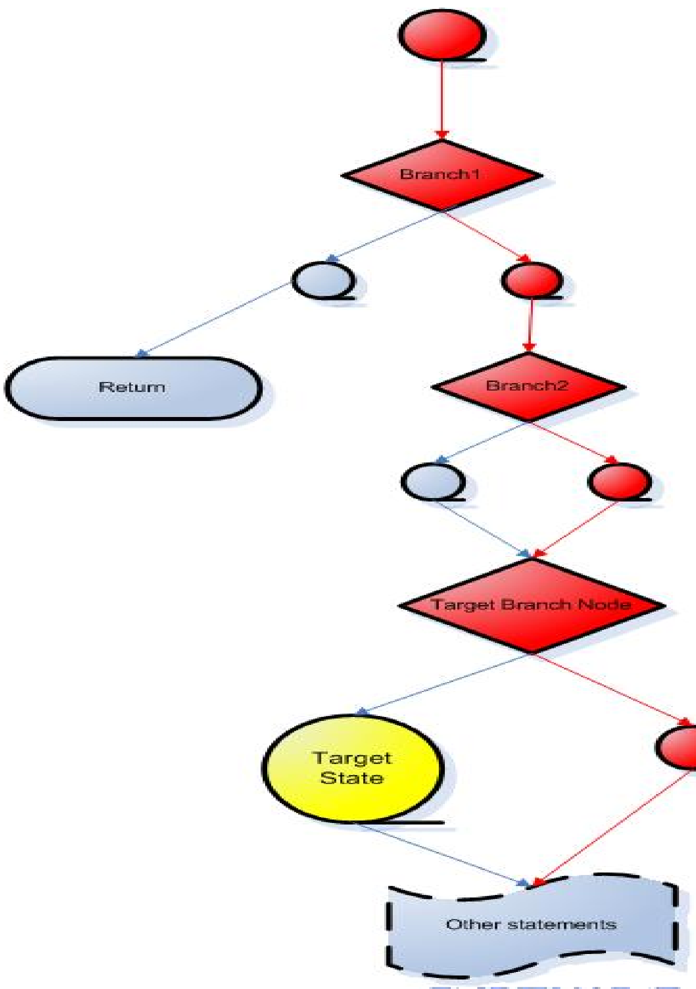

Figure 26: Execution path of testing run 2

This time the execution goes through the other branch instead of reaching the target state. Before it enters the other statements below the target state, CFG guides the path searching to translate this execution and negates the target branch node. The third run executes the following path:

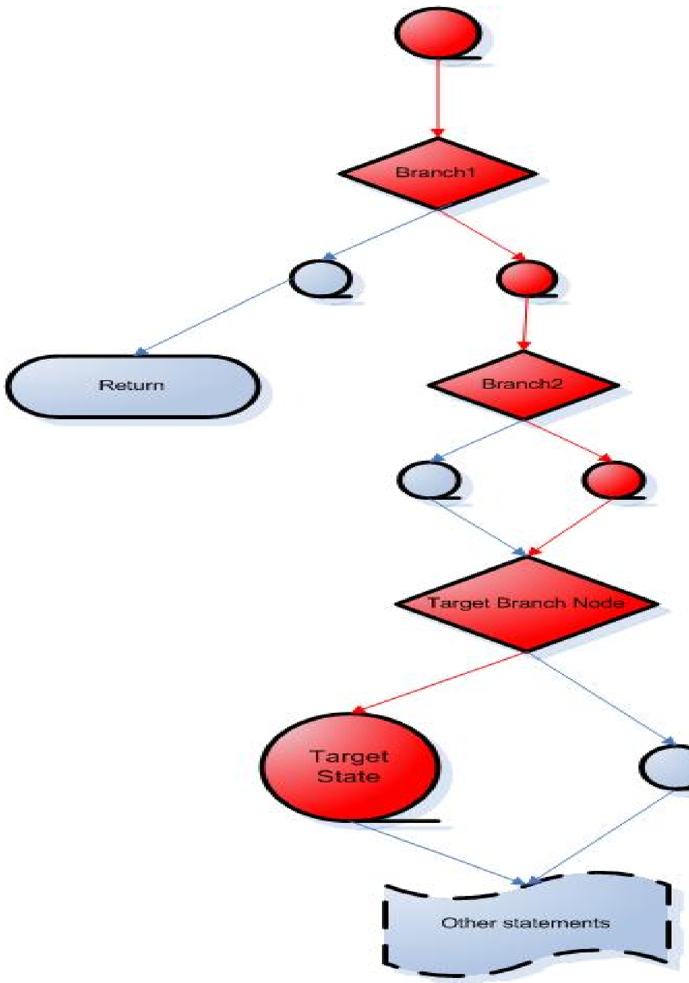

Figure 27: Execution path of testing run 3

In third run, the execution reaches the target state. Note that CFG can prevent our testing from searching the execution paths below the target state.

In this experiment, all buffers read from a file are symbolic. Because our testing framework cannot handle array of symbolic size, let all buffer read by ***fread()*** with a fixed size. After our testing framework generates three test cases, it finds an input to ***vGet8Stylesheet()*** which causes the parameter of ***xmalloc()*** at line 636 of ***stylesheet.c*** to be less than zero.

In other experiment, we check whether the parameter of ***malloc()*** at line 544 of ***summary.c*** can be less than zero. This time we take ***vSet2SummaryInfo()*** as our entry function. After our testing framework generates 24 test cases, it concludes there is no execution to make the parameter be less than zero.

Table 1 compares another concolic testing tool CUTE with our testing framework:

Table 1: Comparison between CUTE and our testing framework

	Objective	**Constraint System**	**Constraint Solving**	**Path Searching**
CUTE	Full Coverage	Byte based	Linear Constraint	DFS(Depth First Searching)
Our testing framework	Feasible state generation	Bit-Vector based	Linear & Non-Linear (Partially)	DFS+CFG guiding

Because the objective of CUTE is full coverage, they simply use DFS to generate test cases covering all execution paths in execution tree. However, our objective is feasible state generation. We use DFS to find “feasible” execution path to our target state. In addition, since there is a target node in execution tree, we use CFG to guide DFS to let the path searching more effectively. On the other hands, because of the ability of CVCL, our testing framework can handle some bit-wise operator which CUTE does not support in its constraint system. In addition, our constraint system also supports some non-linear constraints. However, not all non-linear constraint can be resolve by CVCL. Sometimes CVCL is stuck with a complex non-linear constraint.

2. Conclusions

We implement a testing framework combining concrete run with symbolic execution. We, therefore, have both the logic reasoning capability of static analysis and soundness of concrete run. Such a testing framework can be used to verify the diagnosis reported by static analysis tool. In our work, we use CQual to generate a data flow with potential specification violation. Then our testing framework can verify whether there is a test case leading the data flow to violate specification. In addition, we use CFG information to guide our testing to the target program state. This guiding improves the path searching for the target state.

Currently there are some program behaviors our testing framework cannot handle, especially for handling various kinds of input source. Since files are the most common input source in real program, file input operation is the most important obstacle for us to test a whole real program. In future, we will maintain a symbolic buffer for each file. We will also wrap each file input operation with our specific symbolic execution API to simulate file input operation by reading the symbolic buffer. Finally we will use this buffer to rebuild the input file.

References

[1] A. Gotlieb and M. Petit, "Path-oriented random testing," in *RT '06: Proceedings of the 1st International Workshop on Random Testing,* 2006, pp. 28-35.

[2] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed automated random

testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213-223.

[3] K. Sen, D. Marinov and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 263-272.

[4] C. Cadar and D. Engler, "Execution Generated Test Cases," 2005.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, "EXE: Automatically generating inputs of death," in *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006, pp. 322-335.

[6] J. Yang, C. Sar, P. Twohey, C. Cadar and D. Engler, "Automatically generating malicious disks using symbolic execution," in *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S\&P'06)*, 2006, pp. 243-257.

[7] J. S. Foster, T. Terauchi and A. Aiken, "Flow-sensitive type qualifiers," in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002, pp. 1-12.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, "Extended static checking for java," in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002, pp. 234-245.

[9] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," 2003

[10] D. Avots, M. Dalton, V. B. Livshits and M. S. Lam, "Improving software security with a C pointer analysis," in *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 332-341.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, pp. 752-794, 2003.

[12] D. Beyer, A. J. Chlipala and R. Majumdar, "Generating tests from counterexamples," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 326-335.

[13] E. Clarke, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005); Lecture Notes in Computer Science,* 2005, pp. 570-574.

[14] E. Clarke, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004); Lecture Notes in Computer Science,* 2004, pp. 168-176.

[15] G. C. Necula, S. McPeak, S. P. Rahul and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Computational Complexity,* 2002, pp. 213-228.

[16] C. Barrett and S. Berezin, "CVC lite: A new implementation of the cooperating validity checker," in *Proceedings of the International Conference on Computer Aided Verification (CAV '04); Lecture Notes in Computer Science,* 2004, pp. 515-518.

[17] "Antiword: a free MS word document reader" http://www.winfield.demon.nl/