Inter-procedural Analysis of Concolic Execution

A Thesis Submitted

in Partial Fulfilment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

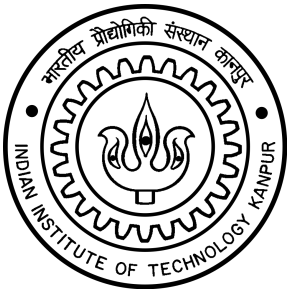
by

Ashwini Kshitij

supervised by

Dr. Subhajit Roy

Dr. Amey Karkare



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June 2015

CERTIFICATE

This is to certify that the work contained in this thesis entitled ***"Inter-procedural analysis of Concolic Execution"***, by **Ashwini Kshitij (Roll No. 10327165)**, has been carried out under my supervision and this work has not been submitted elsewhere for a degree.

Dr. Subhajit Roy
Assistant Professor,
Department of CSE,
IIT Kanpur.

Dr. Amey Karkare
Assistant Professor,
Department of CSE,
IIT Kanpur.

ABSTRACT

Interprocedural analysis is the cornerstone of determining precise program behavioral information. Using this technique we can avoid making overly conversative assumptions about the effects of procedures and the state at call sites. It aims at gathering informations across multiple procedures.

In this thesis, to extend the concept of concolic execution to interprocedural calls we instrument the function call site. For such instrumentation purposes we have used a tool called CIL. Just before the function call we set up the environment for calling the procedure which in turn enables it to follow through the concolic execution of parameters with the control flow from caller to callee. Similarly, just after the call instruction we restore the calling environment by restoring the symbolic values.

Modifying the symbolic execution engine to collect the interprocedural analysis information can have widepread applications in software verification and testing. We can improve the coverage of test suites that are automatically generated by intraprocedural concolic executers.

Acknowledgements

I acknowledge, with gratitude, my debt of thanks to Professor Subhajit Roy for his advise and encouragement and to Professor Amey Karkare for his aid and foresight. They presented me with the opportunity to tackle interesting problems in field of Software Testing. Their patient but firm guidance was critical to successful completion of my research.

I appreciate the support of my friends and wingmates who always provided me with the confidence and courage to tackle even the most challenging problems. Without their help and councel, the completion this work would have been immeasurably more difficult.

I also want to express my sincere gratitude to ***BRNS*** for encouraging our research work. Their encouragement motivated me to follow through this project.

-Ashwini Kshitij

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Symbolic execution has been a recipient of significant attention during the past few years. It is now considered an effective technique in generation of high coverage test suites. The idea has been discovered around three decades ago but it was after significant improvements the potential of the idea was harnessed. One such important improvement was symbolic execution alongside keeping track of concrete values (concolic execution). The main advantage of this technique is that whenever constraint solving complications (like timeouts) occur during classical symbolic execution, it is alleviated using the concrete values.

All practical programs involve procedure calls. The symbolic execution is relatively simple if there are no function calls involved (*intraprocedural analysis*). A function call tranfers the control from the caller to the callee. That function may very well modify the symbolic state of the variables. If we transition the symbolic state correctly through the function call, the symbolic execution will run correctly with the modified symbolic state and collect the precise alteration made to symbolic state within the procedure.

1.1 Motivation

Intraprocedural Analysis is performed on one procedure at a time. It is simple and conservative. But almost all real world programs involve procedural calls. This poses a problem for techniques such symbolic execution since they dont know how the procedure is going to modify the symbolic state. This brings us to the need of extending our analysis techniques to prgrams with functions and procedures.

1.2 Problem Statement

This thesis aims to provide a method that can capture the transformation of the symbolic state of the inputs when the program goes through a function call. Our implementation uses inbuilt concolic execution engine for the purpose. But the engine cannot handle a procedure call. At the call site, the symbolic value is reset to "start state" instead of modifying it as per the operations performed on it by the callee function. Merging our approach with this concolic engine will increase the precision of test case generation.

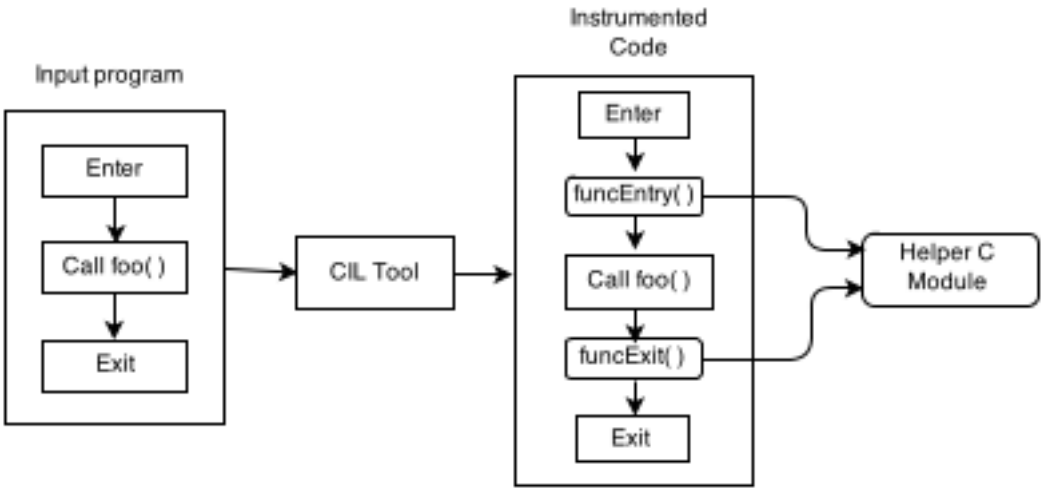


Figure 1.1: Basic Module

Figure 1.1 shown above gives a high-level idea of what our implementation does. We are given a C program with a concolic execution engine integrated with it. Problem arises with execution when it comes across a procedure call instruction. The symbolic state of the input variables is set to default as the program does not have the ability to follow through symbolic execution from caller to callee and then back to the caller.

We aim at solving this problem by instrumenting the call site. For such purposes in our approach we have used a tool called CIL. We first set up an environment before commencing the callee procedure execution (in this case *foo*). The operations for such tasks is carried out by *funcEntry* function which is instrumented just before *foo* is called. *funcEntry* handles the mapping of symbolic and concolic values of actual parameters to formal parameters, the creation of symbolic stack (later explained in *Chapter 3*).

Similarly we instrument *funcExit* function just after *foo* call. The task of *funcExit* is to map the symbolic state of "return variable" to appropriate variable at the call site (if

any) and clear the symbolic values that were generated during call and execution of *foo*. It also manages the symbolic stack appropriately. The implementation of *funcEntry* and *funcExit* is done in C language and is defined in included C modules.

This approach ensures that symbolic execution is carried through correctly to and from the callee. The modified state of symbolic variables is in accordance with operations of concerned callee function (in our case *foo*). This improves the accuracy and correctness of the test suite generation in our tool.

1.3 Contributions

In this thesis we have accomplished the following:

- Designed an approach to apply inter-procedural analysis to concolic execution.
- Application of this technique to improve the automatic test case generation using concolic testing.
- Dynamic support to handle recursive procedure calls.

1.4 Organisation of thesis

A brief summary of the contents of other chapters of this thesis are as follows.

Chapter 2 deals with the concepts which are necessary to understand the thesis. Concolic execution is introduced and its advantages over classical symbolic execution and random testing. An overview of approaches in interprocedural analysis like Function Inlining and Call String have also been discussed in this chapter. We have also briefly mentioned the related work that has been done in this field.

Chapter 3 discusses our implementation in a detailed manner. Some terminology has been introduced to explain techniques employed in our work. But mostly the chapter pertains to the relevant algorithms designed to accomplish the aim of this thesis and how it should improve the scalability and accuracy of concolic IP analysis.

Chapter 4 contains the experimental results and their analysis after we run our modified tool on a set of standard benchmark programs. Analysis in done in terms of coverage improvement and runtime of the tool.

Chapter 5 sheds light on future work and modifications that can be done to improve the accuracy and performance of the approach discussed in this thesis.

Chapter 2

Background

2.1 Interprocedural Data-flow Analysis

This is a technique which is broadly defined as gathering of information across multiple procedures (typically over the entire program). Procedure call poses barrier to program analysis. Its aim is to avoid making conservative assumpions about the effect of procedures and the state at call sites.

Interprocedural analysis is more demanding and challenging than intraprocedural analysis. Here we need to take into account the call-return and parameter passing mechanisms, local variable of the function and function recursion (can be unbounded). In some cases the called procedure is known only in run-time like with function pointers or with virtual functions.

Application : Interprocedural analysis (IPA) enables the compiler to optimize the code across different files (whole-program analysis), and can result in significant performance improvements of the program by removing spurious data dependencies. Integrating IPA with intraprocedural concolic execution will help compute the precise behavior of function calls on the symbolic state of the variables.

2.1.1 Function Inlining

One of the approaches is *function inlining* which is the simplest and most widely used approach for accurate interprocedural symbolic execution. We simply use the copy of procedure's Control Flow Graph (CFG) at each call site. This leads to function being

re-analyzed at every call site. This is can be avoided using *function summaries*. They are implemented by merging all states at the function exit after computing an intraprocedural path constraint in terms of function input.

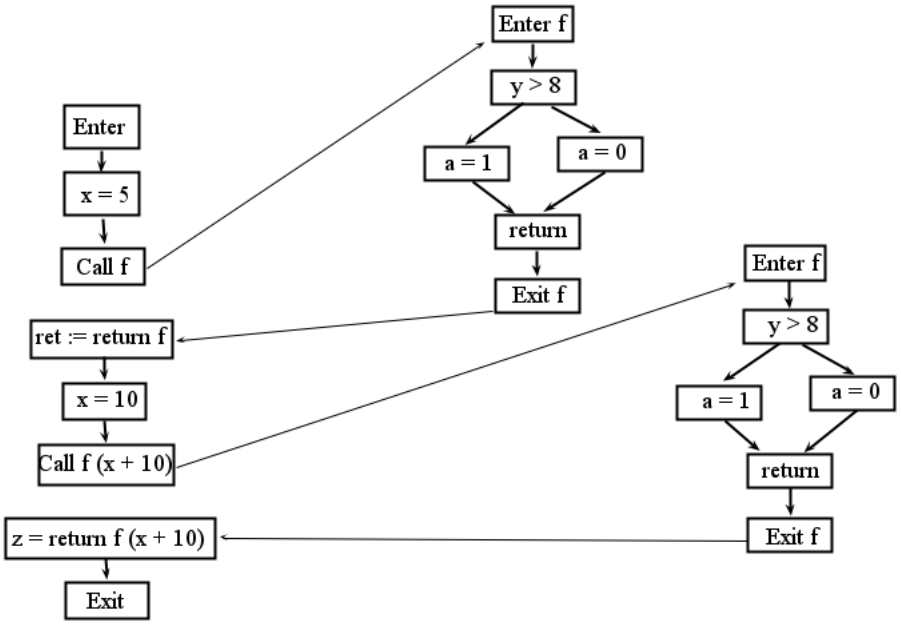


Figure 2.1: Function Inlined

This technique does not require function calling overhead. It also saves the overhead for manipulating function stack. In increases locality of reference by utilizing instruction cache.

But there are significant *drawbacks* with this approach. Firstly, the performance overhead will increase if we increase the size of the code that is to be inlined. The caller function may not fit on the cache causing high cache miss rate. Similarly if there are too many function calls involved, inlining maybe expensive since it will cause an exponential increase in the size of the CFG.

Secondly it is also going to create problems in the recursive procedures. Same can be said more generally for any scenario where there are cycles in the call graph. So basically function inlining has scalability issues pertaining to code size. Lastly, procedure inlining is only possible if the target of the call is known. Hence it will not be possible if call is via a pointer or is "virtual".

2.1.2 Call String Approach

In another IP analysis we observe the CFG's of all the procedures. In this technique,

- When we encouter a function call we interpret it as a goto from the call intruction to the first instruction of the procedure.
- Interpreting every return statement like a goto to the instruction following each call site that invoked that procedure (*Non-Deterministic*).

Using this method non-deterministically will allow non feasible paths in our analysis which will cause loss of accuracy. In a better approach called "Call String" we keep track of the where we came from, that is, the context of the call and where to return. To do this we maintain a "string" that simulates a call stack. A feasible path is a control flow path that is generated in accordance with the stack regime. A perfect solution is keep record of the whole stack. This concept is easy and intuitive according to which every return jumps to the instruction that's immediately after the call site which corresponds to that particular function call. By adding the context of the call to the information in the state we can overcome the problem of passing through invalid paths.

```
int main(){
    int x ;
    c1: x = foo(2);
    ret:[x → 4]
    c2: x = foo(3);
    ret:[x → 6]
    return 0;
}
```

```
int foo(int p){
    c1:[p → 2]
    c1:[p → 3]
    return 2*p;
    c1:[p → 2, $$ → 4]
    c1:[p → 3, $$ → 6]
}
```

Figure 2.2: Example : Call String

Every procedure records their state information when they are invoked. Let us consider the program in *Figure 2.2*. Here the labels *c1* and *c2* are saved in the state information of *foo*. This is so that p gets different values every time function *foo* is called. Another reason to save the labels is because they help in figuring out the respective return values of call *c1* ($\$\$ \rightarrow 4$) and *c2*($\$\$ \rightarrow 6$).

Implementing this algorithm is efficient for small strings. Problem arises when call string generated is large since we can keep track of only limited number of strings. This poses a limitation on the depth of function calls.

2.2 Concolic Testing

One popular approach for automated software testing is *random testing* which involves subjecting the program to be tested to a stream of random data. It is fast but it can find only basic bugs like program crashes, code assertions or memory leaks. But it is not always possible to employ this technique specially when working with binaries since it is very difficult to figure out the expected inputs. But random Testing fails to cover corner cases.

A more deliberate approach can be used by merging symbolic execution with concrete execution (*concolic execution*) and then using a an SMT-solver (like Z3) to generate test inputs. This method is called *Concolic testing*. It is much more efficient than symbolic execution. To understand why let us assume that our costraint solver cannot handle non-linear constraint. Then a path constraint with some *non-linear* function involved cannot be solved and symbolic execution will be stuck. Similar problem arises when we encounter a closed third party library function say *increment()*. Symbolic execution alogorithm doesn't know how to modify the symbolic state according to the behavior of that function. For such cases the test cases cannot be generated by classical symbolic execution. Concolic Tesing addresses these limitations and resolves situations like above by replacing the symbolic values by their concrete values so that the resulting constraint can be solved by the constraint solvers.

The algorithm involves initialting the inputs with randomly generated values. The program is executed and during the execution, on every conditional branch statement program collects symbolic path constraints on inputs. Symbolic constriants are essentially a set of logical constriants on input data. New program path is directed and executed by *negating/flipping* the last condition of the path constraint. This is done until all the feasible program paths are explored. To get the intuition, let us consider the program in *Figure 2.2* for concolic testing.

```
1 #include <stdio.h>
2
3 int main(){
4
5   int x,y;
6   scanf("%d",&x);
7   scanf("%d",&y);
8
9   int z = 3*x;
10
11  if(y == z){
12
13    if(y == x + 10){
14        assert(0);
15        /*error*/
16    }
17
18  }
19  return 0;
20 }
```

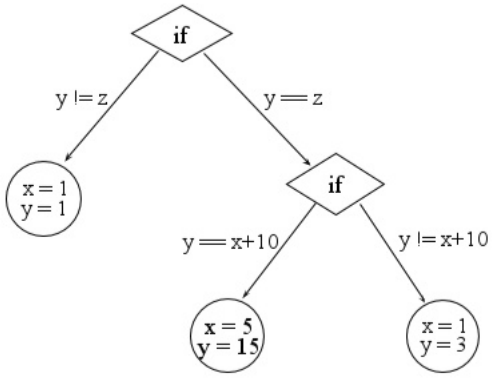


Figure 2.3: Concolic Testing

The program input variables are assigned random concrete values, say $x = 1$ and $y = 1$. The program runs and line 11 is executed. Path condition generated is :

$$\neg(y == 3x) \tag{2.1}$$

Now we negate the last condition in path constraint (PC has only one condition in this case) to execute an alternate path. So the new path constraint generated is :

$$y == 3x \tag{2.2}$$

which is examined by a SMT solver to generate the new input data. Out of many different possible values for equation *2.2* solver picks say $x = 1$ and $y = 3$ which should explore a different path than before. The program runs on these new inputs and generates a new path condition :

$$(y == 3x) \wedge \neg(y == x + 10) \tag{2.3}$$

Like previously we negate the last condition of the path constraint to generate condition for new path exploration. So we pass on the following constraint to solver to generate new inputs

$$(y == 3x) \wedge (y == x + 10) \tag{2.4}$$

Let the new inputs generated be $x = 5$ and $y = 15$ we will lead to the execution of line 13 and consequently line 14 which will hit the error or any other unexpected behavior. Since there are no more paths left to explore, the algorithm will terminate generating a set of input data for complete path coverage of this program.

2.3 Related work

2.3.1 Summary-based Analysis

New techniques have been invented that compute procedure summaries for performing an interprocedural analysis of programs. In summary-based context-sensitive analysis we create "summary" which is succinct description of the observable behavior of each procedure. The purpose of this approach is to prevent renanalysing the behavior of same procedure when there are invoked at each call site.

The representation of every procedure has a single entry point. The analysis is divided into following two phases:

- In fist phase we summarize the effects of a procedure and a transfer function is computed in a bottom-up manner.
- In second phase we propogate the caller information to compute callee result in a top-down manner.

2.3.2 DART

DART is an abbreviation for "*Directed Automated Random Testing*" which is a tool for automated test case generation developed by *Patrice Godefroid, Nils Klarlund* and *Koushik Sen*. It utilizes the concept of concolic execution and is comprised of the following techniques :

- *Automated* extraction of program interface from source code.
- *Random testing* the program interface by generating a test driver.
- Dynamic generation of test cases to *direct* alternate program execution path.

2.3.3 CUTE

CUTE stands for "A Concolic Unit Testing Engine for C" which also addresses automatic test case generation with memory graphs as inputs. This tool is developed by *Koushik Sen*, *Darko Marinov* and *Gul Agha*. It is similar to DART, thereby employs concolic testing technique. It resolves some of the limitations of DART and aims at testing real-world examples of C code.

It provides a method for representing and solving approximate pointer contraints to generate test inputs. The symbolic model being used is more powerful and the theorm solver is both more powerful and is built to be efficient in this system. As opposed to what DART does, CUTE does not automatically extract program interface but lets user decide relation among functions and their preconditions. The work also shows exactly how it made approximations and trade off between speed vs. correctness and scenarios where CUTE will not work correctly.

Chapter 3

Methodology

The algorithm that has been used to implement the interprocedural analysis in the concolic execution engine has been described in detail in this chapter.

3.1 Definitions

We are given a program **P** which has a concolic execution engine **C** integrated with it. Our aim is to combine interprocedural analysis with **P** so that the test case generation can also take into account the effect of function calls. To implement this algorithm we will instrument the code of P with our auxiliary code which will not alter the outcome of the program **P**. Let

- I be the input generated by the tool for program P.
- $\boldsymbol{f}$ be the callee function used in call site.
- *Output(P,I)* be the output of the program P when run on input I.
- $\boldsymbol{S}$ denote Variable Stack explained in *section 3.3.3*.

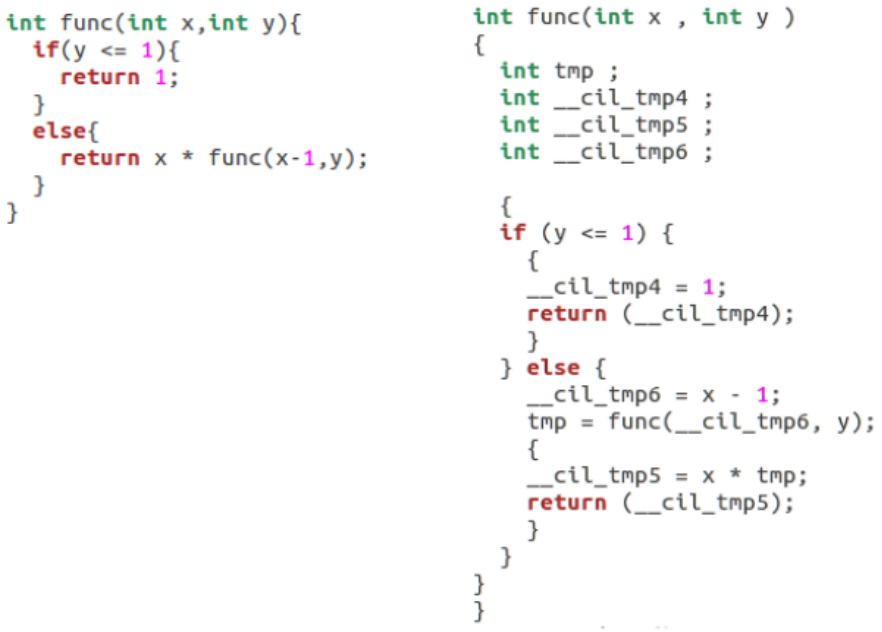
3.2 Static Analyser

We have used a tool called **CIL** (**C** **I**ntermediate **L**anguage) to perform static analysis of program P and source-to-source transformations on it. We do this by tranversing the AST (Abstract Syntax Tree) which is the in-memory data-structure which represents the parsed program P.

3.2.1 Handle Expressions

This module accomplishes two tasks. It involves the simplification at call site with respect to the arguments (if any) passed to the function. It also simplifies the return expression at the end of function (if functions returns anything).

At the call site the arguments of the function are examined one by one. If an argument is not a variable or literal then it must be an expression. In latter case, we store the argument in a temporary local variable and pass that new variable as the parameter instead. Similar approach applies to return values. If the return statement consists of a variable, it is left unchanged otherwise it is transformed.



```
int func(int x,int y){
  if(y <= 1){
    return 1;
  }
  else{
    return x * func(x-1,y);
  }
}
```

```
int func(int x , int y )
{
  int tmp ;
  int __cil_tmp4 ;
  int __cil_tmp5 ;
  int __cil_tmp6 ;

  {
  if (y <= 1) {
    {
    __cil_tmp4 = 1;
    return (__cil_tmp4);
    }
  } else {
    __cil_tmp6 = x - 1;
    tmp = func(__cil_tmp6, y);
    {
    __cil_tmp5 = x * tmp;
    return (__cil_tmp5);
    }
  }
}
}
```

Figure 3.1: Original and Modified Function

We can see in *figure 3.1* the call sites and return statements have been transformed such that in further stages we only have to deal with variables. We will not have to deal with complicated expressions. To create new local temporary variable we use CIL API's. It ensures that the variable have unique names thus avoiding any ambiguity. This phase will have no effect on function calls with no arguments or functions which don't return any value (void).

3.2.2 Variable Renaming

The concolic testing engine we are using, uses a symbolic table structure that is globally defined, to carry out the symbolic execution. The entries in the table are manipulated using variable names of program P as the key. Until now the symbolic execution was intraprocedural, so we didn't have to deal with duplicate variable names.

But now we are dealing with multiple procedures and all of them will be capable of executing symbolically. Two different procedures may very well have variables with the same name. So when these variables are used as the key to manipulate the values in symbol table, the context of the variable will be ambiguous. This will cause undesired behavior in the concolic engine.

Therefore we need to make the variables of all the procedures unique. This is done by renaming the variable in association with the name of the procedure in whose scope they belong. If K is the key of the symbol table and V be the variable name. Then before transformation, we have

$$K = f(V) \tag{3.1}$$

and after transformation we have

$$K = f(V, P) \tag{3.2}$$

where P is the name of the procedure to which V belongs. *Figure 3.2* demonstrates the renaming of the variables to resolve the conflict of symbol table keys. This method will work for non-recursive procedure calls. To handle recursion we have developed an extended version of approach which will be discussed in *section 3.4*.

```
int mult(int x, int y){
        int z;
        z = x * y;
        return z;
}
```

```
int mult(int mult_x , int mult_y )
{
  int mult_z ;

  {
  mult_z = mult_x * mult_y;
  return (mult_z);
}
}
```

Figure 3.2: Variable Renaming

3.2.3 Extend Concolic Execution Engine

The current engine uses approaches such as handling set (or assignment) instructions symbolically and unrolling of loops to carry out the symbolic execution. But these aproaches are only applied on the procedure that the user demands to be tested. The remaining procedures are unchanged.

Now on function call we want the symbolic execution to be able to continue in the callee *f*. Hence we need to instrument all the procedures of program P such that it enables them to be executed symbolically. So instead of applying the loop unrolling and symbolic assignment only on user input procedure, they are applied on every procedure.

3.2.4 Call Site Transformation

Without interprocedural call handling whenever the concolic testing encountered a function call, it used the concrete value of the function result. There interprocedural symbolic analysis could not be done. This hindrance is removed by setting up a function call environment and function return environment, that will facilitate the transfer of symbolic state from caller to callee and back. The pseudo-code of the algorithm of this phase is mentioned on the next page.

Algorithm 1 Description : The input to this algorithm is program P. We inspect all the procedures of P and detect the call sites. It is at these call sites where we need to do transformations such that when control flows from caller to callee, the callee has the symbolic information needed to execute symbolically.

Let one such call site be *c* which invokes a function *f*. Parameters that are passed to the function *f* have certain symbolic and concrete values associated with them. Before *f* starts executing, the mapping of concolic values needs to be done from actual parameters to formal parameters. For that we have to analyse the formal and actual parameters, and figure out a way to send their combined information to *funcEntry*, which does the actual task of concolic mapping.

The function *getArgumentsAsString* in *line 8* has two parts. Firstly, it extracts the information about actual parameters of *f* like if its a literal or variable (its name). Secondly, it analyses the formal parameters of *f*'s prototype and gathers information about their name and type. Then it merges this information into a string, one parameter at a time.

Algorithm 1: Transform Call Site

1 ***Input :*** Program P
2 ***Output :*** P with modified call sites
3 **begin**
4 **for** each procedure f ∈ P **do**
5 InstList = instructionList(f)
6 **for** each instruction I' ∈ InstList **do**
7 **if** I' is a Call Instruction C **then**
8 ArgsList = getArgumentsAsString(C → fname)
9 LocalsList = getLocalsAsString(C → fname)
10 FuncEntryInst = makeCallInstruction(ArgsList, LocalsList)
11 FuncExitInst = makeCallInstruction()
12 InstList = InstList :: [FuncEntryInst :: I' :: FuncExitInst]
13 **end if**
14 **end for**
15 **end for**
16 **end**

This string is passed as a paramter to *funcEntry* along with another string containing the names of local variables (created by *getLocalsAsString()* in *line 9*). The function instrumented after call site *c* is *funcExit* that does the task of cleaning up the intermediate data-structures created by *funcEntry* and were necessary to execute *f* symbolically.

```
1 int main(){
2         int a,b=3,c;
3         scanf("%d",&a);
4         a = fact(a,4);
5         if(a == 20){
6                 c = 1;
7         }
8         else{
9                 c = -1;
10        }
11        printf("%d\n",c);
12        return 0;
13 }
```

Figure 3.3: Original Code

Let us observe an example shown in *figure 3.4*. There is a call instruction at *line 4* invoking function *fact* with two paramters. The information about both actual and formal paramters in analysed and sent to *funcEntry* as string argument at *line 12* in *figure 3.4*. The second argument to *funcEntry* is names of local variables of function *fact*. Using these arguments, environment for *fact* to execute symbolically is set up by *funcEntry*. Similarly, *funcExit* at *line 15 in figure 3.4* handles the task of cleanup of this environment after *fact* has finished executing.

```
1 int main(void)
2 {
3   int a ;
4   int b ;
5   int c ;
6
7   {
8   b = 3;
9   scanf((char const   * __restrict  )"%d", & a);
10
11  //---------Transformed call site-------------------------
12  funcEntry("(int,x,variable,a) (int,y,constant,4)",
13            "tmp", "fact");
14  a = fact(a, 4);
15  funcExit();
16  add_entryToSTable("a", ret_SymValue, ret_ConValue, & a, 1);
17  //-------------------------------------------------------
18
19  if (a == 20) {
20    c = 1;
21  } else {
22    c = -1;
23  }
24  printf((char const   * __restrict  )"%d\n", c);
25  return (0);
26 }
```

Figure 3.4: Transformed Call Site

At *line 16* in *figure 3.4* we also have function *add_entryToSTable* which is used to map the returned concolic values of function *fact* to appropriate variable, which is in our case *a*. We will discuss about this in *section 3.3.4*.

3.3 Set up Environment For Procedure Call

This section involves the runtime analysis of the arguments that are passed to *funcEntry*. We will explain their utilization to create entries in global symbol table, to enable callee function f to execute symbolically.

3.3.1 Organising Argument Details

Using the information of actual and formals parameters passed to the *funcEntry*, we construct a data structure *Argument* that has the following attributes:

- *funcName* : name of the function to which the argument is passed.
- *vname* : name of the associated formal parameter.
- *type* : 1 for int and 2 for real.
- *apname* : if actual argument is a variable, then its name.
- *val* : if actual argument is a literal, then its value.

Note that actual parameter can either be a variable or a literal, implying that only one among *apname* or *val* can have a valid value. The other is going to be *null* so we have to use them accordingly. Recall that we have passed this parameter information along with local variable names of f to *funcEntry* as discussed in *section 3.2.4*. Using string tokenizing and parsing within *funcEntry*, we get a list of *Arguments* type structures.
This step is demonstrated in *figure 3.5* in reference to program shown in *figure 3.4*.

```
funcName : "fact"
vname : "x"
type : 1
apname : "a"
val : null
```

```
funcName : "fact"
vname : "y"
type : 1
apname : null
val : 4
```

Figure 3.5: Two arguments at *line 12* in *figure 3.4*

3.3.2 Populate Symbol Table

After we created a systematic model of arguments, we need to populate the global symbol table accordingly. Besides the arguments, we also need to handle local variable in *f*. For locals, we create empty entries in symbol table. The algorithm used in this implementation is described below

---

Algorithm 2: Populate Symbol Table

```
1  Input : Argument list A, locals list L, symbol table S
2  Output : Modified symbol table S' with entries for A and L
3  begin
4  for each argument a ∈ A do
5    if a is a literal then
6      addEntryToSTable(a→vname, "Constant", a→val)
7    else
8      sym = findSymbolicValue(a→apname)
9      con = findConcreteValue(a→apname)
10     addEntryToSTable(a→vname, sym, con)
11   end if
12 end for
13 for each local variable l ∈ L do
14   createEmptyEntryInSTable(l→name)
15 end for
16 end
```

Algorithm 2 description : This algorithm maps the symbolic and cocrete values of actuals parameters at call site to the formal parameters in the beginning of the function *f*. For symbolic execution in *f*, we also need to create initially empty entries of local variables in the table. Functions *findSymbolicValue* and *findConcreteValue* at *line 8 and 9* search the symbolic and concrete values for a variable in the table. They are defined in the C modules of concolic testing engine.

3.3.3 Variable Stack

As we have seen in the previous section, before the execution of f we have to populate symbol table with respective variables of f. But same importance should be given to systematic clean up after f is done executing, that is, deletion of entries from the table that were temporarily required for symbolic execution of f. This is because these spurious values may not only cause inconsistencies but also caused the symbol table to overflow. For this purpose, we maintain a stack S called *variable stack*. Each element of the stack stores the information regarding variable entries in the symbol table for indiviual function. The contents of stack element are as follows :

- *funcName* : name of the function for which this element is created.
- *args* : string array containing the names of formal paramters of *funcName*.
- *locals* : string array containing names of local variable in the scope of *funcName*.
- *occurence* : used to indicate the instance of *funcName* in the call stack. We will explain this in detail and how it is used to resolve recursive calls in *section 3.4.2*.

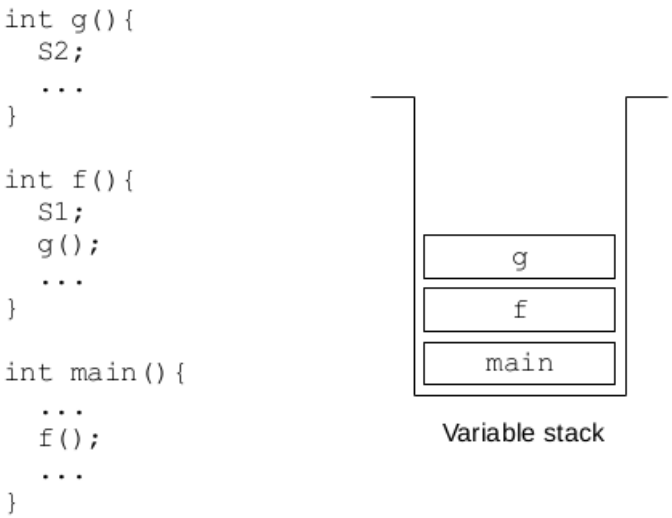


Figure 3.6: Variable Stack

The behavior of variable stack is a lot alike to function call stack. Whenever we encounter a function call we push the variables (both arguments and locals) of the callee function onto the stack S. For example, in *figure 3.6* when f is called in *main*, its variable details are pushed onto S. By the time statement S2 executes in g, stack has three elements

as shown in the figure. When we return from a function call, we pop an element from the stack and delete the entries in the symbol table by referring the popped element.
The task of pushing the element on encountering a function call is done by a method *funcEntry* and popping the element after returning from that function is done by *funcExit*, both of which are defined in C modules.

3.3.4 Return Handling

After the environment is set up for *f*, it executes symbolically covering some path in control flow graph of *f*. It collects a summary that tranforms the symbolic values of parameters to new symbolic return values (if *f* returns something). We need to map the concolic return values to appropriate variable at call site.
For this, just before the function *f* returns, we copy the concolic return value to a global variable. Then at call site, we assign the lvalue of call instruction to this global concolic variable. Thus, the concolic execution flows to and from the callee correctly.

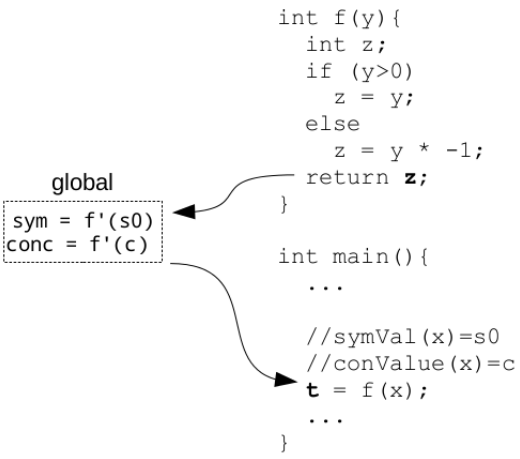


Figure 3.7: Concolic Return

Here in *figure 3.7*, function *f* is called with parameter x. Its symbolic and concrete value is modified according to function f', which is a transition function along some path in *f*. Variable *z* attains this new concolic value and is returned. To ensure transfer of modified symbolic information on function return, we map the concolic value of *z* to global variable and from them to *t* in *main*, which is the intended recipient.

3.4 Recursion

3.4.1 Limitation of previous approach

The approach that we have discussed until now can only handle non-recursive procedure calls. The reason being that our algorithm did not have support for multiple versions of same variable of the same function in the symbolic table. Our variable stack can handle atmost one instance of a function at any given time. {Recall that in *section 3.2.2* we have resolved the issue of same named variables in different procedures.}
To understand this, let us consider an example of recursive code.

```
1 int fac(int fac_n)
2 {
3   int fac_m;
4   if (fac_n == 0)
5      fac_m=1;
6   else
7      fac_m = fac_n * fac(fac_n-1);
8   return fac_m;
9 }
10
11 int main (void)
12 {
13   int i = 5;
14   int j = fac(i);
15   return 0;
16 }
```

Figure 3.8: WCET program for factorial

In this example, at *line 14* a function call has been made to *fac*. Before the call instruction executes, our algorithm maps concolic values from x to *fac_n* and creates an entry in the symbol table using *fac_n* as the key. Now when the *fac* executes, it again encounters a call instruction at *line 7*. Our algorithm should map the values from *fac_n - 1* to "new" *fac_n*. But it has no mechanism implemented to distinguish between different versions of same variable of the same procedure. This leads to overwriting of previously present entry of *fac_n* in the symbol table (refer to *table 3.1*) . More important problem that will arise is that after *fact* at *line 7* executes and returns, it cannot restore the previous symbolic state (i.e. before call was made). The concolic values of previous versions of the recurring function in the stack will be lost.
Therefore, we need to modify the key of symbol table such that we can keep track of multiple instances of same function in the variable stack, allowing recursive call analysis.

key	symVal	conVal
	:	
	:	
fac_n	s0	c0
	:	

(a) call at line 14

key	symVal	conVal
	:	
fac_n	s1	c1
~~fac_n~~	~~s0~~	~~c0~~
	:	

(b) call at line 7

Table 3.1: Symbol Table

3.4.2 Need for Versions

We have to maintain versions of function variables according to number of instances they have in the variable stack. We have to make sure that the entries in the symbol table dont get deleted or overwritten unless they are not needed anymore. One way is to modify the keys (of symbol table) such that they have function version associated with them.
Let **occurence** of a procedure at any time be defined as the number of instances of that same procedure present in the call stack.

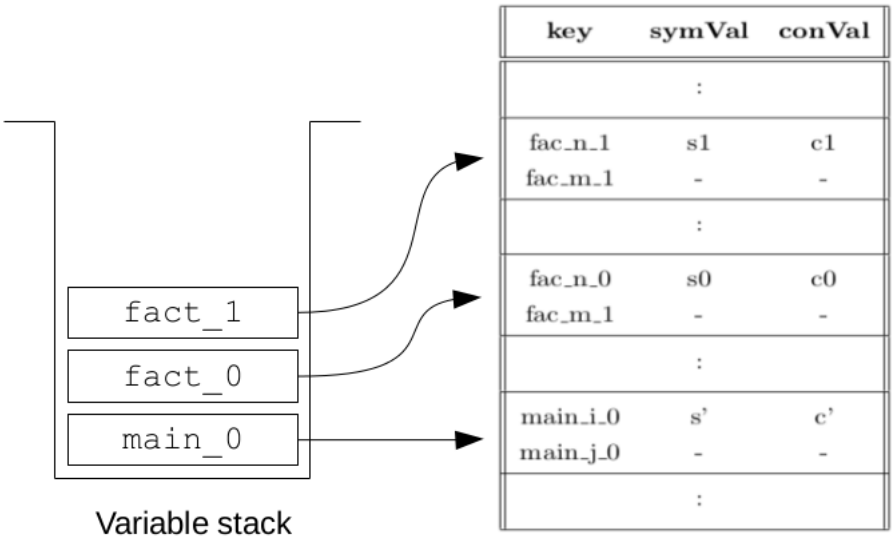


Figure 3.9: Function Variable Versions in Symbol Table

Therefore we remodify the key defined in *section 3.2.2* to keep track its associated variable, procedure to which that variable belongs and that procedure's version (*occurence*)

it the call stack. The new key is:

$$K' = f'(V, P, O_P) \tag{3.3}$$

where O_P is the occurence of P in function call stack. In *figure 3.9* we can see the version handling of multiple instances of same functions in the symbol table. When another instance of function is pushed in the stack (recursively), no value in the symbol table is overwritten. Similarly, when a function instance is popped from the stack, the concolic values of previous version should be accessed.

3.4.3 Variable Hash-Map

The issue that arises with this approach is that we cannot rename variables in conjunction with their versions (like we did in *section 3.2.2* using static analysis). This is because the versions are created dynamically at runtime when function call stack is manipulated. Therefore we create a hashmap M that maps the variable name to key in the symbol table. At any point during execution, if procedure P is in the call stack, there will exist a mapping such that if K = f(V,P) then M has following *(key, value)* pairs

$$M = (K, K') \tag{3.4}$$

Whenever concolic execution needs to refer the symbol table using variable V, we look up the *value* corresponding to V in M and return the *key*. But we still have to maintain the hashmap so that it always give the correct version of the key (the state of the variable stack is dynamic and so are the versions of function variables). To explain this let us consider a call instruction in the program that invokes function *f*. The operations before and after the execution of f are as follows.

Before Function Execution

In *section 3.2.2* we discussed the algorithm to populate the symbol table with specific values before executing a call instruction. We modify that algorithm slightly, instead of using variable name K as key in *addEntryToSTable*, we use our new key K'. Then we add the new mapping (K, K') to *M*. This keeps track of key K' and its concolic values, for when we need to do a query on the symbol table.

After Function Execution

When a function is done executing, the entries in the symbol table for that particular version of function are deleted. To do this, we again use the hashmap to get the current keys (containing current version of variables also) of symbol table that are no longer required.

3.4.4 Bound on Recursion

To keep the size of symbol table in check and prevent the variable stack from overflowing, we have to put an upper limit on the maximum number of instances that the variable stack can have of a given function. This puts a cap on the precision of symbolic execution. If the maximum number of versions allowed for a particular function be C, then the symbolic execution will only continue uptil stack depth of C, after that the function executes using only concrete values. The symbolic execution resumes when the stack depth with respect to that function becomes less than or equal to C.

Chapter 4

Experiments

4.1 Result

4.2 Comparion of Test Case Generation

4.3 Analysis

Chapter 5

Conclusion

Bibliography

key	symVal	conVal
	:	
fac_n_1	s1	c1
fac_m_1	-	-
	:	
fac_n_0	s0	c0
fac_m_1	-	-
	:	
main_i_0	s'	c'
main_j_0	-	-
	:	