國立交通大學

資訊科學與工程研究所

碩 士 論 文

從 COTS/Binary 元件中解析條件參數以輔助隨機測試

Resolving Constraints from COTS/Binary components for Concolic Random Testing

研　究　生：范揚杰
指導教授：黃世昆　教授

中 華 民 國 九 十 六 年 六 月

Resolving Constraints from COTS/Binary Components for Concolic Random Testing
從 COTS/Binary 元件中解析條件以輔助隨機測試

研 究 生：范揚杰　　Student：Yang-Chieh Fan

指導教授：黃世昆　　Advisor：Shih-Kun Huang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

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

從 COTS/Binary 元件中解析條件以輔助隨機測試

學生：范揚杰　　　　指導教授：黃世昆

國立交通大學資訊科學與工程研究所碩士班

摘　　　　要

軟體品質透過軟體測試來達到驗證。軟體測試是想要找到程式缺陷而進行技巧性審查的過程。目前有許多的方式應用在軟體測試上，但是有些臭蟲是卻很難用傳統的測試方法來找到，假如他們不常發生在一般的情況下。為了改進軟體測試的涵蓋率，許多研究中提出自動化產生所有可能輸入資料的技術。

近期，較先進的做法是將具體和符號執行結合應用於隨機測試中。先簡化受測程式的原始碼成基本的指令型式，然後依據不同指令附加額外的程式碼以描述程式的行為，使用者提供具體的輸入值來執行轉換後的程式。在實際執行後，相關的條件參數會被儲存起來，然後再解析條件參數以產生測試的例子。但是這種方法不適合使用在沒有原始碼的程式上，特別是程式中使用到商業軟體或執行檔的部份。我們在 ALERT（自動化隨機與邏輯推理運算測試）中提出一個解決之道，以接近原本程式語意的原則建立符號參數跟相依變數之間的關係。因此，可以保有原本符號的屬性，並且抵達更多的程式路徑。

Resolving Constraints from COTS/Binary Components for Concolic Random Testing

Student : Yang-Chieh Fan **Advisor :** Dr. Chih-Ming Huang

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Software quality is verified through software testing, which is a process of technical investigation that is intended to reveal faults. There are many approaches to software testing, but bugs are difficult to find in conventional testing if they occur infrequently. In order to improve the test coverage, several techniques have been proposed to automatically generate all possible values of the inputs.

Recently, the developed methods are combining concrete and symbolic execution for Random Testing. They first try to instrument the source code of a program under testing. The program is executed on some user-provided concrete input values. After the concrete run, symbolic constraints are stored and then generate concrete test cases by solving these symbolic constraints. Unfortunately, access to instrument source code is often infeasible especially for COTS/Binary component. We present a method in the framework, ALERT (Automatic Logic Evaluation for Random Testing), which approximates the program semantics, and establishes a connection between symbolic parameters and dependent variables. As a result, the symbolic property can be preserved and more paths will be reached.

誌　　　謝

感謝一直以來默默付出的父母，當兵退伍後決定繼續求學，由於他們的支持才能順利地完成學業。黃世昆老師認真地培養實驗室成員分析、解決問題的能力，除了每週固定的 Group meeting 之外，老師還費心地一一指導個人論文的方向，不時地跟大家分享相關領域的新知與願景，從決定題目到論文撰寫，都不遺餘力地指導。蔡昌憲學長幫忙克服許多實作上的困難，適時地給予建議，分配協調 ALERT 計劃的實作細節。彥佑和立文一起參與整個計劃的執行，以及後來熱心加入計劃的學弟們，沒有大家的幫忙就不會有這篇論文的產生，謝謝你們。

范揚杰謹誌

民國九十六年八月二十八日

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Automated Test Generation

Automated test generation problem has been studied since 70's and comes to significant progress in recent years [1-4]. Attributed to the increasing computational power available on modern computers, sophisticated program analysis techniques, such as symbolic execution engines and constraint solvers, becomes more practical on real programs. It regains interest in automated test generation with program analysis and makes a considerable impact on this domain. Based on the approaches of analysis, there are two categories: static and dynamic test generation. Dynamic test generation can be viewed as extending static generation with additional runtime information, and is more general and powerful (Table 1). Hence the latest trend is to blend dynamic test generation and model checking based on symbolic execution to find program errors [5-8]. It can systematically execute all feasible paths of a program and then use run-time checking tools [9] or universal checks for detecting software failures, like null pointer deference, memory leak, buffer overflow, etc. Remarkable achievements are that new bugs can be revealed in their demonstration. This novel work for automated checking indeed has a great impact on modern software industry.

Table 1: Comparison between static and dynamic test generation

	Execute program	Symbolic execution	Target	Problem
Static Test Generation	No	Computation tree	Enumerate all paths	Unresolvable Constraint
Dynamic Test Generation	Yes	Run-time trace	A given location / specific path	Missing Constraint

1.2 Constraint Propagation

Symbolic execution is a well known technique, which represents values of program variables with symbolic values instead of concrete data and manipulates expressions involving symbolic values [10]. It is traditionally used for context checking in program analysis and greatly evolved in automated test generation. The behavior of a program can be analyzed as constraints, which includes operations and predicates. Constraints, like a formula, are then resolved by a constraint solver to generate a test input of a specific path [4]. Naturally this approach suffers when the program contains statements involving constraints outside the scope of reasoning of the solver. As shown in Figure 1, because *hash(y)* can not be resolved statically, symbolic execution can not progress. Dynamic analysis is introduced to cope with this problem [5-8]. At runtime *hash(y)* is nothing but a concrete value, and can be compared with *x* successfully. This kind of problem usually happens in external calls, like system or library functions.

```
int foo(int x, int y)
{
        if (x == hash(y)) return -1;          // error
        return 0;                             // ok
}
```

Figure 1: Example code that can not be reasoned about symbolically

Although dynamic concretization addresses to handle the mentioned problem, this method may result in the imprecise symbolic execution due to dropping some symbolic information (Figure 2). As shown in Figure 2, if *abs(x)* is concretized, and then assigned to *y*, *y* also becomes concrete. Eventually the branch in this code is not perceived in symbolic execution. Performing symbolic execution mostly relies on instrumentation in source code. If there is a section of un-instrumented code, related constraints will be dropped. Unfortunately most COTS or binary components do not provide source code.

```
int foo (int x)
{
        y = abs(x);
        if (y > 10) return -1;      // error
        return 0;                   // ok
}
```

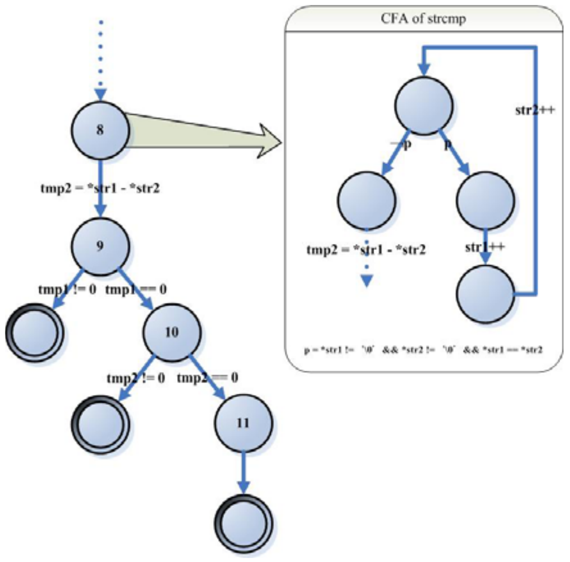
Figure 2: Example code that loses a path due to concretization

1.3 Path Explosion

Programs may have infinite states while the constraint system only has finite states. Therefore symbolic execution can not always simulate the complete program behavior. For example, if the constraint system has a limited number of constraints, says N, the most paths that can be enumerated are 2^N. Moreover only branch constraints are taken into consideration, and the actual paths may be fewer. Since the number of enumerated paths is fixed, how to effectively use this enumeration becomes important. Tested program usually interacts with external functions, and traditional symbolic execution on those also performs exhaustively. It seems useless and expensive because we are only interested in the top level program. That turns into the major factor that causes the unnecessary reach of deeper paths [11, 12]. Figure 3 gives a clear illustration about this problem, and we use a CFA to show the program flow. The successive calls to *strcmp* hamper the search of line 10 and 11. Assume this path enumeration uses DFA strategy, line 10 needs the first input equal to “Hello World”, and this at least wastes 10 paths in *strcmp*. If we want to reach line 11, the extra paths could amount to 10*10. Obviously the redundant paths in external calls obstruct the testing.

```
int strcmp(char *str1, char *str2) {
    while (*str1 != '\0'
                    && *str2 != '\0'
                    && *str1 == *str2) {
        str1++; str2++;
    }
    return *str1 - *str2;
}

void testme() {
1:  int tmp1, tmp2;
2:  char *s1, * s2;

3:  char *const1 = "Hello World";
4:  char *const2 = "Hello ESEC/FSE";

5:  s1 = input();
6:  s2 = input();

7:  tmp1 = strcmp(s1,const1);
8:  tmp2 = strcmp(s2,const2);
9:  if (tmp1 == 0) {
10:     if (tmp2 == 0) {
11:        printf("Success");
        }
    }
}
```

(a)



(b)

Figure 3: (a) Example code (b) CFA (Control Flow Automaton)

1.4 Motivation

The main idea in our work focuses on how to redeem the missing constraints due to un-instrumented function calls. Intuitively, resolving constraints from un-instrumented functions seems to be towards recovering their source semantics and is regarded as a category of reverse engineering. However, either disassembly or decompilation is a difficult task. Annotation-assisted static checking is a mature technology for finding security vulnerabilities and coding mistakes. Although it sometimes generates false warnings and misses real problems, annotation indeed gives an alternative view of functions in semantics. This idea can probably improve the partial coverage of concolic testing due to lack of source codes. We try to instrument a post-condition after each function call which will pass partial symbolic information. The post-condition like annotations is generated from function semantics manually. Although the source code is not available, we still can learn of the specification to interpret its semantics. On the other hand, the programmer can also provide related post-conditions to facilitate testing if they do not want to release source code.

Testing always intends to pass all program paths using different inputs. But the behaviors of external functions usually are not the major concern. Even if there are two inputs that can yield two different paths in an external function, the path in the top level function may be the same. Moreover those redundant paths obstruct our systematic test generation and result in path explosion problem. We compact the constraints of external functions using post-condition to solve this problem.

1.5 Objective

Our objective is to construct an automated test generation framework in order to experiment with post-condition aided symbolic execution, and to build a map that will establish the relationship between symbolic function arguments and its return values according to its semantics. We want to show how post-condition helps to resolve constraints from COTS or binary components. Besides, run-time information can be used to strengthen the aided constraints.

Using specification to interpret functions is analogue to an approach to abstract the semantics of functions. As soon as functions are abstracted, added constraints can be compacted, and then the load of constraint system can also be eased. This method reduces the redundant paths in external functions and helps our testing to achieve deeper path enumeration.

The external function calls should not affect the testing process because what we care is the behavior of the top level function. Our work aims to separate the symbolic execution from external functions with post-conditions. Consequently, the instrumentation to external functions can be omitted and can avoid the unnecessary path enumeration. As the specification is given, the related constraints can be manufactured easily. This functionality is integrated into ALERT framework for more correct and effective testing.

2 Related Work

2.1 Random Testing (RT)

It randomly selects test cases from the input domain, the set of all possible inputs, and can detect certain failures unable to be revealed by deterministic testing. The advantage of RT includes its low cost, ability to generate numerous test cases automatically, and the generation of test cases in the absence of the software specifications and source code. However RT usually consumes much time to yield duplicate test cases and still can not achieve full path coverage. If the failure-causing inputs are clustered together, the fault-detection effectiveness of RT may be incompetent. Systematic testing is preferred because it is directed, usually toward exposing failures. ALERT combining concrete and symbolic execution can be viewed as a systematic testing to walk all the computation paths from program analysis and tries to find potential bugs through universal checking. Only the initial input is random and the power of symbolic execution makes sure that next generated input is for a different path. Upon the ability of completely symbolic evaluation, ALERT can efficiently generate all feasible paths.

2.2 Model Checking

Model checking is an automatic technique for verifying finite state concurrent systems. In the hardware and protocol domains, it has been widely successful in validating and debugging designs by algorithmic exploration of their state spaces. The major limitation is *state-space explosion*, so model checkers only explore the state space of an abstracted system. Recently, there has been significant interest in applying model checking to software, and state enumeration is viewed as computation with predicates that represent state sets [13, 14]. Since software is typically infinite-state, abstraction is even more critical. One approach to model checking software is based on the *abstract-check-refine* paradigm: build an abstract model, then check the desired property, and if the check fails, refine the model and start over. Generally symbolic execution is a developed method to check property and counter-example driven refinement is adopted to improve accurate. Even if refinement loop is taken, model checking still suffer from infinite refinement loop and incorrect reports due to the statically modeled program. ALERT, like an explicit model checking, uses symbolic execution to map abstract counterexamples on concrete executions and to refine the abstraction, by adding new predicated discovered during symbolic execution. The main difference is that ALERT combines symbolic execution with concrete execution to alleviate the infeasible states. Consequently shortcomings described above in model checking will be alleviated.

2.3 Concolic Testing

Concolic testing [5-8] iteratively generates test inputs by combining concrete and symbolic execution, observing that the complexity and imprecision of purely symbolic techniques can be alleviated by using concrete values from random executions. During a concrete execution, a conjunction of symbolic constraints placed on the symbolic input variables along the path of the execution is generated. These constraints are modified and then solved, if feasible, to generate additional test inputs which would direct the program along alternative paths. There are two important works, DART and EGT in this domain, and they are proposed almost in the same period. Later come the CUTE and EXE. We compare those works below:

Table 2: Comparison between CUTE and EXE

	Target	Search Strategy	Bug Diagnosis	Memory Model	Target
CUTE	Full coverage	Bounded Depth-First Search	External tools	Approximate	Unit-testing
EXE	Find bugs automatically	Selective Depth-First Search	Universal checking	Accurate	Kernel code

2.4 Path Compaction

This research is splintered form DART[11, 12], and focus on the path explosion problem for scalability, because the number of feasible execution paths of a program increases exponentially with the increase in the length of an execution path. SMART proposes to perform dynamic test generation compositionally, by adapting known techniques for interprocedural static analysis. LATEST explores the most abstract version of the program, refining the abstraction on demand based on particular executions. They respectively propose top-down and bottom-up fashions to solve this problem. An immediate way is to save symbolic execution in external functions because to traverse all paths in these functions makes no sense about testing. In other words, an external function is essentially treated as a black-box.

SMART summarizes the pre-condition and post-condition for external functions in constraints according to each successive iterations on different function inputs. It applies DART-like search algorithm to compute summaries of functions and then reuses them. Because most external functions can not be reasoned about in symbolic execution without instrumentation, manual analysis in our work assists in breaking this limitation. However, the pre-condition and the post-condition stand for different terms in ALERT. The post-condition can be view as the side effect after a function call, and the pre-condition, often a return value, is used to determine what post-condition should be added. We add post-conditions of the external functions depending on the outcomes concretely.

3 Post-condition Aided Symbolic Execution

3.1 Strongest Post-condition

The requirement of source codes for instrumentation becomes a major factor in the incomplete paths generation. In computer programming, the post-condition is a condition or predicate that must always be true. Calculation of post-conditions has applications in program verification, quality improvement, deriving specifications from programs and quality measurement. During symbolic execution, we can also use the constraints generated from manual analysis beforehand to interpret the semantics of external functions as post-conditions. In this way, the side effect of a function can be carried out in our constraints system. Even though the symbolic execution can not be properly performed in external binary components without the instrumentation of source codes, we can add the necessary constraints after they return to improve the defects of concretization. Here is an example:

*char * strcpy (char * destination, const char * source).*

This function will copy the string pointed by *source* to the location pointed by *destination*. Due to concretization, our symbolic system is not informed about this behavior, and then latter usage of the destination can not propagate the constraints of the source. In other word, the side effect of *strcpy* disappears under symbolic execution. Actually if the *source* can vary with the input data, it is possible to go through all different execution paths determined by the *destination*. When the side effect can be presented with the post-condition, the symbolic propagation can be ensured successfully. For instance, we add a constraint that the first byte in the *destination* is equal to the first byte of the *source*. We expect to correctly simulate the semantics of the external function with post-conditions, so there needs a joint post-condition regardless of the different pre-conditions. In addition the patched constraints merely represent certain but not complete calculation of post-conditions because these are constructed only from the specifications. Therefore strongest post-condition (Figure 4) is adopted to interpret the semantics of external functions. It represents the necessary semantics after the external function calls, and can be easily described in constraints. Strongest post-condition may be imprecise, but somehow it overcomes the missing paths problem caused by concretization. Symbolic information in external function calls can not be ignored if the testing wants to achieve better path coverage. The semantics of functions must be analyzed in advance and this work can not be automated; even so, it indeed helps to redeem the lost constraints and can be reused in later testing.

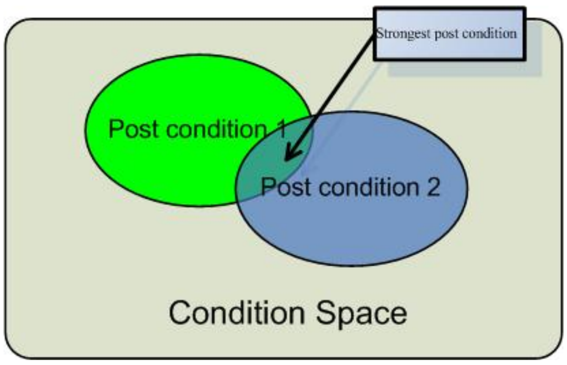


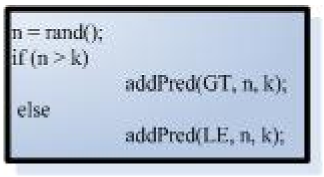
Figure 4: Strongest post condition

3.2 Return Values Driven Constraint

Strongest post-condition only supplies the most common predicates of an external function during symbolic execution. Although it patches the symbolic execution, it is not precise enough. We take the advantage of the run-time information as pre-conditions in order to conduct the suitable constraints. Therefore, the return value of a function can be taken for our selective post-condition. Consider the function in C Standard General Utilities Library which returns a pseudo-random integral number in the range 0 to *RAND_MAX*:

int rand (void).

The return value of the function *rand()* is then assigned to another variable, says *n*. The constraint system does not have any idea about *n* and results in constraint missing. If we use strongest post-condition to some value *n*, the constraint of $(0 \le n) \wedge (n \le RAND\_MAX)$ will be added. Obviously, this will not benefit the symbolic execution anymore because the range is too large. So we make use of the return value to refine the constraints (Figure 5). *k* is the key value that will influence the subsequent flow.



```
n = rand();
if (n > k)
                addPred(GT, n, k);
 else
                addPred(LE, n, k);
```

Figure 5: Refine the post-condition

We add the constrains depending on its return value because *rand()* can be evaluated dynamically. This measure can effectively reduce the range of the post-condition of *rand*() and then can improve the imprecision of strongest post-condition. The difficulty is how to choose the decision value. Basically we use the common behavior to determine, e.g. *strcmp* returns 0 if two strings are the same.

4 Implementation

4.1 ALERT Framework

The development of ALERT project is raised for the purpose of automatic feasible input generation. Symbolic constraint system in ALERT operates like that in DART, but we employ more powerful constraint solver. This feature gives ALERT more accuracy during symbolic execution and more practicality in a real program. In the following sections, we will introduce several important stages in this work.

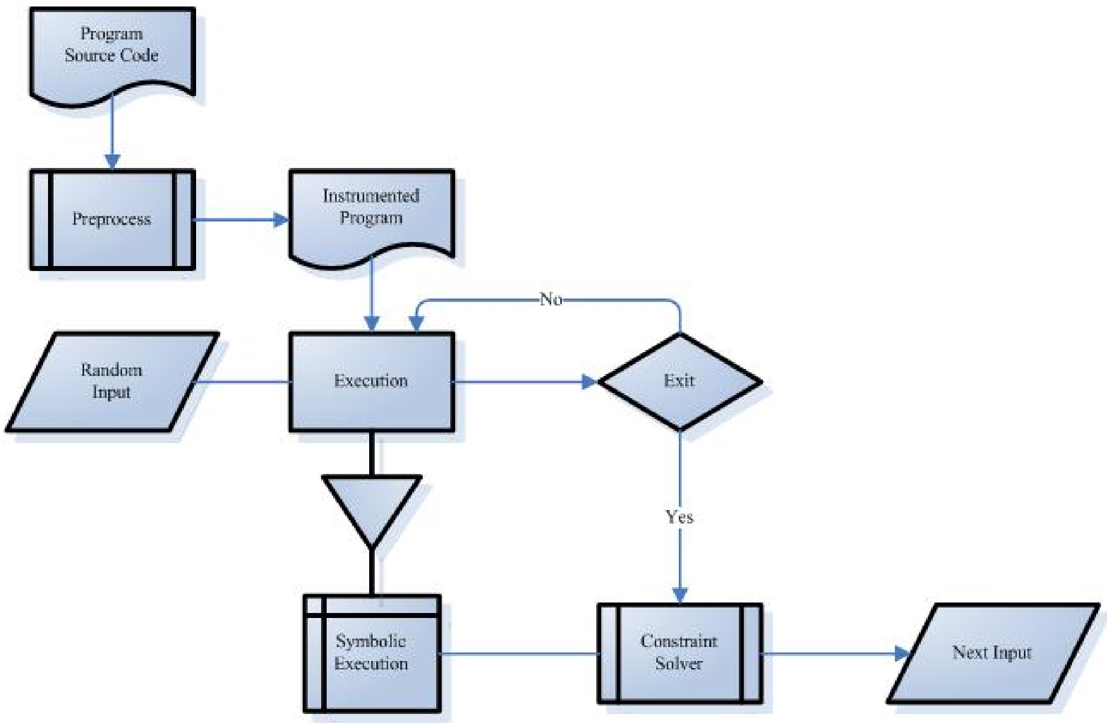


Figure 6: ALERT framework

4.1.1 Preprocess – CIL (C Intermediate Language)

CIL is a high-level representation along with a set of tools that permit easy analysis and source-to-source translation of C programs. It supplies many useful modules for analyzing and manipulating C programs, and we utilize some of them in the preprocess stage.

4.1.1.1 Source to Source Translation

Programs usually contain various structures and complex statements. This makes it difficult to normalize the symbolic execution. For this reason, we need a regular translation to simplify the original program into basic data types and operations with a very clean semantics. This idea can be viewed as a compiler to output a form suitable for the processing by the constraint system. CIL provides several available modules for us to utilize, but there are still some simplifications not handled. Therefore we add a module, *otherSimp*, in CIL to deal with the left jobs. Main modules in CIL we used are *simplify*, *simpleMem*, and *cfg*.

simplify is responsible for the translation of program to 3-address code. Hence composite statements will be reduced to basic statements in the form of atomic operations. Moreover, structures will be interpreted in a base address and offsets to represent different objects. *simpleMem* simplifies all memory expressions. Pointer to pointer and reference to pointer will be simplified via the introduction of well-typed temporaries. After this transformation all lvalues involve at most one memory reference. *cfg* makes the program look more like CFG: for instance, while loop will be translated into a statement *if* and a branch *goto*. *otherSimp* is applied to make the translated code more suitable for a later instrumentation. It handles the arithmetic in memory access and return value expressions.

4.1.1.2 Instrumentation

After the translation, the code of the tested program turns into the simplest form. Most code should be instrumented into an ALERT API used to trigger the representing symbolic execution when the code is executed. We implement a module *alert* in charge of this work. It can automatically identify various statements, different operators, and decide which arguments to be used in the instrumented code.

There are two kinds of statements to be instrumented. One is the conditional branch, and the other is the basic instruction. We instrument a code for the truth predicate in *if-statement* while for the false predicate in *else-statement.* In the case of basic instructions, we instrument the corresponding symbolic operations. Besides, we also instrument code to count the frame context number after the function entry point and before function exit point.

4.1.2 Symbolic Execution – CVCL

Symbolic execution is mainly used to accomplish the automated test input generation. It provides a systematic way to enumerate different paths. All paths in a program can be generated through extending traces in the CFG., i.e. a computation tree. Therefore we can view path enumeration as a search in the computation tree of a program. Our symbolic execution combined with concrete execution uses DFS to systematically enumerate all feasible paths. The method is to gather constrains for symbolic execution during concrete runs, to negate the predicate in the last branch, and then to use the constraints to generate the next input, which can steer the program to an alternative path.

Constraints propagation starts at the tested program entry point, and is carried out during the concrete execution. Initially, only the input values are symbolic in symbolic execution, i.e. the inputs can be any values in the range of their types. This symbolic property is propagated through various operations of assignment. If one of the operands is symbolic, the assigned variable also becomes symbolic, or it is still concrete. Therefore we maintain a hash table to check if the operand is symbolic.

CVC Lite is an automatic theorem prover for the Satisfiability Modulo Theories (SMT) problem. Its features include: support for a variety of theories; interactive as well as C and C++ library interfaces; proof and model generation abilities; predicate subtyping; and suppport for quantifiers. We utilize this constraint system to perform symbolic execution because it can accurately model the operation in memory system and has the interpreted theories to validate some non-linear problems.

4.1.2.1 Symbolic variables

Although CVCL provides a powerful constraint system in real and integer, we choose the bit vector to express different types in C because it has more precision under memory operation. We use 8 bits length bit vector to express a byte, and then concatenate bytes to express different data types in C. So there is a map from a type to size of bytes in ALERT for symbolic execution.

Constraint system carries out the proper procedures relative to the nondeterministic operation in the program as a manner of explicit model checking. The relationship between variables is propagated through this way. Because our constraint system can not maintain temporal property, the previous relation may be modified by later use. So we append a number to each variable in constraint system, and the number will increase whenever the variable is reused in symbolic execution (Figure 7). First, $b + c$ is assigned to a, and then a is assigned to b. If we do not use the variable translation, the constraint system is unable to distinguish the former b and the latter b. Therefore the constraints created in the system are $a = b + c \wedge b = a$, which induces $a = a + c$. This will cause the constraint system to be invalid, and stop the symbolic execution. Besides, we especially append a context number for inter procedures. This number helps our constraint system to make out the local variables of different functions in stack frames.

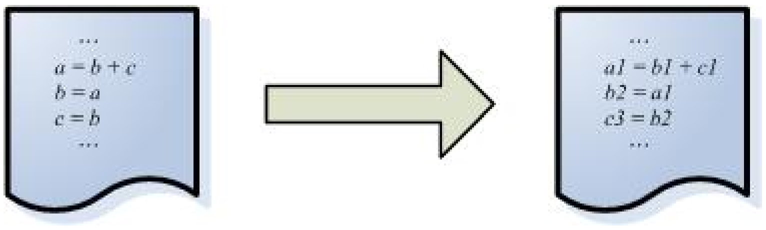


Figure 7: Variable Translation System

4.1.2.2 C Operator

C contains many operator groups: arithmetic, assignment, logical, bitwise, and others. It means that our constraint system should cover all of those operations for accurate symbolic execution. As a theorem prover, CVCL can interpret real and integer arithmetic (linear and some support for non-linear), arrays, records, and bit-vectors. Thanks to this power of CVCL, most atomic operations in C can be reasoned about in symbolic execution. ALERT API indicates CVCL how to perform the symbolic execution depending on a given argument which is determined in the preprocess stage. We will describe how the symbolic execution works at those operators.

Arithmetic:

The basic operators of addition, subtraction and multiplication can be immediately reasoned about in real and integer in CVCL, but modulo and division are only in integer. We use the arithmetic formula: the dividend equals a divisor multiplied by a quotient plus a remainder to set up the constraints in those operations of modulo and division. Therefore the operators of modulo and division can also be reasoned about in real. As for decrement, it is translated to the operation that subtracts the operand with one and then assigns the result to itself in preprocess stage. Actually we do not need to handle the decrement, and so does the increment.

A difference of arithmetic operations of signed values between C and CVCL is that CVCL uses a theory of bit vector to evaluate to the result of an operation, and it differs a little from C. For example, -128 / -1, the result we know is 128, and so does in CVCL. But in C the result is actually -128 because -128 / -1 can be viewed as negation of -128. C uses 2's complement to evaluate the negative value of -128, and then -128 / -1 is still -128 (Figure 8). Therefore we have to implement the real behavior of negation in C.

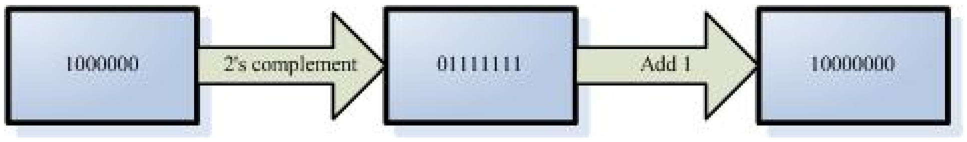


Figure 8: The evaluation of -128 / -1 in C

Assignment:

These all perform an operation on the lvalue and assign the result to the lvalue. However they have been translated to the 3-address code after simplification (Figure 9).

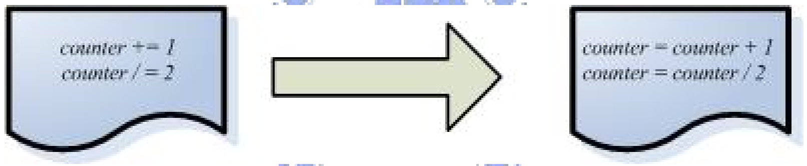


Figure 9: Three-address code translation

So we merely perform the symbolic execution as usual. CVCL provides an API, eqExpr, for the equality of two expressions. This is convenient to build the constraint, but the premise is that the two expressions must be the same type. Simply speaking, the length of bit vector of these expressions should be the same, or the type checking in CVCL will raise an exception.

Logical:

Most of logical operators are used as predicates in branch statements, which control the flow of a program towards different paths. CVCL originally has a powerful capability on such operations, and what we need to do is to maintain the constraints in different branch statements. Whenever the constraint system negates the last predicate of a branch to generate a next input, the constraints in the indicated branch statements require to be removed first. Otherwise those constraints will be misused and induce an anomaly path, which is not expected.

Bitwise:

Shift operations in CVCL are originally designed for bit vector, so the semantics of these operators are not the same as those in C. Left shift is to append successive zeros after the LSB to let the bit vector move toward left. Here is an example of left shift:

0bin0011 << 3 = 0bin0011000

Obviously that does not match in C because the bit length increases after left shift operation. For that reason we need to choose significant bits properly to show the result, so the bits should be extracted with original length from the LSB.

As for right shift operation, we need to take the sign of a bit vector into consideration because the shifted bit vector differs in signed and unsigned types. The original right shift operation in CVCL is like in C. For example,

0bin1100 >> 2 = 0bin0011

However for signed type that will change the MSB and make negative become positive. To overcome this problem, we first execute a signed extension on the signed bit vector, and then extract significant bits from the extended bit vector with correct length (Figure 10).

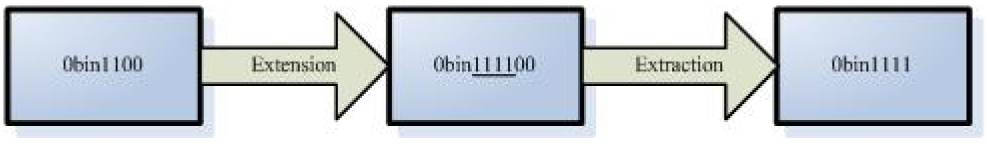


Figure 10: Right shift procedure in symbolic execution

Odds and ends:

All of these operations will be evaluated alone and the results will be assigned to a temporary variable created in the CIL preprocess stage. Then the variable will replace the original operation in instructions. Most operations can be evaluated to a concrete values at run-time, e.g. sizeof(), and are transparent in the constraints system. Only the pointer operation is related to symbolic execution because the test input originally can view as a logic memory map. All memory access operations should take into consideration during symbolic. Next section will detail how to implement them.

4.1.2.3 Symbolic Pointer

All variables used in the tested program are maintained in an object list, which records the name, starting address and size of the object. This list makes our symbolic execution support for pointer alias. Although the names of alias pointers are different, they have the same name in a object list if their starting addresses are equal. Furthermore this list help to check if a memory access is legal. We implement the CRED based memory access, and perform it before every memory reference and dereference. This execution is to calculate the range of the corresponding object and to verify whether the memory access is out of bound (Figure 11). If the address it points to is less than the lower bound or greater than the upper bound, an exception will raise. Because this checking is through our constraint system, its overhead is larger than CRED, which uses directly dynamic checking.

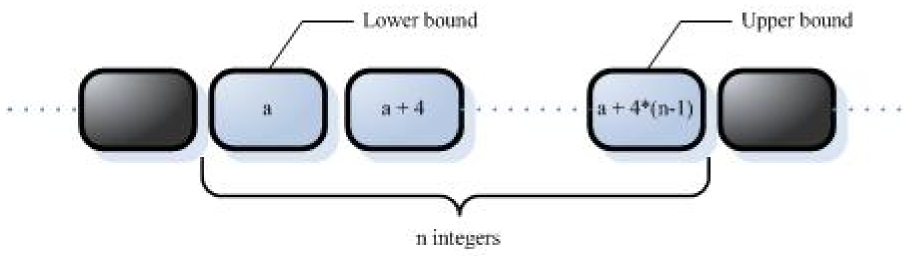


Figure 11: CRED based memory access

The symbolic memory access has three main cases: (1) a symbolic pointer with a concrete offset; (2) a symbolic pointer with a symbolic offset; (3) a concrete pointer with symbolic offset. First case is easy because only one symbolic variable is referenced. The main issue is induced by the symbolic offset since we can not determine what memory location it pointers to in our constraint system. A possible solution proposed in EXE is to interpret the memory access with a disjunction of accessing each location, and we have implemented it in our work. As we know, the addressing space is up to 4 GB in IA32 system, so it is not efficient to interpret an uninitialized pointer. We only handle those which have a limited size in the object list. Consider the example: *p* = **tmp*. The operation is to dereference *tmp* and then to assign its value to *p*. Assume *tmp* is symbolic, the object *tmp* points to is *k* (it can be determined dynamically), and the size of object is *n*, then the value of **tmp* could be **k*, *(*k*+1), *(*k*+2), …, and *(*k*+n-1). Therefore the symbolic read operation is interpreted as $\bigvee_{0 \le i < size(k)} tmp = i \wedge p = *(k+i)$. The symbolic write operation is similar.

4.1.3 Test Driver

This component is responsible for driving the testing procedure. It is merely written as a shell script which invokes the instrumented tested program and feedback the generated input at the end of program in order to drive another execution. Besides, it provides a simple interface to set up some preferences, like depth, iteration, etc.

We maintain a path history of last execution which marks what statements it chose in the passed branches. Because this work uses the systematic search algorithm to sweep out all feasible paths, test driver can determine if this testing should be terminated through this record. There are two cases for normal termination: (1) all paths are traversed (within the bounded depth); (2) an anomaly path. First case happens when only one branch in the history. Second case is that the previous execution does not obey the resolved path. That may be caused by path explosion or errors of the constraint system. Another situation is that the procedure never stops due to the constraints that our solver can not resolve.

4.2 SP module

This module serves as a CIL add-on, and provides a convenient capability to add post-conditions of different functions. Users can easily write the constraints of post-condition with simple type format defined in CIL and instrument them in the preprocess stage. Or they can use our post library and instrument a stub function after the specified function in order to give the constraints.

C standard library is usually linked in our testing program. It is inconvenient to add the source code of the callee functions at each test. Worse, the testers usually leave the matter aside by means of concretization. Therefore we obey the specification of standard C to build several post-conditions of general functions. But only the usual behaviors of functions are represented in our post-conditions. For instance, the return value of the function *strcmp* is 0 if and only if the two strings are equal. If there has any different character between these two strings, the return value naturally is any integer except 0. Actually this constraint is sufficient to deal with most cases, and it certainly improves the path coverage comparing to just concretization.

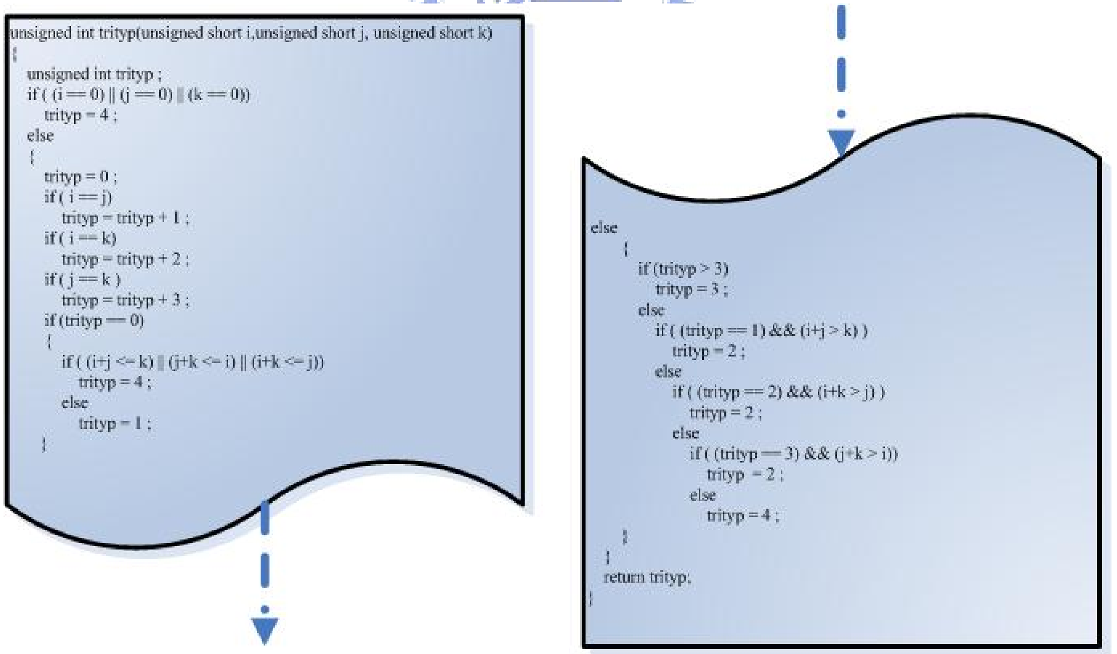
The constraints of post-conditions can be easily constructed using the implication, equivalence, and if-condition expression provided in CVCL. How to effectively interpret the semantics of external functions becomes the major work. Many extreme paths can be triggered on the premise that a branch predicate comprises some variable which is concerned with an external function.

5 Experimental Results

We experiment on ALERT with the well-known tritype program for classification of triangles. And then use two cases to illustrate how the post-condition helps to achieve better coverage for concolic testing.

5.1 A Simple Example: tritype

The *tritype* program is a basic benchmark in test case generation since some extreme paths are hard to trigger. Many random testing tools can not generate all paths, or need to take long time to complete it. *tritype* takes three positive integers as inputs (the triangle sides) and returns 1 if the inputs correspond to any triangle, 2 if the inputs correspond to an isosceles triangle, 3 if the inputs correspond to an equilateral one, 4 if the inputs do not correspond to any triangle. Figure 12 gives the *tritype* program in C code.



```
unsigned int trityp(unsigned short i,unsigned short j, unsigned short k)
{
  unsigned int trityp ;
  if ( (i == 0) || (j == 0) || (k == 0))
    trityp = 4 ;
  else
  {
    trityp = 0 ;
    if ( i == j)
      trityp = trityp + 1 ;
    if ( i == k)
      trityp = trityp + 2 ;
    if ( j == k )
      trityp = trityp + 3 ;
    if (trityp == 0)
    {
      if ( (i+j <= k) || (j+k <= i) || (i+k <= j))
        trityp = 4 ;
      else
        trityp = 1 ;
    }
```

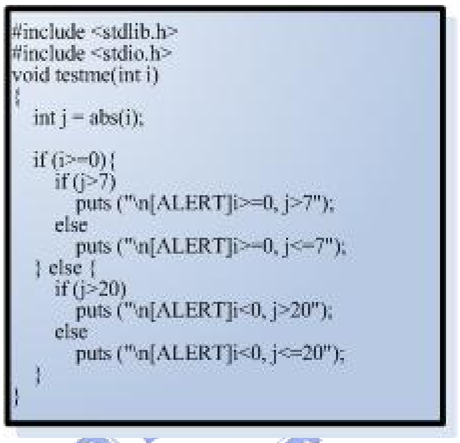
```
else
    {
      if (trityp > 3)
        trityp = 3 ;
      else
        if ( (trityp == 1) && (i+j > k) )
          trityp = 2 ;
        else
          if ( (trityp == 2) && (i+k > j) )
            trityp = 2 ;
          else
            if ( (trityp == 3) && (j+k > i))
              trityp  = 2 ;
            else
              trityp = 4 ;
    }
  }
  return trityp;
}
```

Figure 12: Code of the tritype program

We use this program to validate the capability of a test generation in ALERT. At beginning, ALERT use a default input (0, 0, 0), and then drives this execution to go through all feasible paths. Finally ALERT terminates at 14 iterations, which is the same as the number of paths that CUTE generates. In fact, there are only 14 different paths in the program. The result shows that our systematic test generation can not only enumerate feasible paths, but also avoid the repeated path. It confirms that our tool can exactly generate all inputs data for testing.

5.2 Post-condition: abs

Consider the function *testme* whose code is as follows:



```
#include <stdlib.h>
#include <stdio.h>
void testme(int i)
{
   int j = abs(i);

   if (i>=0) {
      if (j>7)
         puts ("\n[ALERT]i>=0, j>7");
      else
         puts ("\n[ALERT]i>=0, j<=7");
   } else {
      if (j>20)
         puts ("\n[ALERT]i<0, j>20");
      else
         puts ("\n[ALERT]i<0, j<=20");
   }
}
```

Figure 13: Simple program including a library call abs

We use CUTE to test this program to observe the path coverage. The result reports that there are only 2 paths in this program, those are (i >= 0)∧(j <= 7) and (i >= 0)∧(j <= 7). However if we input -50, (i < 0)∧(j > 20) can be induced. Actually another path (i < 0)∧(j <= 20) is also missed in CUTE. This problem is caused by the external library call *abs*. Constraints propagation of i stopped at the invocation of *abs* due to the concretization.

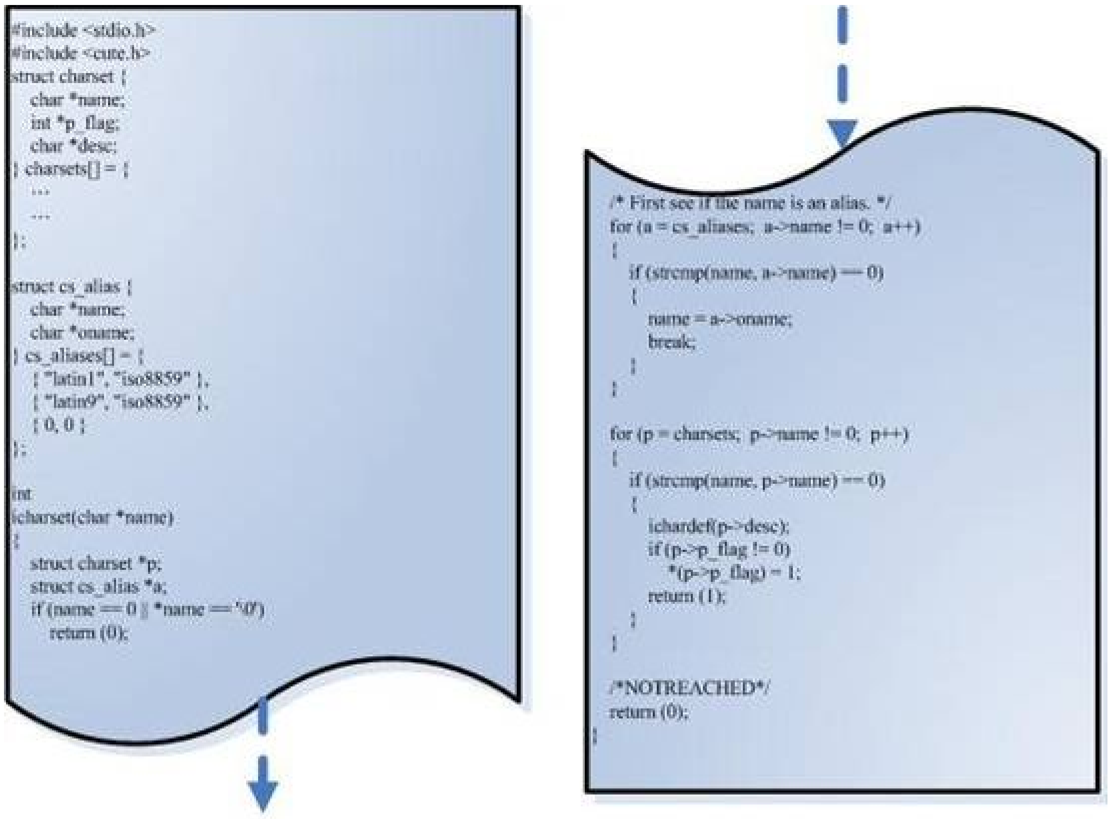
Supported by post-condition of *abs*, our tool can effectively enumerate all paths of this program. What we do is to add the post-condition:

$$\begin{cases} j = -i, & if\ i < 0 \\ j = i, & if\ i >= 0 \end{cases}$$

Through this rule, ALERT can cover four paths.

5.3 Post-condition: strcmp

The following program *icharset* is a function (Figure 14) used in the well-known command *less* that displays text files. It checks if a given character set is valid. We apply the post-condition of *strcmp()* on this test, and observe the result.



```
#include <stdio.h>
#include <cute.h>
struct charset {
   char *name;
   int *p_flag;
   char *desc;
} charsets[] = {
   ...
   ...
};

struct cs_alias {
   char *name;
   char *oname;
} cs_aliases[] = {
   { "latin1", "iso8859" },
   { "latin9", "iso8859" },
   { 0, 0 }
};

int
icharset(char *name)
{
   struct charset *p;
   struct cs_alias *a;
   if (name == 0 || *name == '\0')
      return (0);
```

```
/* First see if the name is an alias. */
for (a = cs_aliases;  a->name != 0;  a++)
{
   if (strcmp(name, a->name) == 0)
   {
      name = a->oname;
      break;
   }
}

for (p = charsets;  p->name != 0;  p++)
{
   if (strcmp(name, p->name) == 0)
   {
      ichardef(p->desc);
      if (p->p_flag != 0)
         *(p->p_flag) = 1;
      return (1);
   }
}

/*NOTREACHED*/
return (0);
}
```

Figure 14: Simple program including a library call strcmp

For the convenience of manual analysis, we assume that the first string is symbolic and the other is concrete, and the length of these strings is *n*. Intuitively we may build a relation between arguments and the return value in *strcmp()* as follows:

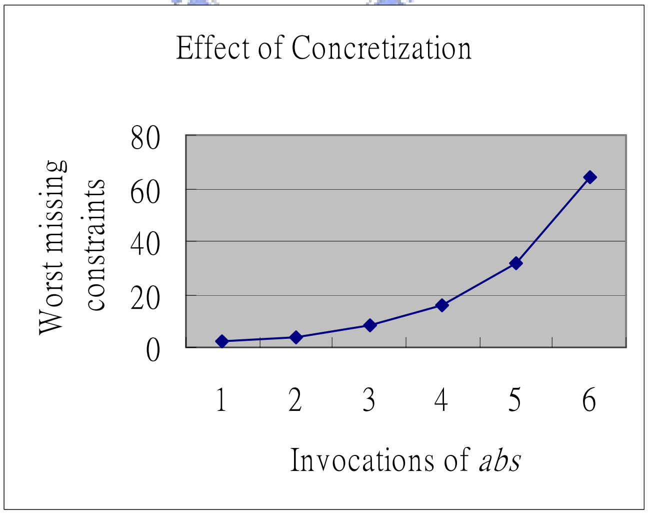
$$\boxed{\begin{array}{l} *str1 = *str2 \vee *(str1+1) = *(str2+1) \vee \cdots \vee *(str1+n-1) = *(str2+n-1) \\ \quad iff \quad strcmp\ (str1, str2) = 0 \end{array}}$$

Whenever the constraint system need to negate *ret = 0* for next test input, this will effect the related symbolic variables, for example, *ret != 0* implies **str1 - *str2 != 0*. Therefore at the next run *str1* and *str2* differ in the first character. The difficulty is how to construct the relation when *strcmp(str1,str2) != 0.* We use the method mentioned in SP module to build the constraint. Our experiment is to demonstrate the utility of the post-condition aided symbolic execution. CUTE consumes 21 iterations in the function *strcmp* with source code available while ALERT consumes only 2 iterations without source code.

5.4 Discussion

It seems that with these results of above cases, with less iterations path coverage can be improved using post-condition aided execution. As for a large program, the invocations of external functions become frequent. And this follows the increase of missed constraints (Table 3). Hence, the effect of post-condition aided execution becomes apparent. In addition, we have not analyzed *strcmp()* well, or the result will be more notable though this analysis could be complex. If this analysis is finished, it can be reused extensively in other testing.

Table 3: The effect when the times of calling abs in increase



We summarize the differences between CUTE and ALERT with post condition in the following table:

	Logical Input Map	Constraint Solver	External Invocation	Symbolic Pointer
CUTE	Byte level	lpsolver	Concretization	Only dereference
ALERT with post condition	Bit level	CVCL	Post-condition	Reference and Dereference

Table 4: Comparison between CUTE and ALERT

6 Conclusions

We address a post-condition aided symbolic execution to redeem the missing constraints due to concretization. Even though this method employs manual analysis, there is no suitable solution proposed yet. Most related works are available based on the assumption that the source codes of all involved function. Generally speaking, the condition is hardly satisfied, especially in COTS components. Recent research uses the binary instrumentation technology to perform the symbolic execution without source code. However, the symbolic execution in that work is extremely restricted by the size of program because the granularity in symbolic execution is too fine to the assembly language level. Moreover, binary instrumentation is difficult to be applied on system calls unless it has instrumented the kennel carefully. In fact, we can avoid the code in kernel level, and perform the effect approximately in symbolic execution using well-designed constraints.

Post-condition is interpreted from the semantics of a function, and it just gives the abstraction of the function. That is why manual analysis can be easily applied. The primary drawback is less precise. We therefore take advantage of the return value to refine the constraints. And the method can indirectly solve the path explosion problem because of its path compaction. SMART and LATEST aim at the same problem and have more precision than our work owing to the help of source code. Those methods can not be applied on un-instrumented programs.

Although manual analysis is necessary for post-condition, the generated constraints can be reused. Our method does not consume any extra overhead in testing because it can be analyzed off-line. The experiment shows that path coverage indeed can be improved and path explosion problem can be alleviated using post-condition aided symbolic execution. The future work is to construct a set of post-conditions which includes most popular functions in libraries. That will provide our test framework more capability in real program.

References

[1] 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.

[2] W. Visser, C. S. reanu and S. Khurshid, "Test input generation with java PathFinder," in *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis,* 2004, pp. 97-107.

[3] W. Visser, C. S. reanu and R. nek, "Test input generation for java containers using state matching," in *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis,* 2006, pp. 37-48.

[4] Tao Xie, Darko Marinov, Wolfram Schulte and David Notkin ER -, *Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution.* 2005, pp. 365-381.

[5] 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.

[6] 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.

[7] C. Cadar and D. E. E. -, *Execution Generated Test Cases: How to make Systems Code Crash itself.* 2005, pp. 2-23.

[8] 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.

[9] C. Csallner and Y. Smaragdakis, "Check 'n' crash: Combining static checking and testing," in *ICSE '05: Proceedings of the 27th International Conference on Software Engineering,* 2005, pp. 422-431.

[10] C. Csallner and Y. Smaragdakis, "Check 'n' crash: Combining static checking and testing," in *ICSE '05: Proceedings of the 27th International*

Conference on Software Engineering, 2005, pp. 422-431.

[11] P. Godefroid, "Compositional dynamic test generation," in *POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* 2007, pp. 47-54.

[12] R. Majumdar and K. Sen, "LATEST : Lazy dynamic test input generation," EECS Department, University of California, Berkeley, Mar, 2007.

[13] Huey-Der Chu, John E. Dobson and I-Chiang Liu ER -, "FAST: a framework for automating statistics-based testing," *Software Quality Journal,* vol. V6, pp. 13-36, 03/01/. 1997.

[14] Saswat Anand, Corina S. PÄƒsÄƒreanu and Willem Visser ER -, *Symbolic Execution with Abstract Subsumption Checking.* 2006, pp. 163-181.