國立交通大學

資訊科學與工程研究所

碩 士 論 文

操 作 符 號 位 址 以 產 生 異 常 執 行 路 徑

Exploiting Symbolic Locations for Abnormal Execution Paths

研 究 生：林孟緯

指導教授：黃世昆　教授

中 華 民 國　一 百 年 九 月

操 作 符 號 位 址 以 產 生 異 常 執 行 路 徑

Exploiting Symbolic Locations for Abnormal Execution Paths

研 究 生：林孟緯　　　　Student：Meng-Wei Lin

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

September 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年九月

操 作 符 號 位 址 以 產 生 異 常 執 行 路 徑

學生：林孟緯　　　　　　　　　　指導教授：黃世昆 老師

國立交通大學資訊科學與工程學系（研究所）碩士班

摘要

程式開發者不能完全避免因疏忽造成的漏洞，因而如今軟體安全是一個重要的議題。擬真測試是一項典型的自動化軟體測試技術，藉由實體測試與符號測試之間的結合交互作用，擬真測試可以達到較高且較精確的程式碼檢測率。但在擬真執行時，若發現路徑條件表示式含有符號型態的位址，它將無法掌握在相反路徑條件下所代表的實體數值，因而無法找到該相反路徑。本論文針對擬真符號測試，提出一個能增加程式碼檢測率的擬真位址模組，為了確保符號測試正常執行，我們暫時將含有符號位址的部分取代掉，並將替換的資訊記錄在符號位址表，最後我們透過路徑條件和符號位址表找出可能的位址解。我們目的為透過求解出來的符號位址，進入我們之前無法執行的路徑，如此一來我們將能提升程式碼檢測率，並找到更多程式錯誤。

Exploiting Symbolic Locations for Abnormal Execution Paths

Student：Meng-Wei Lin Advisors：Dr. Shin-Kun Huang

Department of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

The vulnerability caused by the negligence of the programmer is unavoidable. Software security is an important issue today. Concolic testing is a typical technique in automatic software testing. It achieves high coverage and precise analysis by combining concrete and symbolic execution in a co-operative way. But it cannot handle the situation when the address is symbolic in the path condition, so concolic executer may not find a concrete value which represents the test case of another negated path. This thesis proposes symbolic address module for enhancing the coverage of concolic testing. We use a substitute method to ensure symbolic executor running correctly and construct a symbolic address map to record symbolic address information. According to map information and path conditions, we generate a possible answer for symbolic addresses. We aim to find symbolic address solutions to enter abnormal paths we had never executed before. Then we can find more bugs by improving the code coverage.

誌謝

首先感謝父母在我背後的支持與鼓勵，讓我在求學生涯能無後顧之憂。接著誠摯感謝指導教授 黃世昆老師，老師這兩年來細心的教導讓我更瞭解軟體品質這個領域，在求學過程中不斷碰到看似無法跨越的障礙，但在老師的啟發與鼓勵之下，各種困難逐步迎刃而解，更使得我在這些年中獲益匪淺。感謝口試委員田老師、孔老師以及師母，點出論文中有所不足的部分，讓我更全面的掌握主題及其完整性。

兩年來的日子裡，實驗室裡的點點滴滴，感謝世欣、佑鈞不厭其煩幫我解決學業上的問題，感謝博彥在課業上的互相扶持，感謝翰霖、韋翔、俊維、基傑、偉明、奕任的共同砥礪。最後，拜實驗室 IKEA 雙層床架之賜，讓我得以在緊張的考試前夕獲得片刻的休憩。要感謝的人有很多，無法一一列舉，真的很感謝大家，讓我順利走過這段研究生的日子。

Table of Contents

List of Figures

List of Tables

1. Introduction

Software testing is the process that assuring program quality is identical with our expectation. In the development of complicated software, humanly programmer may miss some requirement or implement redundant functions. Both behaviors will lead to bugs or security problems.

The process of software testing is tedious and labor-consuming so manual testing is unfeasible. In recent years automatic software testing technique is mature gradually, there are many researches proposed to resolve the issues [17, 24, 23, 6, 16]. A typical testing technique named concolic testing [15, 20, 7]; it tests the software by combining concrete and symbolic execution[12] in a co-operative way. This method is feasibility on real program unit. But if the path constraint has the address which is symbolic, concolic executor cannot find the suitable real address solution for negated path constraint and it will abort this negated path.

Our work is base on concolic testing; we propose a new testing feature named symbolic address module. We aim to exploit symbolic address solutions and improve the path coverage. Then we can enter abnormal paths that we had never entered before.

1.1 Background

In the recent years, software security is a serious issue. Because of humanly software testing is not efficient, it's important to use the automatic tool to inspect software for vulnerability likes buffer overflow[10, 14, 22]. We focus on tainted base vulnerability; the overwritten data may cause unexpected behaviors. We describe major researches about our work below.

1.1.1 Common Vulnerabilities

- Stack-based Overflow

  A buffer overflow occurring in the stack memory is referred to as a stack overflow. A common case is that a local variable near the buffer but the program doesn't exam the buffer size. When we manipulate the buffer memory out of range, the local variable will be covered by our input. Not only local variable but we also possibly overwrite function pointer, exception handler or return address. Attacker may use those overwritten data to crash program or executing an unexpected instruction.

- Heap-based Overflow

  A buffer overflow occurring in the heap data area is heap overflow. Heap memory is dynamically allocated by program at run-time. Exploitation is to cause the program overwrite the memory management information which associated with heap memory such as dynamic memory allocation linkage. For example in BSD Phkmalloc, we can overflow metadata of malloc and then overwrite GOT entries or return address.

- Uninitialized Variables

  Uninitialized variable is a new declared variable which the program didn't set an initial value before using it. The value of uninitialized variable cannot be expected but it may tainted by other variable when two variable allocated in the same address range. Attacker can find a specific path to control the uninitialized variable and it may cause the vulnerability.

1.1.2 Program analysis policy

- Static analysis

  Static analysis is performed without actually executing programs. Instead, static analysis just scans the source code to gather information about the possible set of values, parses execution states of the program. It is usually implemented in formal methods such as data-flow analysis, model checking.

Static analysis tool used to detect vulnerability such as buffer overflow. We can check if there are dangerous standard library functions in source code such as *strcpy* and *fgets*. Unfortunately, the drawback of static analysis is high false positive; it cannot promise that all the found vulnerability will occur in actually executing programs.

- Dynamic analysis

Dynamic analysis actually executes the program and detects vulnerabilities at run-time. Such as valgrind[18], a tool for memory debugging and memory leaking analysis tool. It usually needs a large number of test cases and a software testing technique: code coverage observer to explore paths.

Dynamic analysis can promise that all the found vulnerability will occur during executing program. It is more precise than static analysis, but it also needs more time in executing analysis.

1.1.3 Program testing mechanism

- Random testing

Random testing is also named fuzzing; it is commonly used to test program security. It selects random inputs for target program and monitors if there is exceptions such as crashes occurred.

Fuzzing explores random paths very fast, but it wastes a large amount of time to enter the same path. The tool zzuf perform fuzzing testing on target program.

- Symbolic execution

Symbolic execution is useful for software validation because it can prove if the errors may occur. The main idea is to use the tracking symbolic for the input variable. It executes the program symbolically on symbolic variable. It collect symbolic path constrains and then uses constraint solver to explore execution paths. In the result, the explored paths represent by mathematical expressions. The drawback of symbolic execution is it does not scale for large programs because of there is too many execution paths.

- Concolic testing

Concolic testing combines random testing and symbolic execution in a co-operative way. It initializes the input variable with the symbolic variable. As program runs, it first chooses a random value to determine a path and collect the path constraints. In the next run, it negates the last path condition and feeds this new path conditions to the solver, and gets another concrete value which represent the test case of new path. This counterexample technique can be used to find next path conditions and available test case until all the paths is explored.

Concolic testing is focus on finding bugs in the real program. It has higher branch coverage than random testing and has no false positives or scalability problem like in symbolic execution.

1.1.4 Control the branch by symbolic address

In normal concolic execution, the symbolic variable refers to the tainted value. We explore paths with branch conditions which including symbolic variable.

Considering about the path condition has the tainted address, following are two specific types:

i. Constant_buf[symbolic index]=constant_value

ii. Symbolic_pointer[constant value] = constant_value

In type **i**, the base address is constant, but the array index is symbolic. The left of equation is a symbolic array index dereference. If we want to satisfy the condition, we have to find a specific address which it's dereference value is exact equal to right equation constant_value. In type ii, the pointer is symbolic. The left of equation is a symbolic pointer dereference. We can also find a specific address which it's dereference value is equal to right equation constant_value.

Above the first type we called symbolic array index, the second type we called symbolic pointer. Because of their address are symbolic variable, we say that's the symbolic address. Our thesis interest in symbolic address solution, we try to exploit symbolic locations for abnormal execution paths.

1.2 Motivation

Concolic testing is a popular software verification technique, it explore program paths as many as possible and find bugs. In figure 1, the buffer overflow occurs at *Line 6*. The address of pointer *p* is tainted by the standard input. In cocolic testing, the address of pointer *p* became a symbolic variable and it cannot determine the value of *p[0]*. In the situation we may miss the true path and we cannot find the vulnerability at *Line 8*.

```
1 #include <stdio.h>
2 void main()
3 {
4       int *p;
5       char buf[4];
6       fgets(buf,10,stdin);
7       if(p[0]==5)
8               vulnerability;
9       return 0;
10 }
```

Figure 1 the address of pointer **p** is tainded by **stdin**

If we can find a address for p that let the dereference of pointer p is properly 5, then we can enter an abnormal execution path and find the vulnerability.

1.3 Problem Description

For concolic testing, if the pointer or the array index is symbolic in branch condition, the executor doesn't know where to get the proper value from memory. The executor will execute incorrectly and then give up the path. But in some cases if we choose a suitable address for symbolic pointer or symbolic array index, the condition will be satisfied. Then we can enter this execution path.

In order to perform better coverage, we should construct a Symbolic Address Map for recording information of symbolic pointer and symbolic array index; we should add relation constraints into branch condition for exploring abnormal paths.

1.4 Objective

We focus on handling symbolic address to enter abnormal paths, and we can trigger more vulnerability in those paths which we had never entered before. To achieve these goals, we will try to implement two major objects on S^2E[5, 4]:

1. Symbolic Address Map: A table records symbolic addresses information and relationship between each symbolic address.
2. Symbolic Address constraints generator: A generator which generates relation constrains base on Symbolic Address Map for satisfying the abnormal path condition.

2. Related Work

Following tools specify and track symbolic variables and constraints; they fully explore program paths and find bugs. CRED[19, 11] (C Range Error Detector) directly checks the bound of memory accesses; it's just a bound checking tool. DART[9] (Directed Automated Random Testing) dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs. It mainly handles the integer constraints and invokes random testing with symbolic pointer. CUTE[21] is the first concolic testing tool which splintered from DART. It simulates the pointer into array, it can handle some symbolic pointer cases, but it cannot handle symbolic array index. CREST[1] is a concolic testing tool for C, it combine concolic testing with heuristic search strategies to perform high coverage on large software systems. It doesn't handle symbolic array index or symbolic pointer. EXE[3] can handle the more complex pointer access than CUTE. But it cannot handle multi-dimension dereference. It can handle symbolic array index but just in-bound related. SAGE[8] implements a new memory model for handling symbolic array index but just a bound checking tool. Catchconv is a symbolic execution and run-time integer conversion testing tool. It is a module of Valgrind and only focus on testing integer conversion error. SecTAC[25] is Trace-based security testing tool. Each trace is symbolically executed to produce program constrains and security constraints. Its trace can handle neither symbolic array index nor symbolic pointer. KLEE[2] is redesigned from EXE. It is a symbolic virtual machine built on LLVM[13] compiler infrastructure and uses search heuristics to reach high coverage in program. KLEE cannot handle symbolic array index or symbolic pointer. S2E is redesigned from KLEE, it provides the illusion of symbolic execution of an entire software stack, including applications, libraries, OS kernel, device drivers, and even firmware. It has guessing steps for symbolic address but not enough. Alert is developed by our laboratory; it used the memory model of EXE and the execution model of CUTE. It handles both symbolic array index and symbolic pointer, but it cannot handle out-of-bounds array index.

Our research base on S2E executor, we implement a plug-in for handling symbolic address. The comparison of above tools is shown as table 1.

Table 1 remaining symbolic variable

Tool	**Symbolic array index**	**Single-dimension symbolic pointer**	**Multi-dimension symbolic pointer**	**Out-of-bounds checking**
DART	X	X	X	X
Crest	X	X	X	X
Cute	X	O	O	X
Exe	O	O	X	X
Sage	O	O	O	X
KLEE	X	X	X	X
Alert	O	O	X	X
Hsin	O	X	X	O
Wei	O	O	O	O

3. Method and Steps

We provide a plugin out of box for enhancing path coverage by handling symbolic address problem, including symbolic array index dereference and symbolic pointer dereference.

S^2E executor inherited KLEE symbolic executor. In the original edition, when S^2E executor found an address is symbolic in constraints, it didn't know where to get the suitable value in the memory.

Figure 2 shows actual examples for symbolic array index and symbolic pointer for S^{2}E. S^{2}E executor will transfer the concrete value to the symbolic variable at *line 5*. At *Line 7*, because of the buf's address is symbolic, S^{2}E executor will try to assign concrete values to buf's address and fork states to solve the constraints. But in the most of the cases, the number of forking states always reached the maximum number of states to fork when concretizing symbolic value. Unfortunately the original edition failed to solve the symbolic address problem and can't reach *line 8*.

symbolic pointer:

```
1 int main()
2 {
3       int a=5;
4       int *buf;
5       s2e_make_symbolic(&buf, 4, "buf");
6       s2e_enable_forking();
7       if(buf[0]==5)
8               s2e_warning("GOAL");
9       s2e_disable_forking();
10      s2e_kill_state(0, "program terminated");
11      return 0;
12 }
```

symbolic array index:

```
1 int main()
2 {
3       int i, a=5;
4       int buf[3]={0,0,1};
5       s2e_make_symbolic(&i, 4, "i");
6       s2e_enable_forking();
7       if(buf[i]==5)
8               s2e_warning("GOAL");
9       s2e_disable_forking();
10      s2e_kill_state(0, "program terminated");
11      return 0;
12 }
```

Figure 2 symbolic pointer and symbolic array index

But in the left half side of figure 2, if we assign the address of *buf[0]* equal to address of *a*, we can pass the true branch and reach *line 8:"GOAL"*. In the right half side of figure 2, if address of *buf[i]* equal to address of *a (i=4)*, the program can also reach *line 8:"GOAL"*.

In our research, we add a new plug-in named *SymbolicAddress* for S^{2}E. We use a substitute method to ensure S^{2}E running correctly and construct a symbolic address map to record every symbolic address. When S^2E executor state terminating, we will check the state constraints and symbolic address map and then generate a possible answer. The following is detailed conception.

3.1 Symbolic Address classifications

Before solving the symbolic address problem, we have to define what is symbolic address? In our thesis, Symbolic address is classed as two main parts: symbolic pointer and symbolic array index.

Figure 3 shows the possible symbolic pointer classifications.

Example **i** shows a trivial symbolic pointer. Address of pointer *p* is symbolic and "*p[0] == concrete value*" is a condition in branch.

Example **ii** shows multiple symbolic pointers. We have to consider those different symbolic pointers dereference are adding together in a condition.

Example **iii** shows the pointer may have offset. The same pointer but different offset can be bind to different addresses in the memory. It means two different offset of the same pointer have different dereference values, but they have a fixed distance between them in the memory.

Example **iv** shows the pointer to pointer case. Of course, not only pointer to pointer, but also we have to consider triple or more. ex. ***p, *****p…

i. trivial symbolic pointer
   → *int *p and p = symbolic value*
   → *if(p[0] == concrete value)*

ii. extended from trivial
   → *int *p, *q, *r… and p, q, r… = symbolic value*
   → *if(p[0] + q[0] + r[0]… == concrete value)*

iii. multiple offsets of the same pointer address
   → *int *p and p = symbolic value*
   →*if(p[0] + p[1] + p[2]… == concrete value)*

iv. pointer to pointer
   → *int **p and p = symbolic value*
   → *if(p[0][1] == concrete value)*

Figure 3 symbolic pointer classifications

Figure 4 shows the possible symbolic array index.

Example **i** shows a trivial symbolic array index. Because of integer *i* is symbolic, then address of *buf[i]* is also symbolic. "*buf[i] == concrete value*" is a condition in branch.

Example **ii** shows multiple symbolic array indexes, different symbolic array indexes dereference are adding together in a condition.

Example **iii** shows two different base addresses have the same symbolic array index. They bind to different addresses in the memory and may have different values, but they have a fixed distance between them in the memory. Furthermore, The case maybe (*bufA[i] + bufB[i] + bufC[i]*) or more.

Example **iv** shows the multi-level symbolic array index. The entire size of *buf* is *(i × j × k)*. It means the different *(i, j, k)* may cause the *buf[i][j][k]* have the same address in the memory.

i. trivial symbolic array index
   → *int i and i = symbolic value*
   → *if(buf[i] == concrete value)*

ii. extended from trivial
   → *int i, j, k… and i, j, k… = symbolic value*
   → *if(bufA[i] + bufB[j] + bufC[k] … == concrete value)*

iii. multiple base addresses have the same symbolic array index
   → *int i and i = symbolic value*
   → *if(bufA[i] + bufB[i] == concrete value)*

iv. multi-level symbolic array index
   → *int i, j, k… and i, j, k… = symbolic value*
   → *if(buf[i][j][k]… == concrete value)*

Figure 4 symbolic array index classifications

3.2 Symbolic Address variable Substituting

S^2E executor should fork a new state and add a negate constraint to it when execute a branch condition. According to left half side of figure 2, we can get a state constraints diagram figure 5.

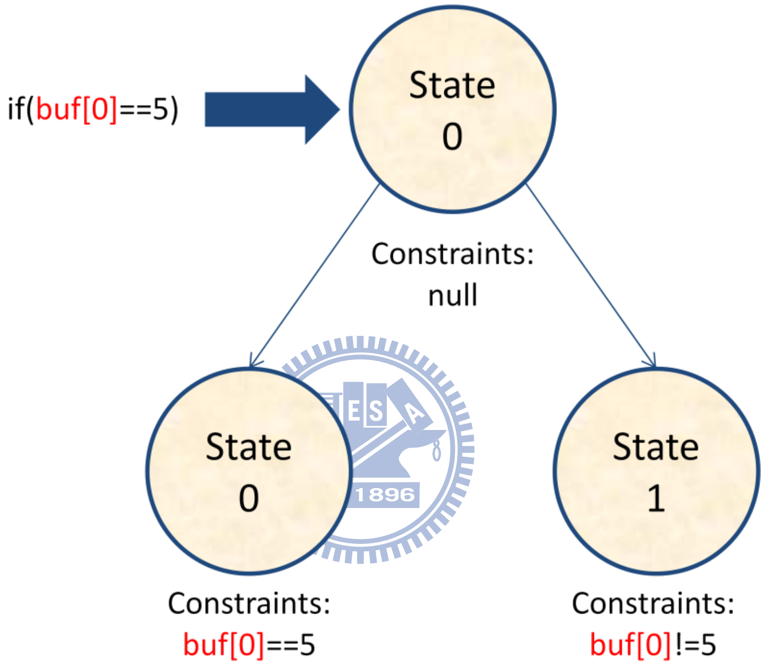


Figure 5 origin state constrains diagram

In the fact, KLEE executor cannot execute correctly when the branch condition has the symbolic address. Although S^2E executor inherited KLEE symbolic executor and has guessing steps to handle symbolic address, it still cannot allocate appropriate address in most of case. In this case, S^2E executor will fork states until reached the maximum number of states to fork when concretizing symbolic value, then fail to find a available address for *buf[0]*.

We have an idea for S^2E executor when found a symbolic address in state condition. We declare a new symbolic variable and then substitute the symbolic address dereference value. Figure 6 shows the state constraints diagram after the substitution. S^2E executor now can execute successfully.

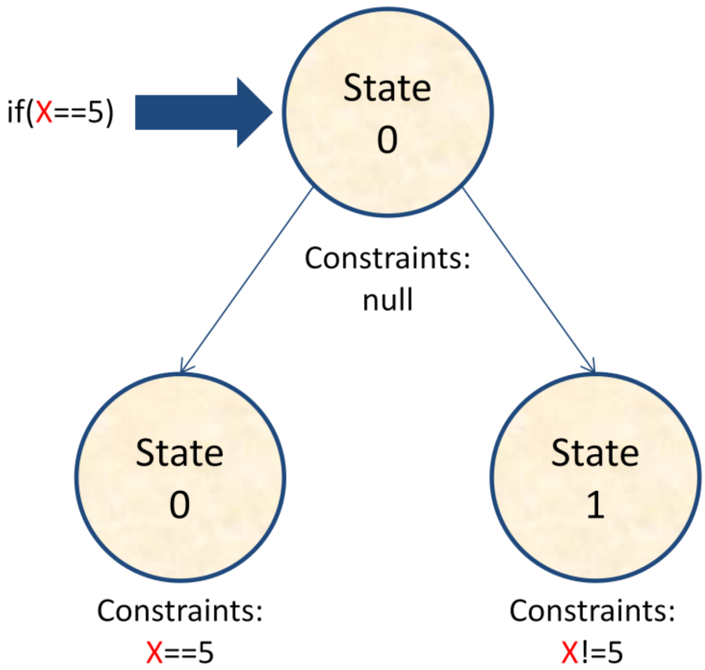
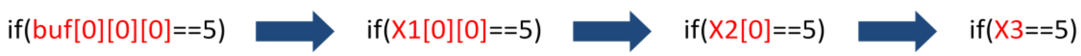


Figure 6 state constrains diagram after substituting

In addition, if *Line 4* in left half side of figure 2 is *"int ***buf"* and *Line 7* is *"if(buf[0][0][0]==5)"*, we have to operator the substitution three times.



Only substitution is not enough, we have to construct the relation sheep between our new made symbolic variables. We will explain the symbolic address map structure in next section.

3.3 Symbolic Address Map

Symbolic address map contains four basic elements: ***Origin Expression, Substituted Expression, Related Address and Target Address***.

i. ***Origin Expression*** is a symbolic address expression, symbolic executor doesn't know where to read it in the memory.

ii. ***Substituted Expression*** is a new declared expression used to substitute ***Origin Expression***. In addition, we add it to Symbolic Table and then Symbolic executor believes it is a symbolic variable. Now symbolic executor can execute it continually.

iii. ***Related Address*** is the same with the address of ***Origin Expression's*** symbolic variable in symbolic table. If two symbolic addresses have the same ***Related Address***, one of them may another one's dereference.

iv. ***Target Address*** is a blank space now. It used to store a concrete address which suits with state constraints.

Figure 7 shows how to construct the symbolic address map. If the branch conditions still have symbolic addresses, it adds symbolic address into the map recursively until there is no symbolic address.

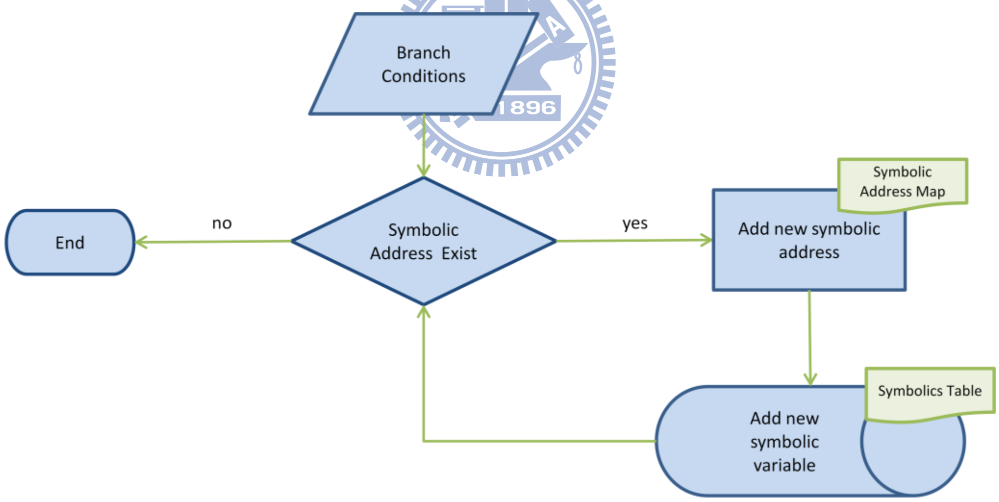


Figure 7 Flowchart of Constructing symbolic address map

Figure 8 is a symbolic address map example. Symbolic executor found a symbolic address in branch condition *(Line 13)*. We declared a new expression named *buf1*, its ***origin expression*** was *buf*, and the ***related address*** was the same with the address of *buf* in Symbolic Table. Besides, we added *buf1* into Symbolic Table. Until now, *buf1* represented the value at address *buf[1]* in memory. Because of *buf[1]* was symbolic, we had to declared a new expression named *buf2* and substitute *buf1*. In the end, *buf3* represented a concrete value at address *buf[1][3][5]* in memory. We finished substituting all the symbolic address in branch condition.

```
#include <stdio.h>
#include "s2e.h"

int main()
{
	int a;
	int b;
	int ***buf;
	s2e_make_symbolic(&a, 4, "a");
	s2e_make_symbolic(&b, 4, "b");
	s2e_make_symbolic(&buf, 4, "buf");
	s2e_enable_forking();
	if( buf[1][3][5]==5 )
		s2e_warning("GOAL");
	s2e_disable_forking();

	s2e_kill_state(0, "program terminated");
	return 0;
}
```

```
0xbffff918: (Read w8 0 buf)
0xbffff919: (Read w8 1 buf)
0xbffff91a: (Read w8 2 buf)
0xbffff91b: (Read w8 3 buf)
0xbffff91c: (Read w8 0 b)
0xbffff91d: (Read w8 1 b)
0xbffff91e: (Read w8 2 b)
0xbffff91f: (Read w8 3 b)
0xbffff920: (Read w8 0 a)
0xbffff921: (Read w8 1 a)
0xbffff922: (Read w8 2 a)
0xbffff923: (Read w8 3 a)
```

Symbolics Table

	Array->name	MemoryObject->address
1.	*a*	0xbffff920
2.	*b*	0xbffff91c
3.	*buf*	0xbffff918
4.	*buf1*	0xbffff918
5.	*buf2*	0xbffff918
6.	*buf3*	0xbffff918

Symbolic Address Map

Related	Origin	Substituted	Target
1. 0xbffff918	*buf*	*buf1*	*buf[1]*
2. 0xbffff918	*buf1*	*buf2*	*buf[1][3]*
3. 0xbffff918	*buf2*	*buf3*	*buf[1][3][5]*

Figure 8 symbolic address map example

3.4 Symbolic Address Constrains Generator

Our goal is finding ***Target Address*** for symbolic address map. We could use ***Target Address*** stored in symbolic address map to generate symbolic address constrains. In the end, we add symbolic address constrains into symbolic execution state, and then symbolic executor will automatically generate a test case for every symbolic address.

Figure 9 shows how to generate symbolic address solutions. If state constraints have any symbolic addresses before symbolic execution state terminated, we choose a combination of addresses from Symbolics Table and pass them into symbolic address map ***Target Address*** field. We use STP solver to identify if the relationships in Symbolic Address Map is satisfy all constrains in symbolic execution state. If answer is yes then we obtain a solution, otherwise we choose next combination addresses from Symbolic Table and use STP solver to identify again.

If all combination of addresses in Symbolic Table is not the solution, then we try to find it in the actual memory. As before, but we choose a combination of addresses from actual memory. If there is no solution in Symbolics Table or actual memory, we say that this path maybe is impossible in the program.

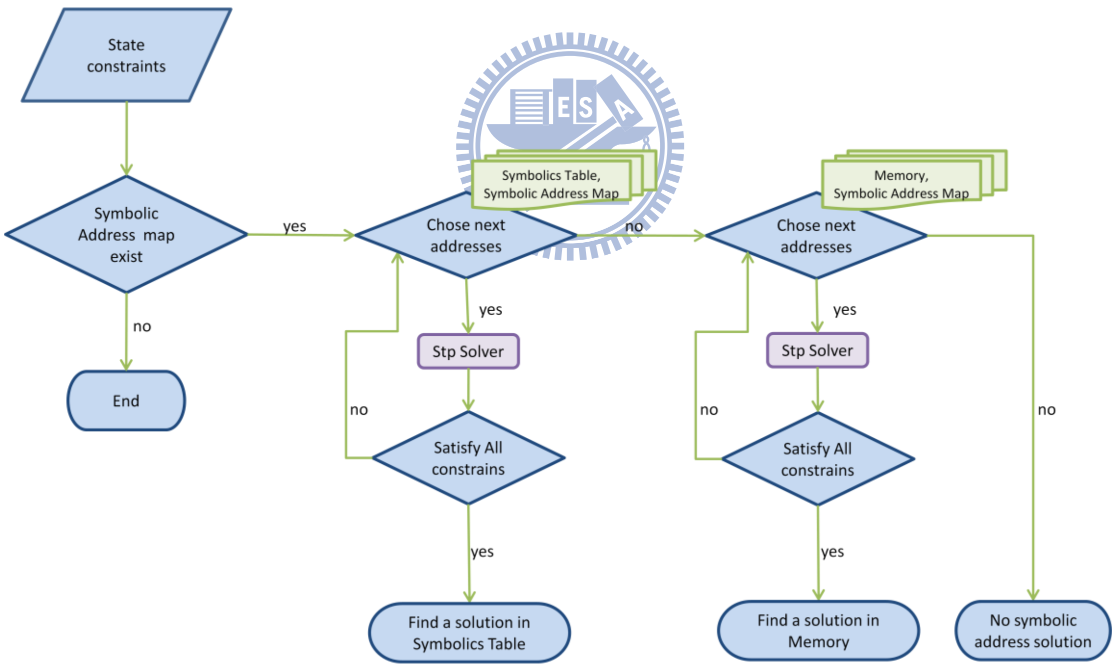


Figure 9 Flowchart of symbolic address solution

4 Implementation

S^2E provides the core symbolic execution engine. All the analysis is done by various plug-in. In this thesis, we write a plug-in named *SymbolicAddress* that uses features of the S^{2}E plug-in infrastructure.

We substitute symbolic pointer dereference and symbolic array index dereference during symbolic execution, and we add them into Symbolic Address Map. Symbolic Address Map describes what expression to be substitute and where address to be substitute.

Before S^{2}E execution state terminated, according to Symbolic Address Map we search available addresses in Symbolics Table or actual memory for every symbolic address. Finally, we add those available address relation constrains into S^{2}E execution state conditions, and the S^{2}E plug-in named *TestCaseGenerater* will generate an available test case for every symbolic variable automatically.

4.1 Symbolic Address Map & Class

As Figure 10 shows, Symbolic Address Map construct from symbolic addresses during symbolic executing.

```
Struct SApoint{
        uint64_t tempAddress;
        ref<Expr> tempExpr;
        const Array* tempArray;
        ref<Expr> targetExpr;
        const Array* targetArray;
        uint64_t targetAddress;
        ref<Expr> targetValueExpr;
};
```

Figure 10 Symbolic Address structure

tempAddress : ***Related Address***.

tempExpr : ***Substituted Expression***.

tempArray : the Aarray object used to store major variable name of *tempExpr*.

targetExpr : ***Origin Expression***.

targetArray : the Aarray object used to store major variable name of *targetExpr.*

targetAddress : ***Target Address***.

targetValueExpr : the content of *targetaddress* in actual memory.

```
Class SymbolicAddress {
Private:
        int SAcounter;
        std::vector<SApoint> SAmap;

public:
        void adjust(S2EExecutionState* state,
                        ref<Expr> expr);
        void solutionGEN(S2EExecutionState* state);
Private:
        bool searchSymbolicMap(S2EExecutionState* state,
                                vector<SApoint>& map);
        bool searchMemoryMap(S2EExecutionState* state,
                                std::vector<SApoint>& map,
                                uint64_t beginMap,
                                uint64_t endMap);
         bool chooseNextSymbolicAddress(S2EExecutionState* state,
                                            std::vector<SApoint>& map);
         bool chooseNextTargetAddress(std::vector<SApoint>& map,
                                         uint64_t beginMap,
                                         uint64_t endMap);
}
```

Figure 11 shows our implementation. *SAcounter* used to calculate how many symbolic addresses in SAmap. Function *adjust* doing the substitute stage when symbolic executor found the symbolic address. Function *solutionGEN* doing the solution searching stage and constrains generating stage at symbolic execution state terminating.

4.2 Symbolic Address Plug-in for S²E

S²E symbolic executor will call the function *handleForkAndConcretize* when instructions have expression. If the expression is constant or state->forkDisabled is on, it will simply pick one possible value and return. Otherwise, if the expression has symbolic address, it will run the guessing steps.

As figure 12, we instrument our function *adjust* to Instead the guessing steps. We pick the moment to substitute symbolic address when *handleForkAndConcretize* find the symbolic address.

$\mathcal{S}$: the current Execution State pointer

$\mathcal{E}$: the current Expr in the branch

S2EExecutor::handleForkAndConcretize ($\mathcal{S}$, $\mathcal{E}$, …){
    …
    If $\mathcal{E}$ is NOT a symbolic address{
        …
        return
    }
    …
    ~~*guessing steps*~~
    adjust($\mathcal{S}$, $\mathcal{E}$)
}

Figure 12 instrumenting function *adjust*

Before every symbolic execution state terminating, S²E handler will call the function *processTestCase*. This function will call the original author plug-in named *TestCaseGenerator*, it will generate an available value for each symbolic variable.

As figure 13, we instrument our function *solutionGEN* before calling the plug-in *TestCaseGenerator*. We search available address for each symbolic address and add related constrains between each other into S²E state constrains. In the end, plug-in

TestCaseGenerator will also generate an available value for each origin symbolic variable and our new made symbolic variable.

```
𝒮 : the current Execution State pointer

S2EHandler::processTestCase ( 𝒮 ){
    ...
    solutionGEN ( 𝒮 )      // add constrains before running s2e::plugins::TestCaseGenerator
    ...
    getPlugin("TestCaseGenerator")
}
```

Figure 13 instrumenting function *solutionGEN*

4.3 Symbolic Address adjust function

Figure 14 shows the pseudo code of function *adjust*. We use integer $\mathcal{N}$ to count how many symbolic addresses now. We also use the String *"SA"*+ *IntToStr*($\mathcal{N}$) as the new name for our new declared expression. *Line 8* to Line 12 will fill the new symbolic address with the substitute information. Now the struct member $\mathcal{P}$.*targetAddress* and $\mathcal{P}$. *targetValueExpr* will be blank, we will use them in other steps.

```
S : the current Execution State pointer
E : the current Expr in the branch
N : the number of symbolic addresses
A : the origin symbolic variable's address in memory
P : a SApoint struct used to store symbolic address information
NAME : a new symbolic variable used to Substitute symbolic address value
SAMAP : the vector used to store every SApoint information in the program

SymbolicAddress:: adjust(S , E ){
    N ← N + 1
    A ← symbolic variable's address in E
    NAME ← String "SA" + IntToStr( N )
    S->createSymbolicArray( A , 4 , NAME )

    P ← new SApoint
    P.tempAddress ← A
    P.targetArray ← findSymbolicObjects( E )->second
    P.targetExpr ← E
    P.tempArray ← findSymbolicObjects( NAME )->second
    P.tempExpr ← Expr:: createTempRead( P.tempArray , 32 )

    SAMAP.push_back( P )
}
```

Figure 14 function *adjust*

Figure 15 is a Symbolic Address Map example of figure 2. Both programs we only consider State 0 when executing true branch, and each state has one symbolic address.

	Left figure, State 0	right figure, State 0
tempAddress	0xbffff91c	0xbffff91c
tempExpr	ReadLSB w32 0 SA1	ReadLSB w32 0 SA1
tempArray	SA1	SA1
targetExpr	ReadLSB w32 0 buf	ReadLSB w32 0 i
targetArray	buf	i
targetAddress	null	null
targetValueExpr	null	null

Figure 15 example of Symbolic Address Map after adjusting

4.4 Symbolic Address solution generator function

Figure 16 shows the pseudo code of function *solutionGEN*. We have two searching algorithm: *searchSymbolicMap* and *searchMemoryMap*. First we search the Symbolics Table for symbolic address solution. If we didn't find an available solution, then we search the entire memory from address $\mathcal{BM}$ to address $\mathcal{EM}$. In our research, we define the region from $\mathcal{BM}$ to $\mathcal{EM}$ as addresses near the symbolic address in stack memory.

When we get an available solution for Symbolic Address Map, we add two constrains (*line 13 ~ 16*) into S^2E execution state. At *Line 13*, symbolic address expression $\mathcal{P}$->*targetExpr* should equal to concrete address $\mathcal{P}$->*targetAddress*. At *Line 14 ~ 15*, our new declared expression $\mathcal{P}$->*tempExpr* should equal to expression $\mathcal{P}$->*targetValueExpr* which address is $\mathcal{P}$->*targetAddress*.

```
S : the current Execution State pointer
B : a boolean variable will be TRUE if the state have a symbolic solution
SAMAP : the vector used to store every SApoint information in the program
BM : the begin of the memory
EM : the end of the memory
P : a SApoint pointer used to store symbolic address information
E : the current symbolic address constrains that should be add to the state

SymbolicAddress:: solutionGEN ( S ){
    If searchSymbolicMap( S , SAMAP ) is true{
        B ← True
    }else if searchMemoryMap(S , SAMAP , BM , EM) is true{
        B ← True
    }else{
        No solution!!
        B ← false
    }

    If B is true{
        for each P ∈ SAMAP {
            E ← EqExpr::create(P ->targetExpr, ConstantExpr::create(P ->targetAddress , 32) )
            E ← =AndExpr::create(E ,
                        EqExpr::create(P ->tempExpr, P ->targetValueExpr))
            S ->constraints.addConstraint(E )
        }
    }
}
```

Figure 16 function *solutionGEN*

4.4.1 Searching solutions from Symbolics Table

Figure 18 shows the pseudo code of function *searchSymbolicMap*. At *Line 2*, function *chooseNextSymbolicAddress* chooses a combination of addresses from Symbolics Table, and those addresses are set in each object *SApoint* member *targetValueExpr* where in Symbolic Address Map. We loop the function until found an available combination or all the combinations are unavailable.

At *Line 9~13*, we handle the case which object *SApoint* member *targetArray* is the same. We have two major cases:

i. Symbolic pointer: multiple offsets of the same pointer address.

if(p[0] + p[1] == concrete value), p is a symbolic pointer.

ii. Symbolic array index: the same symbolic array index with multiple bases.

if(bufA[i] + bufB[i] == concrete value), i ,j are symbolic array index

We can solve the problem by adding a constraint: the offset between two object *SApoint* member *targetArray* should equal to offset between two object *SApoint* member *targetExpr*.

At Line 14~18, we add a constraint to handle the pointer to pointer problem:

*int **p, p is a symbolic pointer, if(p[1][2] == concrete value)*

If X and Y are both the symbolic addresses and Y adjust from X. Figure 17 shows the dereference relationship between X and Y. The value at ***X->targetAddress*** in memory should equal to which subtract the offset of ***Y->targetExpr*** from ***Y->targetAddress***. Moreover, this method can handle not only the pointer to pointer, but also ***p, ****p… or more situation.

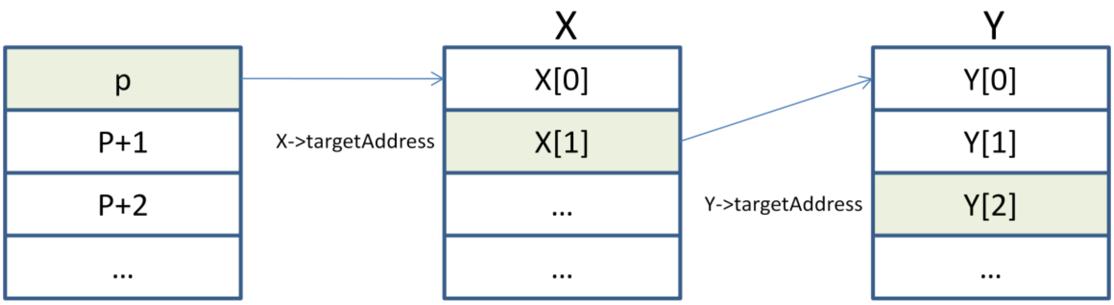


Figure 17 p[1][2] == concrete value

S : the current Execution State pointer

SAMAP : the vector used to store every SApoint information in the program

P : a SApoint pointer used to store symbolic address information

P' : a SApoint pointer used to store symbolic address information

E : the current address constraints used for solver

SUC : a Boolean variable used to determin

```
searchSymbolicMap( S , SAMAP ){
    while chooseNextSymbolicAddress( S , SAMAP ) is true{
        E ← NULL
        for each P ∈ SAMAP {
            E ←AndExpr::create (E ,
                    EqExpr::create(P ->targetExpr, ConstantExpr::create(P ->targetAddress , 32) ))
            E ← AndExpr::create(E ,
                    EqExpr::create(P ->tempExpr, P ->targetValueExpr))
            for each P' ∈ SAMAP && P'->targetArray->name is equal P ->targetArray->name{
                E ← AndExpr::create(E ,
                        EqExpr::create(ConstantExpr::create((P ->targetAddress)-( P' ->targetAddress) , 32),
                        SubExpr::create(P ->targetExpr, P' ->targetExpr)))
            }
            for each P' ∈ SAMAP && P'' is adjust from P' {
            E ← AndExpr::create(E ,
                    EqExpr::create( S ->readMemory( P'->targetAddress , 32 ),
                    ConstantExpr::create( (P'' ->targetAddress)-(offset of P'' ->targetExpr), 32))
            }
        }
        solver->mayBeTrue(Query( S ->constraints , E ) , SUC )
        if SUC is true{
            for each P ∈ SAMAP {
                P ->targetValueExpr ← S->readMemory( P ->targetAddress , 32 )
            }
            return TRUE
        }
    }
    return FALSE
}
```

Figure 18 function *searchSymbolicMap*

4.4.2 Searching solutions from actual Memory

The function *searchMemoryMap* is almost the same with *searchSymbolicMap*. The only different is *Line 2* in figure 18; instead of *chooseNextSymbolicAddress* we use the function *chooseNextTargetAddress*. We search a range of actual memory, including concrete value and symbolic variable. In the fact, if we define the region as entire memory including text, data, heap and stack, it will take a large amount of searching time. In our implement, we search near the memory region in stack which near the symbolic address. We have the best opportunity to find available addresses in this region.

In addition, if the number of origin symbolic variables is more than the number of symbolic address we new declared. We do not need to search anywhere. We set each origin symbolic variable address to each *SApointer* member *targetAddress*. It will be a symbolic address solution.

5 Result and Experiment

We present results of experiments and prove symbolic address module in this section. We use the example of SAGE and our made programs to illustrate the solution of symbolic array index and symbolic pointer. Next, we discuss the efficiency of two searching algorisms when execution state has remaining symbolic variables. In the end, we illustrate the enhancement on path coverage with real programs.

5.1 A simple Example of SAGE

As shown in figure 19, we test the example of SAGE. The program has two symbolic variable x and y. Our goal is to reach the Line 15.

When reached line 14, we executed the true branch first. We found 2 symbolic array indexes in the state constraint. Then we new declared symbolic variables x1 and y2, x1 is the dereference value of buf[x], y2 is the dereference value of buf[y]. We updated our symbolic address map and adjusted the state constraint to

(Eq (ReadLSB w32 0 x1)

(Add w32 2 (ReadLSB w32 0 y2)))

For generating symbolic address solution, we searched a combination of addresses in stack memory. Later, we discovered when x equal 3 and y equal 1, then x1 equal 2 and y2 equal 0, and this was a combination of addresses solution which satisfied the state condition. According to symbolic address map, we added related constrains into the state constraint. In the end, plug-in *TestCaseGenerater* automatically generated a test case for symbolic variables which including our new declared. The false branch symbolic address solution is also generated in the same way.

```
#include <stdio.h>
#include "s2e.h"
int main()
{
	int x,y;
	s2e_make_symbolic(&x, 4, "x");
	s2e_make_symbolic(&y, 4, "y");
	int buf[4];
	buf[0]=x;
	buf[1]=0;
	buf[2]=1;
	buf[3]=2;
	s2e_enable_forking();
	if( buf[x]==buf[y]+2 )
		s2e_warning("GOAL");
	s2e_disable_forking();
	s2e_kill_state(0, "program terminated");
	return 0;
}
```

```
compared 23 times
Found a solution in memory.
state constraint:
(Eq (ReadLSB w32 0 x1)
    (Add w32 2
             (ReadLSB w32 0 y2)))
Symbolic address solution:
TestCaseGenerator: processTestCase of state 0 at address 0x8048477

x: 03  00  00  00
y: 01  00  00  00
x1: 02  00  00  00
y2: 00  00  00  00
```

```
compared 1 times
Found a solution in memory.
state constraint:
(Eq false
    (Eq (ReadLSB w32 0 x1)
        (Add w32 2
                 (ReadLSB w32 0 y2))))
Symbolic address solution:
TestCaseGenerator: processTestCase of state 1 at address 0x8048477

x: 00  00  00  00
y: 00  00  00  00
x1: 00  00  00  00
y2: 00  00  00  00
```

Figure 19 SAGE

5.1.1 More Complicated Symbolic Array Index

In this sub section, we evaluate the test case as shown in figure 20, which has more complicated symbolic array index. This experiment focus on two major classifications for branch condition:

$$if\ (\ bufA[i][j][k]\ +\ bufB[k]\ +\ bufC[l]\ ==\ 10\ )$$

1. Multi-dimension symbolic array index

   → bufA[i][j][k] is a 3-dimension symbolic array index

2. Different base addresses have the same symbolic array index

   → bufA[i][j][k] and bufB[k] have the same symbolic array index k

The program has four symbolic variable i, j, k and l. Our goal is to reach the Line 16.

As our expectation, the true branch symbolic address solution:

$$bufA[1][0][2]\ +\ bufB[2]\ +\ bufC[1]\ =\ i1+\ k2\ +\ l3 = 5 + 2 + 3 = 10$$

In the false branch, because of bufA[0][0][0] is an uninitialed value, so its value is 0xb7ffhf68.

```c
#include <stdio.h>
#include "s2e.h"
int main()
{
        int i,j,k,l;
        s2e_make_symbolic(&i, 4, "i");
        s2e_make_symbolic(&j, 4, "j");
        s2e_make_symbolic(&k, 4, "k");
        s2e_make_symbolic(&l, 4, "l");
        int bufA[2][3][4];
        int bufB[3]={0,0,2};
        int bufC[2]={0,3};
        bufA[1][0][2]=5;
        s2e_enable_forking();
        if(bufA[i][j][k]+bufB[k]+bufC[l]==10)
                s2e_warning("GOAL");
        s2e_disable_forking();
        s2e_kill_state(0, "program terminated");
        return 0;
}
```

```
compared 1312 times
Found a solution in memory.
state constraint:
(Eq 10
    (Add w32 (ReadLSB w32 0 l3)
             (Add w32 (ReadLSB w32 0 k2)
                      (ReadLSB w32 0 i1))))
Symbolic address solution:
TestCaseGenerator: processTestCase of state 0 at address 0x80484f0

i: 01  00  00  00
j: 00  00  00  00
k: 02  00  00  00
l: 01  00  00  00
i1: 05  00  00  00
k2: 02  00  00  00
l3: 03  00  00  00
```

```
compared 74 times
Found a solution in memory.
state constraint:
(Eq false
    (Eq 10
        (Add w32 (ReadLSB w32 0 l3)
                 (Add w32 (ReadLSB w32 0 k2)
                          (ReadLSB w32 0 i1)))))
Symbolic address solution:
TestCaseGenerator: processTestCase of state 1 at address 0x80484f0

i: 00  00  00  00
j: 00  00  00  00
k: 00  00  00  00
l: 00  00  00  00
i1: 68 hf6 �ff �b7 �
k2: 00  00  00  00
l3: 00  00  00  00
```

Figure 20 program 1

5.1.2 Symbolic Pointer classifications

We evaluate the test case as shown in figure 21, which includes all symbolic pointer classifications. This experiment focus on two major classifications for branch condition:

if (pA[3] + pA[4] + pB[0][1][2] == 10)

1. Multi-dimension symbolicpointer

    → pB[0][1][2] is a 3-dimension symbolic pointer

2. different offsets of the same pointer address

    → pA[3] and pA[4] is the same pointer but different with offset

The program has a single-dimension pointer pA and a 3-dimention pointer pB. Our goal is to reach the Line 14.

In the true branch, pA1 is the dereference value of pA[3] and pA2 is the dereference value of pA[4]. Because of pB[0][1][2] is a 3-dimension symbolic pointer, pB3 is the first-dimension dereference value, pB34 is the second-dimension dereference value which adjust from pB3, pB345 is the third-dimension dereference value which adjust from pB34, so pB includes three symbolic addresses. There are total five symbolic addresses in the path condition. The final state constraint is:

(Eq 10

(Add w32 (ReadLSB w32 0 pB345)

(Add w32 (ReadLSB w32 0 pA2)

(ReadLSB w32 0 pA1))))

Figure 22 shows the multi-dimension graph for symbolic pointer pA and pB. The true branch symbolic address solution:

$$pA[3] + pA[4] + pB[0][1][2] = pA1 + pA2 + pB345 = 3 + 5 + 2 = 10$$

In the end, we exploit this abnormal path success.

```
1 #include <stdio.h>
2 #include "s2e.h"
3 int main()
4 {
5       int a=2;
6       int b=3;
7       int c=5;
8       int *pA;
9       int ***pB;
10      s2e_make_symbolic(&pA, 4, "pA");
11      s2e_make_symbolic(&pB, 4, "pB");
12      s2e_enable_forking();
13      if(pA[3]+pA[4]+pB[0][1][2]==10)
14              s2e_warning("GOAL");
15      s2e_disable_forking();
16      s2e_kill_state(0, "program terminated");
17      return 0;
18 }
```

```
compared 12476 times
Found a solution in memory.
state constraint:
(Eq 10
    (Add w32 (ReadLSB w32 0 pB345)
             (Add w32 (ReadLSB w32 0 pA2)
                      (ReadLSB w32 0 pA1))))
Symbolic address solution:
TestCaseGenerator: processTestCase of state 0 at address 0x8048485

pA: 10  f9 �ff �bf �
pB: 10  f9 �ff �bf �
pA1: 03  00  00  00
pA2: 05  00  00  00
pB3: 10  f9 �ff �bf �
pB34: 10  f9 �ff �bf �
pB345: 02  00  00  00
```

```
compared 1640 times
Found a solution in memory.
state constraint:
(Eq false
    (Eq 10
        (Add w32 (ReadLSB w32 0 pB345)
                 (Add w32 (ReadLSB w32 0 pA2)
                          (ReadLSB w32 0 pA1)))))
Symbolic address solution:
TestCaseGenerator: processTestCase of state 1 at address 0x8048485

pA: 08  f9 �ff �bf �
pB: 10  f9 �ff �bf �
pA1: 08  f9 �ff �bf �
pA2: 02  00  00  00
pB3: 10  f9 �ff �bf �
pB34: 08  f9 �ff �bf �
pB345: 10  f9 �ff �bf �
```

Figure 21 program 2

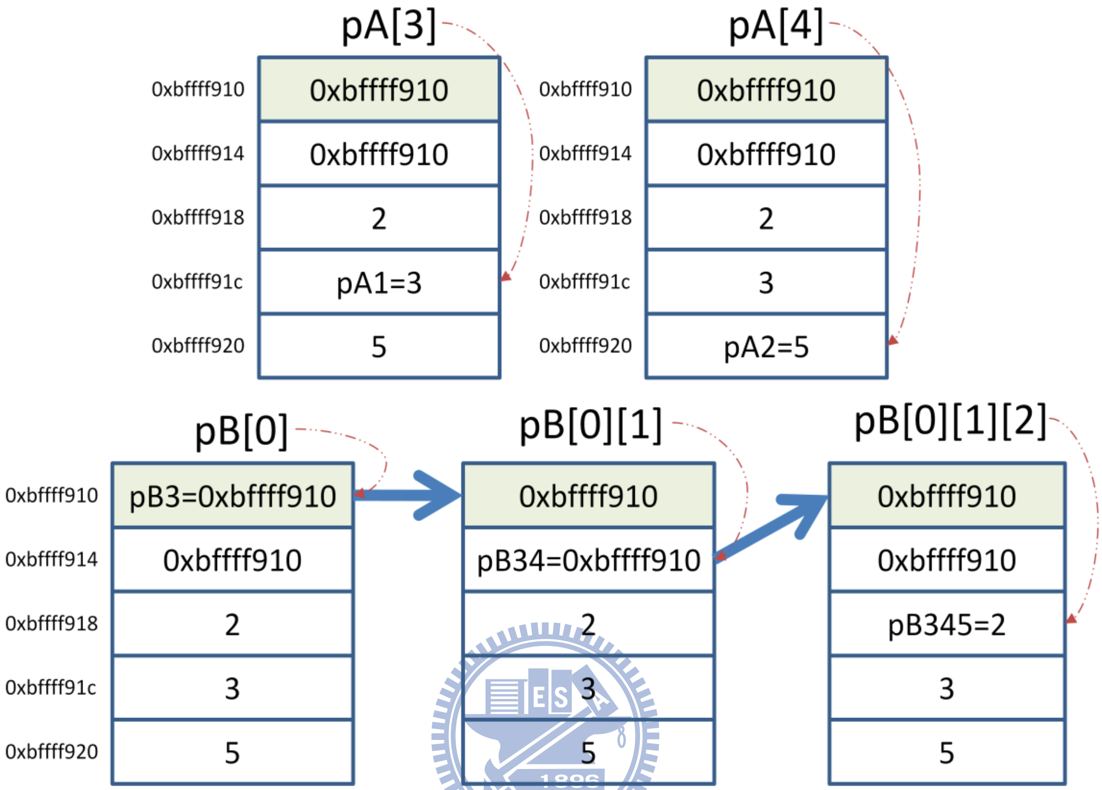


Figure 22 multi-dimension graph for symbolic pointer

5.2 Searching Algorisms analysis

In our thesis, we had implemented 2 algorisms for symbolic address solution generator: searching memory and searching Symbolic Table. Searching memory is a basic searching algorism; we search for the solution in memory. If searching range is too large, it may get stuck in searching step for a long time; if the range is too small, it may miss possible solutions. According to our experiment, we usually set the range to near the symbolic address location. Searching Symbolics Table is an opportunistic algorism. We map the address solutions to Symbolics Table and check if it is a solution. This algorism is useful when remaining symbolic variables which have no relationship with symbolic address exist.

Table 2 shows the number of compares on two algorisms; we test 3 programs which in previous section. N/A means it has no solution in Symbolics Table. Because of there is no any remaining symbolic variable, searching Symbolics Table only improve the program1's false branch.

Table 2 analysis of searching algorism

Program	**Num. of symbolic address**	**Searching memory**		**Searching Symbolics Table**	
		true branch	**false branch**	**true branch**	**false branch**
SAGE	2	23	1	N/A	1
Program1	3	1312	74	N/A	34
Program2	5	12476	1640	N/A	N/A

In the fact, when the buffer overflow happened, it always not only covers the symbolic address. It also covers other variable in most of case. Table 3 shows the situations when remaining symbolic variables are exist. Searching Symbolic Table is a powerful method to reduce the number of compares; in progam2 it reduces the number of compares from 12476 to 92 when remaining variable when there are two remaining symbolic variables. In addition, the compares will increase a little when there are more than 3 remaining symbolic variables. It's because of the Symbolics Table size will increase with remaining symbolic variables. It has to waste some compares on initialization.

Table 3 remaining symbolic variable

Program	**1 remaining**		**2 remaining**		**3 remaining**		**4 remaining**	
	true branch	**false branch**	**true branch**	**false branch**	**true branch**	**false branch**	**true branch**	**false branch**
SAGE	2	1	2	1	2	1	2	1
Program1	34	34	48	48	64	64	82	82
Program2	8274	N/A	92	83	103	102	124	123

6 Conclusion

We propose a new symbolic address module in this thesis. We construct a new symbolic address map on S^2E. We trace and adjust the symbolic address during the symbolic execution step, and then we handle the concrete execution step by our symbolic address solution generator. Two execution steps work in a co-operative way and exploit abnormal paths.

Our objective is to exploit the path condition which has the symbolic address like (**tainted-pointer == concret-value)*. If the program has another remaining tainted variable, we can directly assign *concret-value* to this tainted variable and assign tainted variable address to *tainted-pointer*, and the branch is always be true. In other words, if there is a symbolic address with remaining tainted variable, we can say this branch is completely controllable. If there is no remaining tainted variable, we still possibly find a solution in concrete memory; in the situation we say this branch is possibly controllable.

In future works, it is possibly find the statement (**tainted-pointer = tainted-value)* in the program. If found the situation then we can modify any memory content and fully control the target program followed my inclination.

Reference

[1] J. BURNIM and K. SEN, *Heuristics for scalable dynamic test generation*, IEEE Computer Society, 2008, pp. 443-446.

[2] C. CADAR, D. DUNBAR and D. ENGLER, *KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs*, USENIX Association, 2008, pp. 209-224.

[3] C. CADAR, V. GANESH, P. PAWLOWSKI, D. DILL and D. ENGLER, *EXE: A system for automatically generating inputs of death using symbolic execution*, Citeseer, 2006.

[4] V. CHIPOUNOV, V. GEORGESCU, C. ZAMFIR and G. CANDEA, *Selective symbolic execution*, 2009.

[5] V. CHIPOUNOV, V. KUZNETSOV and G. CANDEA, *S2E: A platform for in-vivo multi-path analysis of software systems*, ACM, 2011, pp. 265-278.

[6] L. CIORTEA, C. ZAMFIR, S. BUCUR, V. CHIPOUNOV and G. CANDEA, *Cloud9: A software testing service*, ACM SIGOPS Operating Systems Review, 43 (2010), pp. 5-10.

[7] O. CRAMERI, R. BACHWANI, T. BRECHT, R. BIANCHINI, D. KOSTIC and W. ZWAENEPOEL, *Oasis: Concolic Execution Driven by Test Suites and Code Modifications*, Technical Report LABOS-REPORT-2009-002, EPFL, 2009.

[8] B. ELKARABLIEH, P. GODEFROID and M. Y. LEVIN, *Precise pointer reasoning for dynamic test generation*, ACM, 2009, pp. 129-140.

[9] P. GODEFROID, N. KLARLUND and K. SEN, *DART: directed automated random testing*, ACM, 2005, pp. 213-223.

[10] E. HAUGH and M. BISHOP, *Testing C programs for buffer overflow vulnerabilities*, Citeseer, 2003.

[11] R. W. M. JONES and P. H. J. KELLY, *Backwards-compatible bounds checking for arrays and pointers in C programs*, Automated and Algorithmic Debugging, 25 (1997).

[12] J. C. KING, *Symbolic execution and program testing*, Communications of the ACM, 19 (1976), pp. 385-394.

[13] C. LATTNER and V. ADVE, *LLVM: A compilation framework for lifelong program analysis & transformation*, IEEE Computer Society, 2004, pp. 75.

[14] W. LE and M. L. SOFFA, *Refining buffer overflow detection via demand-driven path-sensitive analysis*, ACM, 2007, pp. 63-68.

[15] R. MAJUMDAR and K. SEN, *Hybrid concolic testing*, IEEE Computer Society, 2007, pp. 416-426.

[16] R. MAJUMDAR and K. SEN, *Latest: Lazy dynamic test input generation*, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-36, March (2007).

[17] S. NAGARAKATTE, J. ZHAO, M. M. K. MARTIN and S. ZDANCEWIC, *SoftBound: highly compatible and complete spatial memory safety for c*, ACM, 2009, pp. 245-258.

[18] N. NETHERCOTE and J. SEWARD, *Valgrind: a framework for heavyweight dynamic binary instrumentation*, ACM SIGPLAN Notices, 42 (2007), pp. 89-100.

[19] O. RUWASE and M. S. LAM, *A practical dynamic buffer overflow detector*,

Citeseer, 2004, pp. 159¡V169.

[20] K. SEN, *Concolic testing*, ACM, 2007, pp. 571-572.

[21] K. SEN, D. MARINOV and G. AGHA, *CUTE: a concolic unit testing engine for C*, ACM, 2005.

[22] H. SHAHRIAR and M. ZULKERNINE, *Mutation-based testing of buffer overflow vulnerabilities*, IEEE, 2008, pp. 979-984.

[23] D. VANOVERBERGHE, N. TILLMANN and F. PIESSENS, *Test input generation for programs with pointers*, Tools and Algorithms for the Construction and Analysis of Systems (2009), pp. 277-291.

[24] J. YANG, C. SAR, P. TWOHEY, C. CADAR and D. ENGLER, *Automatically generating malicious disks using symbolic execution*, IEEE, 2006, pp. 15 pp.-257.

[25] D. ZHANG, D. LIU, Y. LEI, D. KUNG, C. CSALLNER and W. WANG, *Detecting vulnerabilities in C programs using trace-based testing*, IEEE, 2010, pp. 241-250.