國立交通大學

資訊科學與工程研究所

碩 士 論 文

操作利用非典型之擬真執行

Exploiting Atypical Symbolic Executions

研 究 生：邱世欣

指導教授：黃世昆　教授

中 華 民 國　100 年 7 月

操作利用非典型之擬真執行

Exploiting Atypical Symbolic Executions

研 究 生：邱世欣　　Student：Shih-Hsin Chiu

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 2011

Hsinchu, Taiwan, Republic of China

中華民國 100 年 7 月

操作利用非典型之擬真執行

學生：邱世欣　　　　　　　　　　　指導教授：黃世昆 教授

國立交通大學資訊科學與工程學系（研究所）碩士班

摘要

軟體安全日漸成為重要的研究主題，起因於越來越多的軟體攻擊行為發生，這些狀況有一部份是源自於程式語言本身的缺陷，而另一方面也是程式設計師本身的粗心所導致。因此，我們將藉由軟體偵測技術以減少這些問題。在論文中探討目前被廣泛運用的的程式漏洞-緩衝區溢位(Buffer overflow)，例如西元 2003 年八月造成重大損失的疾風(Blaster)病毒即利用此種漏洞進行破壞。為了防止此類型的漏洞，本論文使用 KLEE 的符號執行模組(symbolic execution model)並引入新的記憶體對映機制(memory map)來探測緩衝區溢位。相較於傳統的檢測工具，本論文所提出的工具可確實產生、可利用的測資來觸發漏洞的行為，進而證實漏洞的存在。這些測資事實上就是一組攻擊字串，有別於駭客手動方式產生，我們將提出自動產生的方法。

Exploiting Atypical Symbolic Executions

Student：Shih-Hsin Chiu Advisors：Dr. Shin-Kun Huang

Department of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

Software security is getting more important recently. There are more and more attacks than before. It is partially due to some design flaws of the programming language and the lack of secure programming practices by programmers. The most serious vulnerability this thesis concerns with is buffer overflow, present in many C/C++ programs, such as the Blaster worm. For preventing from such vulnerabilities, we use symbolic execution with a new memory model supported by KLEE to detect buffer overflow vulnerabilities. This thesis actually generates an exploitable input to trigger buffer overflow and verify the presence of the vulnerability. The input suites we generate are realistic attacks. Unlike the usual hacking methods with manual techniques to reason on the tainting paths, we propose methods to generate exploitable input automatically.

誌謝

首先感謝黃世昆教授願意收留當年懵懂無知的我，讓我可以一窺軟體測試的奧妙，讓我對寫程式產生莫大的熱情。當然也要謝謝我的父母，在我求學的生涯中，給予我鼓勵與支持，在我決定要多讀一年的時候，尊重我的決定，讓我無後顧之憂的去完成我的學業。

謝謝 LAB 中所有學長以及所有學弟妹們，讓我碩士生涯中除了苦悶的研究之外，帶來許多歡笑，幫助我緩和研究上的壓力，感謝琨翰學長在我碩一時不厭其煩的教導我課業上的問題，感謝佑鈞在我研究上給予我極大的幫助，感謝那一群在我碩三壓力最大的時候陪我渡過難關的博彥、孟緯、翰霖、韋翔、偉明、奕任、基傑、俊維。

當然也感謝兩位口試委員，陳澤雄教授和鍾玉芳教授，願意撥空幫我的論文口試，也叮嚀了一些論文寫作上的問題。總之，要感謝的人很多，無法一一列出，不如感謝天吧！

最後，我畢業了！！！！！！！！！！！！！！！！！！！！！！！！！！

Table of Contents

List of Tables

List of Figures

1. Introduction

Software testing is the process of validating program behaviors to conform to the specified requirements. Traditionally, programmers manually generate test suites and actually execute program to verify program behavior. It takes human efforts and is not efficient.

As the development of larger software applications, programs become more complicated. As figure1 shows, there are two additional situations in program development.

1. Requirement is not fully accomplished. It exist some bugs in the program.
2. Implementing additional functions. It causes the security problems.

Manual program validation is not appropriate for program testing. The demand of automated testing becomes essential. There are many related methods proposed to resolve the issues [3,12,13,16,17,21], but few of them can deal with large software systems. In our work, we propose an important issue for automatic test suites generation. In contrast with traditional buffer overflow detecting tools [4,10,14], we aim to actually generate an exploitable input to trigger buffer overflow attacks, and verify the presence of the vulnerability. The input suites we generate are realistic attacks.

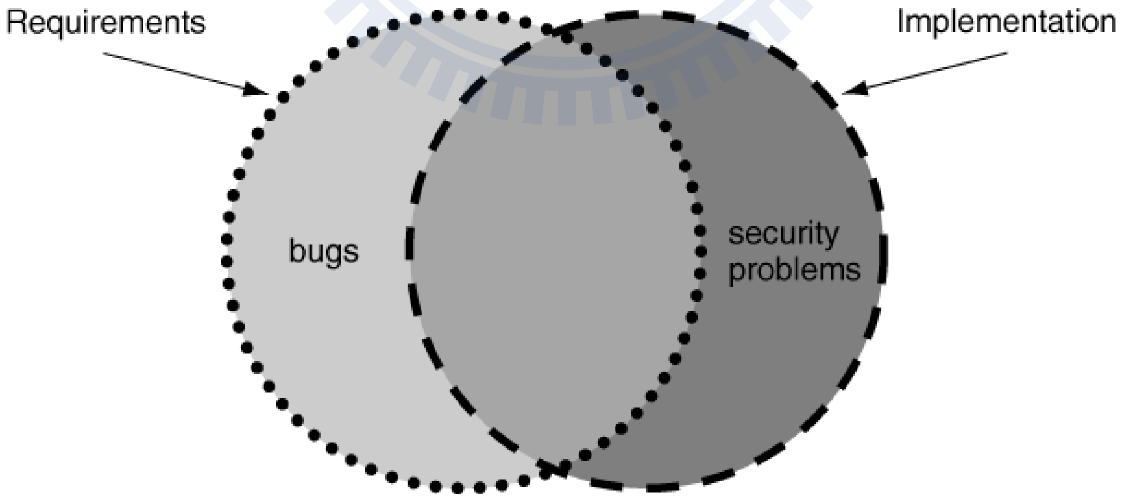


Figure 1 Program development

1.1 Background

Software testing on buffer overflow vulnerability is getting mature gradually, due to the threats originated from buffer overflow flaws. For the security perspective, the buffer overflow defect is an anomaly of program behaviors where a process stores data in a buffer beyond the buffer size declared before. Any data overwritten situations may cause unexpected behaviors.

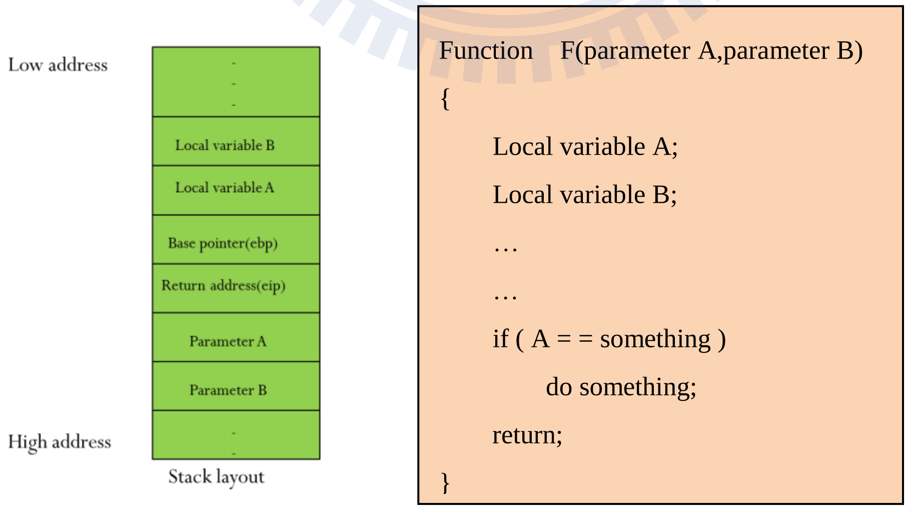
Due to the above-mentioned facts, there are many approaches and researches which have been proposed. We describe the major researches about our works below.

1.1.1 Common Vulnerabilities

From the classifications of security threats, there exist types of serious vulnerabilities, including buffer overflow, uninitialized variables, format string, etc. In this section, we will discuss on several critical vulnerabilities in the following.

1.1.1.1 Stack-based Overflow

During executing program, the information of call functions, including return address, base pointer parameters and local variables is pushed into stack as shown in figure 2, which are accessed via the offset of base pointer.



```
Function    F(parameter A,parameter B)
{
        Local variable A;
        Local variable B;
        …
        …
        if ( A = = something )
                do something;
        return;
}
```

Figure 2 The call stack of Function F

In general, there are several ways to exploit stack-based overflow.

1. Overflowing variables nearby a buffer to change the program behavior, that is, to control the program executing flow if the nearby variables are in the predicates involving some branch conditions, etc.
2. Overflowing function pointers or return address to execute arbitrary codes such as shell-code.

1.1.1.2 Heap-based Overflow

The heap-based overflow [23] is different from stack-based overflow in manner. Memory on heap are dynamically allocated at program run-time, and maintained by metadata.

The exploit is to overflow the memory management information associated with heap memory, such as the metadata of *malloc*. Figure 3 shows the metadata sketch of Phkmalloc in BSD. The Pginfo is used to describe a small or medium page, and is stored in the beginning of the page for small chunks.

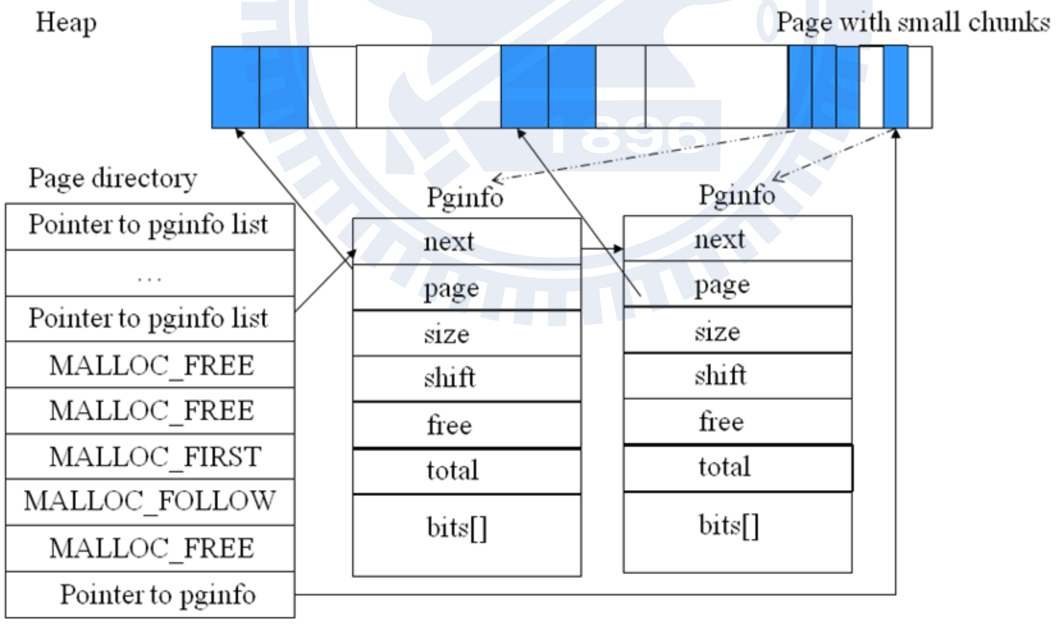


Figure 3 Metadata of Phkmalloc in BSD

As shown in figure 4[23], we can overflow the page-field in Pginfo of chunks to point to target memory location, such as GOT entries and return address, or modify the bits[] array to make all chunks seem free.

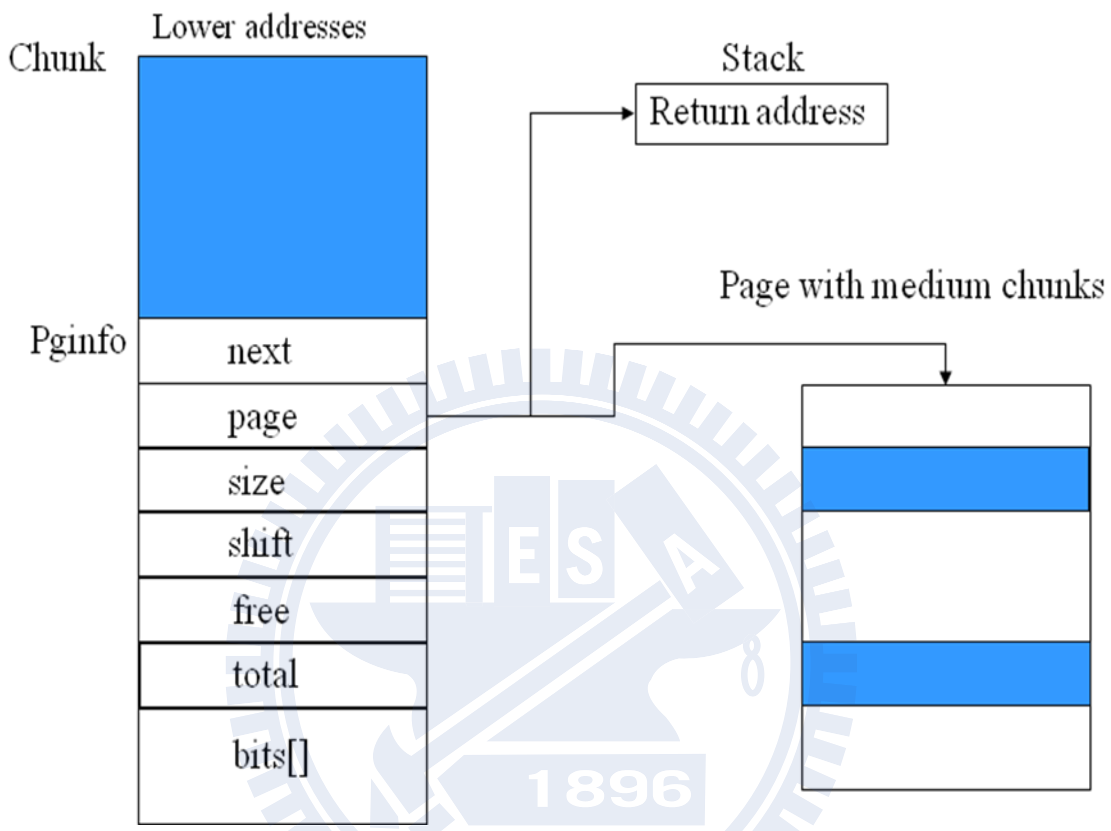


Figure 4 Notion of Heap overflow

1.1.1.3 Uninitialized Variables

In programming language C/C++, the variables of subroutines are allocated at stack region. In fact, it just modulates the stack pointer to indicate a stack frame for allocation without any status check.

The value of an uninitialized variable cannot be expected and it is a common bug in the program. An attacker can find an exploitable path to control the uninitialized variables. As shown in figure 5, the call stack of function A and function B will be allocated in the same address range, that is, the address of local variables in function A and function B may overlap. According to the executing flow of program, we can control the local variable b in function B by adjusting the argument of function A from standard input. For instance, we input value 100 as the argument of function A to pass the if-condition in function B, and finally reach the goal.

```
#include<stdio.h>
void a(int n){
        int a=n;
        printf("%p %d\n",%a,a);
}
void b(int n){
        int b;
        printf("%p %d\n",&b,b);
        if(b==n)
                printf("GOAL!!\n");
}
int main(void){
        int val,tmp=100;
        scanf("%d",&val);
        a(val);
        b(tmp);
        return 0;
}
```

```
[chiush@Laputa:~/test]$ ./cross
100
0xbfb5765c 100
0xbfb5765c 100
GOAL!!
```

Figure 5 Example of uninitialized variable

1.1.2 Existing Detection Tactics

An easy approach for detecting errors is to insert "*printf*" statement manually. However, it is inefficient and is time-wasted. To promote the efficiency, we analyze programs by programs.

The existing detection tactics can be divided into two approaches: one is static analysis and the other one is dynamic analysis.

1.1.2.1 Static Analysis

To perform static analysis, we need variables information, dataflow information and the parse tree of the program. Moreover, some heuristics are needed for detection. There are lots of static analysis tools, such as Uno[24], developed at Bell Labs, which is designed to detect the most common types of vulnerabilities. Static analysis tools just scan the source code without actually executing. For instance in buffer overflow detecting, they check some of dangerous standard library functions by tracking the arguments, such as *strcpy*, where they track the possible size of strings to avoid buffer overflow.

However, these methods are imprecise and with high false positive rate. In fact, it just determines if a buffer overflow might occur.

1.1.2.2 Dynamic Analysis

Dynamic analysis actually executes the target program to detect the vulnerabilities at run-time. To perform it, we need a large number of test cases to explore the target program. Such as valgrind[9], a tool for memory debugging and memory leaking analysis tool.

Dynamic analysis costs more time than original for program exploration. However, it performs precise detection than static analysis. The vulnerabilities found by dynamic analysis are authentic.

1.1.3 Random testing (Fuzzing)

Unlike dynamic analysis that uses selected test inputs, the random testing selects test inputs randomly and leads program to explore more deep. Such as zzuf [25], a tool that performs fuzz testing on target program.

Random testing executes much faster. However, it costs lots of time to select duplicate test inputs. In a 32-bit computer, there are 2^{32} inputs for a variable.

1.1.4 Symbolic Execution

Symbolic execution [20] is a popular testing approach in software validation and program proving [12]. The execution process is by collecting a list of assignment statements and branch predicates with symbolic values in a particular path of execution, and solve these constraints by solvers to get a solution as next program inputs. The drawback of symbolic execution is the lack of scalability because of the much more execution paths for exploration in a large program.

1.1.5 Concolic Testing

Concolic testing [7,8,15] combines the concrete and symbolic (concolic) executions. Based on symbolic execution, it collects symbolic constraints for further executions. The differences between them are the concept of concrete executions, which replace some of the symbolic values with concrete ones, when the constraint solver cannot resolve feasible solutions that satisfy the constraints.

For instance in figure 6, we begin with a simple random testing for variable a and variable b, for example a=b=0. Concolic execution treats a and b as symbolic variables and the test in line 3 fails since a=0!=1. Because we would like to explore different paths on next run, we negate the last path condition and get a=1. And then we encounter the next path condition and it fails again since b=0!='a', thus we negate the path condition and get b='a' and finally we reach the goal.

```
#include<stdio.h>
void testme(int a,char b){
        if(a==1)
                if(b=='a')
                        puts("GOAL");
}
int main(void){
        int a;
        char b;
        testme(a,b);
        return 0;
}
```

Figure 6 Example of concolic execution

1.2 Motivation

Buffer overflow is one of critical security problem, as the statistics of CERT advisories [30] in table 1, buffer overflow is in possession of the percentage near 50% before 2004. These table only list critical advisories.

Table 1 Summary of recent CERT advisories (Last updated 2004)

Year	Advisories	Buffer overflow related advisories	Percentage of Buffer overflows
1996	27	5	19%
1997	28	15	54%
1998	13	7	54%
1999	17	8	47%
2000	22	3	14%
2001	37	19	51%
2002	37	21	57%
2003	28	18	64%
2004	9	7	78%
Total	218	103	47%

The US-CERT Cyber Security Bulletin [31] provides a summary of new vulnerabilities that have been recorded by the National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD) in the past week. We count the high severity Vulnerabilities, determined by the Common Vulnerability Scoring System (CVSS) standard. Because buffer overflow related vulnerability appear in high severity more than medium and low severity.

As shown in table 2, the percentages of buffer overflow vulnerability from January to May in 2011 have 15% in total. The prevention of buffer overflow is essential.

Table 2 Cyber Security Bulletins Summary in 2011

Release Date	Bulletins	Buffer overflow related	Percentage of Buffer overflow
January 3, 2011	18	1	5%
January 10, 2011	18	1	5%
January 17, 2011	64	23	35%
January 24, 2011	58	10	17%
January 31, 2011	15	5	33%
February 7, 2011	55	19	34%
February 14, 2011	103	11	10%
February 21, 2011	22	3	13%
February 28, 2011	53	8	15%
March 7, 2011	48	5	10%
March 14, 2011	26	0	0%
March 21, 2011	16	1	6%
March 28, 2011	13	2	15%
April 4, 2011	15	1	6%
April 11, 2011	18	4	22%
April 20, 2011	59	3	5%
April 27, 2011	59	3	5%
May 2, 2011	10	2	20%
May 9, 2011	55	2	3%
May 16, 2011	34	9	26%
May 23, 2011	22	7	31%
Total	781	120	15%

All of existing software analysis tools only detects vulnerabilities, that is, it does not generate an exploitable input to trigger vulnerabilities. In order to perform precise testing and manifest the vulnerabilities which exist indeed, it is a trivial notion by intuitions, that is, exploiting the vulnerabilities by running the program with abnormal inputs to trigger it. Furthermore, showing the abnormal inputs as the exploitable evidence is very persuasive. Not only crash the checked program, we finally get a test suite to perform the exploit, such as getting the root privilege.

1.3 Problem Description

Buffer overflow [4] indicates that we should write some data beyond the buffer size we declared before. Human are intelligent to know the principle. But for computers, it cannot make sense with it. We describe several case of buffer overflow situations below：

Case 1：Normal overflow in stack region

Given a set of symbolic variables x1, x2, x3,…xn, and a set of non-symbolic variables y1,y2,y3, …, yn. We try to verify if we can convert any non-symbolic variables into symbolic variables. If it is feasible for the conversion, we define this is an exploitable state. In figure 7, X1 is a symbolic character array with size 4, and we can overflow non-symbolic variable Y1 by assigning some value to X1[5] no doubt. Specially, we can overflow the return address.

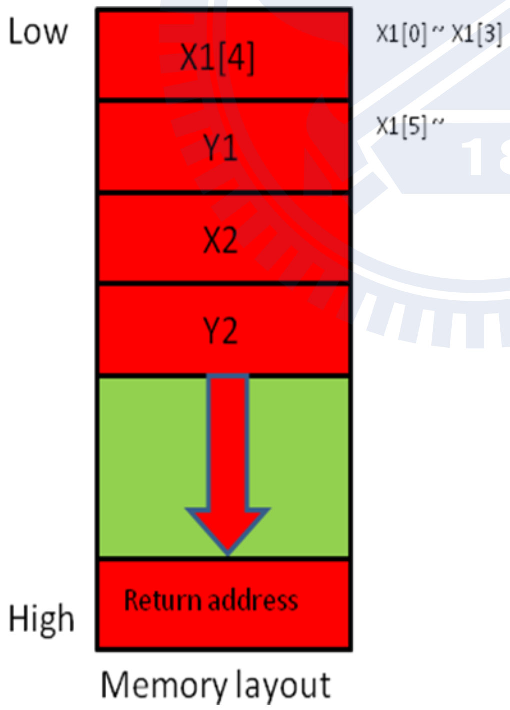
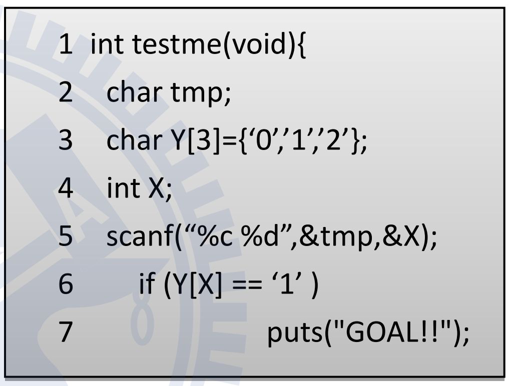


Figure 7 Normal stack overflow with a symbolic character array X1

Case 2：Symbolic array index dereference

In this thesis, we make efforts in this case. Given a concrete array Y handled by a symbolic array index X in branch condition, such as in Figure 8. The program prints “GOAL” when X is assigned to 1, it is the regular example of symbolic index. However, how if we rewrite the branch condition as (Y[X] ==’3’). In this situation, we can not find solution in array Y, but we can try the indices beyond in stack region.

Normally, we don’t need worry about the problem, that is, the symbolic index is rare in program. Nevertheless, it is the key to explore more program path for exploit input generator.



```
int testme(void){
  char tmp;
  char Y[3]={‘0’,’1’,’2’};
  int X;
  scanf(“%c %d”,&tmp,&X);
      if (Y[X] == ‘1’ )
                  puts("GOAL!!");
```

Figure 8 Example of symbolic index

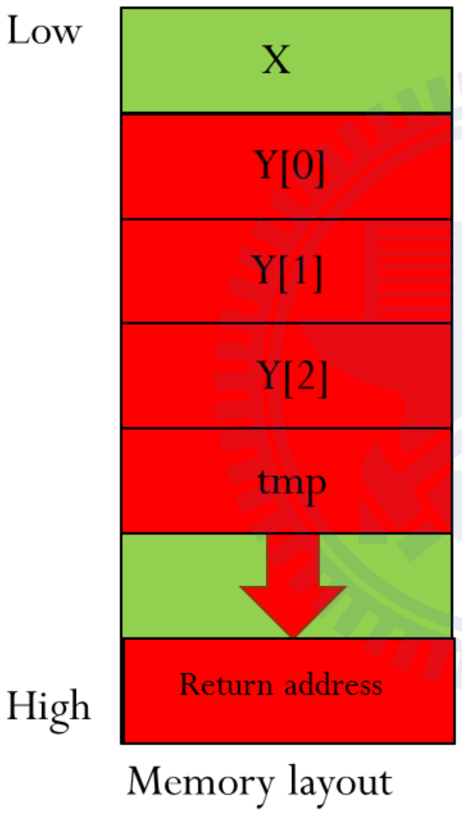


Figure 9 Concrete array Y with symbolic index X

Case 3：Cross-function overflow in stack

According to the stack allocation, a callee function with symbolic variables can taint non-symbolic variables in caller function. Such as in figure 10, the function Y is a callee function called by function X, specially, the symbolic variable Y1 can overflow the non-symbolic variables in X function,

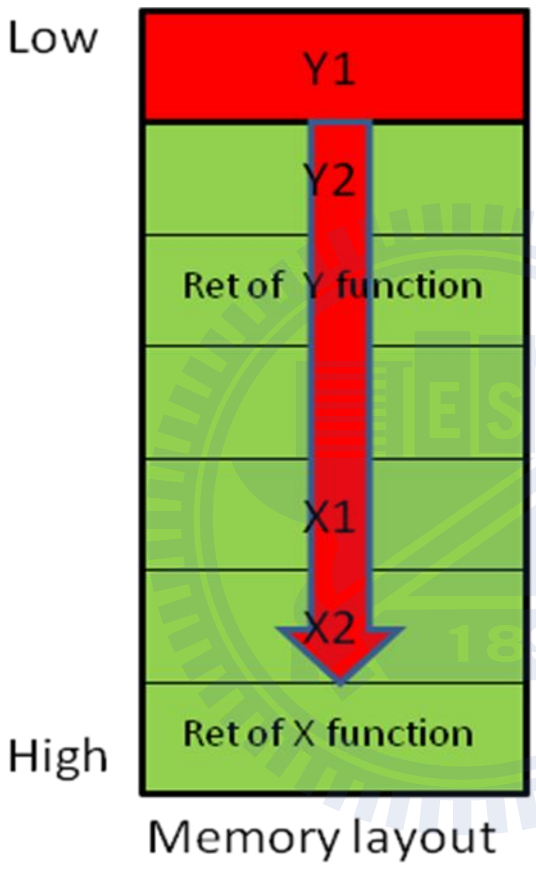


Figure 10 Cross-function overflow in stack with symbolic variable Y1

Case 4：The general case (*tainted-pointer = tainted-value)

Normally, lower-addressed variables are not taintable by a symbolic variable, unless there has a symbolic pointer. In figure 11, the symbolic pointer X2 is mighty. In generally, it can taint not only stack region, but also the whole of memory region, such as the metadata in heap region and GOT, etc.

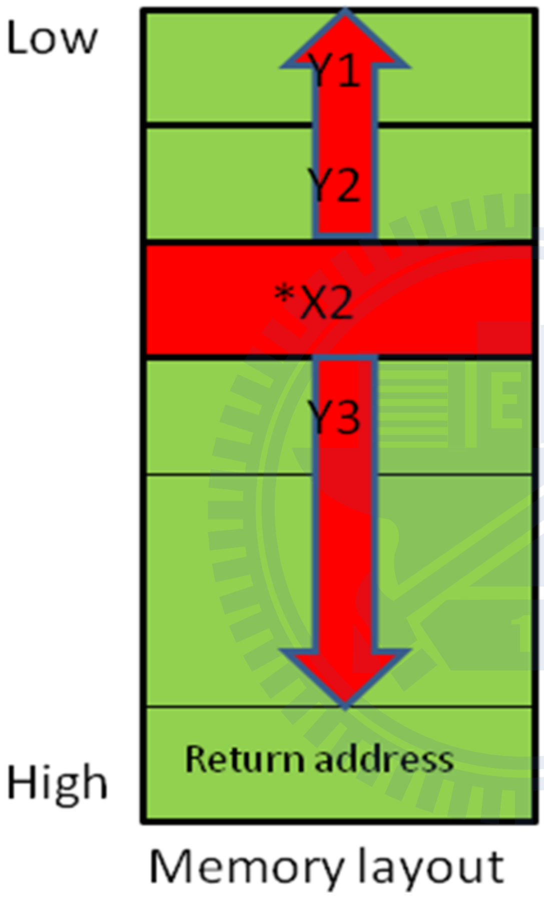


Figure 11 Buffer overflow with a symbolic pointer X2

For the simulation of buffer overflow, that is, we make the exploit process automatic by symbolic execution. To achieve the goal, we should record the relationship between symbolic variables, that is, we must maintain a memory offset map which records buffer index according to the symbolic variable. This allows us to write beyond the buffer size as abnormal inputs to trigger buffer overflow or other vulnerabilities.

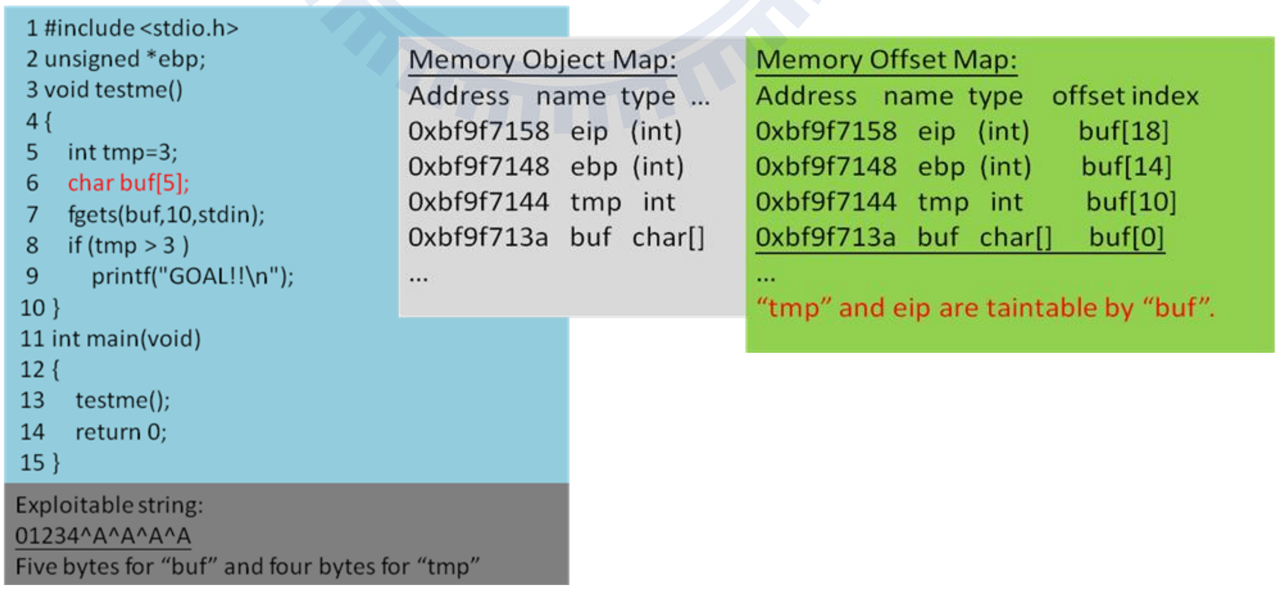
In dynamic input generation, we collect the list of constraints during execution, and solve them by solvers. Furthermore, in order to perform precise buffer overflow, we should generate many constraints to handle symbolic index and symbolic pointer by using memory offset map which records the relations between symbolic variables.

1.4 Objective

We want to improve the precision of dynamic input generation, that is, we generate a precise testing input to explore more feasible execution paths, and to trigger more vulnerability like buffer overflow. To achieve these goals, we will try to implement a new component of dynamic test input generator on KLEE [2] including:

1. Memory offset map: A table records the variable's information and relation between symbolic variables in target code.
2. Symbolic array index: To generate abnormal symbolic solutions in array object dereference.

We depict the memory offset map example below：



```
#include <stdio.h>
unsigned *ebp;
void testme()
{
   int tmp=3;
   char buf[5];
   fgets(buf,10,stdin);
   if (tmp > 3 )
      printf("GOAL!!\n");
}
int main(void)
{
   testme();
   return 0;
}
```

Exploitable string:
01234^A^A^A^A
Five bytes for "buf" and four bytes for "tmp"

Memory Object Map:

Address	name	type ...
0xbf9f7158	eip	(int)
0xbf9f7148	ebp	(int)
0xbf9f7144	tmp	int
0xbf9f713a	buf	char[]
...		

Memory Offset Map:

Address	name	type	offset index
0xbf9f7158	eip	(int)	buf[18]
0xbf9f7148	ebp	(int)	buf[14]
0xbf9f7144	tmp	int	buf[10]
0xbf9f713a	buf	char[]	buf[0]
...			

"tmp" and eip are taintable by "buf".

Figure 12 Example of memory offset map

2. Related Work

Memory map is an essential part for every symbolic execution analysis tools, such as EXE [11], SAGE [18], CUTE [6] and KLEE [2], etc. All of them implement the model in their tools to specify and track symbolic variables and constraints. We base on the KLEE memory model, with improvement to make a precise dynamic input generator:

- CRED [22,27] (C Range Error **Detector**) directly checks the bounds of memory accesses, and the address of a special OOB (out-of-bounds) object is substituted for every out-of-bounds pointer address. In short, it is just a bound checking tool.

- DART [5] integrates random testing and dynamic test input generation using symbolic reasoning, but it only handles integer constraints and invokes random testing when symbolic pointer constraints are used.

- CUTE uses concolic execution to generate test inputs to explore all feasible execution paths. It extends DART by tracking symbolic pointer constraints of the equalities and inequalities. It cannot handle symbolic array index.

- Crest [19] is an open source concolic testing tool for C. by supporting heuristic search strategies to perform high branch coverage in target code. It does not handle symbolic array index.

- EXE is more precise and handles both pointer arithmetic and symbolic pointer with single dereference, but it does not handle symbolic pointer with multi-dimension dereferences and write. Moreover, it has implemented symbolic index, but it cannot handle out-of-bounds array index dereference.

- SAGE performs symbolic array index in dynamic input generation by presenting a new memory model. Furthermore, it claims the symbolic constraints are solvable using SMT solvers. However, it cannot handle out-of-bounds array index dereference

- KLEE is redesigned from EXE. In KLEE, it takes advantage of many constraint solving optimizations, and uses search heuristics to reach high coverage in program. It directly interprets the LLVM [1] assembly language, a RISC-like instruction set, and converts instructions to constraints. However, KLEE records all of memory objects including unnecessary objects and cannot handle symbolic array index dereference.

- ALERT is developed by our laboratory, it has implemented a memory offset map and byte-level precision for symbolic expression. Moreover, it handles symbolic pointer and symbolic array index to explore more feasible execution path. But it cannot handle out-of-bounds array index.

Above tools target to fully explore program paths and find bugs. In our works, we focus on exploiting program bugs to explore more program paths. The comparison of above tools is shown as table 3. There are tools capable of handling symbolic array index. However, our work can handle not only in-bounds array index but also out-of-bounds array index.

Table 3 Comparison of analysis tools in symbolic index

Tool	**Symbolic array index**	**Out-of-bounds index**
DART	X	X
Crest	X	X
Cute	X	X
Exe	O	X
Sage	O	X
KLEE	X	X
Alert	O	X
Hsin	O	O

3. Method and Steps

We target to improve KLEE symbolic execution model. Aiming to handling symbolic array index dereference for path coverage enhancement. Therefore, we trace KLEE to understand the requirement of symbolic array index handling in KLEE and study LLVM intermediate representation.

KLEE is a Symbolic executor built on LLVM. LLVM provides many useful tools and APIs for code analysis, for example, we can use LLVM pass to insert some instructions for statistics. For this reason, we can do some modifications on target code for precise checking.

KLEE interprets LLVM bit-codes and by inserting a special function named *klee_symbolic_make* in target code for symbolic execution. It collects path constraints related to symbolic variables indicated by the special function call, and forks entire states when it encounters a branch, then resolves the path constraints for possible solution according to the branch condition.

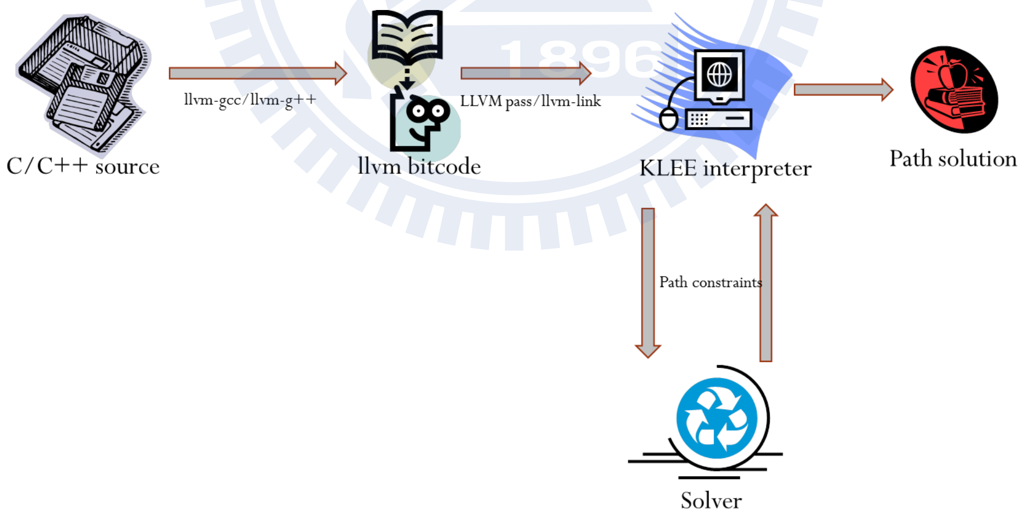


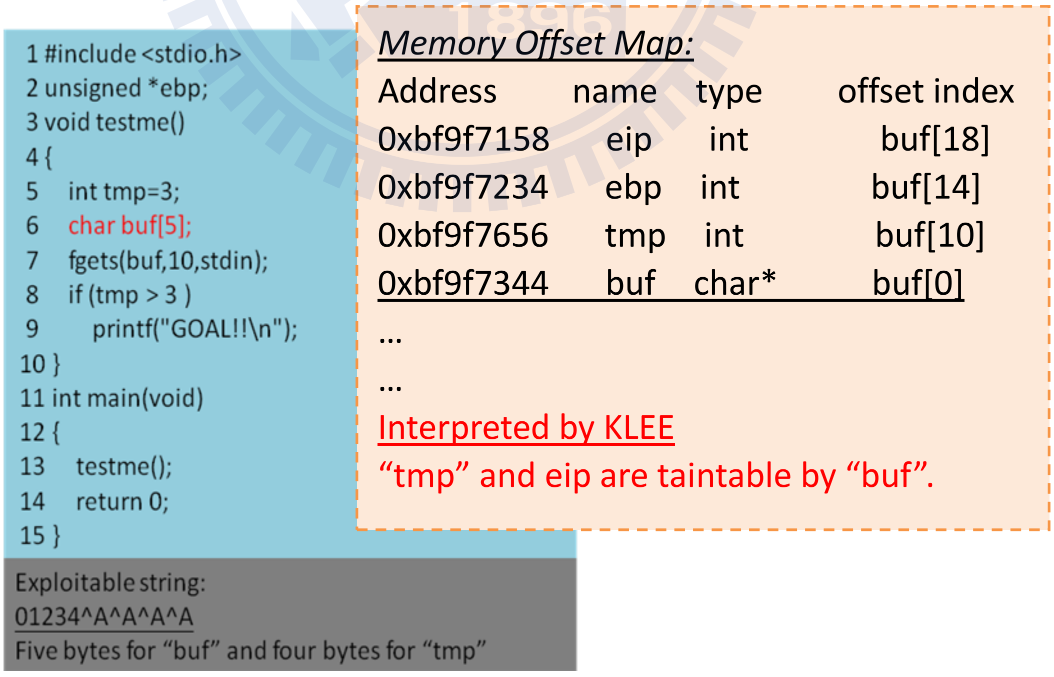
Figure 13 KLEE testing procedure

Based on KLEE symbolic execution model, we construct a new memory offset map and add a new symbolic memory function call named *hsin_make_symbolic* to simulate buffer overflow. Furthermore, we modify the interpretation in memory dereference instruction to handle symbolic array index dereference. We explain the conceptions following.

3.1 New symbolic memory offset map

KLEE use abstract memory model to process memory operation. It takes the target program as the part of KLEE by allocating independent memory space to store variables in target program even if they are in the stack. Therefore, the stack variables of target program are irrelevant in each other. Furthermore, original memory map records too much destructive information like the variables in *uclibc*.

For buffer overflow manipulation, we need the information of tainted variables, including size、type and value, etc. For this reason, we construct a memory offset map to record the relationship of variables with offset information in stack region. The memory offset map supports us for buffer overflow simulation and exploiting input generation.



```
#include <stdio.h>
unsigned *ebp;
void testme()
{
    int tmp=3;
    char buf[5];
    fgets(buf,10,stdin);
    if (tmp > 3 )
        printf("GOAL!!\n");
}
int main(void)
{
    testme();
    return 0;
}
```

Exploitable string:
01234^A^A^A^A
Five bytes for "buf" and four bytes for "tmp"

Memory Offset Map:

Address	name	type	offset index
0xbf9f7158	eip	int	buf[18]
0xbf9f7234	ebp	int	buf[14]
0xbf9f7656	tmp	int	buf[10]
0xbf9f7344	buf	char*	buf[0]

…

…

Interpreted by KLEE
"tmp" and eip are taintable by "buf".

Figure 14 Example of memory offset map

3.2 New symbolic memory function

Original KLEE provides a function named *klee_make_symbolic* to mark symbolic memory by setting the appointed memory contents as changeable for constraints generation. KLEE collects constraints related to symbolic variables and solves them when meets branches.

The symbolic variables can influence program path exploration. KLEE intuitively marks the variables from standard input as symbolic only. As shown in figure 15 at line 6, original KLEE marks the variable buf as symbolic with its size, but in fact, the variable i should be symbolic for constraint generation.

We need a new function to mark nearby memory as symbolic by automatically determining whether a variable is overflow-able through the standard input buffer. Finally, we can integrate the solutions of overflow-able variables with the buffer to generate an abnormal test input by the memory offset map supports.

```
 1 #include<stdio.h>
 2 int main(void){
 3        int i;
 4        char buf[5];
 5        scanf("%s",buf);
 6        //klee_make_symbolic(buf,5,"buf");
 7        hsin_make_symbolic(buf,9,"buf");
 8        if(i==1)
 9             puts("GOAL!!");
10        return 0;
11 }
```

Figure 15 An example of abnormal buffer input

3.3 Symbolic array index dereference approach

This is the main idea in our thesis. For an exploitable input generator, we use symbolic index to explore each memory contents, aiming to rewrite the values and generate more possible program path.

In a C/C++ program, we use array for large related data deposition, manipulating by an index value. The situations of branch condition with array index dereference are listed below:

i. A constant value → if (buf[2] == ‘a’)

ii. A calculated value → i=2; if(buf[i] == ‘a’)

iii. A symbolic value → cin>>i; if(buf[i] == ’a’)

Figure 16 Situation of array index

We interests on symbolic index, it is common in string checking. The concept of symbolic index means that a concrete array handled by a variable index. The index may be tainted by standard input.

Const_buf[symbolic index] = tainted_value

KLEE didn’t support symbolic index dereference because the lack of constraints for index expression. Therefore, KLEE keeps forking states on inaccurate constraints generation if symbolic index exists.

We must provide index constraints for symbolic index solving. To handling symbolic array index dereference, we look into the memory dereference operations and comparison instruction interpreted by KLEE. It is trivial to handle symbolic index at inside scope by adding related index constraints at each index. However, for indices which are at outside scope, we should compare the required value with concrete values at each possible index by memory offset map supporting.

4 Implementation

Based on KLEE symbolic executor, we add a memory offset map for stack object gathering, and store the relations between each stack objects. Rely on the memory offset map, we develop a simple buffer overflow input generator and supports symbolic array index dereference.

In this thesis, we make efforts in handling symbolic array index dereference. To simulate buffer overflow, we supports the array index dereference at outside scope for precise checking.

We use C/C++ and LLVM IR to develop memory offset map and symbolic array index dereference algorithm based on KLEE, and perform the executing script by bash shell script supports.

4.1 Memory offset map

We construct a framework with some rules to collect stack objects, As the figure 17 shows, an stack object includes its name、size and type information, etc. The *simuStack* class constructs a virtual stack region and provides some controlling methods to process stack operation.

In KLEE, the allocating function *ExecuteAlloc* collects all of variables in a simple memory map without classifying. It collects not only stack variables but also variables in *uclibc*, etc. So we should filter out the unnecessary variables. The key is to acquire all of function name in target code by writing an analysis LLVM *pass*. We describe the LLVM *pass* at ending of this section. Afterwards, we can collect stack variables in target code only by setting a comparison condition as shown in figure 18.

```
Class SimuStack{
Private:
        vector<StackObj *> so;
        StackObj *ebp;
        StackObj *eip;
Public:
        void pushObj(StackObj* so);
        void popObj(StackObj* so);
        void setIndex();
        void print();
}
```

```
Class StackObj {
Private:
        string name;
        size_t size;
        int address;
        int index;
        vector<unsigned char*> contents;
public:
        void setContents(std::vector<unsigned char> cont);
}
```

Figure 17 Memory offset map

Following stack as a model, we provide some controlling function to simulate the action on stack. There are about four methods in the *SimuStack* class and one method in *StackObj* class:

In *SimuStack* class:

1. *void pushObj(StackObj* so)*
   Pushing a stack object into the SimuStack.

2. *void popObj(StackObj* so);*
   Popping a stack object from SimuStack.

3. *void setIndex();*
   Setting offset information without alignment.

4. *void print();*
   Printing the memory offset map information.

In *StackObj* class:

1. *void setContents(std::vector<unsigned char> cont);*
   Setting symbolic solutions of symbolic variables in hexadecimal.

```
$\mathcal{M}$ : a memory object allocated by KLEE
$\mathcal{F}$ : the current function name
$\mathcal{O}$ : an stack object used to store variable information
$\mathcal{S}$ : the simuStack

ExecuteAlloc(...){
    If $\mathcal{F}$ exists in target program {
        $\mathcal{O} \leftarrow$ new StackObj($\mathcal{M}$->name, $\mathcal{M}$->address, $\mathcal{M}$->size, ...)
        $\mathcal{S}$->pushObj( $\mathcal{O}$ )
    }
    ...
}
```

Figure 18 Allocating function of KLEE

To get all of function name in target code, we write a function analysis *pass*. As shown in figure 19, we inherit *FunctionPass* and overload *runOnFunction* function to iterate on each function in target program, and then print the function name to standard error.

Depend on the function *pass*, we can get variables in each function without including unnecessary variables in *uclibc*, etc. The function analysis *pass* is important in subsequent section.

```
#include "llvm/Pass.h"
#include "llvm/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;
namespace {
    struct funcAns : public FunctionPass {
        static char ID;
        funcAns() : FunctionPass(ID) {}
        virtual bool runOnFunction(Function &F) {
            errs() << F.getName() << "\n";
            return false;
        }
    };
    char funcAns::ID = 5566;
    static RegisterPass<funcAns> X("funcAns", "Get func name pass", false, false);
}
```

Figure 19 Function analysis pass

4.2 New Making Symbolic Strategy

KLEE provides a framework to insert some function calls or handle some specific function in target program in file *SpecialFunctionHandler.cpp*. By defining the handler, we can instrument a function to collect some information in target code.

Depend on the stack layout, the stack variables are allocated in light of declaring sequence as shown in figure 20. According to the relations in stack, we define a new symbolic memory function named *hsin_make_symbolic* as shown in figure 21, iteratively determining nearby stack objects of the standard input buffer as symbolic. Afterwards, KLEE gathers constraints related to symbolic variables and solves the constraints by STP solver for path exploration.

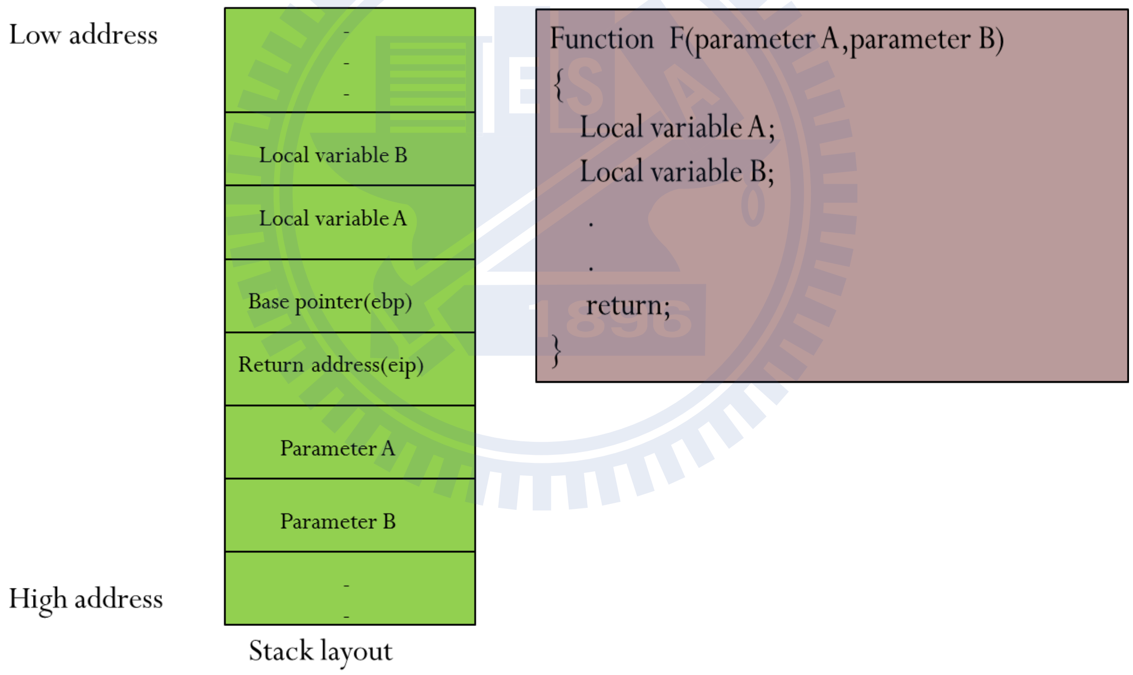


Figure 20 Example of stack allocation

In this function, the first parameter is address of the standard input buffer, continuing with the buffer size and name. The size is immaterial, just to record the overflow-able size we guess. At first, we mark the standard input as symbolic by calling *executeMakeSymbolic* function of KLEE, and iterate on each stack variables by memory offset map supports. For each stack variable, we determine if it can be overflowed by the standard input and mark as symbolic if yes.

$\mathcal{S}$: the simuStack

$\mathcal{O}$: the stack object in target program

$\mathcal{B}$: address of object

$\mathcal{SIZE}$: size of object

$\mathcal{NAME}$:name of object

```
Hsin_make_symbolic($\mathcal{B}$, $\mathcal{SIZE}$, $\mathcal{NAME}$){
    executeMakeSymbolic($\mathcal{B}$, $\mathcal{NAME}$)
    for each $\mathcal{O} \in \mathcal{S}${
        ...
        if $\mathcal{O}$ is overflow-able by $\mathcal{B}$
            executeMakeSymbolic($\mathcal{O}$)
        ...
    }
}
```

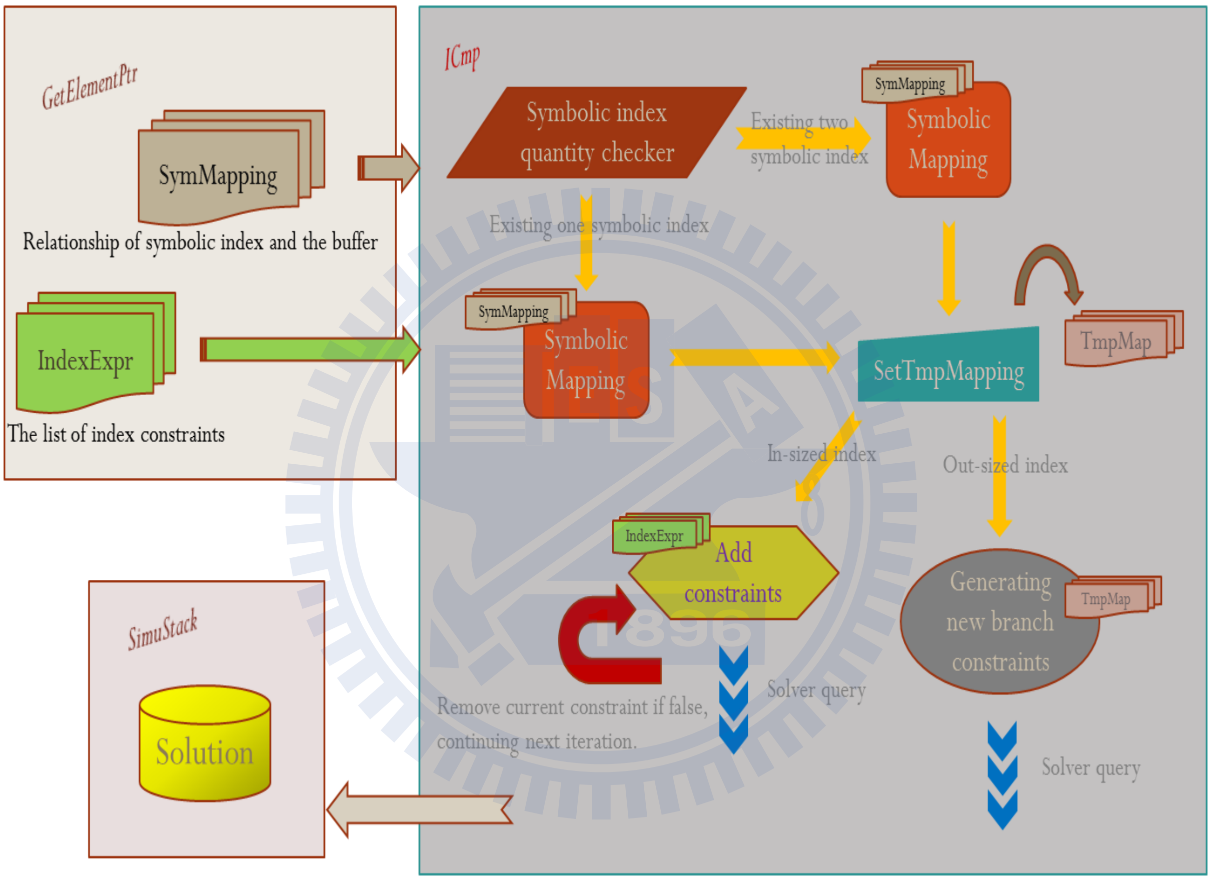
Figure 21 The *hsin_make_symbolic* function

As the example shown in Figure 15, the variable i is overflow-able by variable buf. In normal execution, KLEE iteratively generates the symbolic solutions for variable buf without including variable i. Because variable i is not symbolic, KLEE do not collect the constraints about variable i.

Handling by our new symbolic memory function, we automatically mark the overflow-able variables as symbolic for constraints generation. Therefore, the second parameter is immaterial, because we automatically determine the total size the buf can overflow.

4.3 Symbolic array index dereference handling

To handle symbolic array index, we modify the *executeInstruction* function, an instruction interpreter on KLEE. In *GetElementPtr* instruction, we generate index constraints to determine the index value and modify the *Icmp* instructions to compare each array offset for the index solutions.



GetElementPtr
SymMapping
Relationship of symbolic index and the buffer
IndexExpr
The list of index constraints
SimuStack
Solution
ICmp
Symbolic index quantity checker
Existing two symbolic index
SymMapping
Symbolic Mapping
Existing one symbolic index
SymMapping
Symbolic Mapping
SetTmpMapping
TmpMap
In-sized index
Out-sized index
IndexExpr
Add constraints
Generating new branch constraints
TmpMap
Solver query
Remove current constraint if false, continuing next iteration.
Solver query

Figure 22 Flowchart of the implementation

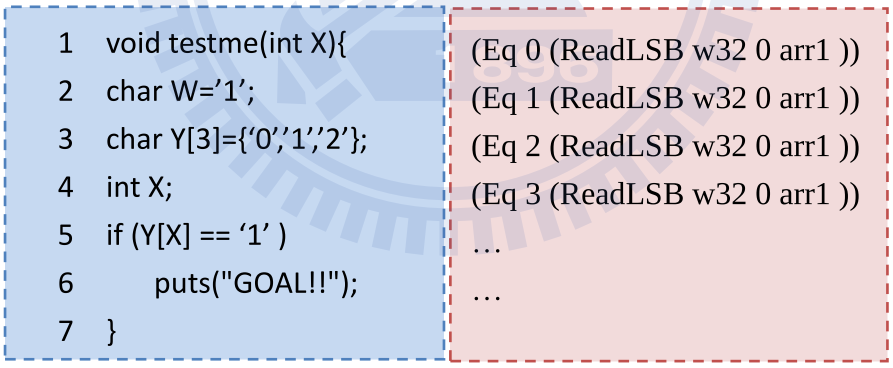
4.3.1 Memory dereference in LLVM IR form

LLVM uses *getelementptr* instruction to get the address of a subelement of an aggregate data structure. The arguments are a pointer value <ptrval> with its type <pty>*, and the index value <idx> with its type <ty>.

Table 4 The getelementptr instruction in LLVM

```
Syntax:
  <result> = getelementptr <pty>* <ptrval>{, <ty> <idx>}*
  <result> = getelementptr inbounds <pty>* <ptrval>{, <ty> <idx>}*
```

In general, the index value <idx> is a constant value or a calculated value, that is, a concrete value. However, we would get a symbolic value while symbolic index occurs. For instant as shown in figure 23 at line5, the array index is loaded from a symbolic variable X. KLEE represents symbolic index by an expression in *Kleaver* form and we need a constraint related to the expression to determine its value.



```
1   void testme(int X){
2   char W='1';
3   char Y[3]={'0','1','2'};
4   int X;
5   if (Y[X] == '1' )
6         puts("GOAL!!");
7   }
```

```
(Eq 0 (ReadLSB w32 0 arr1 ))
(Eq 1 (ReadLSB w32 0 arr1 ))
(Eq 2 (ReadLSB w32 0 arr1 ))
(Eq 3 (ReadLSB w32 0 arr1 ))
…
…
```

Figure 23 Example of symbolic index with its index constraints

We modify the instruction *GetElementPtr* interpreted by KLEE. Originally, KLEE would get a concrete index value at line2 in figure 24, and calculate the offset at line3. To handle symbolic array index, we construct constraints at each array index including overflow-able indices by memory offset map supports between line2 and line3.

```
$\mathcal{K}$ : the current instruction
$\mathcal{S}$ : the current execution state
$\mathcal{B}$ : the start address of the array
$\mathcal{E}$ : the element size of the array
$\mathcal{I}$ : the index expression

case   Instruction::GetElementPtr {
    $\mathcal{I} \leftarrow$ eval($\mathcal{K}$, $\mathcal{S}$ ).value     // Getting index from previous instruction
    $\mathcal{B}' \leftarrow$ AddExpr::create($\mathcal{B}$ , MulExpr::create( $\mathcal{I}$ , $\mathcal{E}$) ) // Calculating the offset
    bindLocal($\mathcal{K}$, $\mathcal{S}$, $\mathcal{B}'$ )        // Pushing the result into stack for next instruction
}
```

Figure 24 Original Instruction *GetElementPtr*

As shown in figure 25, we confirm the target buffer and identify if the index is symbolic. For each variable which allocated behind the target buffer, we generate index constraints for each offset. For instance, we generate four index constraints for an integer variable.

```
$\mathcal{IndexExpr}$  : a container used to store index constraints
$\mathcal{S}$ : the simuStack
$\mathcal{O}$ : the stack object in target program
$\mathcal{IC}$ : the index constraint
$\mathcal{I}$ : the index expression

for each $\mathcal{O} \in \mathcal{S}$ && $\mathcal{O}$ is current buffer && $\mathcal{I}$ exists {
  for each $\mathcal{O}'$ is overflow-able by the buffer {
    for each offset in $\mathcal{O}'$ {
      $\mathcal{IC} \leftarrow$ EqExpr::create( offset , $\mathcal{I}$ )   // To create an index constraint for current offset
      Store $\mathcal{IC}$ in $\mathcal{IndexExpr}$
    }
  }
}
```

Figure 25 Index constraints generation

4.3.2 Branch condition evaluation

LLVM use *icmp* instructions to process comparison in branch as shown in figure 26, including EQ、NE、LTE ,etc. In section 4.3.1, we generate index constraints in *GetElementPtr* instruction. In this section, we use the index constraints to determine the symbolic index value.

Original KLEE interprets comparison instructions as figure 27 shows. It simply compares the values, and pushes the result into the stack for value propagation.

```
%9 = getelementptr inbounds [3 x i8]* %buf, i32 0, i32 %8 ; <i8*> 2 [#uses=1]
%10 = load i8* %9, align 1                                ; <i8> [#uses=1]
%11 = icmp eq i8 %10, 48                                   ; <i1> [#uses=1]
br i1 %11, label %bb, label %bb1
```

Figure 26 Example of *icmp* instruction

$\mathcal{K}$: the current instruction

$\mathcal{S}$: the current execution state

$\mathcal{L}$: left value in condition

$\mathcal{R}$: right value in condition

$\mathcal{RES}$: the comparison constraint

```
case   ICmpInst::ICMP_EQ {
    $\mathcal{L} \leftarrow$ eval($\mathcal{K}$, 0 , $\mathcal{S}$).value
    $\mathcal{R} \leftarrow$ eval($\mathcal{K}$, 1 , $\mathcal{S}$).value
    $\mathcal{RES} \leftarrow$ EqExpr::create($\mathcal{L}$ ,$\mathcal{R}$)
    bindLocal($\mathcal{K}$, $\mathcal{S}$, $\mathcal{R}$)
    break
}
```

Figure 27 Comparison in Original KLEE

We write a function named *getSymIndexSolution* and insert between line4 and line5 in figure 27 to handle symbolic array index solution generation. For indices at inside scope, we iteratively add the each index constraint and query solver to determine the symbolic index solution as shown in figure 28 at line2 to line3.

```
IndexExpr : a container used to store index constraints
S : the current execution state
IC : the index constraint
RES : the comparison constraint
SIC : a containter used to store solutions
Sol : the solutions of symbolic index

1   for each IC ∈ IndexExpr at inside scopes {
2         addConstraint( S , IC)
3         Sol ← solver->query( S,RES)
4         if Sol exists                    // For solver
5               SIC.insert( Sol);
6   }
```

Figure 28 Symbolic array index handling at inside scope

To handle symbolic index at outside scope, due to KLEE did not support array index dereference at outside scope, we write a function named *setTmpMapping* to get the real values of overflow-able variables , and construct comparison constraints to compare the real value with the required value. As shown in figure 29, we can get the value of variable W with array index value 3.

```
void testme(int X){
char W='1';
char Y[3]={'0','1','2'};
int X;
if (Y[X] == '1' )
      puts("GOAL!!");
}
```

0	48
1	49
2	50
3	49

Figure 29 Example of the result of *setTmpMappinp*

KLEE records the value of each variable in class *ObjectState*. We use the object iterator to process each stack object behind the tainted buffer. As shown in figure 30 at line4, we process some complicated mapping works to get the corresponding array object, and then use the method *findObject* in class *addressSpace* to obtain the object state. Rely on the corresponding *objectstate*, we can use the method *read8* to get real values per byte.

```
S : the current execution state
OS : the current object state
SS: the simuStack
O : stack object
IT : the stack object iterator points to beginning of the tainted buffer
Array : the object represented in Kleaver constraint
Value : the real value in byte
MAP : a containter used to store the real values

1   setTmpMapping( S, MAP ,IT ){
2       for each O $\in$ SS  behind the tainted buffer {
3           O $\leftarrow$ *IT
4           Array $\leftarrow$ findArrayObject( O )
5           OS $\leftarrow$ S.addressSpace.findObject(Array)
6           IT $\leftarrow$ IT - 1
7            for each byte in O {
8               Value $\leftarrow$ OS->read8()
9               MAP.insert(O ,Value);
10           }
11      }
12  }
```

Figure 30 Function *setTmpMappinp*

Because KLEE didn't accept the array index dereference at outside scopes, we generate comparison constraint by ourselves.

$\mathcal{P}$ is an enumeration defined in LLVM used to identify the type of comparison. Depend on the result of *setTmpMapping*, we can construct the comparison constraints and query solver to get symbolic array index solutions at outside scope as shown in figure 31.

$\mathcal{S}$: the current execution state

IT : the stack object iterator points to beginning of the tainted buffer

MAP : a containter used to store the real values

$\mathcal{P}$: the comparison type in LLVM

$COND$: the constraint

$Index$: the index from the beginning of the tainted buffer

$\mathcal{R}$: the required value in original comparison constraint

SOL : the solutions of symbolic index

SIC : a containter used to store solutions

```
setTmpMapping( $\mathcal{S}$, $MAP$ ,$IT$)
for each $Index$  at outside scope {
      switch ( $\mathcal{P}$ ){
           case ICMP_EQ:
                $COND$ $\leftarrow$ EqExpr::create($MAP$ , $Index$ , $\mathcal{R}$ )
                break;
           case ICMP_NE:
                $COND$ $\leftarrow$ NeExpr::create($MAP$ , $Index$ , $\mathcal{R}$ )
                break;
           case ICMP_UGT:
                $COND$ $\leftarrow$ UgtExpr::create($MAP$ , $Index$ , $\mathcal{R}$ )
                break;
           …
           …
      }
      $SOL$ $\leftarrow$ solver->query ($\mathcal{S}$ , $COND$);
      if $Sol$ exists                          // For solver
             $SIC$.insert( $Sol$ );
 }
```

Figure 31 Symbolic array index handling at outside scope

In our implementation, we can also handle two or more symbolic array index in a branch condition by simply doubling the symbolic array index checking strategy as shown in figure 32.

IndexExpr : a container used to store index constraints

S : the current execution state

IC1 : the index constraint

IC2 : the index constraint

RES : the comparison constraint

SIC : a containter used to store solutions

Sol: the solutions of symbolic index

```
for each IC1 ∈ IndexExpr at inside scopes {
    for each IC2 ∈ IndexExpr at inside scopes {
        ... more for loop ...
        addConstraint( S , IC1 )
        addConstraint( S , IC2 )
        ... more constraints adding ...
        Sol ← solver->query( S,RES)
        if Sol exists                       // For solver
            SIC.insert( Sol);
    }
}
```

Figure 32 More than one symbolic array index handling at inside scope

4.4 Symbolic solution management

To preserve the correct symbolic execution, we generate the symbolic index solution and insert in corresponding position between each variable per iteration. KLEE handles solver querying in function *runAndGetCexForked* in file *solver.cpp*, the variable $\mathcal{POS}$ is a share memory used to record symbolic solutions. Original KLEE cannot generate symbolic solution when symbolic index appears. Therefore, we insert symbolic index solution when the symbolic index appears as shown in figure 33 at line2 to line5. We use array object to identify symbolic index and insert the solutions into the $\mathcal{POS}$ shared memory, the symbolic index solutions is propagated by $\mathcal{SIC}$.

$\mathcal{A}rray$: the object represented in Kleaver constraint

$\mathcal{SIC}$: a containter used to store solutions

$\mathcal{S}ol$: the solutions of symbolic index

$\mathcal{POS}$: the shared memory used to store symbolic solutions after solver querying

```
1   for each $\mathcal{A}rray$ in KLEE{
2       if $\mathcal{A}rray$ is symbolic index {                // our implementation
3           for each $\mathcal{S}ol \in \mathcal{SIC}$
4               if $\mathcal{S}ol$ is the solution of the $\mathcal{A}rray$
5                   $\mathcal{POS} \leftarrow \mathcal{S}ol$
6       } else                                              // original works
7           $\mathcal{POS} \leftarrow$ solver->getsolution($\mathcal{A}rray$)
8   }
```

Figure 33 Symbolic solution fixing in *solver.cpp*

5 Result and Experiment

In this section, we present results of preliminary experiments with our symbolic array index algorithm. At first, we use the example of SAGE to simply illustrate the purpose of symbolic array index. Next, we evaluate on some real programs to verify our algorithm and analyze some well-known exploitable program on symbolic tainting. We evaluate our algorithm on a machine with a Intel(R) Pentium(R) 4 3.40GHz cpu and 1.5GB of RAM and perform the experiments under Ubuntu 2.6.32-24-generic-pae. We use llvm-gcc 4.2-2.7 to compile our test programs. The experiment results were supplied by the statistics in KLEE functions.

5.1 Trivial example of SAGE

At first, we test the example of SAGE. As shown in figure 34, the program has two inputs X and Y. Our goal is to pass the branch at line10. In our implementation, we will collect index constraints for X and Y, and add these constraints in each kinds of constraint pair for X and Y. Finally we can get not only one solution, there are two kinds of solution pair in this program (X,Y)=(0,3) and (X,Y)=(1,3).

```
1 #include<stdio.h>
2 int main(void){
3         int X,Y;
4         char A[4];
5         scanf("%d %d",&X,&Y);
6         A[0]=X;
7         A[1]='0';
8         A[2]='1';
9         A[3]='2';
10        if(A[X] == A[Y]+2)
11                puts("GOAL!!");
12        return 0;
13 }
```

```
KLEE: done: total instructions = 9753
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
```

Figure 34 Example of SAGE tool

5.2 Evaluations on real programs

Depend on our symbolic array index algorithm, we can explore more program paths by feeding some atypical input to programs. To verify our methods, we do some immaterial modification on our benchmarks.

1. Extract the target function as a main function with standard inputs
2. Change the target buffer from integer buffer or others to a character buffer
3. Insert *klee_make_symbolic* or *hsin_make_symbolic* into target code

Our purpose is to illustrate the enhancement on path coverage, the modifications are not affect out results. Experiments are performed with a wargame in our program security course, Snort[32] : a network intrusion detection system and Asterisk[33] : a transformer of a computer to a communication server.

In wargame1, the index value i is probably overflowed by variable buf. Therefore, variable i will be a symbolic variable, and we can collect related constraints for solver as shown in figure 35 at line 22. In figure 36, the function *prmAddRule* of Snort construct the rule mapping with specific ports, hence it is a situation of symbolic array index dereference.

In function *__ast_string_field_index_build_va* of Asterisk as shown in figure 37, the variable index is different to the variable dport in Sonrt. The variable dport is restricted at line3 in figure 36. However, the variable index do not have any restriction, hence it can be assigned to some bigger enough value to look over the out-of-bounds scope for solutions.

Results are summarized in table 5. We solve the symbolic array index to explore the two sides of branch with no doubt. Especially in Asterisk, we may feed an out-of-bounds value to variable index to pass the branch predicate.

Table 5 Results on experiments of symbolic array index

Program	WallTime	Num. of test case	Num. of symbolic index	Path coverage
example of SAGE	0.136289	2	2	2
wargame1	0.368502	27	1	32
snort 2.9.0.4	0.098151	4	1	4
asterisk-1.4.38-rc1	0.110749	4	1	4

5.3 Atypical symbolic analysis

In this section we analyze the symbolic tainting on some vulnerable programs. According to symbolic execution, a variable becomes symbolic if its value is assigned from a symbolic variable. More the variables we can taint, more the program paths we can explore. Therefore, we analyze how many non-symbolic variables we can taint.

Experiments are performed with four programs about buffer overflow vulnerability announced in CVE [34], including CoreHTTP 0.5.3.1, Htget 0.93, Nocompress 4.2.4 and iwconfig. We count the number of tainted variables in vulnerable function, and divide into four types.

1. Tainted arguments: If a callee function argument is type of call-by-reference(address), it may taint other variables in caller function, hence we focus on this kind of argument only.
2. Symbolic pointer: If a function exist symbolic pointer, we may taint variables at any address.
3. Symbolic index: Similar to symbolic pointer, we may taint variables at any address by giving an appropriate index value.
4. Tainted variables: Except the mentions above.
5. Tainted eip: Theoretically, eip is taintable if the vulnerable function includes strcpy() calls.

Results are summarized in table 6. Normally, a symbolic buffer may taint the variables at higher memory address, even the eip, ebp and arguments. Therefore, if a function exist lots of local variables, the number of tainted variables is comparatively many, and the possibility of exploit also increases. Especially in CoreHTTP 0.5.3.1, we may execute malicious codes by assigning the start address of malicious code to the symbolic pointer without the assistance of tainted eip.

Table 6 Results of tainting analysis

Program	**Num. of symbolic variables**	**Num. of symbolic arguments**	**Num. of symbolic index**	**Num. of symbolic pointer**	**Tainted eip**	**Tatal**
CoreHTTP 0.5.3.1	8	0	0	2	Y	10
Htget 0.93	2	4	0	0	Y	6
Nocompress 4.2.4	3	0	0	0	Y	3
iwconfig	1	1	0	0	Y	2

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
char pass[8];
int main(void){
        FILE *fp;
        int i = 0, auth = 0;
        char buf[8];

        printf("Input passwd: ");
        fgets(buf, 20, stdin);

        if ((fp = fopen("/home/chiush/wargame/passwd", "r")) == NULL) {
                return 1;
        }
        fgets(pass, sizeof(pass), fp);
        pass[strlen(pass)-1] = '\0';

        for ( ; i < strlen(buf); ++i)
                if (buf[i]<'a'|| buf[i]>'z')
                        return 1;
        if (!strcmp(buf, pass))
                auth = 1;
        if (auth == 1 && buf[0] == '0'){
                char fname[32];
                uid_t uid = getuid();
                sprintf(fname, "/home/chiush/wargame/checkin/%u", uid);
                open(fname, O_CREAT | O_WRONLY, 0000);
        }
        return 0;
}
```

Figure 35 Wargame1

```
int prmAddRule( PORT_RULE_MAP * p, int dport, int sport, RULE_PTR rd )
{
    if( dport != ANYPORT && dport < MAX_PORTS ){
        p->prmNumDstRules++;
        if(p->prmDstPort[dport] == NULL){
            p->prmDstPort[dport] = (PORT_GROUP *)calloc(1, sizeof(PORT_GROUP));
            if(p->prmDstPort[dport] == NULL)
                return 1;
        }
        if(p->prmDstPort[dport]->pgCount==0) p->prmNumDstGroups++;
        prmxAddPortRule( p->prmDstPort[ dport ], rd );
    }
    …
    …
```

Figure 36 Symbolic index in Snort

```
void __ast_string_field_index_build_va(struct ast_string_field_mgr *mgr,
                                       ast_string_field *fields, int num_fields,
                                       int index, const char *format, va_list ap1, va_list ap2){
    size_t needed;
    size_t available;
    char *target;
    if (fields[index][0] != '\0') {
        target = (char *) fields[index];
        available = strlen(fields[index]) + 1;
    } else {
        target = mgr->pool->base + mgr->used;
        available = mgr->space;
    }
    needed = vsnprintf(target, available, format, ap1) + 1;
    va_end(ap1);
    if (needed > available) {
        if (needed <= mgr->space) {
            target = mgr->pool->base + mgr->used;
        } else {
            size_t new_size = mgr->size * 2;
            while (new_size < needed)
                new_size *= 2;
            if (add_string_pool(mgr, new_size))
                return;
            target = mgr->pool->base + mgr->used;
        }
        vsprintf(target, format, ap2);
    }
    if (fields[index] != target) {
        fields[index] = target;
        mgr->used += needed;
        mgr->space -= needed;
    }
}
```

Figure 37 Symbolic index in Asterisk

6 Conclusion

The issue on buffer overflow is getting more and more important recently. In this thesis, we propose a new viewpoint on buffer overflow. Unlike other analysis tools, we focus on exploiting program bugs to explore more program paths. By exploiting symbolic array index, we can perform abnormal control flow and execution flow finding to explore more program path we cannot reach before. To reach the goal, we construct a new memory offset map on KLEE and modify KLEE symbolic execution model.

The ultimate objective is to generate an exploitable program input. A general notion for exploiting is (**tainted-pointer = tainted-value)*, that is, if we has a pointer which can be tainted by standard input and a large enough serial memory space to insert shell code, then we can fully control the target program to follow our inclinations.

7 Future works

We just propose a concept and simply write an algorithm to meditate our idea based on KLEE. However, KLEE is a testing platform at most because of the lack of reality. We suggest to research on a real environment to obtain the real memory address information, and we can exploit buffer overflow in true.

In this thesis we handle buffer overflow problem only on stack region, but the program is common on heap region too. To handle buffer overflow on heap region, we need the real address of all of objects. Therefore, a real memory model is needed.

By constructing a symbolic memory model to record all address of symbolic variables and writing an algorithm to determine the value (address) of a symbolic pointer, we can control program path to follow our inclinations.

Reference

[1] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In CGO , 2004.

[2] C. Cadar, D. Dunbar, D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,In Stanford University 2008

[3] S. Nagarakatte, J. Zhao, M. Martin, S. A. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C, In University of Pennsylvania 2009

[4] E. Haugh and M. Bishop. Testing C programs for buffer overflow vulnerabilities. In Proceedings of the Network and Distributed System Security Symposium, February 2003.

[5] P. Godefroid, N. Klarlund, K. Sen. DART: directed automated random testing, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, June 12-15, 2005, Chicago, IL, USA

[6] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. Technical Report UIUCDCS-R-2005-2597, UIUC, 2005.

[7] R. Majumdar and K. Sen. Hybrid concolic testing. In 29th International Conference on Software Engineering (ICSE'07), pages 416{426. IEEE, 2007.

[8] Koushik Sen. Concolic testing. ASE 2007

[9] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation, in Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 89-100.

[10] W. Le and M. L. Soffa. Refining buffer overflow detection via demand-driven path-sensitive analysis, in Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2007, pp. 63-68.

[11] Cristian Cadar, Paul Twohey, Vijay Ganesh, Dawson Engler. EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution, 2006

[12] J.Yang, C. Sar, P. Twohey, C. Cadar and D. Engler. Automatically Generating Malicious Disks using Symbolic Execution , Stanford University Computer Systems Laboratory

[13] Z. Lin X. Zhang D. Xu. Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach, Department of Computer Sciences and CERIAS Purdue University

[14] H. Shahriar and M. Zulkernine. Mutation-based Testing of Buffer Overflow

Vulnerabilities , School of Computing Queen’s University, Kingston, Ontario, Canada

[15] O. Crameri, R. Bachwani, T. Brecht, R. Bianchini, D. Kostic, W.Zwaenepoel. Oasis: Concolic Execution Driven by Test Suites and Code Modifications , EPFL Technical report

[16] D.Vanoverberghe , N. Tillmann , F. Piessens. Test Input Generation for Programs with Pointers, Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York UK, March 22-29 2009

[17] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, G. Candea. Cloud9: A Software Testing Service , School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

[18] B. Elkarablieh P. Godefroid M.Y. Levin. Precise Pointer Reasoning for Dynamic Test Generation, 2009

[19] J. Burnim K. Sen. CREST : Heuristics for Scalable Dynamic Test Generation. Presented at 23rd IEEE/ACM International Conference on Aitomated Software Engineering, ASE 2008

[20] J. C. King. Symbolic Execution and Program Testing, Communications
of the ACM, vol. 19, no. 7, pp. 385–394, 1976.

[21] R. Majumdar and K. Sen. Latest: Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, 2007.

[22] O. Ruwase and M. S. Lam. CRED : A practical dynamic buffer overflow detector. In Proceedings of the 11th Annual Network and Distributed System Security Symposium,pages 159–169, 2004

[23] Y. Younan , W.Joosen and F. Piessens. Security of memory allocators for C and C++. Department of Computer Science, K.U.Leuven, 2005

[24] Uno : http://spinroot.com/uno/

[25] ZZUF : http://caca.zoy.org/wiki/zzuf

[26] You-Siang Lin. CAST: Automatic and Dynamic Software Verification Tool, NCTU , Master thesis, 2009

[27] Richard W M Jones and Paul H J Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. Department of Computing Imperial College if Science, Technology and Medicine 180 Queen’s Gate, London.

[28] KLEE : http://klee.llvm.org/

[29] LLVM : http://llvm.org/

[30] CERT advisorie : http://www.cert.org/advisories/
[31] Cyber Security Bulletins : http://www.us-cert.gov/cas/bulletins/
[32] Snort 2.9.4.0 : http://www.snort.org/
[33] Asterisk : http://www.asterisk.org/downloads
[34] CVE : http://nvd.nist.gov/home.cfm