Software Security

Program Analysis Techniques（3）

Symbolic Execution

Guosheng Xu,

guoshengxu@bupt.edu.cn

Outline

- Techniques used in Security Analysis
- Basic Program Analysis
  - Control flow analysis
  - Data flow analysis
- Taint Analysis
- **Symbolic execution**
- **Fuzzing**

Symbolic Execution

- What is symbolic execution?
  - Concrete execution versus symbolic execution
  - Symbolic execution tree
- Applications of symbolic execution
  - test input generation, infeasible paths detection, bug finding, program repair, debugging
- The three challenges
  - Path explosion
  - Modeling statements and environments
  - Constraint solving
- Implementation and symbolic execution tools

Take Vulnerability Detection As an Example..

- Researchers have explored program analysis methods to find flaws that may lead to vulnerabilities
- Dynamic analysis
  - ***Dynamic program analysis*** *is the analysis of computer software that is performed by running the program*
  - Can perform dynamic analysis easily on binaries
- Static analysis
  - ***Static program analysis*** *is the analysis of computer software that is performed without running the program*
  - Generally, use the source code, but not required

Dynamic Analysis

- Given the code, find whether the code may reach an execution state that fails an invariant
  - Choose sequences of inputs to provide to the program
    - The choice of sequences can include random inputs (*fuzzing*)
  - See what happens – does it violate an invariant?
    - Does it crash?
  - Keep trying – *must try all paths on all inputs*
    - Not much in the way of feedback usually

Dynamic Analysis - Limits

- Inherently results in false negatives
  - Attacks require special conditions
    - Current working directory, links, …
    - Can't run all cases…

Static Analysis

- Given the code, find whether the code may reach an execution state that fails an invariant
  - grep – find existence of a dangerous command
    - Syntactic
  - Examine possible executions
    - Semantic
    - "Run program in aggregate" – maintain an approximation of the possible values that can reach a statement
    - "Run in a non-standard way" – evaluate each function separately and stitch their executions together
- There is a lot of theory and experience on this!

Static Analysis

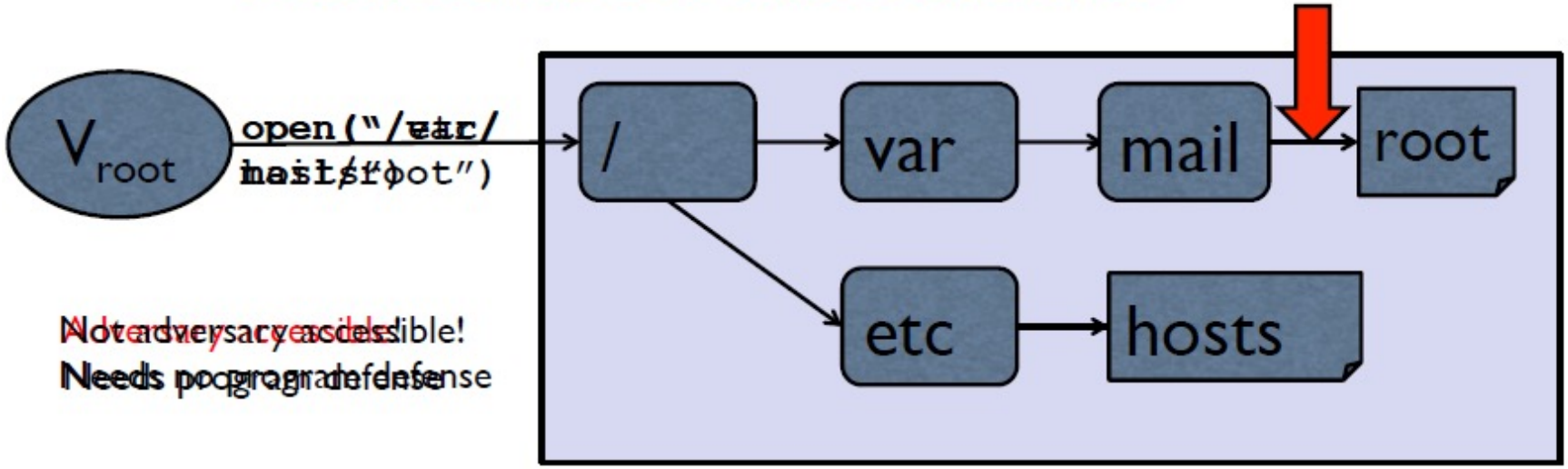
- "Run program in aggregate" – maintain an approximation of the possible values that can reach a statement
- "Run in a non-standard way" – evaluate each function separately and stitch their executions together

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10*
4 :     if (*arg == '\\') {                                  11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:           i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:       } else if (*arg != '\0')
14:         *buffer++ = *arg++;
15:     } else if (*arg == '[') {                            12*
16:       arg++;                                             13
17:       i = *arg++;                                        14
18:       if (*arg++ != '-') {                               15!
19:         *buffer++ = '[';
20:         arg -= 2;
21:         continue;
22:       }
23:       ac = *arg++;
24:       while (i <= ac) *buffer++ = i++;
25:       arg++;       /* Skip ']' */
26:     } else
27:       *buffer++ = *arg++;
28:   }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

Figure 1: Code snippet from MINIX's `tr`, representative of the programs checked in this paper: tricky, non-obvious, difficult to verify by inspection or testing. The order of the statements on the path to the error at line 18 are numbered on the right hand side.

Static Analysis - Limits

- Analyze program to find potentially vulnerable name resolution calls
  - Due to complexity of checks, mainly limited to TOCTTOU
- Deficiencies
  - False positives due to adversary inaccessibility
  - Our runtime study found only around 5% of name resolutions were accessible to adversaries



Symbolic Execution

- Analysis of a program using symbolic values rather than concrete
  - Similar to dynamic analysis in that programs are "executed" sequentially
    - Must evaluate all paths to find all flaws
    - But, not for all values
  - Similar to static analysis in that a symbolic value represents all possible values
    - Maintain symbolic values representing all possible
    - But, build model incrementally with execution
      - Eliminate unnecessary paths, take preferred paths

Concrete Execution Versus Symbolic Execution

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i ;
        return i;
}
```

i = 1

i = 1, j = 2

i = 2, j = 2

i = 4, j = 2

return 4

Concrete Execution Versus Symbolic Execution

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        return i;
}
```

i_{input}

$i = i_{input}$, $j = 2 * i_{input}$

$i = i_{input} + 1$, $j = 2 * i_{input}$

$i = 2 * i_{input} \wedge 2 + 2 * i_{input}$

Concrete Execution Versus Symbolic Execution

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        return i;
}
```

i_{input}

$i = i_{input}$, $j = 2 * i_{input}$

$i = i_{input} + 1$, $j = 2 * i_{input}$

$i = 2 * i_{input}\,\hat{}\,2 + 2 * i_{input}$

$i = -2 * i_{input}\,\hat{}\,2 - 2 * i_{input}$
$(2 * i_{input}\,\hat{}\,2 + 2 * i_{input} < 1)$

OR

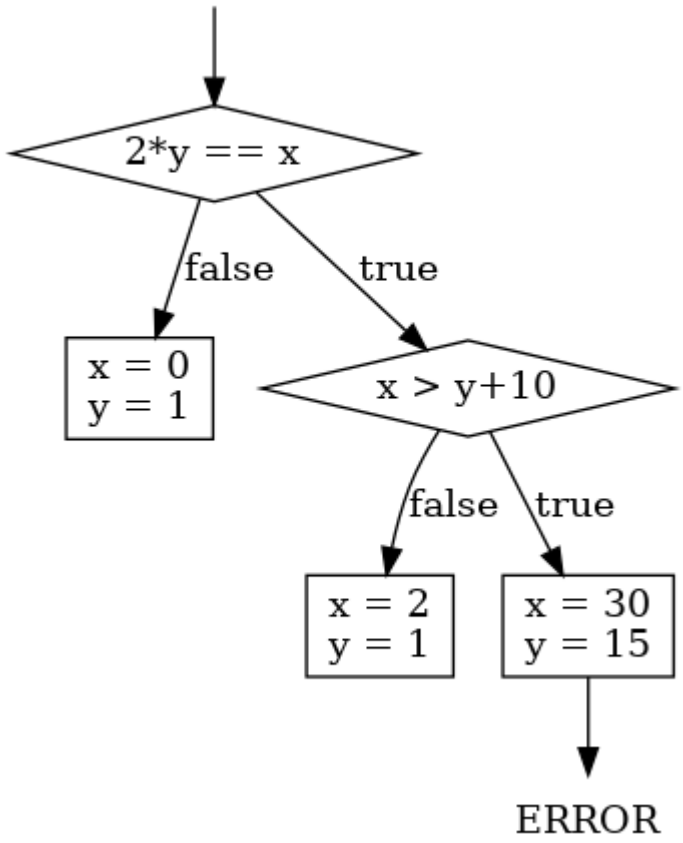
$i = 2 * i_{input}\,\hat{}\,2 + 2 * i_{input}$
$(2 * i_{input}\,\hat{}\,2 + 2 * i_{input} >= 1)$

Some Insights about Symbolic Execution

- **'Execute' programs with symbols:** we track symbolic state rather than concrete input
- **'Execute' many program paths simultaneously:** when execution path diverges, fork and add constraints on symbolic values
- **When 'execute' one path, we actually simulate many test runs**, since we are considering all the inputs that can exercise the same path

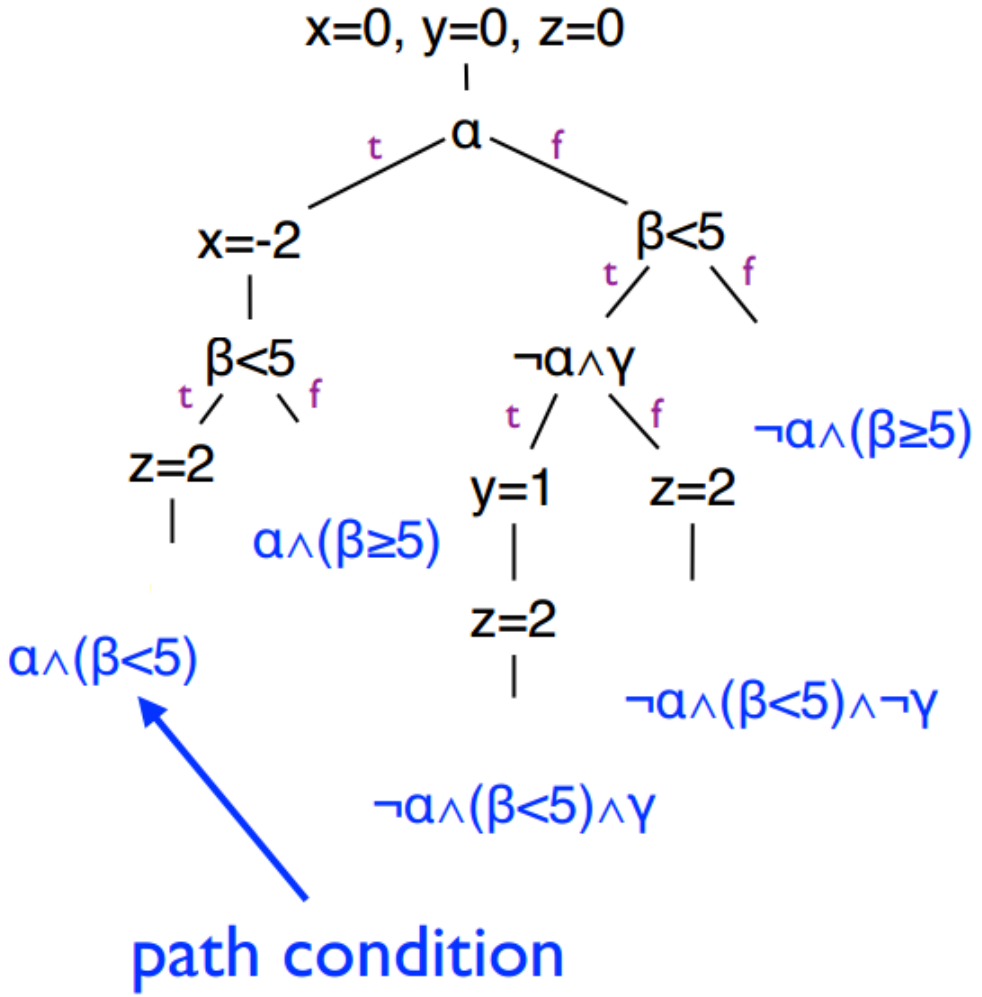
Symbolic Execution Tree

```
void testme(int x, int y) {
        z = 2*y;
        if (z == x) {
                if (x > y+10){
                ERROR;
                }
        }
}
int main()
{
        x = sym_input();
        y = sym_input();
        testme(x, y);
        return 0;
}
```



Symbolic Execution Tree

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



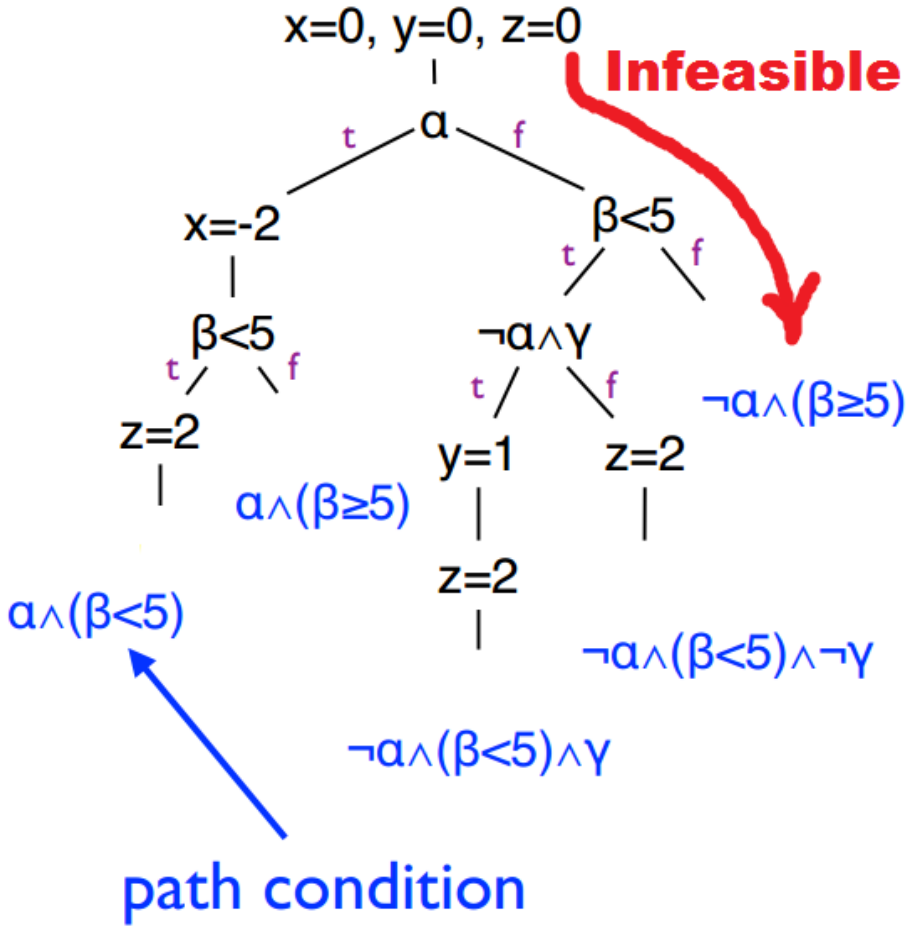
Applications of Symbolic Execution

- General goal: identifying semantics of programs
- Basic applications:
  - Detecting infeasible paths
  - Generating test inputs
  - Finding bugs and vulnerabilities
  - Proving two code segments are equivalent
- Advanced applications:
  - Generating program invariants
  - Debugging
  - Repair programs

Detecting Infeasible Paths

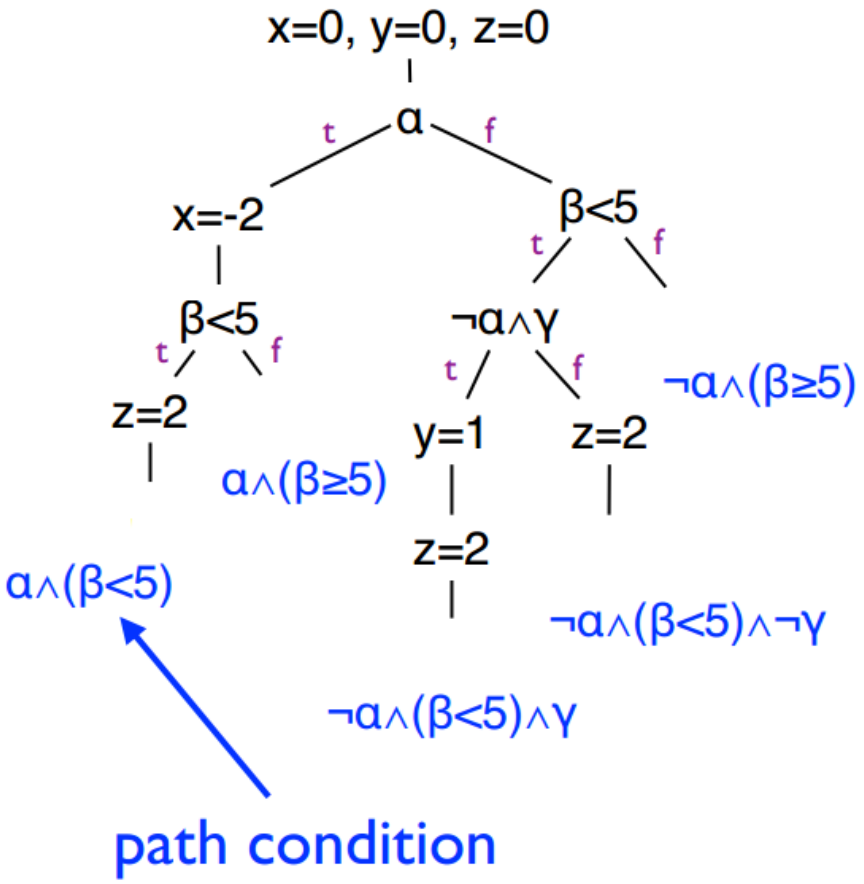
Suppose we require $\alpha = \beta$

```
int a = α, b = β, c = γ;
                  // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Test Input Generation

```
int a = α, b = β, c = γ;
                  // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Path 1: $\alpha = 1, \beta = 1$
Path 2: $\alpha = 1, \beta = 6$
Path 3 ...

Bug Finding

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        i = j/i;
        return i;
}
```

i_{input}

True branch:

$2 * i_{input} \hat{} 2 + 2 * i_{input} < 1$

$i = - 2 * i_{input} \hat{} 2 - 2 * i_{input}$

$i == 0$

False Branch:

$2 * i_{input} \hat{} 2 + 2 * i_{input} >= 1$

$i = 2 * i_{input} \hat{} 2 + 2 * i_{input}$

$i == 0$

Bug Finding

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        i = j/i;
        return i;
}
```

i_{input} i_{input} = -1 Trigger the bug

True branch:

$2* i_{input} ^2 + 2* i_{input} < 1$

$i = - 2* i_{input} ^2 - 2* i_{input}$

$i == 0$

False Branch: **always safe**

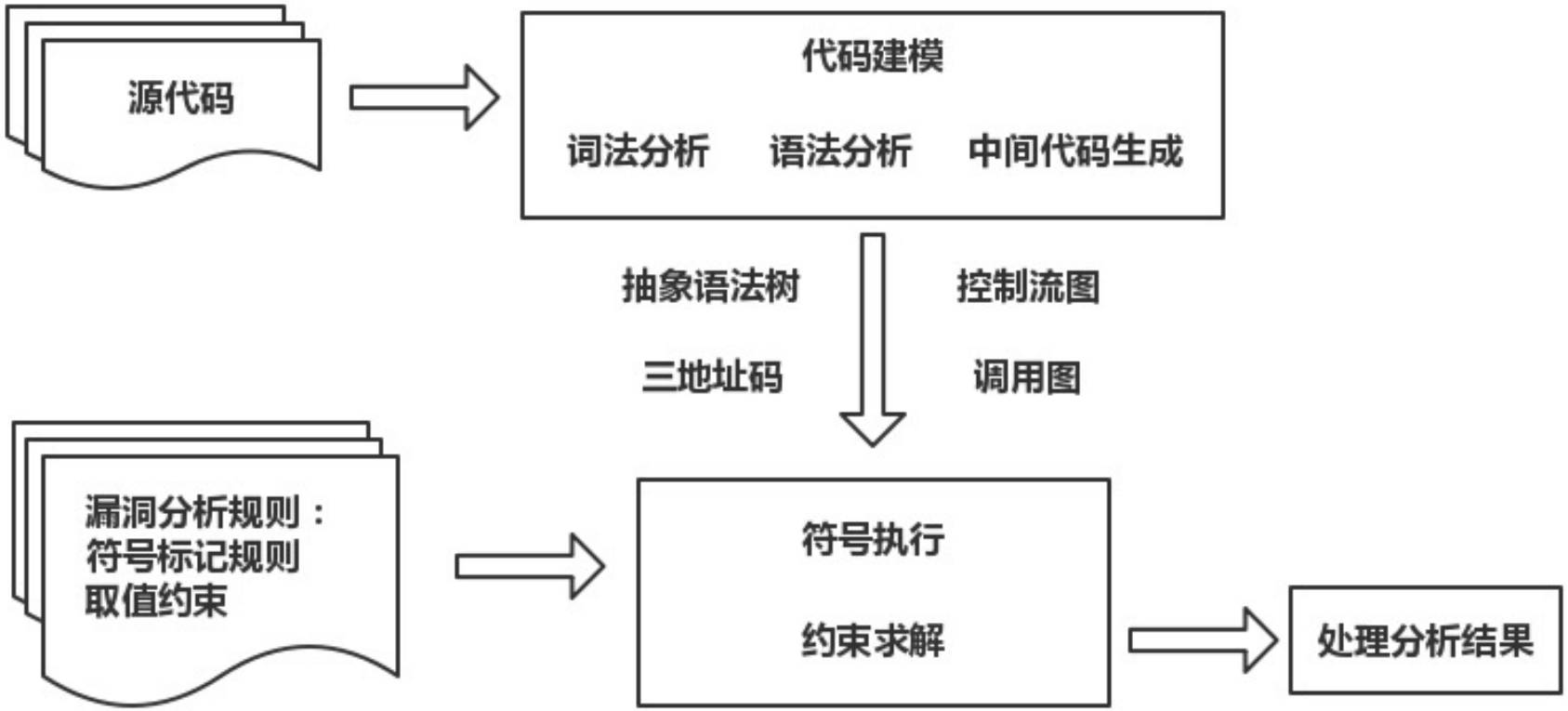
$2* i_{input} ^2 + 2* i_{input} >= 1$

$i = 2* i_{input} ^2 + 2* i_{input}$

$i == 0$

程序中变量的取值可以被表示为符号值和常量组成的计算表达式，而一些程序漏洞可以表现为某些相关变量的取值不满足相应的约束，这时通过判断表示变量取值的表达式是否可以满足相应的约束，就可以判断程序是否存在相应的漏洞。

Implementation of Symbolic Execution



正向的符号执行

- 正向的符号执行用于全面地对程序代码进行分析，可分为过程内分析和过程间分析。
- **过程内分析**逐句地地过程内的程序语句进行分析
- 声明语句分析
  - 通过声明语句，变量被分配到一定大小的存储空间，在检测缓冲区溢出漏洞时，需要记录这些存储空间的大小。
  - 分析声明语句的另一个目的是发现程序中的全局变量，记录全局变量的作用范围，这将有助于过程间分析。
- 赋值语句分析
  - 将赋值变量的取值表示为符号和常量的表达式。
  - 在检查程序漏洞时，常常对数组下标进行检查，判断对数组元素的访问是否存在越界。
  - 对于和指针变量有关的赋值语句，不仅需要考虑指针变量本身的取值，还需要考虑其指向的内容。

正向的符号执行

- 控制转移语句分析
  - 将路径条件表示为符号取值的约束并进行求解，可以判断路径是否可行，进而对待分析的路径进行取舍。
- 调用语句分析
  - 一些过程调用语句会进入符号，在分析过程中，将表示程序输入的变量的取值用符号表示，而程序可以通过过程调用接收程序的输入。对于指针变量，命令行参数同样使用符号表示其取值。
  - 通过过程调用语句，变量被分配的存储空间的大小常常是在分析时所需要记录的。
  - 对于一些关键的过程调用，需要对其使用情况进行检查，如strcpy，需要检查参数以判断是否存在缓冲区溢出。
  - 对于一些库函数或者系统调用等非程序代码实现的过程，用摘要描述所关心的分析过程和结果，可以避免重复分析。

正向的符号执行

- **过程间分析**常常需要考虑按照怎样的顺序分析程序语句，如深度优先遍历和广度优先遍历
- 另外在进行分析时，需要先确定一个分析的起始点，可以是程序入口点、程序中某个过程的起始点或者某个特定的程序点

逆向的符号执行

- 逆向的符号执行用于对可能存在漏洞的部分代码进行有针对性的分析
- 通过分析这些程序语句，可以得到变量取值满足怎样的约束表示程序存在漏洞，将这样的约束记录下来，在之后的分析中，通过逆向分析判断程序存在漏洞的约束是否是可以满足的。通过不断地记录并分析路径条件，检查程序是否可能存在带有程序漏洞的路径

逆向的符号执行

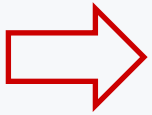
```
if (j > -6)
{
        a = i;
        i = j + 6;
        if (i < 15) {
                if (flag == 0) {
                        a [i] = 1;
                }
        }
}
```

从语句 a[i]=1 开始，逆推

⇒ i<15 and flag==0 and i<0 and i>len(a)

逆向的符号执行

```
if (j > -6)
{
        a = i;
        i = j + 6;
        if (i < 15) {
                if (flag == 0) {
                        a [i] = 1;
                }
        }
}
```



j+6<15 and flag==0 and j+6<0 and j+6>len(a)

碰到赋值语句且赋值变量和路径条件相关时，可以根据赋值语句所示的变量取值之间的关系更新当前路径条件

逆向符号执行的过程间分析

- 当过程内分析中遇到不能根据语义进行处理的过程，这些过程是程序实现的，并且影响所关心的存在漏洞的约束时
  - 通常选择直接对调用的过程进行过程内分析。
- 当过程内分析已经到达过程的入口点，且仍然无法判断存在漏洞的约束是否一定不可满足时
  - 可以根据调用图或其他调用关系找到调用该过程的过程，然后从调用点开始继续逆向分析。

实例分析

```
#define ISDN_MAX_DRIVERS 32
#define ISDN_CHANNELS 64

static struct isdn driver *drivers[ISDN_MAX_DRIVERS];
static struct isdn driver *get_drv_by_nr(int di) {
    unsigned long flags;
    struct isdn driver *drv;
    if (di < 0)
        return NULL;
    spin_lock_irqsave(&drivers lock, flags);
    drv = drivers[di];
    ......
}
static struct isdn slot *get_slot_by_minor(int minor) {
    int di, ch;
    struct isdn driver *drv;
    for (di = 0; di < ISDN_CHANNELS; di++) {
        drv = get_drv_by_nr(di);
        ......
    }
}
```

Challenges of Symbolic Execution

- Path explosion
- Constraint solving

Path Explosion

- Exponential in branching structure

```
1. int a = α, b = β, c = γ;    // symbolic
2. if (a) ... else ...;
3. if (b) ... else ...;
4. if (c) ... else ...;
```

  - Ex: 3 variables, 8 program paths

- Loops on symbolic variables even worse

```
1. int a = α;    // symbolic
2. while (a) do ...;
3.
```

  - Potentially 2^31 paths through loop!

How to address path explosion?

- **Search Strategies: Naive Approach**
- **DFS (depth first search),**

  BFS (breadth first search)
- **The two approaches purely are based on the structure of the code**
- You cannot enumerate all the paths
  - DFS: search can stuck at somewhere in a loop
  - BFS: very slow to determine properties for a path if there are many branches

Search Strategies: Random Search

How to perform a random search?

- Idea 1: pick next path to explore uniformly at random
- Idea 2: randomly restart search if haven't hit anything interesting in a while
- Idea 3: when have equal priority paths to explore, choose next one at random
- ...

Drawback: reproducibility, probably good to use psuedo-randomness based on seed, and then record which seed is picked

Search Strategies: Coverage Guided Search

Goal: Try to visit statements we haven't seen before

Approach:

- Select paths likely to hit the new statements
- Favor paths on recently covering new statements
- Score of statement = # times its been seen and how often; Pick next statement to explore that has lowest score

Pros and cons:

- Good: Errors are often in hard-to-reach parts of the program, this strategy tries to reach everywhere.
- Bad: Maybe never be able to get to a statement

Search Strategies: Generational Search

- Hybrid of BFS and coverage-guided search
- Generation 0: pick one path at random, run to completion
- Generation 1: take paths from gen 0, negate one branch condition on a path to yield a new path prefix, find a solution for that path prefix, and then take the resulting path
- ...
- Generation n: similar, but branching off gen n-1 (also uses a coverage heuristic to pick priority)

Search Strategies: Combined Search

- Run multiple searches at the same time and alternate between them
- Depends on conditions needed to exhibit bug; so will be as good as *best* solution, with a constant factor for wasting time with other algorithms
- Could potentially use different algorithms to reach different parts of the program

Constraint solving

SAT: find an assignment to a set of Boolean variables that makes the Boolean formula true

Complexity: NP-Complete

Constraint solving

The State of the Art: Handle linear integer constraints

Challenges:

- Constraints that contain non-linear operands, e.g., sin(), cos()
- Float-point constraints: no theory support yet, convert to bit-vector computation
- String constraints: a = b.replace('x', 'y')
- Quantifies: $\exists$, $\forall$
- Disjunction

Symbolic Execution Tools

KLEE [1]

- Open source symbolic executor
- Runs on top of LLVM
- Has found lots of problems in open-source software

SAGE [3]

- Microsoft internal tool
- Symbolic execution to find bugs in file parsers - E.g., JPEG, DOCX, PPT, etc
- Cluster of n machines continually running SAGE

Further Reading

- All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution

https://users.ece.cmu.edu/~aavgerin/papers/Oakland10.pdf

- Symbolic execution for software testing: three decades later

https://people.eecs.berkeley.edu/~ksen/papers/cacm13.pdf