Introduction to Symbolic Execution

Yang Hu
huyang@utexas.edu
Sep. 4, 2024

Guest Lecture for CS6501, UVA

Basics in Software Testing

- **Test Oracle:** Test Input ✕ Test Output → {Buggy, Normal}
- **Coverage Metric**
- **Testing Approaches**

Basics in Software Testing

- **Test Oracle:** Test Input ✕ Test Output → {Buggy, Normal}
- **Coverage Metric**
- **Testing Approaches**
  - Reasoning based approaches (e.g., symbolic execution)

Basics in Software Testing - Symbolic Execution

Basic Idea: collect and solve path conditions to generate test inputs

```
void buggy_function(char a, char b, char c, char d) {
    if (a + b == 'B') // C1
        if (b + c == 'U') // C2
            if (c + d == 'G') // C3
                if(d == '!') // C4
                    crash(); /* buggy basic block */
}
```

Reasoning Problem: C1 ∧ C2 ∧ C3 ∧ C4 → **Automated Reasoning Tool** → **Test Input**

Basics in Software Testing - Symbolic Execution

Basic Idea: collect and solve path conditions to generate test inputs

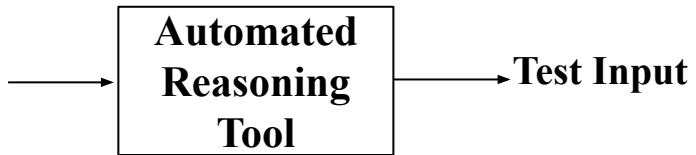
```
void buggy_function(char a, char b, char c, char d) {
    if (a + b == 'B') // C1
        if (b + c == 'U') // C2
            if (c + d == 'G') { // C3
                d = d + 1; // d's value is changed!!!
                if(d == '!') // C4
                    crash(); /* buggy basic block */
            }
}
```

Basics in Software Testing - Symbolic Execution

Basic Idea: collect and solve path conditions to generate test inputs

```
void buggy_function(char a, char b, char c, char d0) {
    if (a + b == 'B') // C1
        if (b + c == 'U') // C2
            if (c + d0 == 'G') { // C3
                char d1 = d0 + 1; // C5
                if(d1 == '!') // C4
                    crash(); /* buggy basic block */
            }
}
```

Reasoning Problem: C1 ∧ C2 ∧ C3 ∧ C4 ∧ C5

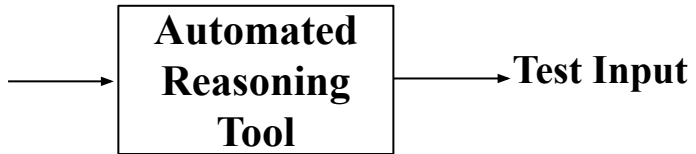


Basics in Software Testing - Symbolic Execution

Basic Idea: collect and solve path conditions to generate test inputs

```
void buggy_function(char a, char b, char c, char d) {
    if (a + b == 'B') // C1 // d <- dinit
        if (b + c == 'U') // C2 // d <- dinit
            if (c + d == 'G') { // C3 // d <- dinit
                d = d + 1; // d <- dinit + 1
                if(d == '!') // C4 // d <- dinit + 1
                    crash(); /* buggy basic block */
            }
}
```

Reasoning Problem: $C1 \wedge C2 \wedge C3 \wedge C4$



Basics in Software Testing - Symbolic Execution

Basic Idea: collect and solve path conditions to generate test inputs

```
1. int a = α, b = β, c = γ;
2.                  // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   z = 2;
10.}
11.assert(x+y+z!=3)
```

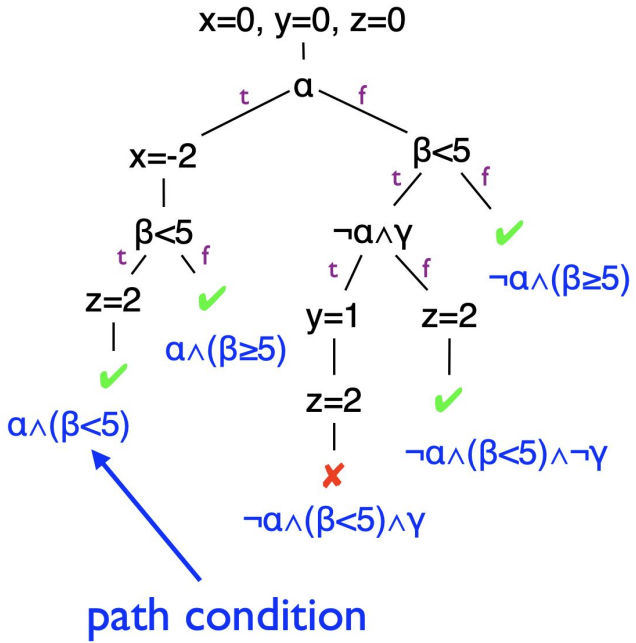


Figure collected from https://www.cs.umd.edu/~mwh/se-tutorial/symbolic-exec.pdf

History of Symbolic Execution Research

- **Theory has been proposed since 1970s**
- **Limited applications in the last century due to**
  - **Limited performance of computer systems in the last century**
  - **Scalability Concern**
    - Path Exploration Challenge
    - Constraint Solving Challenge

History of Symbolic Execution Research

- **Much broader applications today**
  - **Modern computers have much better performance**
  - **Research breakthrough on addressing those challenges**
    - Concolic execution, hybrid fuzzing/analysis, machine learning assisted symbolic execution, …
    - Highly optimized automated reasoning tools (e.g., Z3 and CVC5) and symbolic execution tools (e.g., KLEE)
    - …

Symbolic Execution Tool: KLEE

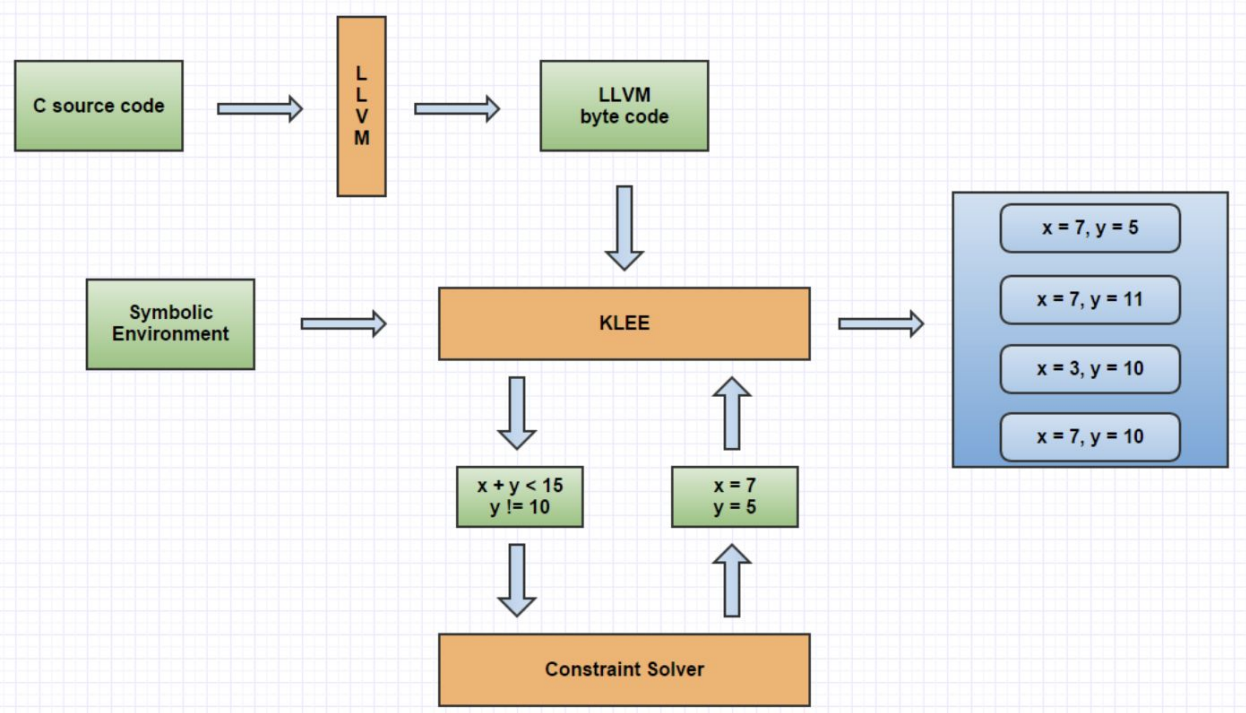
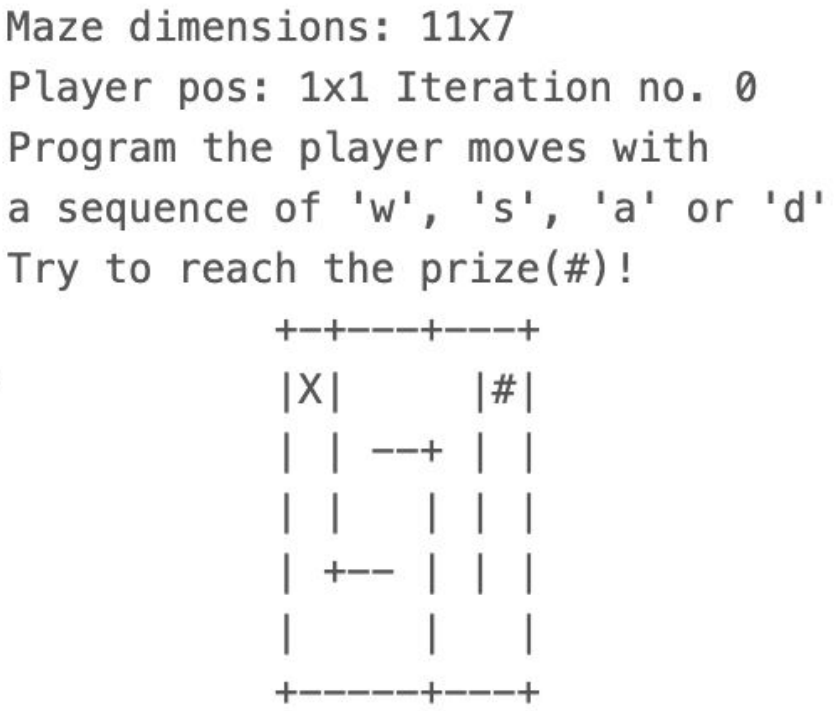


Figure collected from https://events.static.linuxfound.org/sites/events/files/slides/xendev15_chunjie.pdf

Symbolic Execution Tool: KLEE



Figures collected from https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

Symbolic Execution Tool: KLEE

```
#define H 7
#define W 11
char maze[H][W] = { "+-+---+---+",
                    "| |     |#|",
                    "| | --+ | |",
                    "| |   | | |",
                    "| +-- | | |",
                    "|     |   |",
                    "+-----+---+" };
```

Figures collected from https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

```
int
main (int argc, char *argv[])
{
    int x, y;     //Player position
    int ox, oy;   //Old player position
    int i = 0;    //Iteration number
    #define ITERS 28
    char program[ITERS];
```

The initial player position is set to (1,1), the first free cell in the map. And the player 'sprite' is the letter 'X' ...

```
    x = 1;
    y = 1;
    maze[y][x]='X';
```

At this point we are ready to start! So it asks for directions. It reads all actions at once as an array of chars. It will execute up to ITERS iterations or commands.

```
    read(0,program,ITERS);
```

Figures collected from https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

```
while(i < ITERS)
  {
    ox = x;    //Save old player position
    oy = y;
```

Different actions change the position of the player in the different axis and directions. As "usual"; a is Left, d is Right, w is Up and s is Down.

```
    switch (program[i])
    {
        case 'w':
                    y--;
            break;
        case 's':
                    y++;
            break;
        case 'a':
                    x--;
            break;
        case 'd':
                    x++;
            break;
        default:
                    printf("Wrong command!(only w,s,a,d accepted!)\n");
                    printf("You lose!\n");
                    exit(-1);
    }
```

Figures collected from https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

```
if (maze[y][x] == '#')
{
        printf ("You win!\n");
        printf ("Your solution \n",program);
        exit (1);
}
```

If something is wrong do not advance, backtrack to the saved state!

```
if (maze[y][x] != ' ' &&
    !((y == 2 && maze[y][x] == '|' && x > 0 && x < W)))
            {
                    x = ox;
                    y = oy;
            }
```

If crashed to a wall or if you couldn't move! Exit, You lose!

```
if (ox==x && oy==y){
        printf("You lose\n");
        exit(-2);
}
```

Figures collected from https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

```
#include <klee/klee.h>
```

```
    klee_make_symbolic(program,ITERS,"program");
```

```
llvm-gcc -c -Ipath/to/klee -emit-llvm maze_klee.c -o
maze_klee.bc
klee maze.bc
```

Figures collected from https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

Basics in Software Testing

- **Test Oracle**
- **Coverage Metric**
- **Testing Approaches**
  - Reasoning based approaches (e.g., symbolic execution)

Basics in Software Testing

- **Test Oracle**
- **Coverage Metric**
- **Testing Approaches**
  - Reasoning based approaches (e.g., symbolic execution)
  - Mutation based approaches (e.g., fuzzing)
    - Seed (i.e., test input for mutation purposes)
    - Seed Pool
    - Seed Scheduling

Symbolic Execution vs. Fuzzing

- **Symbolic Execution**
  - **Coverage guarantee (if program modeling is complete)**
  - **Slow test generation**
  - **…**
- **Fuzzing**
  - **Fast test generation**
  - **No coverage guarantee (only limited guidance)**
  - **…**

Symbolic Execution vs. Fuzzing

- **Symbolic Execution**
  - **Coverage guarantee (if program modeling is complete)**
  - **Slow test generation**
  - **…**
- **Fuzzing**
  - **Fast test generation**
  - **No coverage guarantee (only limited guidance)**
  - **…**

Figure collected from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9152719

Symbolic Execution for Detecting Access Control Vulnerabilities in the Linux Kernel

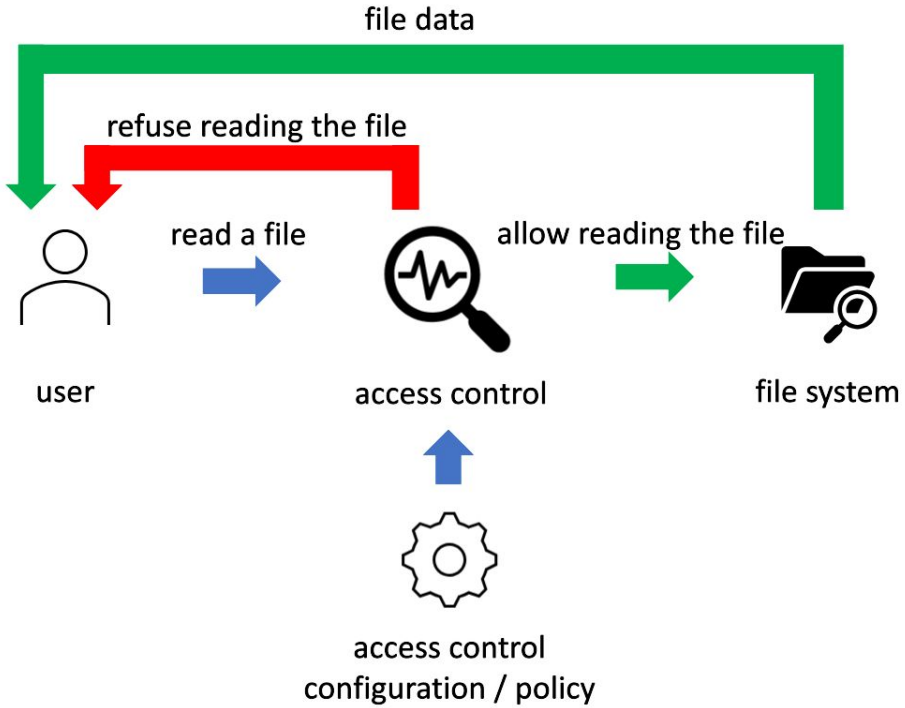
based on our paper published in ESEC/FSE'21:
ACHyb: a hybrid analysis approach to detect kernel access control vulnerabilities

Background: Testing Linux Kernel

- **A test input is a sequence of system calls**
- **Common Test Oracles for Linux Kernel:**
  - crashes
  - sanitizers
  - …
- **Common Coverage Metrics**
  - basic blocks, branches, (control flow) paths, values of shared variables (for fuzzing drivers), etc

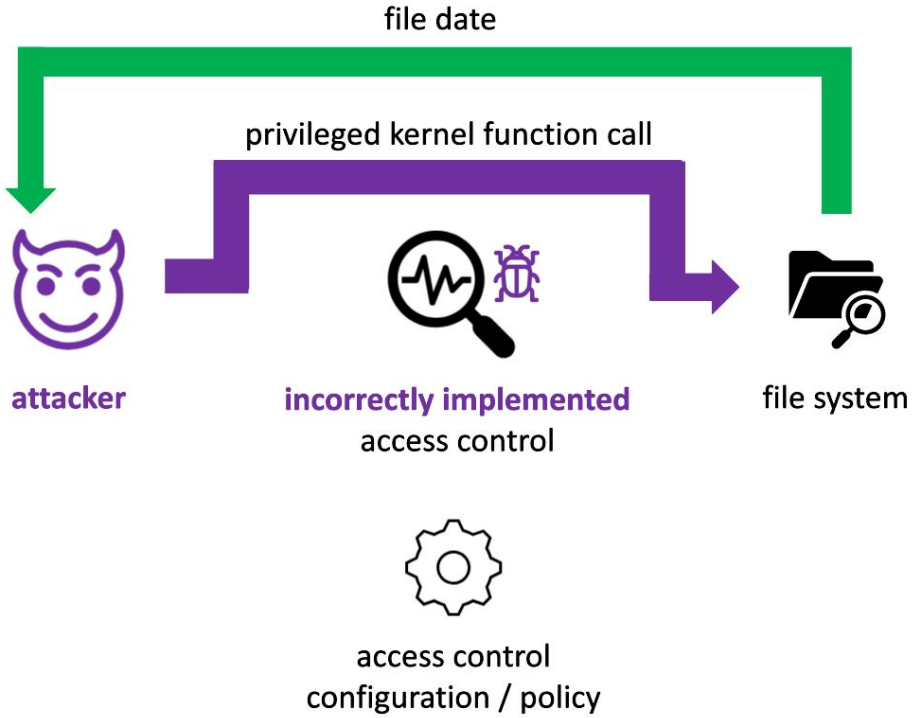
Background: Access Control in Linux Kernel

- Access control (e.g., DAC, SELinux, and Linux Capabilities) is a fundamental security mechanism in Linux Kernel



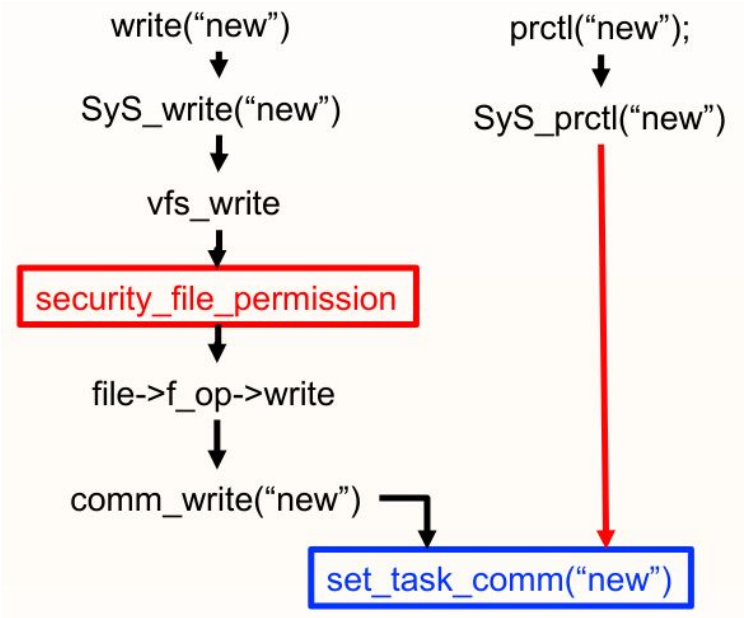
Background: Access Control in Linux Kernel

- *Access Control Vulnerabilities* can be exploited to bypass access control



Background: Access Control in Linux Kernel

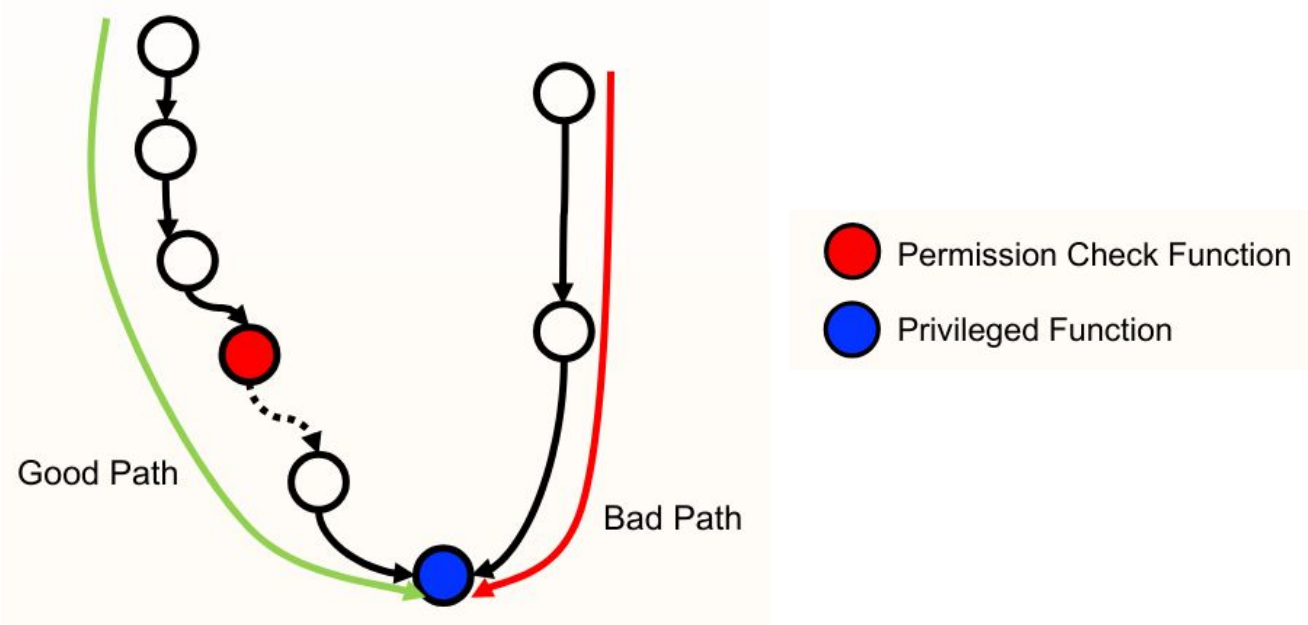
- Access Control Vulnerabilities can be exploited to bypass access control
  - For example, *Missing Permission Check* vulnerability



* figure collected from https://www.usenix.org/sites/default/files/conference/protected-files/sec19_slides_zhang-tong.pdf

Background: Access Control in Linux Kernel

- **Existing detection technique**: static program analysis (e.g,, **[PeX, USENIX SEC'19]**)



* figure collected from https://www.usenix.org/sites/default/files/conference/protected-files/sec19_slides_zhang-tong.pdf

Background: Access Control in Linux Kernel

- **Existing detection technique**: static program analysis (e.g,, **[PeX, USENIX SEC'19]**)
  - *significant false positives*

```
int vfs_dedupe_file_range (...) {
        ...
        bool is_admin = capable(CAP_SYS_ADMIN); // a callsite to a permission check
        ...
        for (...) {
                ...
                if (...) {
                        ...
                } else if (!( is_admin ||...) ) { // incorrect condition
                 ...
                } else {
                        // a callsite to a privileged function
                        deduped = dst_file->f_op->dedupe_file_range(...);
                        ...
                }
        }
}
```

Our Work: Fuzzing Access Control in Linux Kernel

- **Our Solution:** *no false positives*

```
int vfs_dedupe_file_range (...) {
        ...
        bool is_admin = capable(CAP_SYS_ADMIN); // a callsite to a permission check
        ...
        for (...) {
                ...
                if (...) {
                        ...
                } else if (!( is_admin ||...) ) { // incorrect condition
                 ...
                } else {
                        // a callsite to a privileged function
                        deduped = dst_file->f_op->dedupe_file_range(...);
                        ...
                }
        }
}
```

Our Work: Fuzzing Access Control in Linux Kernel

- **Test Oracle:** *invariant check*

```
int vfs_dedupe_file_range (...) {
        ...
        bool is_admin = capable(CAP_SYS_ADMIN); // a callsite to a permission check
        ...
        for (...) {
                ...
                if (...) {
                        ...
                } else if (!( is_admin ||...) ) { // incorrect condition
                 ...
                } else {
                        if(!is_admin) report_acv(); // invariant check
                        // a callsite to a privileged function
                        deduped = dst_file->f_op->dedupe_file_range(...);
                        ...
                }
        }
}
```

Our Work: Fuzzing Access Control in Linux Kernel

- **Straightforward Testing Approach:** Symbolic Execution
  - **Path Constraints:** c1 ∧ c2 ∧ c3 ∧ c4

```
int vfs_dedupe_file_range (...) {
        ...
        bool is_admin = capable(CAP_SYS_ADMIN); // a callsite to a permission check
        ...
        for (...) { // condition c1
                ...
                if (...) { // condition c2
                        ...
                } else if (!( is_admin ||...) ) { // incorrect condition c3
                 ...
                } else {
                        if(!is_admin) report_acv(); // invariant check with condition c4
                        // a callsite to a privileged function
                        deduped = dst_file->f_op->dedupe_file_range(...);
                        ...
                }
        }
}
```

Our Work: Fuzzing Access Control in Linux Kernel

- **Straightforward Testing Approach:** Symbolic Execution
  - **Path Constraints:** c1 ∧ c2 ∧ c3 ∧ c4

Challenge: State Explosion (parameters, pointers, global variables, …)

```
int vfs_dedupe_file_range (...) {
        ...
        bool is_admin = capable(CAP_SYS_ADMIN); // a callsite to a permission check
                ...
                if (...) { // condition c2
                        ...
                } else if (!( is_admin ||...) ) { // incorrect condition c3
                 ...
                } else {
                        if(!is_admin) report_acv(); // invariant check with condition c4
                        // a callsite to a privileged function
                        deduped = dst_file->f_op->dedupe_file_range(...);
                        ...
                }
        }
}
```

Our Work: Fuzzing Access Control in Linux Kernel

- **Our Approach:** *Hybrid Fuzzing* (i.e., Symbolic Execution + Fuzzing)
  - **Step 1. Intra-Procedural Symbolic Execution to Identify Potentially Vulnerable Paths**
  - **Relaxed Path Constraints:** `c1` ∧ `c2` ∧ `c3` ∧ `c4` **(with variables only in the function body)**

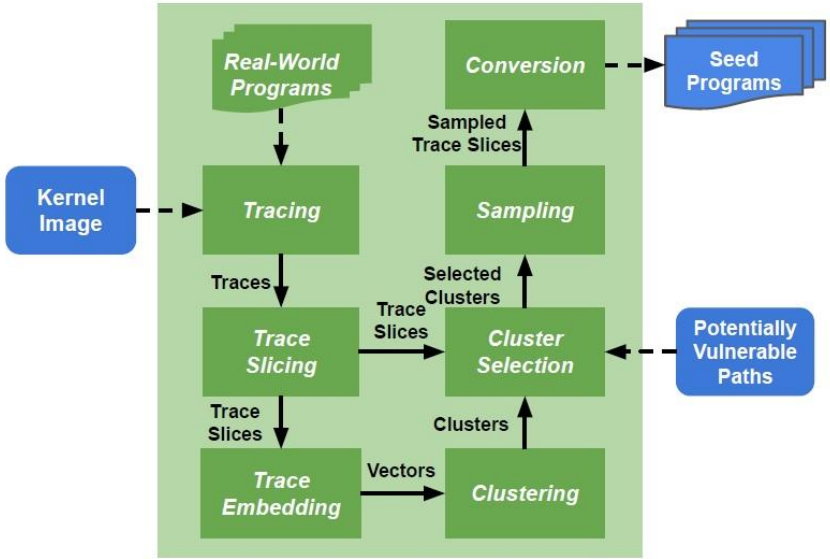
```
int vfs_dedupe_file_range (...) {
        ...
        bool is_admin = capable(CAP_SYS_ADMIN); // a callsite to a permission check
        ...
        for (...) { // condition c1
                ...
                if (...) { // condition c2
                        ...
                } else if (!( is_admin ||...) ) { // incorrect condition c3
                 ...
                } else {
                        if(!is_admin) report_acv(); // invariant check with condition c4
                        // a callsite to a privileged function
                        deduped = dst_file->f_op->dedupe_file_range(...);
                        ...
                }
        }
}
```

Our Work: Fuzzing Access Control in Linux Kernel

- **Our Approach:** *Hybrid Fuzzing/Analysis* (i.e., Symbolic Execution + Fuzzing)
  - **Step 2. Verify potentially vulnerable paths by greybox fuzzing**

Our Work: Fuzzing Access Control in Linux Kernel

- **Our Approach:** *Hybrid Fuzzing/Analysis* (i.e., Symbolic Execution + Fuzzing)
  - **Step 2. Verify potentially vulnerable paths by greybox fuzzing**
    - Seed Distillation with Unsupervised Learning



Our Work: Fuzzing Access Control in Linux Kernel

- **Evaluation**
  - **Baseline: PeX**
    - 2088 potential ACVs in more than 11 hours
    - 14 of them are true ACVs
  - **Our Approach: ACHYB**
    - 22 ACVs in less than 8 hours
- **Tool**
  - Functional and Reusable Badges from ESEC/FSE'21 Artifact Evaluation Track 
  - Open Source Project: https://github.com/githubhuyang/achyb

Thanks!