Prevention: Program Analysis

- Any *automated* analysis *at compile or dynamic time* to find potential *bugs*

- Broadly classified into
  - Dynamic analysis
  - Static analysis
Dynamic Analysis

- Analyze the code when it is running
  - Detection
    - E.g., dynamically detect whether there is an out-of-bound memory access, for a particular input
  - Response
    - E.g., stop the program when an out-of-bound memory access is detected
Dynamic Analysis Limits

- **Major advantage**
  - After detecting a bug, it is a real one
  - No false positives

- **Major limitation**
  - Detecting a bug for a particular input
  - Cannot find bugs for uncovered inputs
Can we build a technique that identifies **all bugs**?

- Turns out that we can: static analysis
Static Analysis

- Analyze the code before it is run (during compile time)
- Explore all possible executions of a program
  - All possible inputs
- Approximate all possible states
  - Build abstractions to “run in the aggregate”
  - Rather than executing on concrete states
  - Finite-sized abstractions representing a collection of states
- But, it has its own major limitation due to approximation
  - Can identify many false positives (not actual bugs)
Static Analysis

- Broad range of static-analysis techniques:
  - Simple syntactic checks like `grep`
    ```
    grep " gets(" *.cpp
    ```
  - More advanced greps: ITS4, FlawFinder
    - A database of security-sensitive functions
      - Gets, strcpy, strcat, ...
      - For each one, suggest how to fix
Static Analysis

- More advanced analyses take into account **semantics**
  - dataflow analysis, abstract interpretation, **symbolic execution**, constraint solving, model checking, theorem proving
- Commercial tools: Coverity, Fortify, Secure Software, GrammaTech
Tool Demo: SWAMP

- **Software Assurance Market (SWAMP)**
  - [https://continuousassurance.org/](https://continuousassurance.org/)
  - Provides free access to some static analysis tools, including some commercial ones

- On homework 3 code
Agenda

- Math/logic preliminaries
- Symbolic Execution
Math Preliminaries
Propositional Logic

- True, False
- $p_1, p_2, ...$: for atomic sentences
  - $p_1 = x > 3$
  - $p_2 = x < 10$
- $p_1 \land p_2$
  - e.g., $x > 3 \land x < 10$
- $p_1 \lor p_2$
  - E.g., $x > 3 \lor x < 10$
- $\neg p_1$
  - $\neg (x > 3)$
- $p_1 \rightarrow p_2$
  - $(x > 3) \rightarrow (x > -10)$
  - $p_1 \rightarrow p_2 = \neg p_1 \lor p_2$
  - $p \rightarrow \text{True}$
  - False $\rightarrow \text{P}$
  - $(p_1 \rightarrow p_2) \land p_1 \rightarrow p_2$ vs. $(p_1 \rightarrow p_2) \rightarrow p_1 \rightarrow p_2$
- $p_1 \leftrightarrow p_2$
  - Same as $(p_1 \rightarrow p_2) \land (p_2 \rightarrow p_1)$
Predicate Logic: Universal and Existential Quantification

- $\forall x. P(x)$
  - e.g. $\forall x. x < 10 \rightarrow x < 3$

- $\exists x. P(x)$
  - e.g. $\exists x. x > 10$
  - e.g. $\exists y. 4 = y \times y$

Examples
- $\forall x. \exists y. y > x.$
- For all square numbers, they are greater than or equal to zero
  - $\forall x. (\exists y. x = y \times y) \rightarrow x \geq 0$
Symbolic Execution

* Some slides adapted from the lectures by Richard Kemmerer at UCSB
Symbolic Execution (SE)

- AKA symbolic evaluation
- Treat program input symbolically and evaluate programs
- A special kind of static analysis (or abstract interpretation)
- Closely related to Hoare Logic
  - But SE goes forward and can also be formulated as a dynamic analysis
Program Syntax

\[
S ::= X := E \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } B \text{ then } S \text{ else } S \\
\quad \mid \text{while } B \text{ do begin } S \text{ end} \\
\quad \mid \text{assume } B \mid \text{assert } B
\]

- Use X, Y, Z etc. for variables
- E is an arithmetic expression
  - An expression that generates a numeric value
  - E.g., X+Y*Z
- B is a boolean expression
  - An expression that generates a boolean value
  - E.g., X>Y+Z
An Example

1 assume (N >= 0);
2 X := 0;
3 Y := 1;
4 while X < N do begin
5     X := X + 1;
6     Y := Y * X
7 end;
8 assert (Y = N!);
Concrete Execution

- Inputs are concrete values
  - For the previous example, e.g., N=3
- All the states as a result are concrete states
  - E.g., when N=3, and after line 3, we have the state \{X=0, Y=1, N=3\}
- Execution of a program statement
  - Go from an input concrete state to an output concrete state
  - E.g., “X=X+1” goes from state \{X=0, Y=1, N=3\} to \{X=1, Y=1, N=3\}
Symbolic Execution

- Inputs are represented symbolically
  - $\alpha_1, \alpha_2, \alpha_3, \ldots$
- Variables get symbolic values
- A symbolic value is
  - Either a constant (e.g., an integer constant),
  - Or $\alpha_i$,
  - Or an expression formed from $\alpha_i$ and constants
    - E.g., $\alpha_1 + \alpha_2, 3\alpha_3$
Symbolic States

- A concrete state holds concrete values for variables
- In contrast, a symbolic state consists of
  - A variable state (VS)
    - A mapping from variables to symbolic values
    - E.g., $\sigma = \{X: \alpha_1 + \alpha_2, Y: \alpha_1 - \alpha_2\}$
  - A path condition (PC)
    - A boolean condition that must hold when the program’s control reaches this point
    - Record the condition when a particular control-flow path is taken
    - E.g., $$(\alpha_1 + \alpha_2 = 0) \land (\alpha_1 > 0)$$
Symbolic Values for Program Expressions

- Suppose $\sigma$ is a variable state
- $\sigma(E)$ stands for the symbolic value for expression $E$
- For instance,
  - Suppose $\sigma = \{X: \alpha_1 + \alpha_2, Y: \alpha_1 - \alpha_2\}$
  - Then $\sigma(X+Y) = 2\alpha_1$
  - Then $\sigma(X-Y) = 2\alpha_2$
Notation

- For a statement $S$
  - $VS_o$ denotes the old variable state when execution reaches the entry of $S$
  - $VS_n$ denotes the new variable state when execution reaches the exit of $S$
  - $PC_o$ denotes the old path condition when execution reaches the entry of $S$
  - $PC_n$ denotes the new path condition when execution reaches the exit of $S$

- There is one symbolic execution rule for each kind of statements

- The initial symbolic state
  - Every input variable assigned a distinct symbolic variable
  - The path condition is the proposition True
Symbolic Evaluation Rule for “X := E”

- Compute the exit symbolic state from the entry symbolic state as follows
  - Get the symbolic value of E in the entry symbolic state; that is, VS_o(E)
  - The result becomes the new value of X in VS_n
  - Path condition is unchanged

- More formally
  - VS_n = VS_o [X ↦ VS_o(E)]
  - PC_n = PC_o

- The computation goes forward
A Simple Example

// input variables: A,B,X,Y,Z
{A:α₁, B:α₂, X:α₃, Y:α₄, Z:α₅}, True

X := A + B;
{A:α₁, B:α₂, X:α₁+α₂, Y:α₄, Z:α₅}, True

Y := A - B;
{A:α₁, B:α₂, X:α₁+α₂, Y:α₁-α₂, Z:α₅}, True

Z := X + Y
{A:α₁, B:α₂, X:α₁+α₂, Y:α₁-α₂, Z:(α₁+α₂)+(α₁-α₂)}, True
{A:α₁, B:α₂, X:α₁+α₂, Y:α₁-α₂, Z: 2α₁}, True
Rule for “assume B”

- Variable state unchanged
  - $VS_n = VS_o$
- Path condition adds the assumption
  - $PC_n = PC_o \land VS_o(B)$
Rule for “assert B”

- If $PC_o$ implies $VS_o(B)$
  - $VS_n = VS_o$
  - $PC_n = PC_o$
- If $PC_o$ does not imply $VS_o(B)$
  - print “assertion failed”
  - Terminate the evaluation
Example

\{A:α_1, B:α_2, X:α_3, Y:α_4, Z:α_5\}, True

assume (A>B);

\{A:α_1, B:α_2, X:α_3, Y:α_4, Z:α_5\}, α_1>α_2

X := A + B;

\{A:α_1, B:α_2, X:α_1+α_2, Y:α_4, Z:α_5\}, α_1>α_2

Y := A - B;

\{A:α_1, B:α_2, X:α_1+α_2, Y:α_1-α_2, Z:α_5\}, α_1>α_2

Z := X + Y

\{A:α_1, B:α_2, X:α_1+α_2, Y:α_1-α_2, Z:(α_1+α_2)+(α_1-α_2)\}, α_1>α_2

assert (X=A+B ∧ Y=A-B ∧ Z=2*A ∧ Y>0);
Verification Condition for the Preceding Example

\[ \alpha_1 > \alpha_2 \rightarrow (\alpha_1 + \alpha_2 = \alpha_1 + \alpha_2 \land \alpha_1 - \alpha_2 = \alpha_1 - \alpha_2 \land \alpha_1 + \alpha_2 + \alpha_1 - \alpha_2 = 2\alpha_1 \land \alpha_1 - \alpha_2 > 0) \]

- How do we check if this holds?
Digression: Theorem Provers

- In general, a theorem prover
  - Takes a logical formula
  - Decides whether the formula is satisfiable or not
  - If the formula is satisfiable, the prover can give a satisfying solution (counter-example)

- SMT (Satisfiability modulo theories) Provers
  - E.g., Z3 by Microsoft Research
Digression: Z3 Demo

; Variable declarations
(declare-fun a () Int)
(declare-fun b () Int)

; if the negation of P is unsatisfiable, then P is always true
(assert (not (=> (> a b)
    (and (= (+ a b) (+ a b))
        (= (- a b) (- a b))
        (= (+ (+ a b) (- a b)) (* 2 a))
        (> (- a b) 0)))))

; Solve
(check-sat)
(get-model)
Rule for “if B then S1 else S2”

- If \( PC_o \rightarrow VS_o(B) \) then execute S1
  - \( PC_n = PC_o \land VS_o(B) \)
  - \( VS_n = VS_o \)
- If \( PC_o \rightarrow \neg VS_o(B) \) then execute S2
  - \( PC_n = PC_o \land \neg VS_o(B) \)
  - \( VS_n = VS_o \)
- If neither \( PC_o \rightarrow VS_o(B) \) nor \( PC_o \rightarrow \neg VS_o(B) \) holds, then two cases to be considered
  - Case 1: \( VS_o(B) \) is true
    - \( PC_n = PC_o \land VS_o(B) \)
    - \( VS_n = VS_o \)
    - Execute S1
  - Case 2: \( VS_o(B) \) is false
    - \( PC_n = PC_o \land \neg VS_o(B) \)
    - \( VS_n = VS_o \)
    - Execute S2
An Example

//input variables are X and Y
1: assume (TRUE);
2: if X< 0
3: then Y := -X;
4: else Y := X;
5: assert (Y>=0)
Branching Behavior

- Can use a tree structure to represent symbolic execution
- Each node represents a statement in the program
- Each branch point corresponds to a forking IF
\( \{X: \alpha_1, Y: \alpha_2\}, \text{True} \)

\( \{X: \alpha_1, Y: \alpha_2\}, \alpha_1 < 0 \)

\( \{X: \alpha_1, Y: -\alpha_1\}, \alpha_1 < 0 \)

\( \{X: \alpha_1, Y: \alpha_2\}, \alpha_1 \geq 0 \)

\( \{X: \alpha_1, Y: \alpha_1\}, \alpha_1 \geq 0 \)

\( \text{VC: } \alpha_1 < 0 \rightarrow -\alpha_1 \geq 0 \)

\( \text{VC: } \alpha_1 \geq 0 \rightarrow \alpha_1 \geq 0 \)
Rule for “while B do S”

- If $PC_o \rightarrow VS_o(B)$ then execute $S$ followed by “while B do S”
- If $PC_o \rightarrow \neg VS_o(B)$ then execute the statement following the While statement
- If neither $PC_o \rightarrow VS_o(B)$ nor $PC_o \rightarrow \neg VS_o(B)$, then two cases to be considered
  - Case 1: $VS_o(B)$ is true
    - $PC_n = PC_o \land VS_o(B)$
    - $VS_n = VS_o$
    - execute $S$ followed by “while B do S”
  - Case 2: $VS_o(B)$ is false
    - $PC_n = PC_o \land \neg VS_o(B)$
    - $VS_n = VS_o$
    - execute the statement following the WHILE statement
1 assume (N >= 0);
2 X := 0;
3 Y := 1;
4 while X < N do begin
5     X := X + 1;
6     Y := Y * X
7 end;
8 assert (Y = N!);
\{N: \alpha_1, X: \alpha_2, Y: \alpha_3\}, \text{True}

\{N: \alpha_1, X: \alpha_2, Y: \alpha_3\}, \alpha_1 \geq 0

\{N: \alpha_1, X: 0, Y: \alpha_3\}, \alpha_1 \geq 0

\{N: \alpha_1, X: 0, Y: 1\}, \alpha_1 \geq 0

\{N: \alpha_1, X: 0, Y: 1\}, \alpha_1 = 0

\{N: \alpha_1, X: 0, Y: 1\}, \alpha_1 > 0

\text{VC: } \alpha_1 = 0 \rightarrow 1 = \alpha_1 !

\{N: \alpha_1, X: 1, Y: 1\}, \alpha_1 > 0

\{N: \alpha_1, X: 1, Y: 1\}, \alpha_1 > 0

\{N: \alpha_1, X: 1, Y: 1\}, \alpha_1 = 1

\{N: \alpha_1, X: 1, Y: 1\}, \alpha_1 > 1

\text{VC: } \alpha_1 = 1 \rightarrow 1 = \alpha_1 !

\vdots
How to Deal with Infinite Execution Tree?

- Approach 1: add loop invariants as annotations
  - Change “while B do S” to “Inv I while B do S”
    - Means that I is a loop invariant for the while loop
  - Pro: an efficient verification tool
  - Cons: need users to add loop invariants

- Approach 2: Search all paths with some kind of bound
  - AKA dynamic symbolic execution
  - Not a verification tool, but a bug finding tool
  - The approach adopted by tools such as EXE and KLEE
How Can This be Used for Security?

- The following example code
  - Assume a is an array of length 100
  - Assume J is some user input

1. \( X := 0; \)
2. if \( J \geq 0 \) then
3.       while \( J < 50 \) do begin
4.           \( X := X + a[2*J]; \)
5.           \( J := J + 1; \)
6.       end;
7. else skip

How do we know \( a[2*J] \) is memory safe?
Insert Security Assertions and Perform Symbolic Execution

1  X := 0;
2  if J>=0 then
3       while J<50 do begin
4           assert (0 <= 2*J < 100);
5           X := X + a[2*J];
6           J := J + 1;
7       end;
8  else skip

After SE, we know a[2*J] is memory safe (w.r.t. to some search bound)
What about this code?

1. $X := 0;$
2. if $J \geq 0$ then
3. while $J \leq 50$ do begin
4. assert ($0 \leq 2*J < 100$);
5. $X := X + a[2*J]$;
6. $J := J + 1$;
7. end;
8. else skip
The previous example

- Need to check the following formula
  \[ \alpha_1 \geq 0 \land \alpha_1 \leq 50 \rightarrow 0 \leq \alpha_1 < 50 \]
- Clearly doesn’t hold
- The SMT solver gives a counter example
  - \( \alpha_1 = 50 \)
  - This is an input that makes the program to perform illegal memory access
- This is the idea behind the paper “EXE: Automatically Generate Input of Death”
Limitations of Classic Symbolic Execution

- **Loops and recursions**: Requiring annotation of loop invariants or infinite execution tree
- **Path explosion**: exponentially many paths due to branches and loops
- **Coverage Problem**: may not reach deep into the execution tree, specially when encountering loops.
- **SMT solver limitations**: dealing with complex path constraints
- **Heap modeling**: symbolic data structures and pointers
- **Environment modeling**: dealing with native/system/library calls/file operations/network events
White-Box Fuzzing (Combining Testing and Symbolic Execution)

Some slides borrowed from Suman Jana
(with contributions from Baishakhi Ray, Omar Chowdhury, Saswat Anand, Rupak Majumdar, Koushik Sen)
Recall: Fuzz Testing

- Black-box fuzzing
  - Treating the system as a blackbox during fuzzing; not knowing details of the implementation

- Grey-box fuzzing
  - Coverage-based fuzzing (e.g., AFL)

- White-box fuzzing
  - Combines fuzzing with test generation
  - Test generation based on static analysis and/or symbolic execution
  - Rather than randomly generating new inputs and hoping that they enable a new path to be executed, compute inputs that will execute a desired path
Solution: Concolic Execution

**Concolic** = **Concrete** + **Symbolic**

Combining Classical Testing with Automatic Program Analysis

Also called **dynamic symbolic execution**

Program is simultaneously executed with concrete and symbolic inputs

Start off the execution with a random input

The intention is to visit deep into the program execution tree

**Concolic execution implementations**: SAGE (Microsoft), CREST
Concolic Execution Steps

- Generate a random seed input to start execution
- Concretely execute the program with the random seed input and record the path taken by that input
- Symbolic execute the path and collect the path constraints along branches
- Negate the last path constraint to get a new path condition
- Solve the new path condition to get a new input
- Example: \( a \&\& b \&\& c \)
  - In the next iteration, negate the last conjunct to obtain the constraint \( a \&\& b \&\& \neg c \)
  - Solve it to get an input which matches all the branch decisions except the last one

Why not from the first?
void testme (int x, int y) {
    z = 2*y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2 * y;
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
void testme(int x, int y) {
    z = 2 * y;
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}

Concolic execution example

Concrete Execution

Symbolic Execution

Concrete state

$x = 22, y = 7, z = 14$

Symbolic state

$x = a, y = b, z = 2b$

Path condition
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2* y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

Solve: 2*b == a
Solution: a = 2, b = 1

x = 22, y = 7, z = 14
x = a, y = b, z = 2*b

2*b != a

Concolic execution example
```c
void testme (int x, int y) {
    z = 2 * y;
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```
Concolic execution example

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

Concrete Execution
- `x = 2, y = 1, z = 2`

Symbolic Execution
- `x = a, y = b, z = 2*b`

Path Condition
Concolic execution example

void testme (int x, int y) {
    z = 2 * y;
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
Concolic execution example

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

Concrete Execution
- Concrete state
  - \( z = 2 \times y \)
  - If \( z = x \) and \( x > y+10 \), ERROR

Symbolic Execution
- Symbolic state
  - \( z = 2 \times b \)
  - \( 2 \times b = a \) and \( a < b + 10 \)

Path Condition
- \( 2 \times b = a \)
- \( a < b + 10 \)
void testme (int x, int y) {
    z = 2 * y;
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
void testme (int x, int y) {
    z = 2* y;

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

Concrete Execution

symbolic state

Concrete state

Program Error

x = 30, y = 15
z = 30

x = a, y = b

2*b == a
a > b+10

Concolic execution example
Further reading

Symbolic execution and program testing - James King

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs - Cadar et. al.

Symbolic Execution for Software Testing: Three Decades Later - Cadar and Sen

DART: Directed Automated Random Testing - Godefroid et. al.

CUTE: A Concolic Unit Testing Engine for C - Sen et. al.