Program Analysis
Learning objectives

• Understand how automated program analysis complements testing and manual inspection
  - Most useful for properties that are difficult to test
• Understand fundamental approaches of a few representative techniques
  - Lockset analysis, pointer analysis, symbolic testing, dynamic model extraction: A sample of contemporary techniques across a broad spectrum
  - Recognize the same basic approaches and design trade-offs in other program analysis techniques
Why Analysis

• Exhaustively check properties that are difficult to test
  - Faults that cause failures
    • rarely
    • under conditions difficult to control
  - Examples
    • race conditions
    • faulty memory accesses

• Extract and summarize information for inspection and test design
Why automated analysis

• Manual program inspection
  - effective in finding faults difficult to detect with testing
  - But humans are not good at
    • repetitive and tedious tasks
    • maintaining large amounts of detail

• Automated analysis
  - replace human inspection for some class of faults
  - support inspection by
    • automating extracting and summarizing information
    • navigating through relevant information
Static vs dynamic analysis

• Static analysis
  - examine program source code
    • examine the complete execution space
    • but may lead to false alarms

• Dynamic analysis
  - examine program execution traces
    • no infeasible path problem
    • but cannot examine the execution space exhaustively
Concurrent faults

- Concurrency faults
  - deadlocks: threads blocked waiting each other on a lock
  - data races: concurrent access to modify shared resources

- Difficult to reveal and reproduce
  - nondeterministic nature does not guarantee repeatability

- Prevention
  - Programming styles
    - eliminate concurrency faults by restricting program constructs
    - examples
      - do not allow more than one thread to write to a shared item
      - provide programming constructs that enable simple static checks
        (e.g., Java synchronized)

- Some constructs are difficult to check statically
  - example
    - C and C++ libraries that implement locks
Memory faults

• Dynamic memory access and allocation faults
  - null pointer dereference
  - illegal access
  - memory leaks

• Common faults
  - buffer overflow in C programs
  - access through *dangling* pointers
  - slow leakage of memory

• Faults difficult to reveal through testing
  - no immediate or certain failure
Example

} else if (c == '%') {
    int digit_high = Hex_Values[*(++eptr)];
    int digit_low  = Hex_Values[*(++eptr)];

• fault
  - input string terminated by an hexadecimal digit
  - scan beyond the end of the input string and corrupt memory
  - failure may occur much after the execution of the faulty statement

• hard to detect
  - memory corruption may occur rarely
  - lead to failure more rarely
Memory Access Failures

(explicit deallocation of memory - C,C++)

- **Dangling pointers**: deallocating memory accessible through pointers
  - no immediate failure
  - may lead to memory exhaustion after long periods of execution
    - escape unit testing
    - show up only in integration, system test, actual use

- **Memory leak**: failing to deallocate memory not accessible any more
  - no immediate failure
  - may lead to memory exhaustion after long periods of execution
  - can be prevented by using
    - program constructs
      - saferC (dialect of C used in avionics applications) limited use of dynamic memory allocation -> eliminates dangling pointers and memory leaks (restriction principle)
    - analysis tools
      - Java dynamic checks for out-of-bounds indexing and null pointer dereferences (sensitivity principle)
    - Automatic storage deallocation (garbage collection)
Symbolic Testing

• Summarize values of variables with few symbolic values
  - example: analysis of pointers misuse
    - Values of pointer variables: null, notnull, invalid, unknown
    - other variables represented by constraints

• Use symbolic execution to evaluate conditional statements

• Do not follow all paths, but
  - explore paths to a limited depth
  - prune exploration by some criterion
Path Sensitive Analysis

- Different symbolic states from paths to the same location
- Partly context sensitive
  (depends on procedure call and return sequences)
- Strength of symbolic testing
  combine path and context sensitivity
  - detailed description of how a particular execution sequence leads to a potential failure
  - very costly
  - reduce costs by memoizing entry and exit conditions
    - limited effect of passed values on execution
    - explore a new path only when the entry condition differs from previous ones
Summarizing Execution Paths

- Find all program faults of a certain kind
  - no prune exploration of certain program paths (symbolic testing)
  - abstract enough to fold the state space down to a size that can be exhaustively explored

- Example:
  analyses based on finite state machines (FSM)
  - data values by states
  - operations by state transitions
Pointer Analysis

- Pointer variable represented by a machine with three states:
  - invalid value
  - possibly null value
  - definitely not null value

- Deallocation triggers transition from non-null to invalid

- Conditional branches may trigger transitions
  - E.g., testing a pointer for non-null triggers a transition from possibly null to definitely non-null

- Potential misuse
  - Deallocation in possibly null state
  - Dereference in possibly null
  - Dereference in invalid states
Merging States

• Flow analysis
  merge states obtained along different execution paths
  - conventional data flow analysis: merge all states encountered at a particular program location
  - FSM: summarize states reachable along all paths with a set of states

• Finite state verification techniques
  never merge states (path sensitive)
  - procedure call and return:
    • complete path- and context-sensitive analysis → too expensive
    • throwing away all context information → too many false alarms
    • symbolic testing: cache and reuse (entry, exit) state pairs
Buffer Overflow

```c
... int main (int argc, char *argv[]) {
    char subject[] = "AndPlus+%26%2B+%0D%";
    char sentinel_post[] = "26262626";
    char *outbuf = (char *) malloc(10);
    int return_code;

    printf("First test, subject into outbuf\n");
    return_code = cgi_decode(subject, outbuf);
    printf("Original: %s\n", subject);
    printf("Decoded: %s\n", outbuf);
    printf("Return code: %d\n", return_code);

    printf("Second test, argv[1] into outbuf\n");
    printf("Argc is %d\n", argc);
    assert(argc == 2);
    return_code = cgi_decode(argv[1], outbuf);
    printf("Original: %s\n", argv[1]);
    printf("Decoded: %s\n", outbuf);
    printf("Return code: %d\n", return_code);
}
...```

Output parameter of fixed length
Can overrun the output buffer
Dynamic Memory Analysis (with Purify)

[I] Starting main
[E] ABR: Array bounds read in printf {1 occurrence}
   Reading 11 bytes from 0x00e74af8 (1 byte at 0x00e74b02 illegal)
   Address 0x00e74af8 is at the beginning of a 10 byte block
   Address 0x00e74af8 points to a malloc'd block in heap 0x00e70000
   Thread ID: 0xd64
...
[E] ABR: Array bounds read in printf {1 occurrence}
   Reading 11 bytes from 0x00e74af8 (1 byte at 0x00e74b02 illegal)
   Address 0x00e74af8 is at the beginning of a 10 byte block
   Address 0x00e74af8 points to a malloc'd block in heap 0x00e70000
   Thread ID: 0xd64
...
[E] ABWL: Late detect array bounds write {1 occurrence}
   Memory corruption detected, 14 bytes at 0x00e74b02
   Address 0x00e74b02 is 1 byte past the end of a 10 byte block at 0x00e74af8
   Address 0x00e74b02 points to a malloc'd block in heap 0x00e70000
   63 memory operations and 3 seconds since last-known good heap state
   Detection location - error occurred before the following function call
   printf {MSVCRT.dll]
...
   Allocation location
   malloc {MSVCRT.dll]
...
[I] Summary of all memory leaks... {482 bytes, 5 blocks}
...
[I] Exiting with code 0 (0x00000000)
   Process time: 50 milliseconds
[I] Program terminated ...
Memory Analysis

• Instrument program to trace memory access
  - record the state of each memory location
  - detect accesses incompatible with the current state
    • attempts to access unallocated memory
    • read from uninitialized memory locations
  - array bounds violations:
    • add memory locations with state *unallocated* before and after each array
    • attempts to access these locations are detected immediately

allocate

Unallocated (unreadable and unwriteable)

dereference

deallocate

Allocated and unitialized (writeable, but unreadable)

dereference

dereference

Allocated and initialized (readable and writeable)

initialize
Data Races

- Testing: not effective (nondeterministic interleaving of threads)
- Static analysis: computationally expensive, and approximated
- Dynamic analysis: can amplify sensitivity of testing to detect potential data races
  - avoid pessimistic inaccuracy of finite state verification
  - Reduce optimistic inaccuracy of testing
Dynamic Lockset Analysis

- Lockset discipline: set of rules to prevent data races
  - Every variable shared between threads must be protected by a mutual exclusion lock
  - …. 

- Dynamic lockset analysis detects violation of the locking discipline
  - Identify set of mutual exclusion locks held by threads when accessing each shared variable
  - INIT: each shared variable is associated with all available locks
  - RUN: thread accesses a shared variable
    - intersect current set of candidate locks with locks held by the thread
  - END: set of locks after executing a test = set of locks always held by threads accessing that variable
    - empty set for v = no lock consistently protects v
**Simple lockset analysis: example**

<table>
<thead>
<tr>
<th>Thread</th>
<th>Program trace</th>
<th>Locks held</th>
<th>Lockset(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread A</td>
<td>lock(lck1)</td>
<td>{lck1}</td>
<td>{lck1, lck2}</td>
</tr>
<tr>
<td></td>
<td>x=x+1</td>
<td>{lck1}</td>
<td>lck1 held</td>
</tr>
<tr>
<td></td>
<td>unlock(lck1)</td>
<td>{}</td>
<td>{lck1}</td>
</tr>
<tr>
<td>tread B</td>
<td>lock{lck2}</td>
<td>{lck2}</td>
<td>lck2 held</td>
</tr>
<tr>
<td></td>
<td>x=x+1</td>
<td>{lck2}</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>unlock(lck2)</td>
<td>{}</td>
<td>Empty intersection potential race</td>
</tr>
</tbody>
</table>

INIT: all locks for x

Intersect with locks held

(c) 2007 Mauro Pezzè & Michal Young

Ch 19, slide 20
Handling Realistic Cases

- simple locking discipline violated by
  - initialization of shared variables without holding a lock
  - writing shared variables during initialization without locks
  - allowing multiple readers in mutual exclusion with single writers
Extracting Models from Execution

- Executions reveals information about a program
- Analysis
  - gather information from execution
  - synthesize models that characterize those executions
Example: AVL tree

```java
private AvlNode insert( Comparable x, AvlNode t ){
    if( t == null )
        t = new AvlNode( x, null, null );
    else if( x.compareTo( t.element ) < 0 ){
        t.left = insert( x, t.left );
        if( height( t.left ) - height( t.right ) == 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
    } else if( x.compareTo( t.element ) > 0 ){
        t.right = insert( x, t.right );
        if( height( t.right ) - height( t.left ) == 2 )
            if( x.compareTo( t.right.element ) > 0 )
                t = rotateWithRightChild( t );
            else
                t = doubleWithRightChild( t );
    } else
        // Duplicate; do nothing
    t.height = max( height( t.left ), height( t.right ) ) + 1;
    return t;
}
```

Behavior model at the end of insert:

- father > left
- father < right
- diffHeight one of {-1,0,1}
Automatically Extracting Models

- Start with a set of predicates
  - generated from templates
  - instantiated on program variables
  - at given execution points
- Refine the set by eliminating predicates violated during execution
## Predicate templates

### over one variable

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$x = a$</td>
</tr>
<tr>
<td>uninitialized</td>
<td>$x = \text{uninit}$</td>
</tr>
<tr>
<td>small value set</td>
<td>$x = {a, b, c}$</td>
</tr>
</tbody>
</table>

### over a single numeric variable

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>in a range</td>
<td>$a \leq x \leq b$</td>
</tr>
<tr>
<td>nonzero</td>
<td>$x \neq 0$</td>
</tr>
<tr>
<td>modulus</td>
<td>$x = a \mod b$</td>
</tr>
<tr>
<td>nonmodulus</td>
<td>$x \neq a \mod b$</td>
</tr>
</tbody>
</table>

### over the sum of two numeric variables

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear relationship</td>
<td>$y = ax + b$</td>
</tr>
<tr>
<td>ordering relationship</td>
<td>$x \leq y, x &lt; y, x = y, x \neq y$</td>
</tr>
</tbody>
</table>
private static void testCaseSingleValues() {
    AvlTree t = new AvlTree();
    t.insert(new Integer(5));
    t.insert(new Integer(2));
    t.insert(new Integer(7));
}

private static void testCaseRandom(int nTestCase) {
    AvlTree t = new AvlTree();

    for (int i = 1; i < nTestCase; i++) {
        int value = (int) Math.round(Math.random() * 100);
        t.insert(new Integer(value));
    }
}
### Derived Models

#### model for **testCaseSingleValues**
- father one of \{2, 5, 7\}
- left == 2
- right == 7
- leftHeight == rightHeight
- rightHeight == diffHeight
- leftHeight == 0
- rightHeight == 0
- fatherHeight one of \{0, 1\}

---

#### model for **testCaseRandom**
- father >= 0
- left >= 0
- father > left
- father < right
- left < right
- fatherHeight >= 0
- leftHeight >= 0
- rightHeight >= 0
- fatherHeight > leftHeight
- fatherHeight > rightHeight
- fatherHeight > diffHeight
- rightHeight >= diffHeight
- diffHeight one of \{-1, 0, 1\}
- leftHeight - rightHeight + diffHeight == 0

---

- limited validity of the test case: the tree is perfectly balanced
- additional information: all elements are non-negative
- the tree is balanced
- useless (redundant) information
Model and Coincidental Conditions

- **Model:**
  - **not** a specification of the program
  - **not** a complete description of the program behavior
  - a representation of the behavior experienced so far

- conditions may be coincidental
  - true only for the portion of state space explored so far
  - estimate probability of coincidence as the number of times the predicate is tested
Example of Coincidental Probability

father \geq 0 \text{ probability of coincidence:}
0.5 \text{ if verified by a single execution}
0.5^n \text{ if verified by n executions.}

threshold of 0.05

two executions with father = 7
father = 7 \text{ valid}
father \geq 0 \text{ not valid (high coincidental probability)}

two additional execution with father positive
father = 7 \text{ invalid}
father \geq 0 \text{ valid}

father \geq 0 \text{ valid for testCaseRandom (300 occurences)}
not for testCaseSingleValues (3 occurences)
Using Behavioral Models

- Testing
  - validate tests thoroughness
- Program analysis
  - understand program behavior
- Regression testing
  - compare versions or configurations
- Testing of component-based software
  - compare components in different contexts
- Debugging
  - Identify anomalous behaviors and understand causes
Summary

• Program analysis complements testing and inspection
  - Addresses problems (e.g., race conditions, memory leaks) for which conventional testing is ineffective
  - Can be tuned to balance exhaustiveness, precision, and cost (e.g., path-sensitive or insensitive)
  - Can check for faults or produce information for other uses (debugging, documentation, testing)

• A few basic strategies
  - Build an abstract representation of program states by monitoring real or simulated (abstract) execution