Automatic Software Verification

Instructor: Mooly Sagiv
TA: Oded Padon

Slides from Eran Yahav and the Noun Project, Wikipedia
Course Requirements

• Summarize one lecture
• 10% one lecture notes
• 45% homework assignment
• 45% exam/project
Inspired by Other Courses

- **CS395T: Automated Logical Reasoning** (UT Austin) Isil Dillig
- **SAT/SMT Solver and Applications**
  Graduate Seminar
  W2013 University of Waterloo (Vijay Ganesh)
Software is Everywhere
Exploitable Software is Everywhere
Exploitable Software is Everywhere

Sony PlayStation

Stuxnet Worm Still Out of Control at Iran's Nuclear Site

Experts Say RSA's combo

Stuxnet in Flash:

Security Advisory for Adobe Flash

Player, Adobe Reader

This vulnerability could

potentially allow

RSA tokens may be

behind major

network security

problems at

April 2011 Secunia Scorecard

control of the
void foo (char *x) {
    char buf[2];
    strcpy(buf, x);
}

int main (int argc, char *argv[]) {
    foo(argv[1]);
}

> ./a.out

Segmentation fault
Buffer Overrun Exploits

```c
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0) auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0) auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(check_authentication(argv[1])) {
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("    Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    } else
        printf("Access Denied.\n");
}
```

(source: “hacking – the art of exploitation, 2nd Ed”)
Attack

evil input → Application

AAAAAAAAAAAAAAAA

Access Granted. 65

AAAAAAAAAAAAAAAA
Automatic Program Verification

Program $P$

Desired Properties $\varphi$

Solver

*Is there a behavior of $P$ that violates $\varphi$?*

Counterexample

Proof
```c
Example

int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0) auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0) auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(check_authentication(argv[1])) {
        printf("\n-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==\n");
        printf("     Access Granted.\n");
        printf("-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==-==\n");
    } else
        printf("\nAccess Denied.\n");
}
```
Undecidability

• The Halting Problem
  – Does the program P terminate on input I

• Rice’s Theorem
  – Any non-trivial property of partial functions, there is no general and effective method to decide if program computes a partial function with that property
Handling Undecidability

- Permits occasional divergence
- Limited programs (not Turing Complete)
- Unsound Verification
  - Explore limited program executions
- Incomplete Verification
  - Explore superset of program executions
- Programmer Assistance
  - Inductive loop invariants
Limited Programs

• Finite state programs
  – Finite state model checking
    • Explicit state SPIN, CHESS
    • Symbolic model checking SMV

• Loop free programs
  – Configuration files
Unsound Verification

• Dynamic checking
  – Valgrind, Parasoft Insure, Purify, Eraser
• Bounded Model Checking
• Concolic Executions
The SAT Problem

• Given a propositional formula (Boolean function)
  \[ \varphi = (a \lor b) \land (\neg a \lor \neg b \lor c) \]
• Determine if \( \varphi \) is valid
• Determine if \( \varphi \) is satisfiable
  – Find a satisfying assignment or report that such does not exist
• For \( n \) variables, there are \( 2^n \) possible truth assignments to be checked
• Effective heuristics exist
Bounded Model Checking

Program P

Input Bound k

Desired Properties $\phi$

FrontEnd

Propositional Formula $\lbrack P(k) \rbrack \land \neg \phi$

SAT Solver

Assignment

UNSAT
A Simple Example

Program

```c
int x;
int y=8,z=0,w=0;
if (x)
    z = y - 1;
else
    w = y + 1;
assert (z == 5 || w == 9)
```

Constraints

```c
y = 8,
z = x ? y - 1 : 0,
w = x ? 0 : y + 1,
z != 5,
w != 9
```

SAT

counterexample found!

```c
y = 8, x = 1, w = 0, z = 7
```
A Simple Example

Program

```c
int x;
int y=8,z=0,w=0;
if (x)
    z = y - 1;
else
    w = y + 1;
assert (z == 7 || w == 9)
```

Constraints

```
y = 8,
z = x ? y - 1 : 0,
w = x ? 0 : y + 1,
z != 7,
w != 9
```

UNSAT
Assertion always holds!
Summary Bounded Model Checking

- Excellent tools exist (CBMC, Alloy)
- Many bugs occur on small inputs
- Useful for designs too
- Scalability is an issue
- Challenging features
  - Bounded arithmetic
  - Pointers and Heap
  - Procedures
  - Concurrency
Concolic Testing

• Combine runtime testing and symbolic execution

• Runtime testing
  – Effectiveness depends on input test

• Symbolic Execution
  read(x);
  y = 2 * x ;
  assert y != 12;
  – Need constraint solver
  – Can be complex

• Concolic testing aims to improve both
A Motivating Example

```c
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```
The Concolic Testing Algorithm

1. Classify input variables into symbolic / concrete
2. Instrument to record symbolic vars and path conditions
3. Choose an arbitrary input
4. Execute the program
5. Symbolically re-execute the program
6. Negate the unexplored last path condition
7. Is there an input satisfying constraint?
Example Concolic Testing

```c
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```
Summary Concolic Testing

• Quite effective:
  – SAGE (Microsoft Research)
  – Datarace detection (Candea, EPFL)
• Instrumentation can be tricky
• Scalability is an issue
• Coverage is an issue
• Limitations of theorem provers
• Data structures
Invariant

• An assertion $I$ is an invariant at program location if $I$ holds whenever the execution reaches this location.

• An invariant is inductive at a loop “while $B$ do $C$” if whenever $C$ is executed on a state which satisfies $B$ and $I$ it can only produce states satisfying $I$. 
rotate(List first, List last) {
  if (first != NULL) {
    last -> next = first;
    first = first -> next;
    last = last -> next;
    last -> next = NULL;
  }
}
Inductive Invariants

\[ x = 2; \]
\[ \text{while true do } \{ x > 0 \} \]
\[ x = 2 \times x - 1 \]

Non-inductive  Inductive

\[ x > 0 \]  \[ x > 1 \]
Deductive Verification

Program P

Goal F

Candidate Invariant inv

VC gen

Inv is inductive w.r.t. P
Inv \Rightarrow F

SAT Solver

Counterexample

Proof
Summary Deductive Verification

• Existing Tools
  – ESCJava, Dafny, CAVEAT

• Hard to write inductive invariants
  – Need to consider all corner cases
  – Small program change can lead to huge change in the invariant
  – The lack of specification languages

• Deduction can be hard
Deduction

\( x = 2; \)

while true do {\( x > 1 \)}

\( x = \frac{2*x*x + x - 1}{x + 1} \)
Transition Systems

• The program semantics can be described as (potentially infinite) graph of reachable states
  – Values of program variables
• Program statements and conditions are relations between states
• Proving a safety property usually means showing that certain state cannot be reached
  – A bad reachable state indicate a bug
• Bounded model checking and concolic testing explore subsets of reachable states
Example Transition System

1: x = 2;
2: while true do
   3: x = 2* x – 1
4: 
Abstract Interpretation

• Automatically prove that the program is correct by also considering infeasible executions
• Abstract interpretation of program statements/conditions
• Conceptually explore a superset of reachable states
• Sound but incomplete reasoning
• Automatically infer sound inductive invariants
Automatic Program Verification

- Program $P$
- Desired Properties $\varphi$

Solver

Is there a behavior of $P$ that violates $\varphi$?

- Counterexample
- Unknown
- Proof
Interval Based Abstract Interpretation

1: \( x = 2; \)
2: while true \{ \( x > 0 \) \} do
  3: \( x = 2 \times x - 1 \)
4: 

\[ \text{pc: int}(x) \]

1: \([0, 0]\)
2: \([2, 2]\)
3: \([2, 2]\)
4: \([3, 3]\)
Interval Based Abstract Interpretation

1: \( x = 2; \)
2: while true \(\{x > 0\}\) do
   3: \( x = 2\times x - 1 \)
4: 

\textbf{pc: int}(x)

1: \([0, 0]\)
2: \([2, \infty]\)
3: \([2, \infty]\)
4: \([3, \infty]\)
Interval Based Abstract Interpretation

1: x = 2, y = 2
2: while true {x = y} do
   3: x = 2 \times x - 1,
      y = 2 \times y - 1
4:

pc: int(x), int(y)

1: [0, 0], [0, 0]
2: [2, 2], [2, 2]
3: [2, 2], [2, 2]
4: [3, 3], [3, 3]
Shape-Based Abstract Interpretation

node search(node h, int v) {
  1: node x = h;
  2: while (h != NULL) {
      3: if (x->d == v) return x;
      4: assert x != null; x = x->n ;
  }
  5: return (node) NULL
node search(node h, int v) {
1: node x = h;
2: while (x != NULL) {
   3: if (x->d == v) return x;
   4: assert x != null; x = x->n;
}
5: return (node) NULL
Odd/Even Abstract Interpretation

1: while (x != 1) do {
    2: if (x % 2) == 0 {
        3: x := x / 2; }
    else
        4: x := x * 3 + 1;
    5: assert (x % 2 == 0); }
6: }

1: ?
2: ?
3: E
4: O
5: E
6: O
Abstract Interpretation

Concrete

Sets of stores

Descriptors of sets of stores

Abstract
Odd/Even Abstract Interpretation

All concrete states

\{x: x \in \text{Even}\} \{-2, 1, 5\}

\{0,2\}

\{0\} \{2\}

\emptyset

E

O

?
Odd/Even Abstract Interpretation

All concrete states

\{-2, 1, 5\}
\{x: x \in Even\}
\{0, 2\}
\{0\} \{2\}
\emptyset

Even
Odd
?
Odd/Even Abstract Interpretation

All concrete states

\{ x: x \in \text{Even} \}

\{-2, 1, 5\}

\{ 0, 2 \}

\{ 0 \} \quad \{ 2 \}

\emptyset
(Best) Abstract Transformer
Odd/Even Abstract Interpretation

1: while (x != 1) do {
   2: if (x % 2) == 0 {
      3: x := x / 2;
   } else {
      4: x := x * 3 + 1;
   }
   5: assert (x % 2 == 0);
} 6: }

1: ?
2: ?
3: E
4: O
5: E
6: O
Summary Abstract Interpretation

• Conceptual method for building static analyzers

• A lot of techniques:
  – join, meet, widening, narrowing, procedures

• Can be combined with theorem provers
## Tentative Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/3</td>
<td>Introduction</td>
</tr>
<tr>
<td>20/3</td>
<td>SAT Solvers for propositional logic</td>
</tr>
<tr>
<td>20/3 10-13, Scriber 309</td>
<td>SAT Solvers for propositional logic</td>
</tr>
<tr>
<td>24/3</td>
<td>Beyond propositional logic: SMT Solvers</td>
</tr>
<tr>
<td>14/4</td>
<td>Symbolic Reasoning with SMT Solvers</td>
</tr>
<tr>
<td>21/4</td>
<td>Concolic Testing</td>
</tr>
<tr>
<td>28/4</td>
<td>Bounded Model Checking</td>
</tr>
<tr>
<td>5/5</td>
<td>SAT based inductive reasoning(Interpolants)</td>
</tr>
<tr>
<td>12/5</td>
<td>Introduction to Abstract Interpretation</td>
</tr>
<tr>
<td>29/5</td>
<td>Pointers and Shape Analysis</td>
</tr>
<tr>
<td>2/6</td>
<td>Applications to Shape Analysis</td>
</tr>
<tr>
<td>9/6</td>
<td>Property Directed Abstract Interpretation</td>
</tr>
</tbody>
</table>