Outline

Overview

Program understanding

Program verification and property checking

Bug finding

How static analysis works

Summary
Recap

We’re looking at

- **principles and tools**

for ensuring software security.

This lecture looks at:

- further **example uses** of static analysis
- an **example of dynamic analysis** for information flow
- some hints about **how static analysis works**
Advanced static analysis jobs

Static analysis is used for a range of tasks that are useful for ensuring secure code.

Basic tasks include **type checking** and **style checking**, described last lecture.

More advanced tasks are:

- **Program understanding**: inferring meaning
- **Property checking**: ensuring no bad behaviour
- **Program verification**: ensuring correct behaviour
- **Bug finding**: detecting likely errors
Outline

Overview

Program understanding

Program verification and property checking

Bug finding

How static analysis works

Summary
Program understanding tools

Help developers understand and manipulate large codebases.

- Navigation swiftly inside the code
  - finding definition of a constant
  - finding call graph for a method
- Support *refactoring* operations
  - re-naming functions or constants
  - move functions from one module to another
  - needs internal model of whole code base
- Inferring *design* from *code*
  - Reverse engineer or check informal design

**Outlook:** may become increasingly used for security review, with dedicated tools. Close relation to tools used for malware analysis (reverse engineering).
Commercial example: Structure101
Research example: Fujaba and Reclipse
How Reclipse works

We’ll explain some of these processes later.

See Fujaba project at University of Paderborn
Outline

Overview

Program understanding

Program verification and property checking

Bug finding

How static analysis works

Summary
Program verification

- The gold standard, ultimate guarantee
- Uses **formal methods** techniques, e.g.,
  - theorem proving
  - model checking
- Drawback: needs precise **formal specification** to verify against
- Very expensive to industry
  - time consuming
  - needs experts (logic/maths)
- Currently only used in safety critical domains
  - e.g., railway, nuclear, aeronautics
  - emerging: automobile, security

Example tools: SPARK, Event-B. See also general purpose **interactive theorem provers** such as Coq and Isabelle/HOL.
Property checking

*Lightweight* formal methods

- Make specifications be *standard* and *generic*
- this program cannot raise NullPointerException
- all database connections are closed after use

**Static checking** (not verification)

- Prevent many violations of specification, not all
- May produce *counterexamples* to explain violations
- Chain pre-conditions (*requires*) and post-conditions (*ensures*)
  - allows *inter-procedural* analysis

Assertion checking

Many languages have support for *assertions*. These are dynamic (runtime) checks that can be used to test properties the programmer expects to be true.

```markdown
assert(exp)
```

- fails if `exp` evaluates to false
- assertion tests *usually disabled*
  - treated as comments
  - may be enabled for testing during development
  - or when running unit tests

**Question.** What could happen if tests are run only with assertions enabled?
private static int addHeights(int ah, int bh) {
    assert ah > 0 && bh > 0 : "parameters should be positive";
    return ah+bh;
}

Notice above method is private.

- API (public) functions should *always* test constraints
  - throw exceptions if not met
  - eliminate clients (or attackers) who break API contract
- Internal functions may rely on local properties
  - if maintained in same class, easier to check/ensure
Assertions for security

We could use assertions as safety checks for functions that are at risk of being used in a buggy way.

```c
assert(alloc_size(dest) > strlen(src));
strcpy(dest, src);
```

`alloc_size()` is not a standard C function, but GCC, for example, has support for trying to track the size of allocated functions with function attributes.
From dynamic to static

With static analysis, we may be able to automatically determine whether assertions (if enabled) will:

1. always succeed
2. may sometimes fail (unknown)
3. will always fail

Easy cases:

1. assert(true);
2. x=readint(); assert(x>0);
3. assert(false);

The perfect case would be showing that assertions in a program can only succeed: thus they do not need to be checked dynamically.

**Question.** what troubles can you see with case 2?
Programming with contracts

This can be useful to increase confidence in programs, or use a *contract* based programming approach, where pre- and post-conditions are explicitly described by the programmer.

Some static analysis tools use assertions (entirely) internally; others allow an interface using *annotations*.
Split (2001): a stronger Lint

Allows *annotations* to be added by programmer, specifically for static analysis tool.

```c
void *strcpy(char *s1, char *s2)
    /*modifies *s1 */
    /*requires maxSet(s1) >= maxRead(s2) */
    /*ensures maxRead(s1) == maxRead (s2) */;
```

- `maxSet(x)`: the greatest offset (index) that may be written to in `x`
- `maxRead(y)`: the greatest that may be *read* from in `y`
strncat

strncat(dest, src, num): appends the first num characters of source to destination, plus a terminating null character. If the length of the C string in src is less than num, only the content up to the terminating null-character is copied.

```
char *strncat (char *s1, char *s2, size_t n)
    /*requires maxSet(s1) >=maxRead(s1) + n*/

void f(char *str){
    char buffer[256];
    strncat(buffer, str, sizeof(buffer) - 1);  
    return;
}
```
Splint warning messages

```c
char *strncat (char *s1, char *s2, size_t n)
    /*requires maxSet(s1) >= maxRead(s1) + n*/

void f(char *str){
    char buffer[256];
    strncat(buffer, str, sizeof(buffer) - 1);
    return;
}
```

strncat.c:4:21: Possible out-of-bounds store:
    strncat(buffer, str, sizeof((buffer)) - 1);
Unable to resolve constraint:
    requires maxRead (buffer strncat.c:4:29) <= 0
needed to satisfy precondition:
    requires maxSet (buffer strncat.c:4:29)
        >= maxRead (buffer strncat.c:4:29) + 255
derived from strncat precondition:
    requires maxSet (<parameter 1>)
        >= maxRead (<parameter1>) + <parameter 3>
Reasoning with assertions

How does a static analyser reason?

Computations about assertions can be chained through the program, using a *program logic* inside the tool.

E.g., build up a set of facts known before each statement:

```plaintext
x = 1;  // { } (nothing known)
y = 1;  // { x = 1 }
assert (x < y);  // { x = 1, y = 1 }
```

```plaintext
assert (x < y);  // FAIL
```
Symbolic evaluation

This can work also with variables, whose value is not known statically:

```plaintext
// { } (nothing known)
x = z;           // { x = z }
y = z+1;          // { x = z, y = z+1 }
assert (x < y);   // SUCCEED (provided z<MAXINT)
```
Conditionals and loops

These make static analysis much harder, of course.

```plaintext
// {} (nothing known)
x = v;
// {x=v}
if (x < y)

//

y = v;
// {x=v, x<y}
assert (x < y)
// Either: {x=v, y=v}: FAIL
// Or: {x=v, ¬(x<y)}: FAIL
```

For conditionals, we need to either

- explore every path
- merge information at join-points

For loops, we need to either

- unroll for a finite number of iterations
- capture variation using logical invariants
Using logical (or other) reasoning techniques, there are various different types of assertions that are useful for security checking, for example:

- **Bounds and range analysis**
- **Tainted data analysis**
- **Type state** and **Resource** tracking

**Exercise.** What kinds of security issues can these assertions help with? What kinds of security issues would need other assertions?
Bound/range Analysis

`alloc_size(dest) > strlen(src)`

`array_size(a) > n` before `a[n]` access

- Check integers are in required ranges
Taint analysis approximates information flow where information may be leaked indirectly as well as directly.

- Tracks whether data can be affected by adversary.
- Tainted input shouldn’t be used for security sensitive choices.
- And should be sanitized before being output.

\[ \text{tainted}(\text{mypageinput}) \]
\[ \text{untainted}(\text{newkey}) \]
Type State (Resource) Tracking

- **nonnull**(ptr), **nonnull**(ptr)
- **isopen_for_read**(handle), **isclosed**(handle)
- **uninitialized**(buffer), **terminatedstring**(buffer)

- Tracks status of data value held by a variable
- Helps enforce API usage contracts to avoid errors
  - e.g., DoS
- Usage/lifecycle may be expressed with automaton
Example: avoiding double-free errors
Null Pointers in CodeSonar

Not all null pointer analyses are equal! Some compilers spot only "obvious" null pointer risks, others perform deeper analysis like CodeSonar. IDE analysis may be in between.
For Java, there is a language called JML which adds similar pre- and post-conditions (requires/ensures). Open source JML toolsets have been through several versions but have had trouble keeping up with Java, Eclipse changes.
Outline

Overview

Program understanding

Program verification and property checking

Bug finding

How static analysis works

Summary
Bug finding

Bug finding tools look for suspicious patterns in code. **FindBugs** is an example:

- Finds possible Java bugs according to *rules*
  - rules are suspicious patterns in code
  - designed by experience of buggy programs
  - ... collected from real world and student(!) code

- Warnings are categorized by
  - **severity**: how serious in general the problem is
  - **confidence**: tool’s belief of true problem
Example bugs

Common accidents

An error found in Sun’s JDK 1.6:

```java
public String foundType() {
    return this.foundType();
}
```

Misunderstood APIs

```java
public String makeUserId(String s) {
    s.toLowerCase();
    return s;
}
```
Anti-idiom: double-checked locking in Java

```java
if (this.fitz == null) {
    synchronized (mylock) {
        if (this.fitz == null) {
            this.fitz = new Fitzer();
        }
    }
}
```

[dice]da: findbugs Fitz.class
M M DC: Possible doublecheck on Fizz.fitz in Fitz.getFitz()
    At Fitz.java:[lines 1-3]
Findbugs GUI

Findbugs: Findbugs GUI

Method may fail to close stream

The method creates an IO stream object, does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. It is generally a good idea to use a finally block to ensure that streams are closed.

http://findbugs.sourceforge.net/
Clang Static Analyser

An open source tool for C, C++, Objective-C included in XCode.
Clang Static Analyser HTML reports

openssl-1.0.0 - scan-build results

<table>
<thead>
<tr>
<th>User:</th>
<th>user@localhost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working Directory:</td>
<td>/home/user/Exercise-4/openssl-1.0.0</td>
</tr>
<tr>
<td>Command Line:</td>
<td>make</td>
</tr>
<tr>
<td>Clang Version:</td>
<td>clang version 3.4 (tags/RELEASE_34/final)</td>
</tr>
<tr>
<td>Date:</td>
<td>Fri Jan 17 12:03:31 2014</td>
</tr>
</tbody>
</table>

Bug Summary

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>Quantity</th>
<th>Display?</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Bugs</td>
<td>269</td>
<td>✔️</td>
</tr>
<tr>
<td>API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Argument with ’nonnull’ attribute passed null</td>
<td>7</td>
<td>✔️</td>
</tr>
<tr>
<td>Dead store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dead assignment</td>
<td>203</td>
<td>✔️</td>
</tr>
<tr>
<td>Dead increment</td>
<td>11</td>
<td>✔️</td>
</tr>
<tr>
<td>Dead initialization</td>
<td>2</td>
<td>✔️</td>
</tr>
<tr>
<td>Logic error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assigned value is garbage or undefined</td>
<td>3</td>
<td>✔️</td>
</tr>
<tr>
<td>Branch condition evaluates to a garbage value</td>
<td>1</td>
<td>✔️</td>
</tr>
<tr>
<td>Dereference of null pointer</td>
<td>30</td>
<td>✔️</td>
</tr>
<tr>
<td>Division by zero</td>
<td>1</td>
<td>✔️</td>
</tr>
<tr>
<td>Result of operation is garbage or undefined</td>
<td>7</td>
<td>✔️</td>
</tr>
<tr>
<td>Uninitialized argument value</td>
<td>4</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Reports

<table>
<thead>
<tr>
<th>Bug Group</th>
<th>Bug Type</th>
<th>File</th>
<th>Line</th>
<th>Path Length</th>
<th>View Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Argument with ’nonnull’ attribute passed null</td>
<td>ssl/d1_both.c</td>
<td>1015</td>
<td>9</td>
<td>View Report</td>
</tr>
<tr>
<td>API</td>
<td>Argument with ’nonnull’ attribute passed null</td>
<td>ssl/d1_srvr.c</td>
<td>1184</td>
<td>10</td>
<td>View Report</td>
</tr>
<tr>
<td>API</td>
<td>Argument with ’nonnull’ attribute passed null</td>
<td>ssl/3_srvr.c</td>
<td>1725</td>
<td>10</td>
<td>View Report</td>
</tr>
<tr>
<td>API</td>
<td>Argument with ’nonnull’ attribute passed null</td>
<td>crypto/asn1/a_bytes.c</td>
<td>295</td>
<td>21</td>
<td>View Report</td>
</tr>
</tbody>
</table>
Outline

Overview

Program understanding

Program verification and property checking

Bug finding

How static analysis works

Summary
Basic overview

Source Code → Build Model → Perform Analysis → Present Results

Security Knowledge
Building a program model

Starts off like a compiler, in stages. Simpler/older static analysis tools only use first stages.

1. **Lexical analysis**: tokenise input
2. **Parsing**: builds a *parse tree* from grammar
3. **Abstract Syntax Tree**: simplify parse tree
4. **Semantic analysis**
   - check program well-formedness
   - including **type-checking**
5. Produce an **Intermediate Representation** (IR)
   - higher level than for compiler
6. Produce **model** to capture control/data flows
   - *control-flow* and *call graphs*
   - variable-contains-data relationships
   - pointer analysis: aliasing, points-to
Control flow graphs

```java
if (a > b) {
    nConsec = 0;
} else {
    s1 = getHexChar(1);
    s2 = getHexChar(2);
}
return nConsec;
```

The CFG consists of *basic blocks* and the paths between them.
A trace is a possible sequence of basic blocks.

Above: [bb0, bb1, bb3] and [bb0, bb2, bb3].

Traces can be used to check against security constraints (e.g., as automata), to construct counterexamples. The CFG is also used to combine/chain assertions.
Call graphs

```c
int a(int x) {
    if (x) { b(1); } else { c(); }
}
int b(int y) {
    if (y) { c(); b(0); } else { c(); }
}
int c() { /* empty */ }
```

- Call graphs are used for *inter-procedural* analysis
- Check requires-ensures contracts connect together
Putting them together: local and global
Outline

Overview

Program understanding

Program verification and property checking

Bug finding

How static analysis works

Summary
Take away points

Program analysis tools can help find security flaws.

- static: examine millions of lines, repeatedly
- dynamic: equip code with *self-checking*

Some tools are generic bug finding, built into IDE.
Others are specific to security, may include:

- risk analysis, including impact/likelihood
- issue/requirements tracking, metrics

Expect these to become mainstream

- current frequency of security errors unacceptable
- incentives will eventually affect priorities
References and credits

Some of this lecture is based Chapters 2-4 of

- *Secure Programming With Static Analysis* by Brian Chess and Jacob West, Addison-Wesley 2007.

Recommended reading: