Recap

We’re looking at
▶ principles and tools
for ensuring software security.

This lecture looks at:
▶ further example uses of static analysis
▶ some details of 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

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

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
- 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
- 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

Examples: SPARK, Event-B. Also general purpose interactive theorem provers (HOL, ACL2, Isabelle, Coq). Many research-quality or legacy tools.

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

Examples: Code Contracts, Splint, JML, Grammatech CodeSonar, PolySpace, ThreadSafe, PRQA, Facebook Infer.

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.

assert(exp)

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

Question. What is the risk with running tests only with assertions enabled?

Assertions in Java APIs

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

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

Question. Do you think this is a good use of assertions?

Note 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?

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:

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

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

```
x = z; // { } (nothing known)
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.

```
x = v; // { } (nothing known)
if (x < y) // {x=v}
y = v; // {x=v, x>y}
assert (x < y) // Either: {x=v,y=v}: 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

Security assertions
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
Check integers are in required ranges:

- `alloc_size(dest)>strlen(src)`
- `array_size(a)>n` before `a[n]` access
Taintedness

tainted(mypageinput)
untainted(newkey)

▶ 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
▶ Taint analysis approximates information flow
  ▶ information may be leaked indirectly as well as directly

Type State (Resource) Tracking

isNull(ptr), nonull(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.

Code Contracts in .NET

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();
        }
    }
}
```

Findbugs GUI

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

Clang Static Analyser HTML reports

Basic architecture of a static analysis tool
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

The CFG consists of basic blocks and the paths between them.

Control flow graph:

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

- 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

```
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

Take away points

Static analysis tools can help find security flaws.
Massive benefits:
- examine millions of lines of code, repeatedly
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 (gradually?) 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: