Secure Programming Lecture 14: Static Analysis II

David Aspinall

Informatics @ Edinburgh

Outline

Recap

We’re looking at

- **principles and tools**

for ensuring software security.

This lecture looks at:

- further **example uses** of static analysis
- 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
- Summary

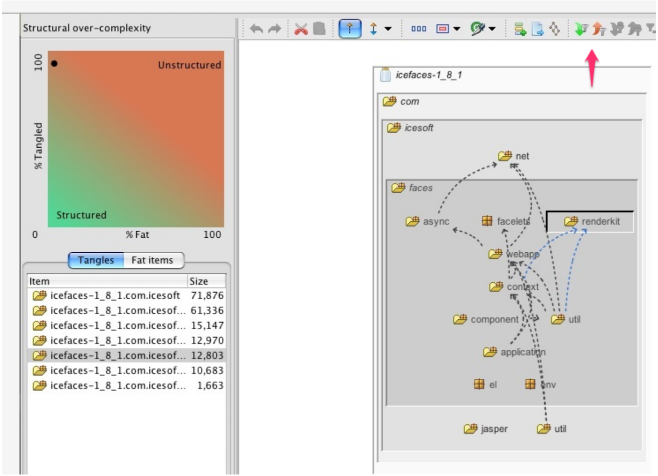
Program understanding tools

Help developers understand and manipulate large codebases.

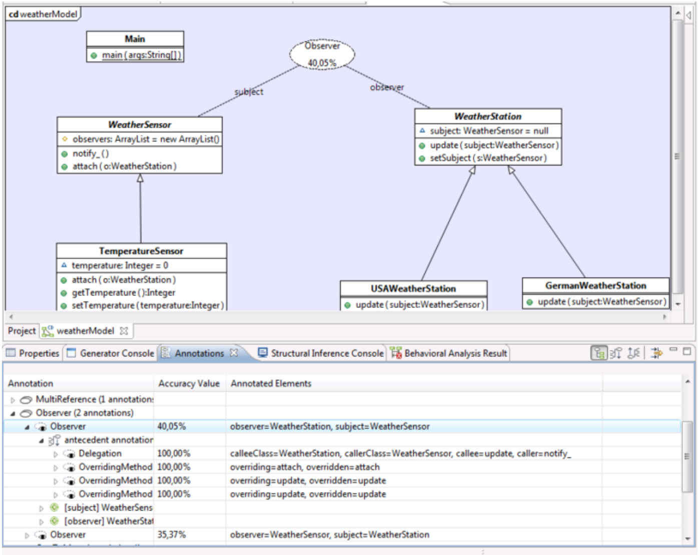
- Navigation swiftly inside the code
  - finding definition of a constant
  - finding call graph for a method
- Support *refactoring* operations
  - re-naming functions or constants
  - move functions from one module to another
  - needs internal model of whole code base
- Inferring *design* from *code*
  - 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) such as IDA Pro and Ghidra.

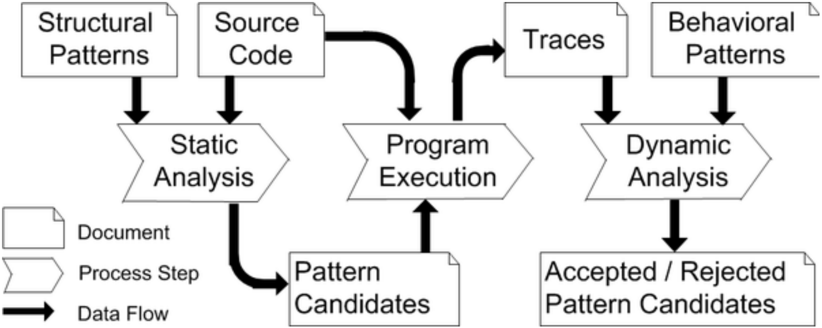
Commercial example: Structure101



Research example: Fujaba and Reclipse (2011)



How Reclipse works



See Fujaba project archive at University of Paderborn

Model-based testing

If we have a precise (formal) model of the system we can check it satisfies security properties.

- Test or staticallly check properties of models
- Models may be from design or extracted from code

Example tools include: Alloy.

General purpose tools like theorem provers or SMT solvers may be used as well.

Outline

- Overview
- Program understanding
- Program verification and property checking
- Bug finding
- Summary

Program verification

The “gold standard”, best guarantee for correctness.

- Uses **formal methods**
  - theorem proving
  - model checking
- Drawback: needs precise **formal specification**
- Drawback: expensive to industry
  - time consuming, needs experts (logic/maths)
  - . . . but investment up front may pay off
- Currently mainly used in safety critical domains
  - e.g., railway, nuclear, aeronautics
  - emerging: automobile, *security*

Example tools include: nuXmv and SPARK.

General purpose **Interactive Theorem Provers** such as Coq and Isabelle/HOL are also used to verify code.

See our course Formal Verification for more.

Property checking

AKA **Lightweight formal methods**

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

Static checking

- Prevent many violations of specification, not all
- Preconditions (`requires`) & postconditions (`ensures`)
- May produce *counterexamples* to explain violations

Examples (using a range of underlying techniques):
Code Contracts, Splint, JML, Grammatech CodeSonar, PolySpace, ThreadSafe, Facebook Infer.

Null References: A Billion Dollar Mistake?

Tony Hoare introduced Null references in ALGOL W back in 1965 “simply because it was so easy to implement”.

He later called it “my billion-dollar mistake”.

. . . but called the C `gets` function a multi-billion dollar mistake!

See his 2009 talk.

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**
  - 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?

Assertions in Java

```java
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 (potential 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.

```
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

Using assertions used in a static or dynamic way can be used to increase confidence in programs being correct.

Some static analysis tools use assertions (entirely) internally; others allow an interface using **annotations**.

So called **contract-based** programming uses explicit pre- and post- conditions supplied by the programmer when developing code.

Design by contract

Design by Contract (TM) aims to build a system as a set of components whose interaction is governed by mutual obligations, or *contracts*.

The idea was promoted by Bertrand Meyer in his design of the Eiffel OOP language (1986).

It adapts and extends ideas from *Hoare Logic* used for program verification, in particular, the use of pre-conditions and post-conditions.

Traditionally a *Hoare triple* is written like this:

$$\{P\}C\{Q\}$$

where C is a program command, P is a pre-condition and Q is a post-condition.

Example contract for insertion into dictionary

	Obligations	Benefits
Client	Table isn’t full, key is non-empty string	Get updated table with element added for given key
Supplier	Record given element in table associated with given key	No need to check for full table or empty key

Question. What are the preconditions and postconditions for the code here?

Specification in Eiffel

```
put (x: ELEMENT; key: STRING) is
        -- Insert x so that it will be retrievable through key.
    require
        count <= capacity
        not key.empty
    do
        ... Some insertion algorithm ...
    ensure
        has (x)
        item (key) = x
        count = old count + 1
    end
```

As well as pre and post conditions, other contract features include *class invariants* which must be established when an object is created and maintained whenever it is modified.

Relationship to defensive programming

"Defensive programming" adds checks to code to ensure that pre-conditions are met (coding assertions explicitly).

```
put (x: ELEMENT; key: STRING) is
    do
    if key.empty then
       error "Empty key supplied"
    ...
    end
```

With contracts these checks are **not** added: they are replaced by contract checking.

Contract checking may use static verification, or dynamic checking (or some combination).

Besides products sold by Eiffel Software, there is an open source free Eiffel tool chain developed by the Gobo Eiffel Project.

Contracts in Java

The **Java Modeling Language** allows specifications in the same way as Design by Contract.

```
/* requires 0 < n;
   assignable elems;
   ensures elems.length == n;
*/
publicBoundedStack(int n) {
   elems =newObject[n];
}
```

The OpenJML project implements a contract checking tool for JML.

Exercise. Try to understand the examples on the OpenJML home page. The *loop invariants* are complex but overall requires and ensures should be comprehensible.

Splint: Secure Programming Lint

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

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

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

Splint was introduced in 2001, it has a Github repo but isn't in active development by original academic authors.

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

```
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:

```
                         //  { }   (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:

```
                        //  { }   (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.

```
                                // {}    (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*

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

alloc_size(dest)>strlen(src)

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

- Check integers are in required ranges

Type State (Resource) Tracking

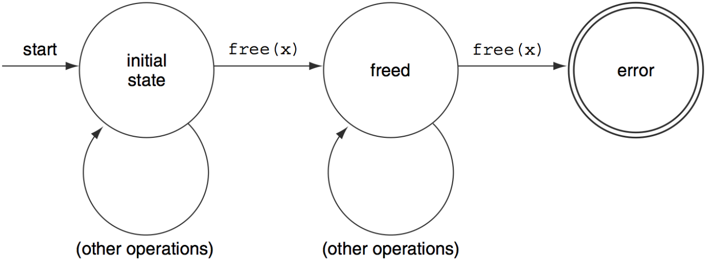
isnull(ptr), **nonull**(ptr)

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



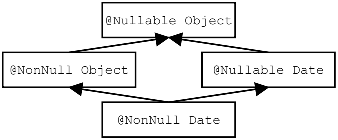
Extensible Type Systems

One approach to implementing type-state like systems is to use an extensible type system.

This allows “plugins” for type-based analysis.

An example is the Checker Framework for Java.

Example: Nullness Checkers in Java

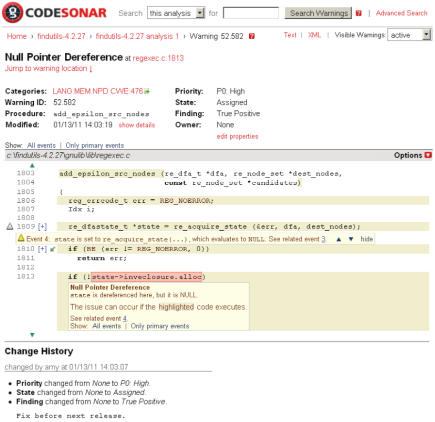


The **type annotation** `Nullable` is a type whose values may be `null`, whereas `NonNull` can never be `null`. This interacts with the class type hierarchy as above.

Exercise. Describe how these types help infer precise information and give errors. For example, inside a check 'if (date != null) ... ' we know the type of 'date' is 'NonNull Date'. APIs can use type annotations. Design other checkers for restricted integers, strings, etc.

Uber's recent `NullAway` tool is an example implementation of this analysis. See Nullness checker and NullAway on Github which is advertised as "giving great bank for your buck."

Null Pointers in CodeSonar



Not all null pointer analyses are equal! Some compilers spot only "obvious" null pointer risks, other tools perform deeper analysis like CodeSonar and NullAway.

Code Contracts in .NET

```
public string ReturnFirstThreeCharacters(string s) {
  return s.Substring(0, 3);
}
```

```
string string.Substring(int startIndex, int length)  (+ 1 overload(s))
Retrieves a substring from this instance. The substring starts at a specified character position and has a specified length.

Exceptions:
   System.ArgumentOutOfRangeException

Contracts:
   [Pure]
   requires 0 <= startIndex
   requires 0 <= length
   requires startIndex + length <= this.Length
   ensures result != null
   ensures result.Length == length
```

Unfortunately Code contracts aren’t supported in more recent versions of .NET, it isn’t clear why. Microsoft’s documentation suggests using Nullable reference types instead.

Outline

Overview

Program understanding

Program verification and property checking

Bug finding

Summary

Bug finding

Bug finding tools look for suspicious patterns in code. Traditionally they may be unsound (flag potential bugs that may not actually be bugs).

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

FindBugs is no longer maintained, and is now replaced by SpotBugs.

Example bugs

Common accidents

An error found in Sun's JDK 1.6:

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

Misunderstood APIs

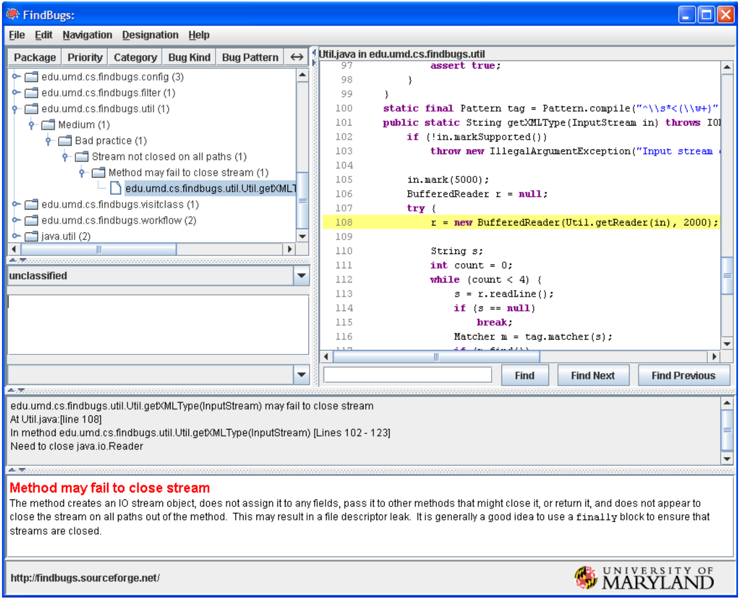
```
public String makeUserid(String s) {
   s.toLowerCase();
   return s;
}
```

Anti-idiom: double-checked locking in Java

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

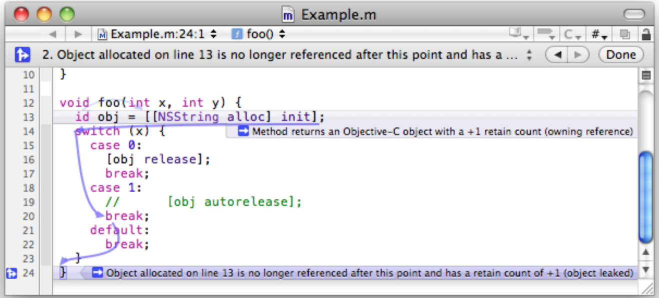
```
[dice]da: findbugs Fitz.class
M M DC: Possible doublecheck on Fizz.fitz in Fitz.getFitz()
        At Fitz.java:[lines 1-3]
```

FindBugs GUI



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

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

Bug Summary

Bug Type	Quantity	Display?
All Bugs	**269**	☑
API		
Argument with 'nonnull' attribute passed null	7	☑
Dead store		
Dead assignment	203	☑
Dead increment	11	☑
Dead initialization	2	☑
Logic error		
Assigned value is garbage or undefined	3	☑
Branch condition evaluates to a garbage value	1	☑
Dereference of null pointer	30	☑
Division by zero	1	☑
Result of operation is garbage or undefined	7	☑
Uninitialized argument value	4	☑

Reports

Bug Group	Bug Type ▾	File	Line	Path Length	
API	Argument with 'nonnull' attribute passed null	ssl/d1_both.c	1015	9	View Report
API	Argument with 'nonnull' attribute passed null	ssl/d1_srvr.c	1184	10	View Report
API	Argument with 'nonnull' attribute passed null	ssl/s3_srvr.c	1725	10	View Report
API	Argument with 'nonnull' attribute passed null	crypto/asn1/a_bytes.c	295	21	View Report

Outline

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 for general bugs, built into IDEs. Others are specific to security, may include:

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

These are becoming more mainstream

- frequency of security errors unmanageable
- $\rightsquigarrow$ deeper, wider automatic code analysis and repair
- integration into source code platforms like GitHub

Tools use program analysis to track properties of data being computed on, sometimes aided by annotations.

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:

- Al Bessey et al. *A few billion lines of code later: using static analysis to find bugs in the real world*, CACM 53(2), 2010.