# System and Software Testing
# Rendszer és szoftvertesztelés (GKN/LM_INTA/INTM057)

Csapó Ádám - csapo.adam@sze.hu
Varjasi Norbert – norbert.varjasi@sze.hu

Széchenyi István University
Department of Information Technology

## Topics for today – based on “Secure Programming with Static Analysis”, by B. Chess & J. West

- A deeper look into static analysis
    - Capabilities and limitations
    - Kinds of problems that can be solved
    - Difficulties with static analysis (in theory and practice)

## Static analysis in general

Any tool that analyses code wo/ executing it is a static analysis tool

In the most useful sense, static analysis is like spell checking

- A quick-and-dirty way of finding well-understood bugs
- Useful even for good programmers! (just like a spell checker)

Even good programmers can experience confusion, momentary lapses or problems with sensorimotor coordination

## Static analysis in general

Static analysis tools are great because:

They protect us from our biases (evaluating all parts of the code)

They often point to root causes (dynamic testing often encourages ‘palliative’ solutions)

They can find bugs early in development, before the program is run even once!

They are great in detecting security flaws, too. High-end tools often check for exploitation vulnerabilities that most programmers don’t normally think of

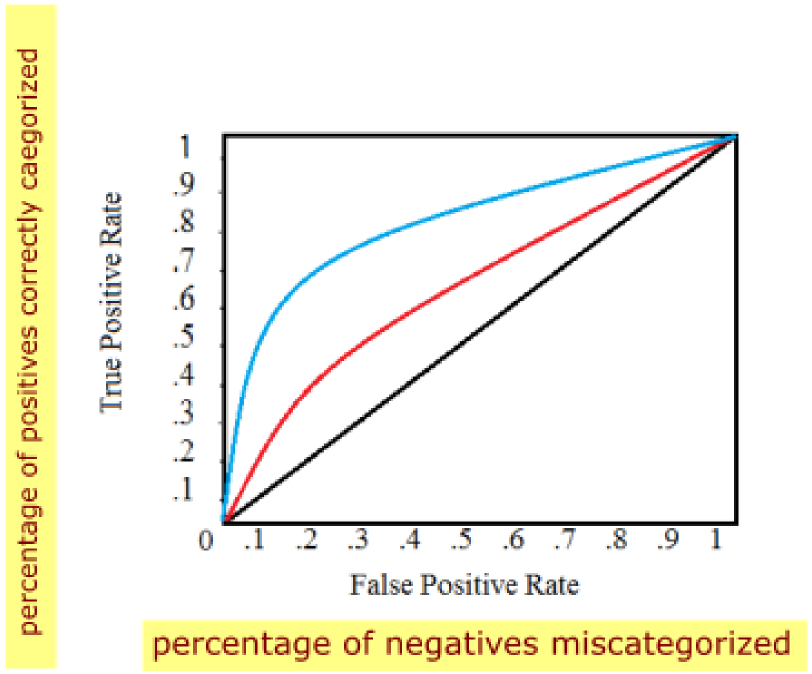
## Static analysis in general

A common frustration is the number of false positives…

but without false positives, false negatives can be higher

Think of this in terms of the ROC curve

If our threshold of decision is at 0, everything is a positive, so all true positives are correctly detected, and all true negatives are incorrectly detected (TR corner)



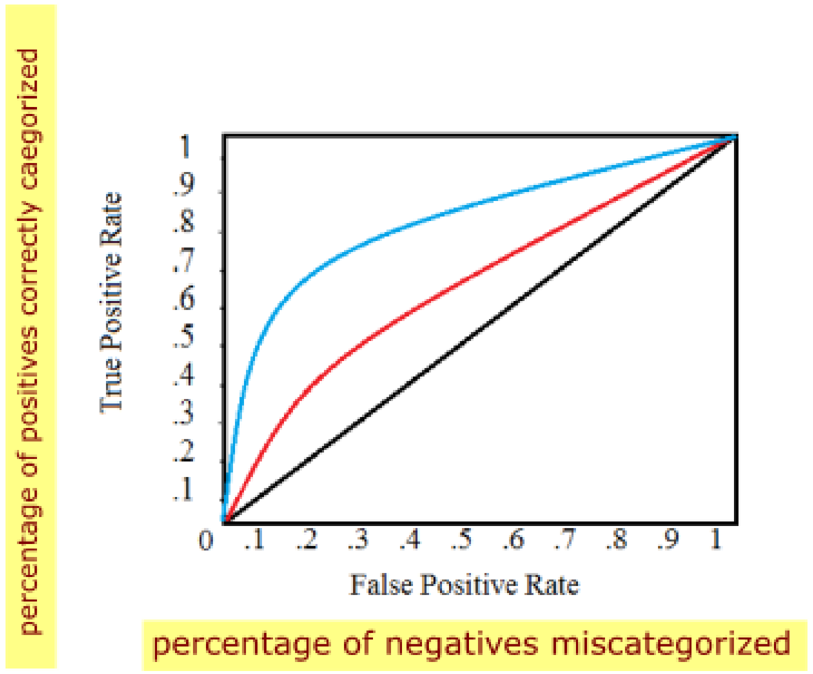
## Static analysis in general

A common frustration is the number of false positives…

but without false positives, false negatives can be higher

Think of this in terms of the ROC curve

If our threshold of decision is at 1, everything is a negative, so no true positives are correctly detected, and no true negatives are incorrectly detected (BR corner)



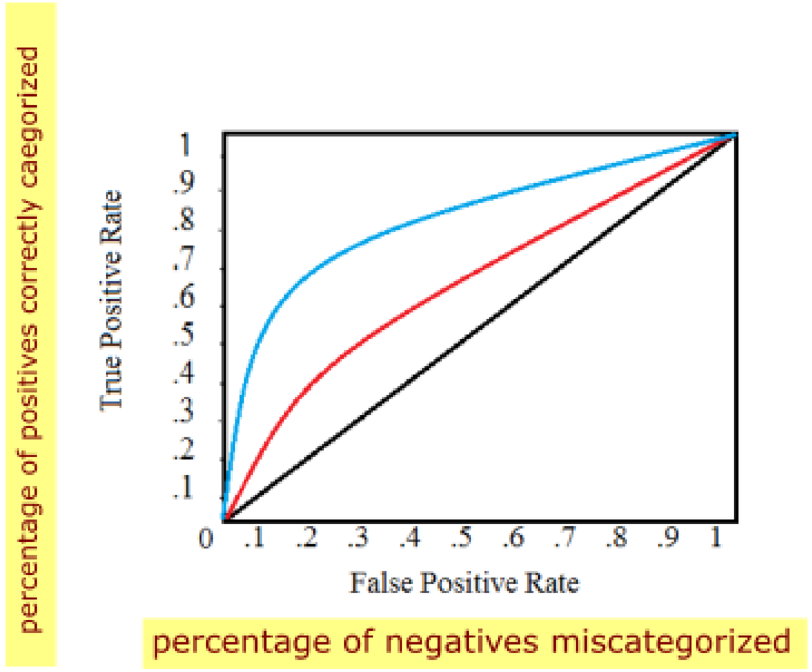
## Static analysis in general

A common frustration is the number of false positives…

but without false positives, false negatives can be higher

Think of this in terms of the ROC curve

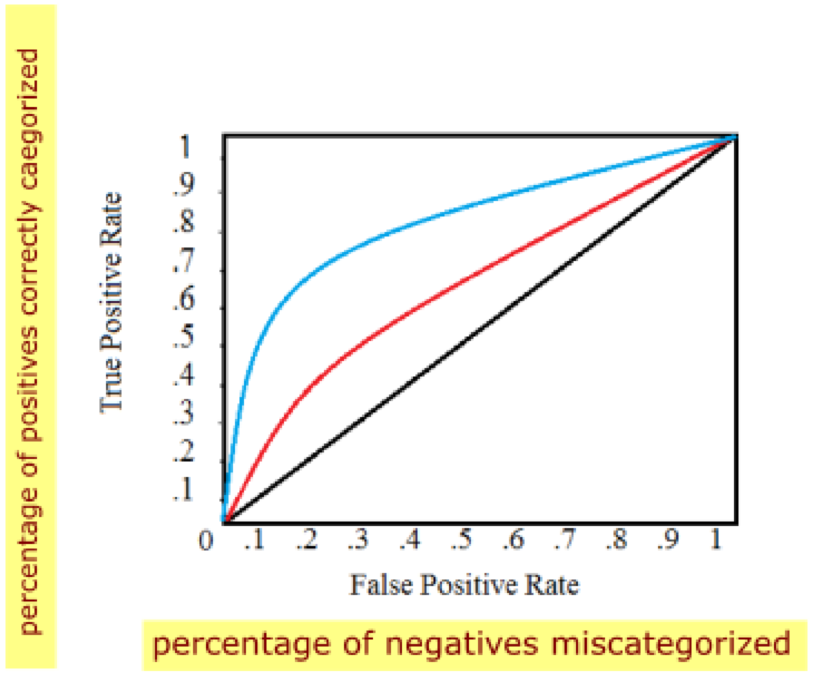
For a good categorizer, a decision threshold exists at which almost all positives are correctly, and no negatives are incorrectly categorized (like the blue curve but more extreme)



## Static analysis in general

So where are the false positives on this figure? They are the percentage of negatives miscategorized. If this number is low, then our true positive rate will also be lower!

This means that if one is serious (the costs of erring are high), one should prefer to deal with more false positives rather than sacrificing the detection of true positives!



## Static analysis in general

That said, static analysis tools for different purposes will likely use different thresholds of detection

- Are we looking for ‘garden-variety’ or security-related bugs?
- Are we looking for architectural risks?
  - The latter is not really amenable to (automated) tools
  - Code review by humans is indispensable!

## Kinds of problems solved by static analysis

There are many varieties of static analysis tools (most programmers may be unaware!) depending on their goal

- Type checking
- Style checking
- Program understanding
- Program verification
- Property checking
- Bug finding
- Security review

## Type checking

The most widely used form of static analysis

If the language is compiled -> no-brainer

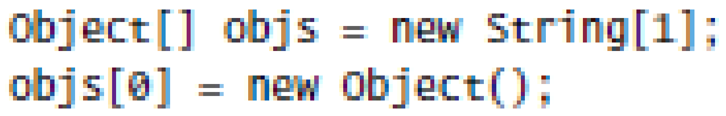
- Still, it’s static analysis! It prevents bugs like assigning an int value to object

Conceptually this is weird, but even type checking suffers from false positive / false negative challenges! Here is a false positive for example in Java:

```
short s = 0;
int i = s;   /* the type checker allows this */
short r = i; /* false positive: this will cause a
                type checking error at compile time */
```

## Type checking

Whereas this is a false negative:



```
Object[] objs = new String[1];
objs[0] = new Object();
```

The type checker doesn’t complain bc nominally, objs is an array of Objects. But at runtime, problems will ensue bc that array of Objects actually contains Strings (which are a kind of Object) and heterogeneous arrays are not allowed

## Style checking

Style checkers are generally more nit-picky than type checkers

- They look for style policies which, when transgressed, can lead to more buggy / less maintainable code
- Do not address bugs leading to errors directly

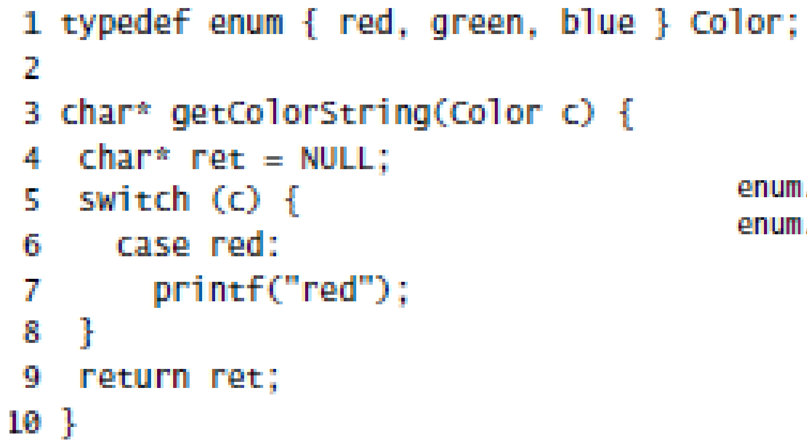
In the general case, style includes whitespaces, naming conventions, deprecated functions, commenting, control structure etc.

In most real-world cases, programmers pick and choose which conventions to enforce. A good example is rulesets in PMD…

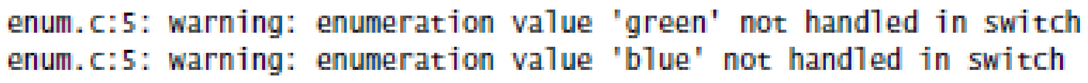
## Style checking

With time, many compilers have adopted some style-related policies.

E.g. with gcc, one can use the –Wall flag. Here is a suspicious case statement from C



```
 1 typedef enum { red, green, blue } Color;
 2
 3 char* getColorString(Color c) {
 4  char* ret = NULL;
 5  switch (c) {
 6    case red:
 7      printf("red");
 8  }
 9  return ret;
10 }
```



```
enum.c:5: warning: enumeration value 'green' not handled in switch
enum.c:5: warning: enumeration value 'blue' not handled in switch
```

## Program understanding

These tools help users make sense of a large codebase

Most IDE-s have tools like these, including:

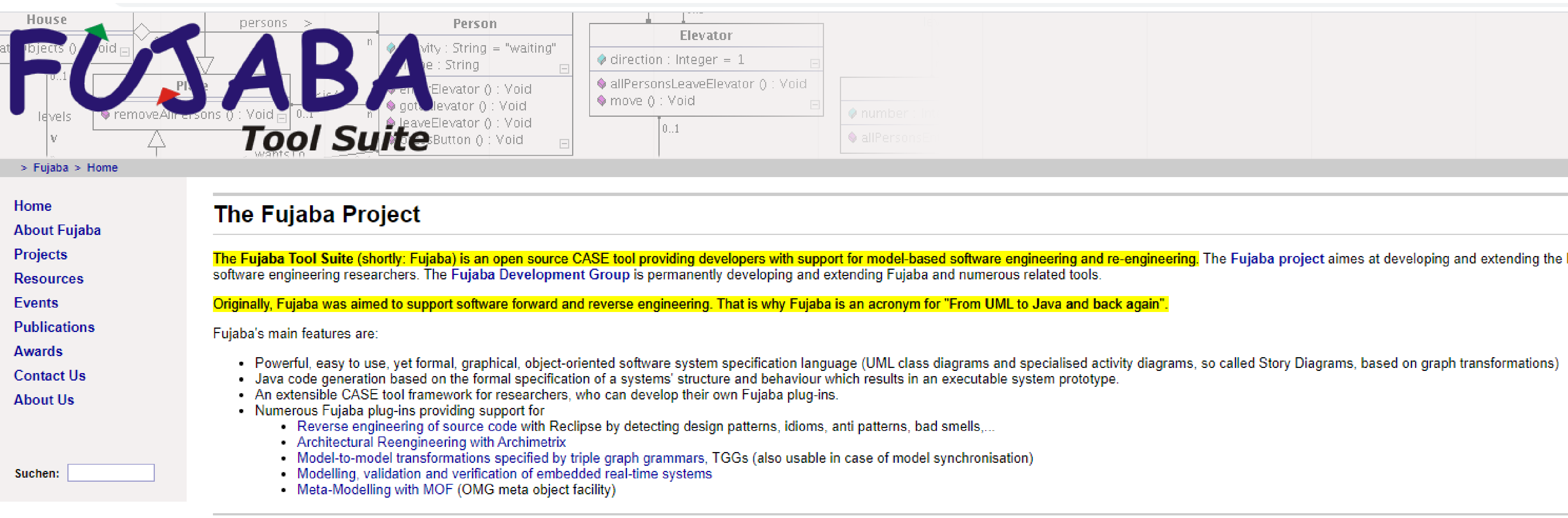
- Find all… replace all occurrences (rename class even!)
- Find the definition of this function

More advanced tools make suggestions on code refactoring

- e.g. bad naming conventions
- e.g. functions that are too long / do too much

# Program understanding

Fujaba is an open-source tool that allows users to interactively map between UML diagrams and the source code



> Fujaba > Home

Home
About Fujaba
Projects
Resources
Events
Publications
Awards
Contact Us
About Us

Suchen:

## The Fujaba Project

The **Fujaba Tool Suite** (shortly: Fujaba) is an open source CASE tool providing developers with support for model-based software engineering and re-engineering. The Fujaba project aimes at developing and extending the
software engineering researchers. The Fujaba Development Group is permanently developing and extending Fujaba and numerous related tools.

Originally, Fujaba was aimed to support software forward and reverse engineering. That is why Fujaba is an acronym for "From UML to Java and back again".

Fujaba's main features are:

- Powerful, easy to use, yet formal, graphical, object-oriented software system specification language (UML class diagrams and specialised activity diagrams, so called Story Diagrams, based on graph transformations)
- Java code generation based on the formal specification of a systems' structure and behaviour which results in an executable system prototype.
- An extensible CASE tool framework for researchers, who can develop their own Fujaba plug-ins.
- Numerous Fujaba plug-ins providing support for
  - Reverse engineering of source code with Reclipse by detecting design patterns, idioms, anti patterns, bad smells,...
  - Architectural Reengineering with Archimetrix
  - Model-to-model transformations specified by triple graph grammars, TGGs (also usable in case of model synchronisation)
  - Modelling, validation and verification of embedded real-time systems
  - Meta-Modelling with MOF (OMG meta object facility)

## Program verification / property checking

Program verification tools try to prove that a code is faithful to a specification based on the code and its specification

- If the specification is meant to be complete, it is a.k.a. equivalence checking
    - Less relevant to sw and more so to hw like circuits!
    - For sw, the costs outweigh the benefits

Most commonly for sw, the specification at hand is partial

This is known as property checking. It is done by logical inference or through model checking.

## Program verification / property checking

One useful category is temporal safety properties

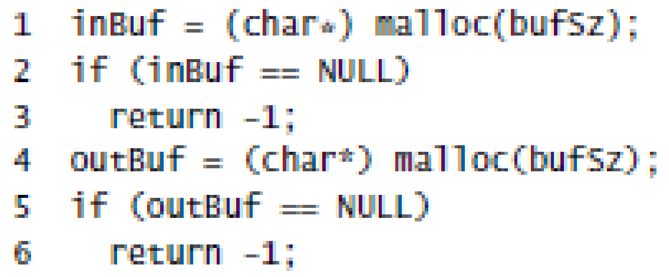
This is an ordered sequence of events that a program must not carry out

- E.g. a memory location should not be read after being freed

A related case is a state that must not arise

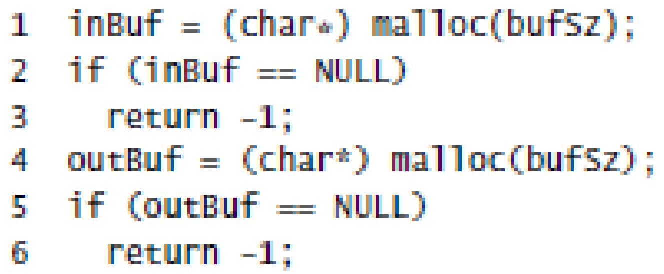
- E.g. two threads cannot hold the same semaphore at the same time

Here is a potential memory leak:



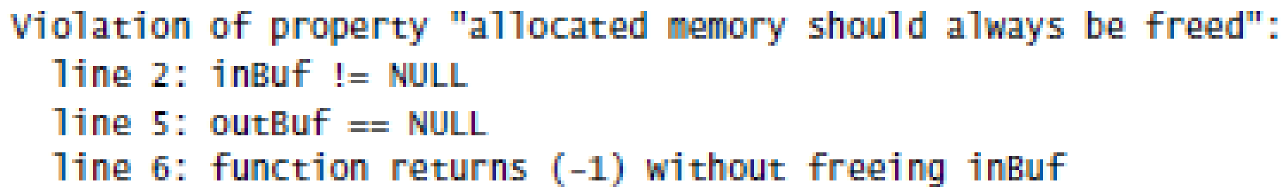
```
inBuf = (char*) malloc(bufSz);
if (inBuf == NULL)
  return -1;
outBuf = (char*) malloc(bufSz);
if (outBuf == NULL)
  return -1;
```

## Program verification / property checking



```
1  inBuf = (char*) malloc(bufSz);
2  if (inBuf == NULL)
3    return -1;
4  outBuf = (char*) malloc(bufSz);
5  if (outBuf == NULL)
6    return -1;
```

Property checkers often report a counterexample when they issue a warning:

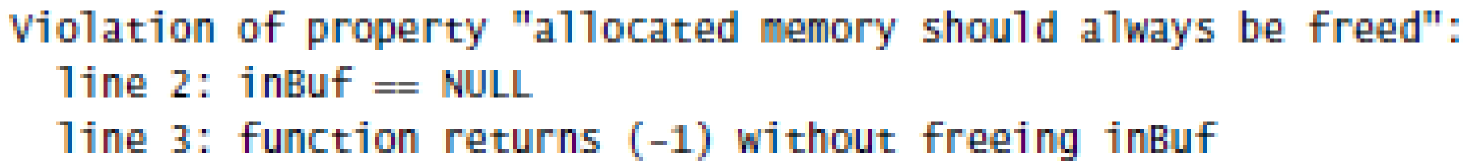


```
Violation of property "allocated memory should always be freed":
  line 2: inBuf != NULL
  line 5: outBuf == NULL
  line 6: function returns (-1) without freeing inBuf
```

Of course, false positives and false negatives are common in the case of verification / property checking tools, like any other

## Program verification / property checking

Here, the tool did not understand that if malloc() returns NULL, no memory was acquired:



```
Violation of property "allocated memory should always be freed":
  line 2: inBuf == NULL
  line 3: function returns (-1) without freeing inBuf
```

Examples of property checkers:

- Praxis High Integrity Systems (for the Ada language)
- Escher Technologies (a language that can be compiled into C/C++
- Polyspace, Grammatech
- Many university projects

## Bug finding

Bug finders usually go through a list of well-known bug types and search for them

Different from plain type checking and very different from style checking

A well-known open source tool for Java is FindBugs. For C++, there is Cppcheck. Visual Studio's compiler also has an /analyze flag

Sophisticated bug finders might even learn from the programmer's code – if a check is done in 99 instances, probably the 100th should be checked as well…

Bug finders often aim to be "sound w.r.t counterexamples" – if a bug is found, it should really be a bug. This means tolerance to false negatives

## Security review

Such tools usually aim for less false negatives and more false positives…

The earliest tools (ITS4, RATS, Flawfinder) usually searched for notorious functions like strcpy() or malloc()

- Kind of like style checkers…

Modern tools are more like a combo of property checkers and bug finders

- Many security vulnerabilities can be expressed as program properties. For example, buffer overflow = “the program never addresses memory that is out of bounds”
- Bug finding is relevant because programmers often use the “same trick” to solve common problems. However, false positives are more desirable here
- Examples: Fortify Software, Ounce Labs

## Difficulties with static analysis

There exists no perfect static analysis method

  (There exists no perfect dynamic analysis method)

The theory behind that is supplied by the famous Halting problem (Entscheidungsproblem) and Rice's theorem

  Key idea: assume we have a "superprogram" that takes a program and decides whether it halts or goes into an infinite loop.

  You can get a contradiction by constructing a program that loops infinitely exactly when the superprogram says that it would halt, and halt otherwise

## Difficulties with static analysis

In practice, though, we are managing risks and do not need the perfect solution

- Saying that the theoretical impossibilities of static analysis are too big of a problem is like saying that our biggest problem is not being able to fly at the speed of light

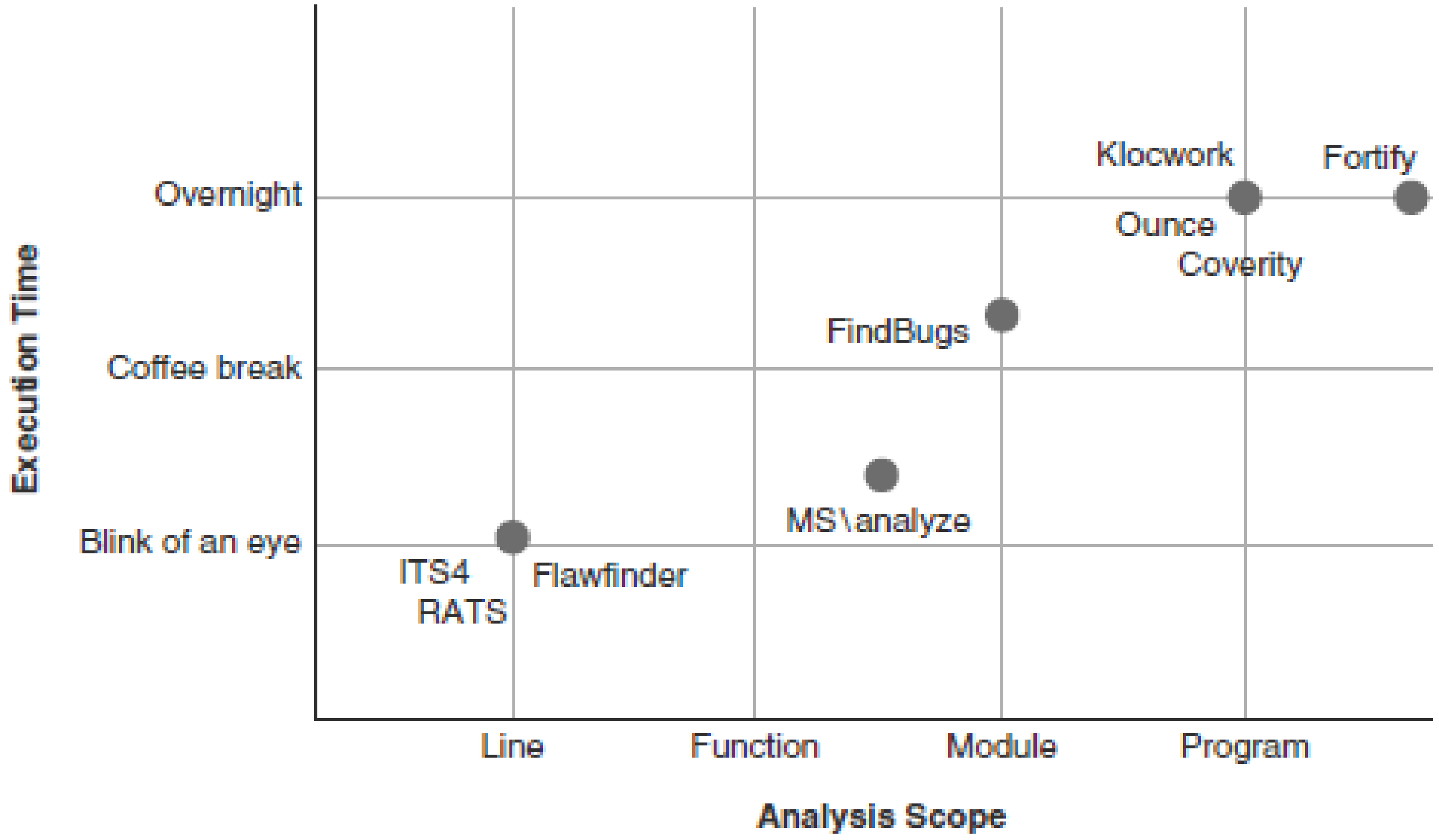
Main challenges include:

- making sense of the program, at scale, in relatively little time
- Analyze source or compiled code? (variable-width instructions?)

## Difficulties with static analysis

- Making sense of the program means not only understanding its structure but also the libraries behind it. Also, it means understanding what the compiler will do (optimizations?)
- What’s even worse, many real-life programs are written in more than one language
- To do all this work, static analyzers have to surrender either precision, depth or scalability
  - The most precise analyzers can run out of memory easily
  - The best analyzers require an execution time of 8-10 hours

## Difficulties with static analysis



## Homework assignment

Try out a static analysis tool like PMD or /analyze in Visual Studio

Try out different rule sets

Experiment with how the tool can be integrated into the development process