HTWG

Hochschule Konstanz
Fakultät Informatik

Software Security

AIN

Hanno Langweg

04 Source Code Analysis

Supply Chain Security

Trust

https://amturing.acm.org/award_winners/thompson_4588371.cfm

- Ken Thompson,
  Turing Award 1983
- Award lecture:
  Reflections on trusting trust
  - *To what extent should one trust a statement that a program is free of Trojan horses?*
    Perhaps it is **more important to trust the people** who wrote the software.
- Cannot set up everything (processes, products, services) to prepare for vandalism

How to trust code?

- Evaluate binary
  - Review source code, compile yourself
  - Evaluate compiler
    - Review compiler source code? +compiler's compiler?
    - Review compiler machine code?
  - Evaluate operating system
  - Evaluate hardware, firmware

- Evaluate persons, processes

How They Did It: An Analysis of Emission Defeat Devices in Modern Automobiles

Moritz Contag*, Guo Li†, Andre Pawlowski*, Felix Domke‡,
Kirill Levchenko†, Thorsten Holz*, and Stefan Savage†

* Ruhr-Universität Bochum, Germany, {moritz.contag, andre.pawlowski, thorsten.holz}@rub.de
† University of California, San Diego, {gul027, klevchen, savage}@cs.ucsd.edu
‡ tmbinc@elitedvb.net

***Abstract*—Modern vehicles are required to comply with a range of environmental regulations limiting the level of emissions for various greenhouse gases, toxins and particulate matter. To ensure compliance, regulators test vehicles in controlled settings and empirically measure their emissions at the tailpipe. However, the black box nature of this testing and the standardization of its forms have created an opportunity for evasion. Using modern electronic engine controllers, manufacturers can programmatically infer when a car is undergoing an emission test and alter the behavior of the vehicle to comply with emission standards, while exceeding them during normal driving in favor of improved performance. While the use of such a defeat device by Volkswagen has brought the issue of emissions cheating to the public's attention, there have been few details about the precise nature of the defeat device, how it came to be, and its effect on vehicle behavior.**

determined that the vehicle was not under test, it would disable certain emission control measures, in some cases leading the vehicle to emit up to 40 times the allowed nitrogen oxides [15].

Defeat devices like Volkswagen's are possible because of how regulatory agencies test vehicles for compliance before they can be offered for sale. In most jurisdictions, including the US and Europe, emissions tests are performed on a chassis dynamometer, a fixture that holds the vehicle in place while allowing its tires to rotate freely. During the test, a vehicle is made to follow a precisely defined speed profile (i.e., vehicle speed as a function of time) that attempts to imitate real driving conditions. The conditions of the test, including the speed profile, are both standardized and public, ensuring

ICT Supply chain

- Software composed of many parts
- Parts acquired from suppliers
- Insider threat
  - Hard to detect backdoors
  - Less skill needed to hide malicious code than to find
- Secure process of binary creation from source to deployed application
- Research@HTWG: How to introduce vulnerabilities into unfamiliar source code?

Insiders – inside and outside

- **Outsourcing and offshore development** risks
  - **Untrustworthy staff**
    - Contract workers, contractors, sub-contracting
    - Insider threat, traceability of code origin/modification
  - **Untrustworthy suppliers**
    - Products: Libraries, components
    - Services: Deployment, operation

- M. Naedele and T. Koch (2006). "Trust and tamper-proof software delivery". 2006 International Workshop on Software Engineering for Secure Systems. Pp. 51-58. http://dx.doi.org/10.1145/1137627.1137636

Trust models for ICT supply chain

- "Blind trust", "Trust by directive", "Liability"
  ➔ Sufficient to evaluate 1st supplier
- "Reputation", "Strong interest"
  ➔ Evaluate 1st supplier + 3rd party components
- "Proven in use"
  ➔ All relevant components known and evaluated?
- Higher risk
  - "Weak interest"
    ➔ Supplier without active interest
  - "Idealism"
    ➔ Supplier might be harmed by others

Software Inspections

Motivation for inspections (i)

Reduce vulnerabilities in code

- Effectiveness: how much reduction?
- Efficiency: at what cost?
  - Manual reviews: Skilled labor is scarce
  - Automation (but manual follow-up)
- Risk-based analysis/testing
  - Reduce scope

Motivation for inspections (ii)

- Code inspections mandatory for CC EAL4 and above
- Ensure that more than one person
  has seen every piece of code
- Prospect of someone else reviewing
  your code raises quality
- Force developers to document decisions
- Train junior developers to get familiar with code base

Methods to find vulnerabilities

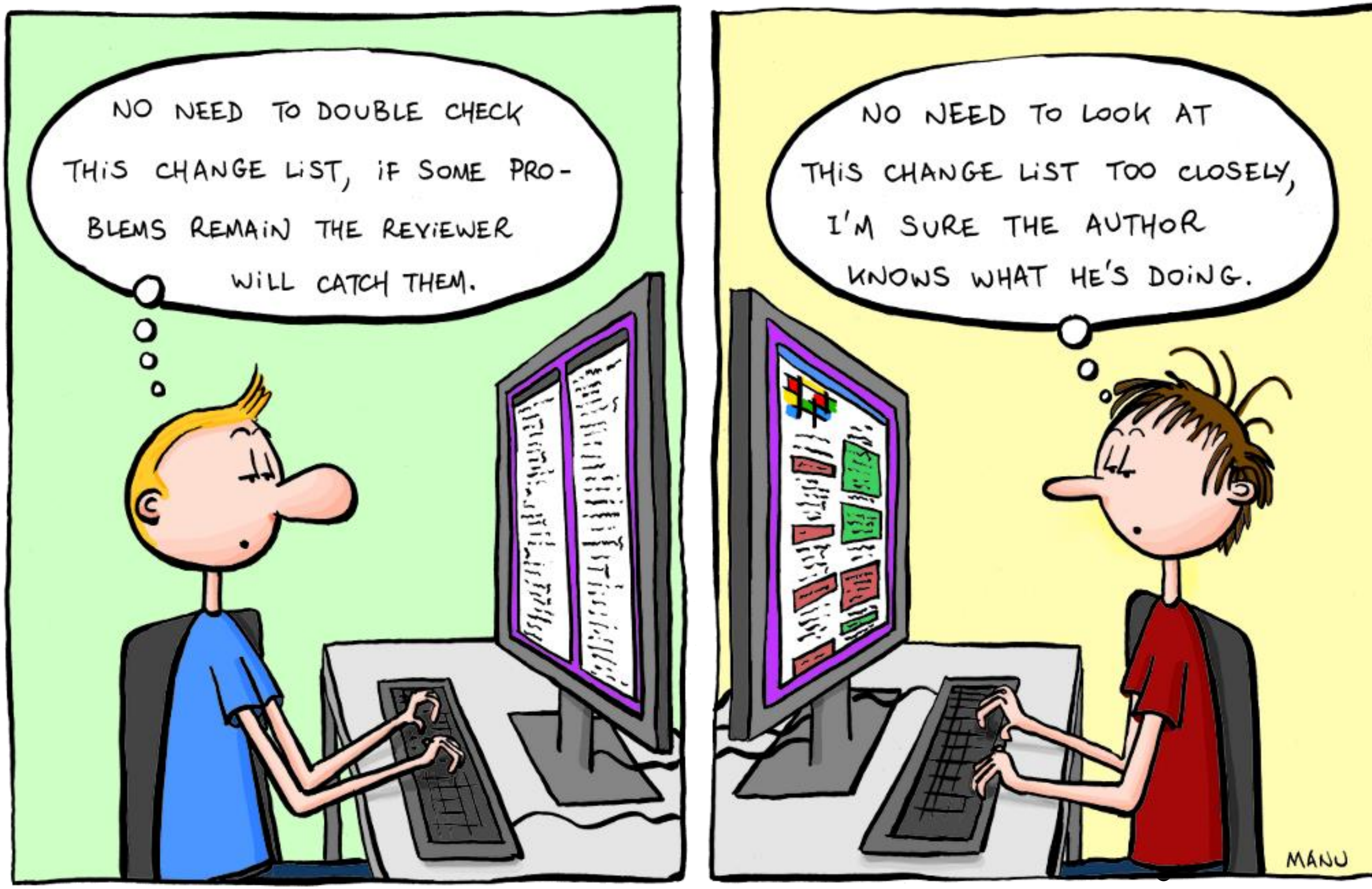
- Black box
  - Same situation as attackers
- White box
  - Insider knowledge, e.g., review of source code
- Grey box

Code inspection

- **Creative** process
  - Come up with unexpected situations
  - Cannot force creativity
- **Skills**
  - Know application **context**
  - Know programming **language**
  - Infer original **intention**
- **Hypothesize effects** of unexpected situations

Inspection process (example)

- Define **scope**
- **Collect** information
- **Review**: design, code; Test
- **Document** findings
- **Analyse** findings: severity, remediation effort
- (Support remediation)


NO NEED TO DOUBLE CHECK THIS CHANGE LIST, IF SOME PRO-BLEMS REMAIN THE REVIEWER WILL CATCH THEM.
NO NEED TO LOOK AT THIS CHANGE LIST TOO CLOSELY, I'M SURE THE AUTHOR KNOWS WHAT HE'S DOING.
MANU

Define scope of inspection

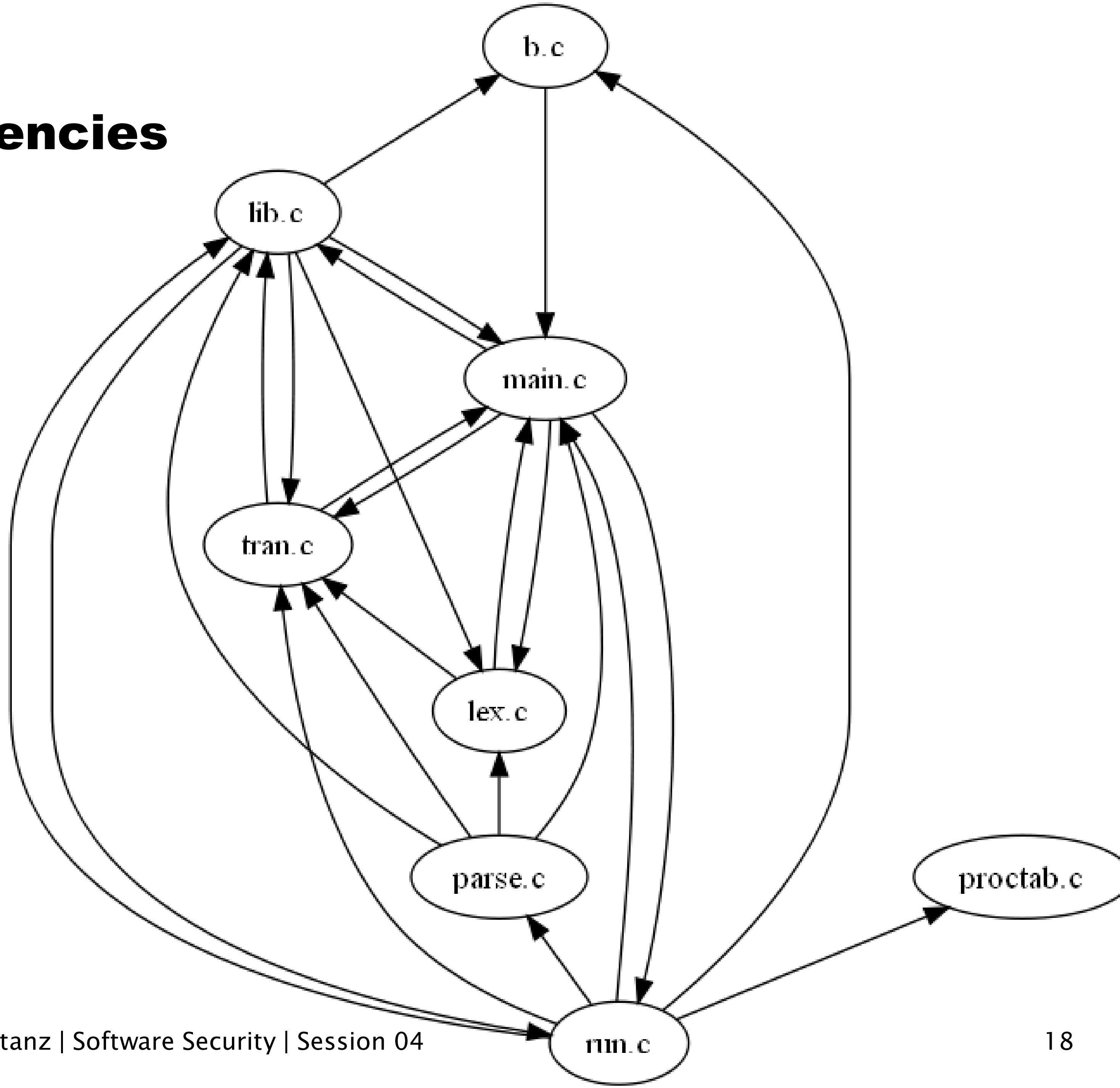
- **Goal** of inspection
  - Ease of detection, severity, code coverage, depth
  - Business **context**
  - **Compliance** with regulations
- Type of access
  - Source
  - Binary, binary+debug info
  - Black box access
    to interfaces

Tools: scope

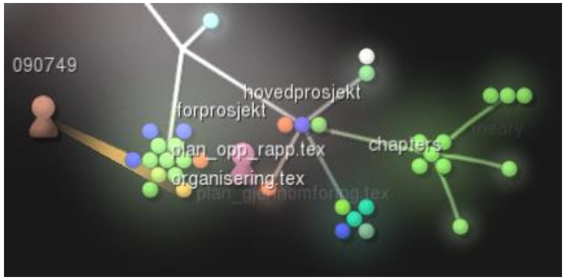
- Coverage
  - Execute test cases, measure coverage, i.e. which parts of code affected
  - Inspect affected code (assumption: what is not tested is less relevant code)
- Call graphs
  - Identify modules used by many
  - E.g. CScout

Dependencies

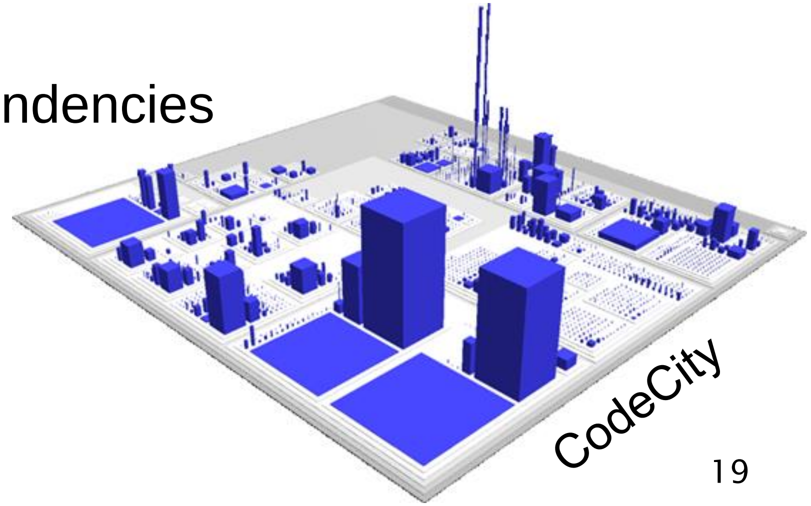
CScout sample output



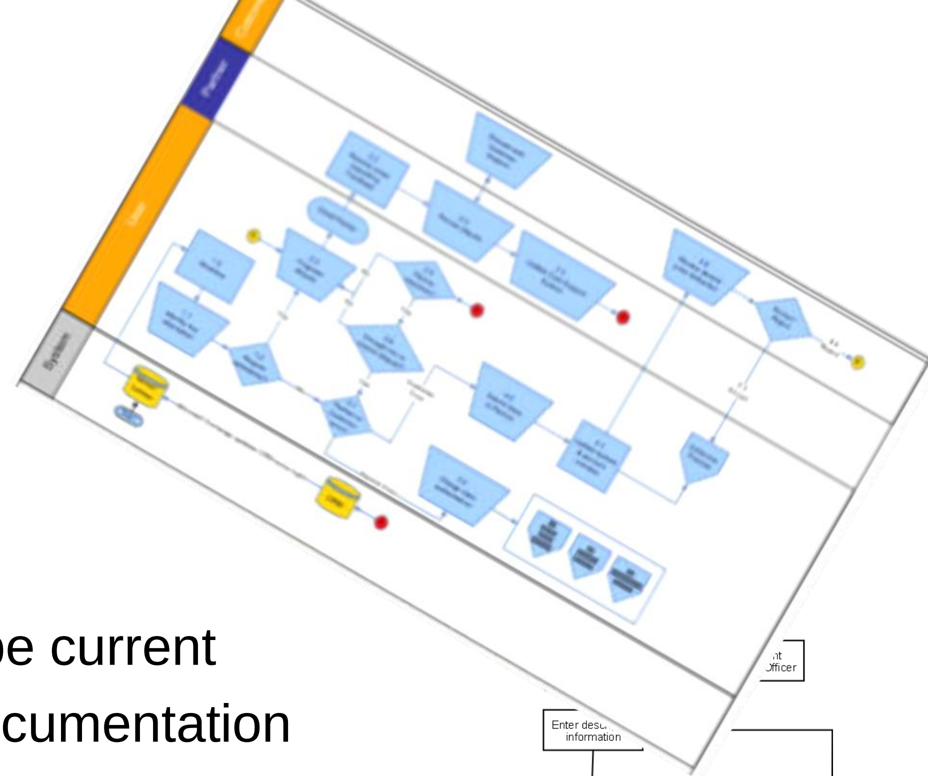
Tools: scope



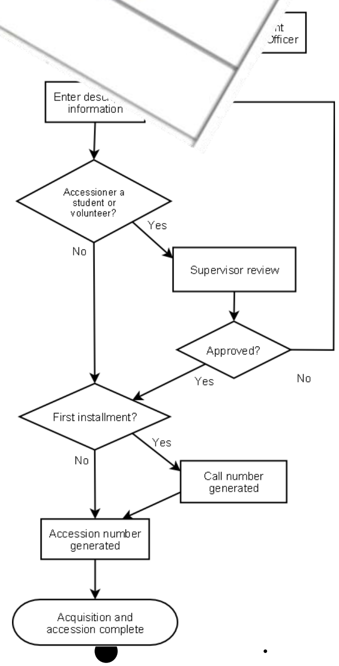
- Hot spots
  - Issue trackers: modules mentioned often (past experience)
  - Version control logs
    - Code modified often
    - Code modified often because of bug fixes
    - Defect prediction (type of change, dev. experience, affected LOC, affected #files)
  - Code metrics: complexity, #dependencies
  - Code visualisation
    - E.g. CodeCity, Gource

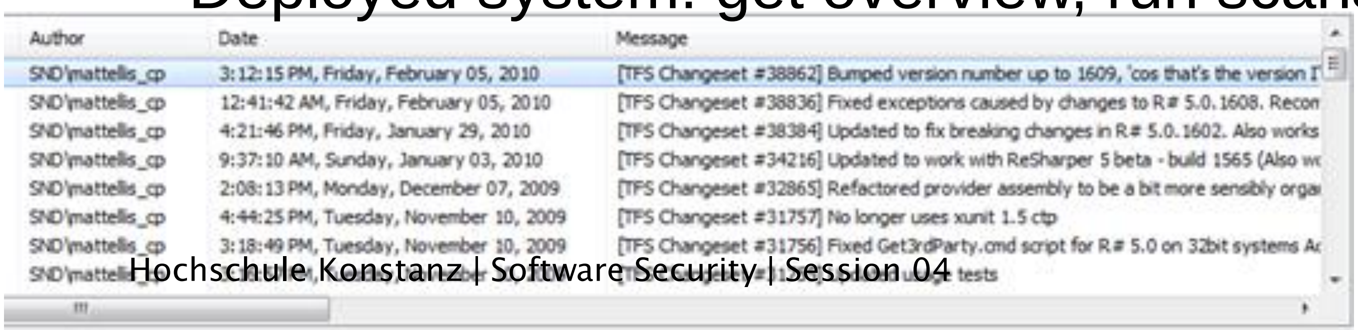


Collect information



– System documentation
– Developer documentation
  – Might not exist or might not be current
  – Version control log is also documentation
– Interviews: be polite
– Documentation of standards that are used
– Source code: get overview, get feeling
– Deployed system: get overview, run scans





Author	Date	Message
SND\mattellis_cp	3:12:15 PM, Friday, February 05, 2010	[TFS Changeset #38862] Bumped version number up to 1609, 'cos that's the version I
SND\mattellis_cp	12:41:42 AM, Friday, February 05, 2010	[TFS Changeset #38836] Fixed exceptions caused by changes to R# 5.0.1608. Recon
SND\mattellis_cp	4:21:46 PM, Friday, January 29, 2010	[TFS Changeset #38384] Updated to fix breaking changes in R# 5.0.1602. Also works
SND\mattellis_cp	9:37:10 AM, Sunday, January 03, 2010	[TFS Changeset #34216] Updated to work with ReSharper 5 beta - build 1565 (Also wo
SND\mattellis_cp	2:08:13 PM, Monday, December 07, 2009	[TFS Changeset #32865] Refactored provider assembly to be a bit more sensibly orga
SND\mattellis_cp	4:44:25 PM, Tuesday, November 10, 2009	[TFS Changeset #31757] No longer uses xunit 1.5 ctp
SND\mattellis_cp	3:18:49 PM, Tuesday, November 10, 2009	[TFS Changeset #31756] Fixed Get3rdParty.cmd script for R# 5.0 on 32bit systems Ac

Plan, review, reflect

- Decide what to look for and how
  - High level (design) vs low level (implementation)
  - Assets, entry points, trust boundaries, relationships
- Perform the review
  - Vary: people, perspective
  - Take notes, annotate source code
  - Stay focused, do not get lost in details
- Reflect: What worked well, what did not?
  - Relate to review goals

Inspection strategies

- Code comprehension
  - Analyse source code, gain understanding
  - Trace input, analyse module/algorithm/class
- Candidate points
  - List potential issues, then examine source code
  - Tools often used to create list
- Design generalisation
  - Recover design from source code
  - Identify missing trust boundaries

Source level: tools

- Code browsers
  - Navigation in code, often integrated in IDE
  - Cross-referencing variable/function definition/use
  - Formatting/pretty-printing, make code easier to read, e.g. Artistic Style (http://astyle.sourceforge.net)
  - Call graphs, functional dependencies
- Version control logs
- Issue trackers (linked to version control)

Time

- Reviewers differ in speed
  - Background, experience
  - Overview vs detail orientation
- Speed differences in literature: 1:10
- Own observation: 1:7 difference between people

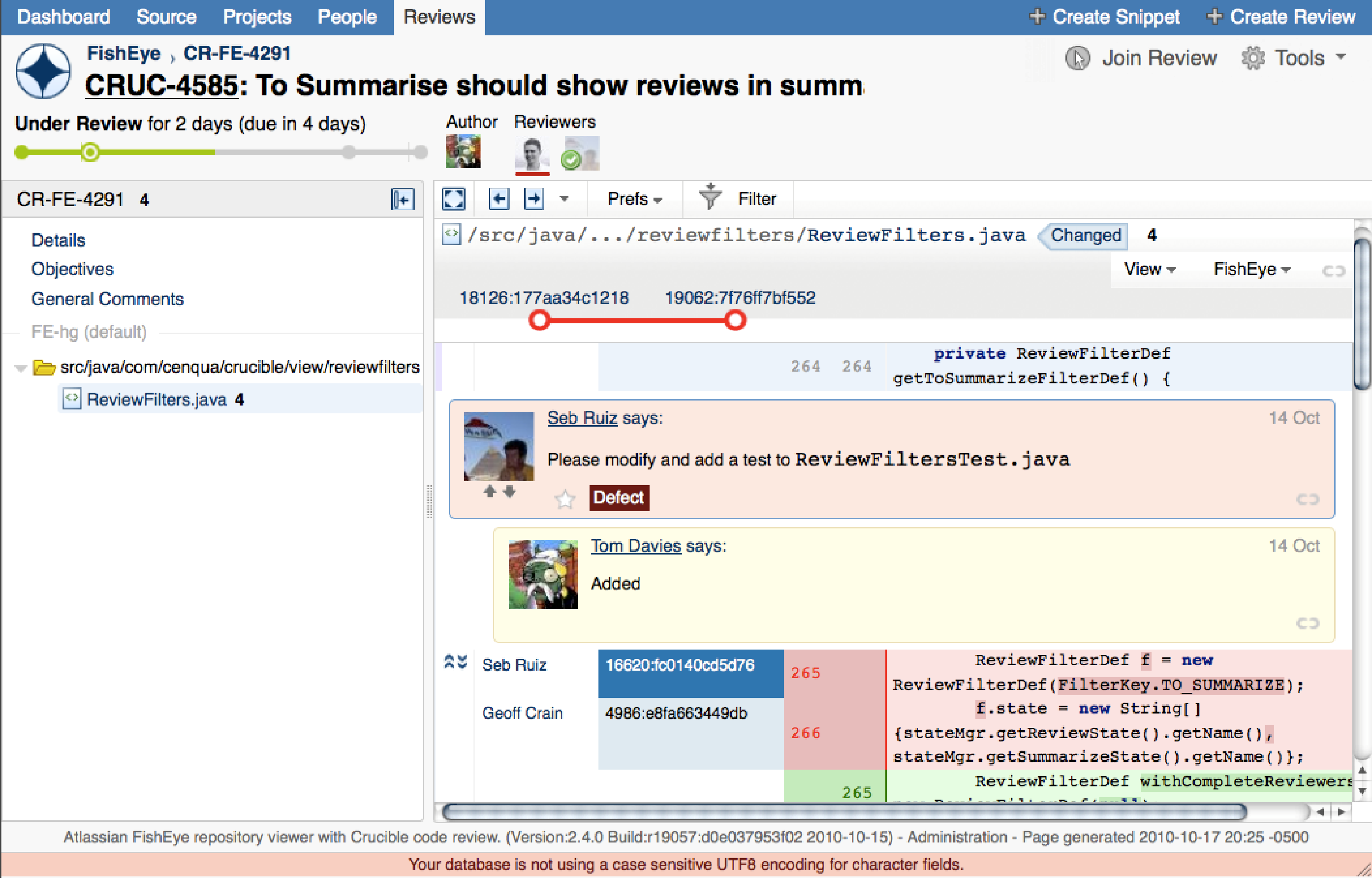
https://upload.wikimedia.org/wikipedia/commons/3/3b/Time_study_stopwatch.JPG

Test

- White box
  - All manufacturer's knowledge available
- Grey box
- Black box testing
  - No insider knowledge
- Verify **presence** of vulnerabilities
- **Automation** enables large number of test cases

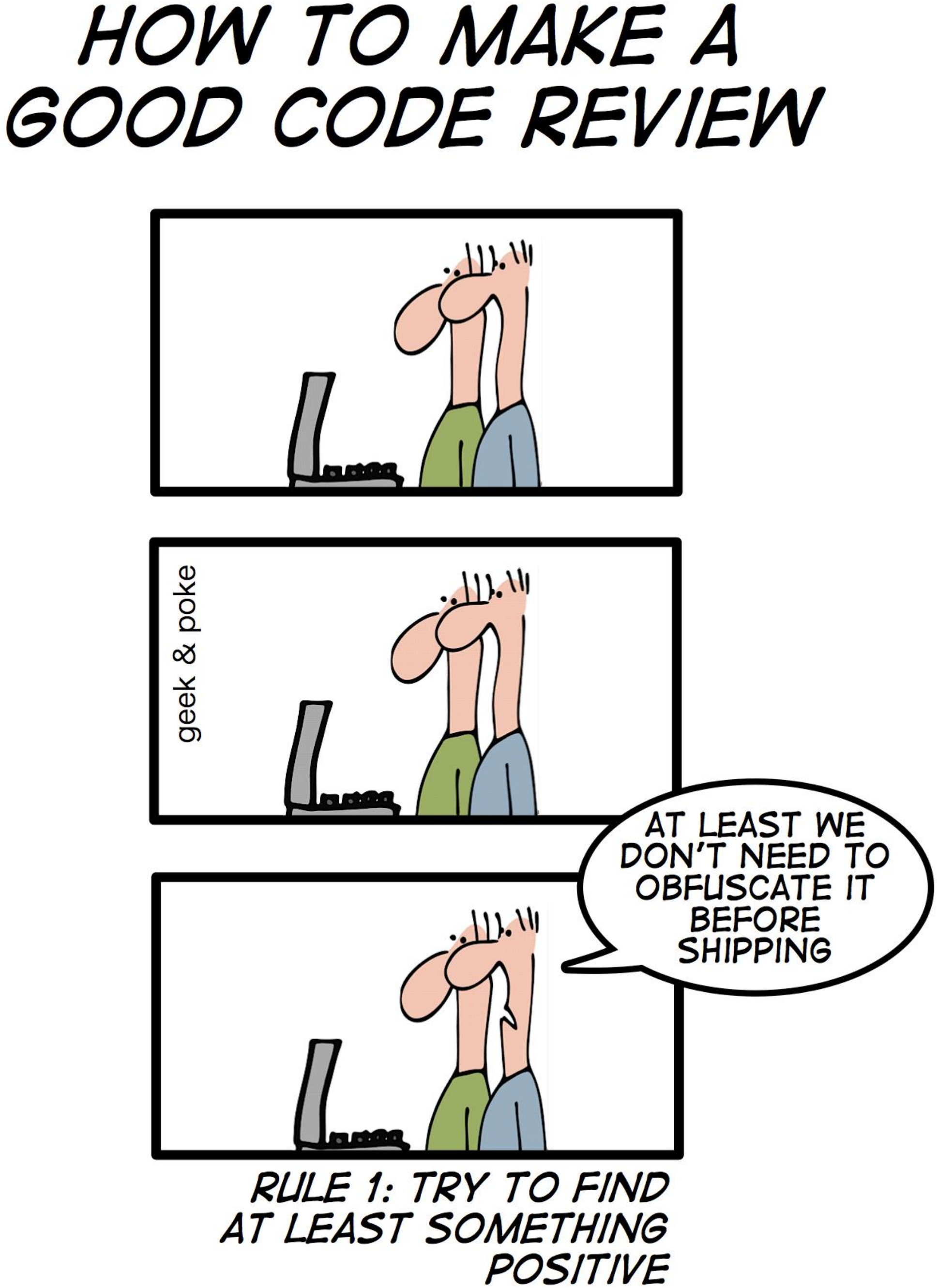
Document and analyse findings

- Method used
- Code coverage
- Findings
  - Threat, description, impact, location
  - Proposed remediation
    (steps, effort)


Dashboard
Source
Projects
People
Reviews
Create Snippet
Create Review
FishEye › CR-FE-4291
CRUC-4585: To Summarise should show reviews in summ
Join Review
Tools
Under Review for 2 days (due in 4 days)
Author
Reviewers
CR-FE-4291 4
Details
Objectives
General Comments
FE-hg (default)
src/java/com/cenqua/crucible/view/reviewfilters
ReviewFilters.java 4
Prefs
Filter
/src/java/.../reviewfilters/ReviewFilters.java
Changed
4
View
FishEye
18126:177aa34c1218
19062:7f76ff7bf552
264 264 private ReviewFilterDef getToSummarizeFilterDef() {
Seb Ruiz says:
14 Oct
Please modify and add a test to ReviewFiltersTest.java
Defect
Tom Davies says:
14 Oct
Added
Seb Ruiz
16620:fc0140cd5d76
Geoff Crain
4986:e8fa663449db
265 ReviewFilterDef f = new ReviewFilterDef(FilterKey.TO_SUMMARIZE);
266 f.state = new String[] {stateMgr.getReviewState().getName(), stateMgr.getSummarizeState().getName()};
265 ReviewFilterDef withCompleteReviewers
Atlassian FishEye repository viewer with Crucible code review. (Version:2.4.0 Build:r19057:d0e037953f02 2010-10-15) - Administration - Page generated 2010-10-17 20:25 -0500
Your database is not using a case sensitive UTF8 encoding for character fields.

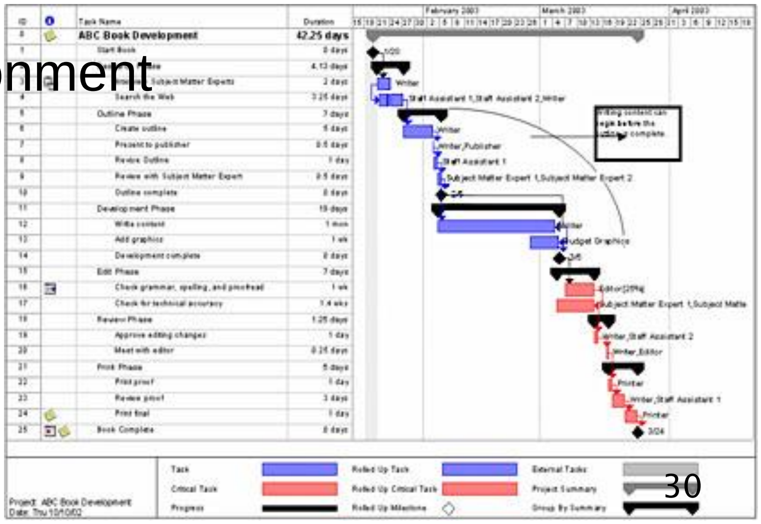
More complex inspection process

– NASA-STD-8739.9 Software Formal Inspections Standard
  http://www.hq.nasa.gov/office/codeq/doctree/ns87399.htm
  Not limited to security vulnerabilities


HOW TO MAKE A GOOD CODE REVIEW
geek & poke
AT LEAST WE DON'T NEED TO OBFUSCATE IT BEFORE SHIPPING
RULE 1: TRY TO FIND AT LEAST SOMETHING POSITIVE

Support remediation

– Not part of review, but **consequence** of review
– **Knowledge gained in discovery** useful for fixing
– **To fix or not to fix**
    – Estimate effort
    – Business context
    – Alternatives: change configuration, reduce functionality, change environment
– **Review fix**



Motivation for inspections

Reduce vulnerabilities in code

- Effectiveness: how much reduction?
- Efficiency: at what cost?
  - Manual reviews: Skilled labor is scarce
  - **Automation** (but manual follow-up)
- Risk-based analysis/testing
  - Reduce scope

Static code analysis

Type systems

- Support security by
  - imposing discipline/restrictions on developer
  - enforcing abstractions on developer
- **Memory safety**
  - Programming language ensures that **only allocated** (and initialised) memory can be **referenced**
- **Type safety**
  - Types annotate program elements to assert invariant properties (e.g. integer, array, string)
  - Type checking verifies the assertions

Type safety

- Type
  - Give meaning to sequence of bits (memory, object)
  - Type checking: verifying/enforcing constraints of types
  - Resulting type of expression combining types
  - **Static** typing: Type checking at **compile-time**
  - **Dynamic** typing: Type checking at **run-time**
  - Important that evaluation of expressions yields defined and predictable results

```
var x := 5;      // (1)
var y := "37";   // (2)
var z := x + y;  // (3)
```

VB

Type safety

- Type
  - Give meaning to sequence of bits (memory, object)
  - Type checking: verifying/enforcing constraints of types
  - Resulting type of expression combining types
  - **Static** typing: Type checking at **compile-time**
  - **Dynamic** typing: Type checking at **run-time**
  - Important that evaluation of expressions yields defined and predictable results

```
var x := 5;     // (1)
var y := "37";  // (2)
var z := x + y; // (3)
```

VB

```
int x = 5;
char y[] = "37";
char* z = x + y;
```

C

Type information

```
public class Demo{
  static private string greeting = "Hello ";
  final static int CONST = 43;

  static void  Main (string[]  args){
     foreach (string name in args){
        Console.Writeline(sayHello(name));
     }
  }

  public static string sayHello(string name){
     return greeting + name;
  }
}
```

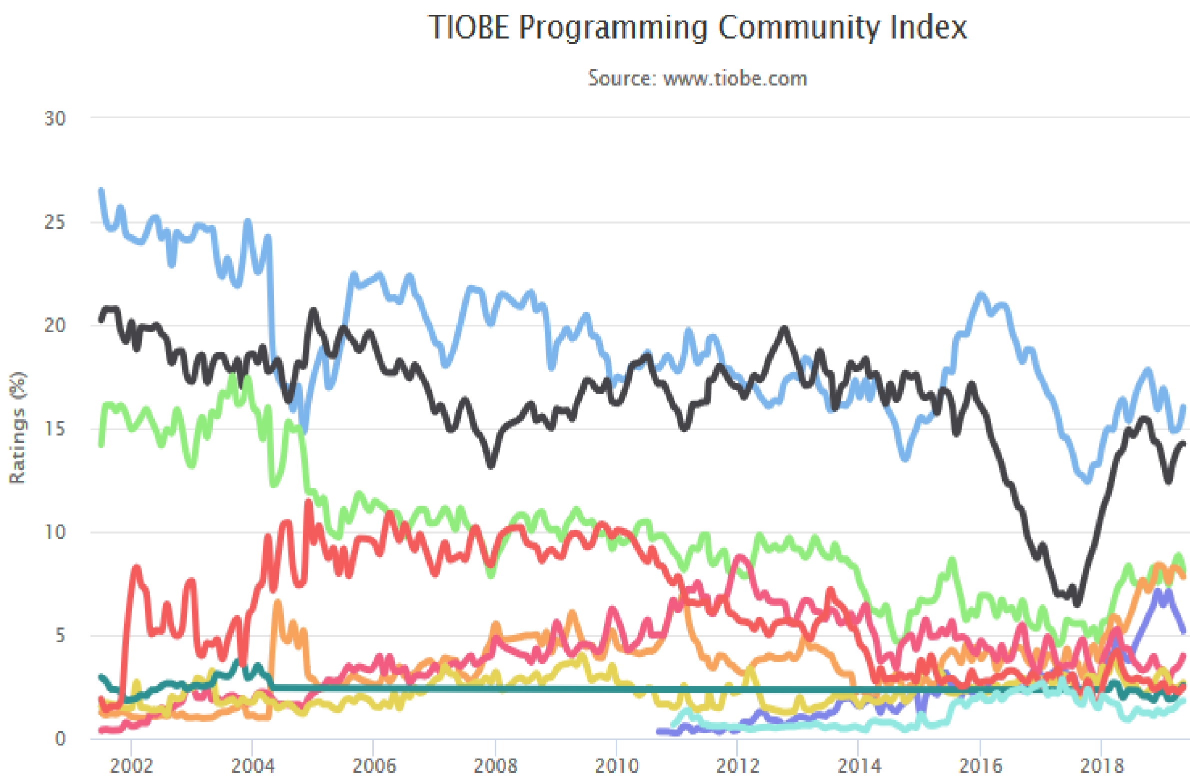
field greeting only accessible in class Demo

CONST will *always* be 43

sayHello will always return a string

sayHello will always be called with 1 parameter of type string

Programming languages



Programming Language	Ratings
Java	16.005%
C	14.243%
C++	8.095%
Python	7.830%
Visual Basic .NET	5.193%
C#	3.984%
JavaScript	2.690%
SQL	2.555%
PHP	2.489%
Assembly language	1.816%
Objective-C	1.626%
Delphi/Object Pascal	1.406%
Perl	1.394%

– **Challenge: Type-safe and memory-safe languages do not dominate**

The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. Observe that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written.

Formal methods in development

- **Static analysis** used in Fortran 1957 for optimisation
- Type checking, variable initialisation/bounds, control flow (conditions, cases, bracketing, termination)
- Specification, proof of properties, construction/verification
- **Labour-intensive** approach limits adoption
  - Smart cards, cryptographic protocols, space probes, small OS kernels
- Promising: **automation**
  - Static analysis tools to **repeatably** cover **large parts** of code base
  - Test cases and tool comparison: https://samate.nist.gov

Why automation, tool support?

- Manual inspections
  - Results depend on reviewer skills
  - Limited (LOC/time, labour costs)
- Automated inspections
  - Increase throughput
  - Increase coverage
  - Repeatable results
  - Increase efficiency
- Often automated and manual inspections combined

Product size and complexity

Firefox
22,000,000 LOC

Windows Server 2003
50,000,000 LOC

Debian 7.0
419,000,000 LOC

Manual inspection almost impossible ➔ Static code analysis can help identifying potential problems

Results of automated analysis

	Vulnerability exists	No vulnerability exists
Vulnerability reported	Vulnerability detected, can be dealt with ➔ success *True positive (TP)*	Vulnerability wrongly detected, needs to be verified ➔ increased effort *False positive (FP)*
Vulnerability not reported	Vulnerability not detected, product vulnerable ➔ increased risk *False negative (FN)*	No problem, no report, no effort ➔ good *True negative (TN)*

Static analysis

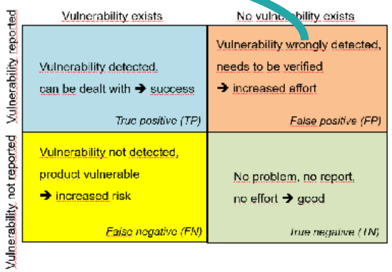
- Detection of errors in a program without executing it
  - No overhead in execution
  - Often **automated**
  - Focus on **implementation vulnerabilities**
    - Beneficial for software quality in general
    - Avoiding vulnerabilities is one aspect
  - Often detects **more (and different) errors than manual** review
- http://www.cerias.purdue.edu/news_and_events/events/security_seminar/details/index/tbk895g0ob5tfbi3056e30q164

Static analysis

- Analysis at compile-time
  - Syntactic checks (often caught by compiler)
  - Type checking (often done by compiler)
  - Program semantics
    - Unused variables, unreachable code, missing initialisation
    - Data flow analysis, control flow analysis, abstract interpretation, program verification, model checking, ...
  - In addition to code inspection, testing
    - Examine unusual execution paths

Static analysis

- **False positives**
  - Static analysis tool reports non-error
  - **Low rate** of false positives **important**, critical for acceptance
  - Big issue with existing code



- False negatives
  - Tool does not report error
  - Static analysis does not cover all classes of vulnerabilities, even if you buy all the combined tools on the market

Classes of program errors

- Data
  - Are all variables initialized before use?
  - Have all constants been named?
  - Array lower bounds: 0, 1, k?
  - Array upper bounds: size? size-1?
  - Can a value be controlled by adversarial input?

Find the errors

```
public static void main(String[] Args) {
    int x;
    double sintable[] = new double[90];
    System.out.println(x);
    for (int angle=1; angle <= 90; angle++) {
    sintable[angle] =
        Math.sin(3.14*(double)angle/180.0);
    }
}
```

Detectable by static analysis

```
public static void main(String[] Args) {
    int x;
    double sintable[] = new double[90];
    System.out.println(x);
    for (int angle=1; angle <= 90; angle++) {
    sintable[angle] =
        Math.sin(3.14*(double)angle/180.0);
    }
}
```

Classes of program errors

- Control flow
  - Correct conditions for conditional statements?
  - Termination of loops?
  - Correct bracketing?
  - All cases accounted for in switch statements?

Find the errors

```
public static void main(String[] args) {
   int i;
   ControlFaults a = new ControlFaults(Integer.parseInt(args[0]));
   if (a.x < 1) System.out.println("Negative");
   for (i=0; i > a.x; i++)
      System.out.println(i);
      System.out.println(i+1);
   switch (a.sign) {
      case -1: System.out.println("Negative."); break;
      case 1: System.out.println("Positive."); break;
   }
}
```

Detectable by static analysis

```
public static void main(String[] args) {
   int i;
   ControlFaults a = new ControlFaults(Integer.parseInt(args[0]));
   if (a.x < 1) System.out.println("Negative");
   for (i=0; i > a.x; i++)
      System.out.println(i);
      System.out.println(i+1);
   switch (a.sign) {
      case -1: System.out.println("Negative."); break;
      case 1: System.out.println("Positive."); break;
   }
}
```

Hard to detect by static analysis

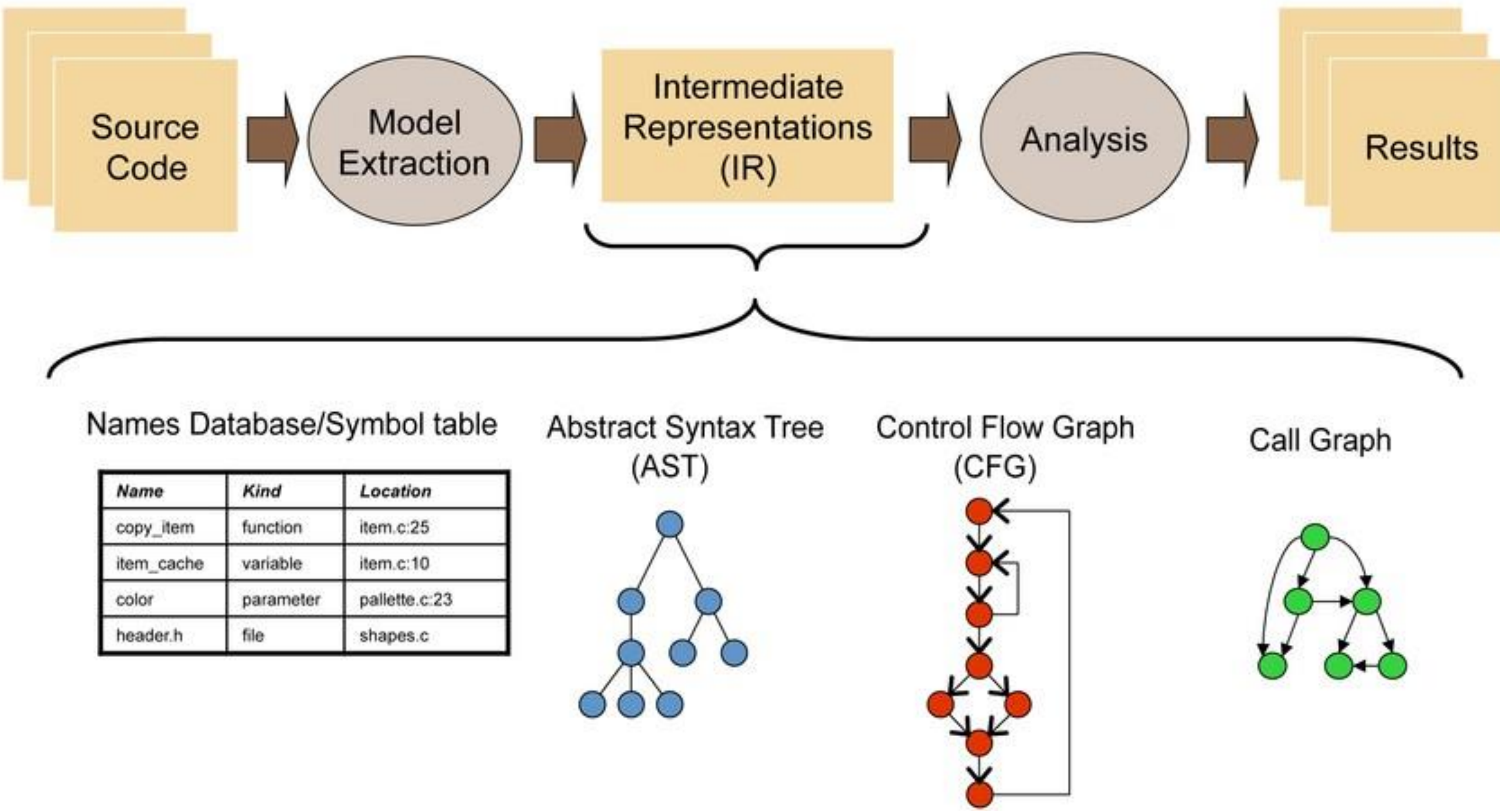
```
public static void main(String[] args) {
   int i;
   ControlFaults a = new ControlFaults(Integer.parseInt(args[0]));
   if (a.x < 1) System.out.println("Negative");
   for (i=0; i > a.x; i++) {
      System.out.println(i);
      System.out.println(i+1); }
   switch (a.sign) {
      case -1: System.out.println("Negative."); break;
      case 1: System.out.println("Positive."); break;
   }
}
```

Approaches to static analysis

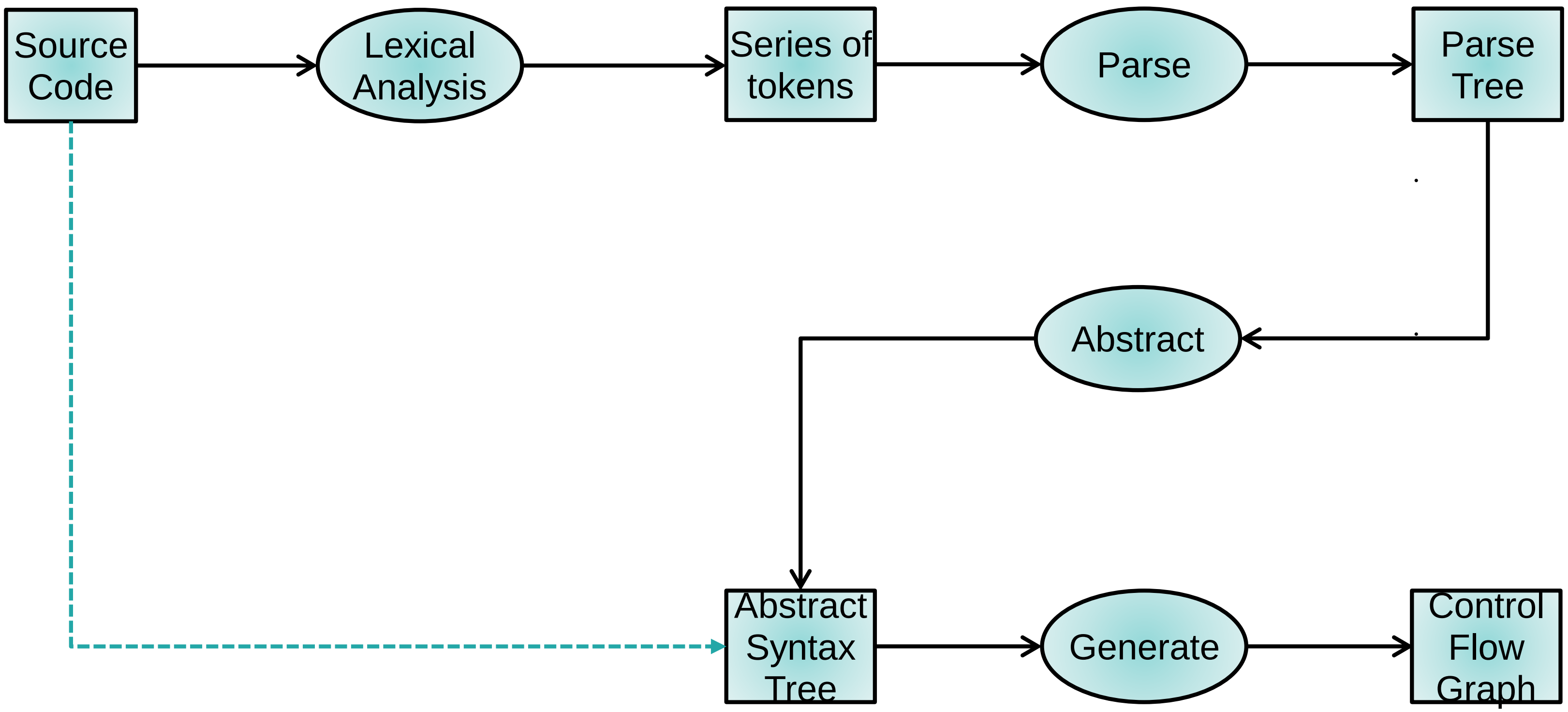
Approaches

- Syntactical analysis (find strings in source code)
- Lexical analysis [regular expressions]
  - Match sequence of tokens against pattern
- Data-flow analysis [finite state machines]
  - Determine possible values for variables
- Abstract interpretation
  - Model execution on (simpler) abstract machine
  - Define abstract security requirements in advance, e.g., access to resources

Intermediate representations



Model extraction



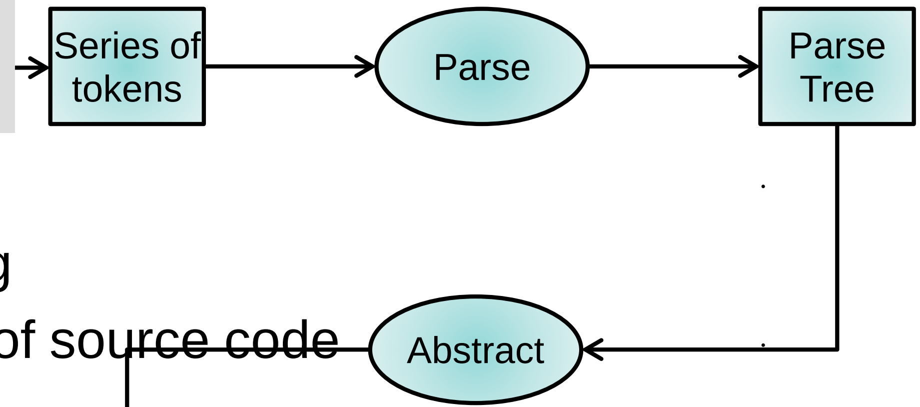
Model extraction

– Lexical analysis:
  – Discard unimportant information (white space, comments etc.)
  – Often regular expressions

```
if ( $DEBUG ) // comment
{
        $i = 10;
        echo( $_GET[‘id’] );
        ...
}
```

```
IF LPAREN ID(DEBUG) RPAREN
LCURLY  ID(i) EQUAL
LITERAL_INT(10) SEMI
ID(echo) LPAREN ID($_GET)
LBRACKET CHAR_LITERAL(id)
RBRACKET SEMI … RCURLY
```

```
if ( $DEBUG ) // comment
{
        $i = 10;
        echo( $_GET['id'] );
        ...
}
```

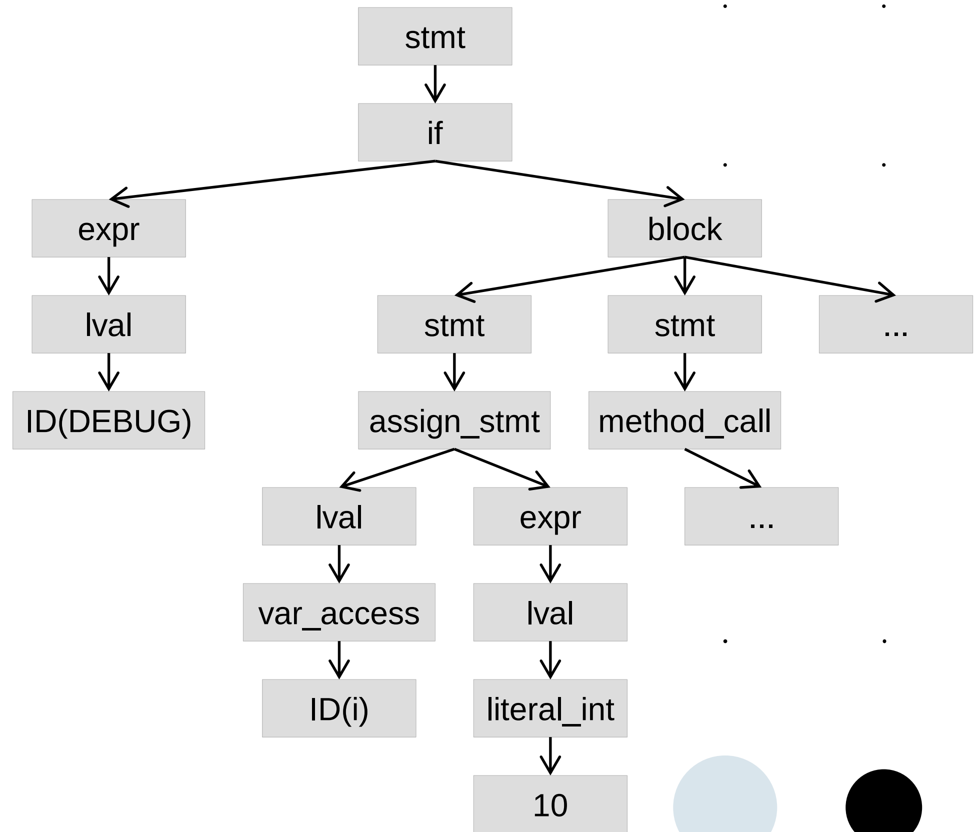


- Parsing:
  - Transform token string
  - Direct representation of source code
  - Not useful for analysis

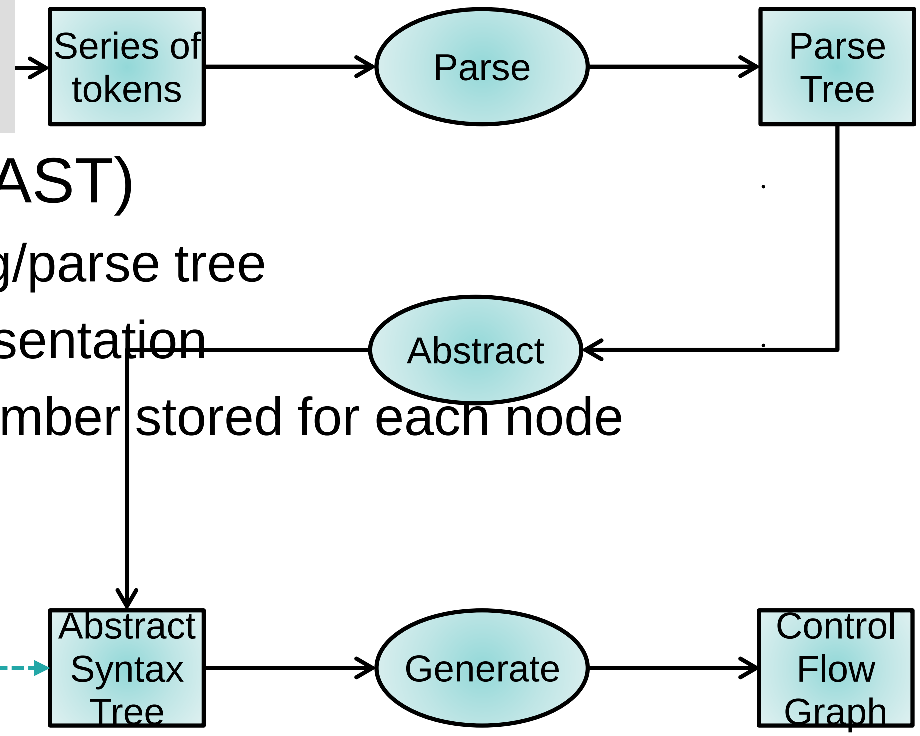
```
stmt := if_stmt | assign_stmt | method_call
assign_stmt := lval EQUAL expr SEMI
expr : = lval
lval := ID | literal_int
...
```

```
if ( $DEBUG ) // comment
{
        $i = 10;
        echo( $_GET[‘id’] );
        ...
}
```

– Parse tree



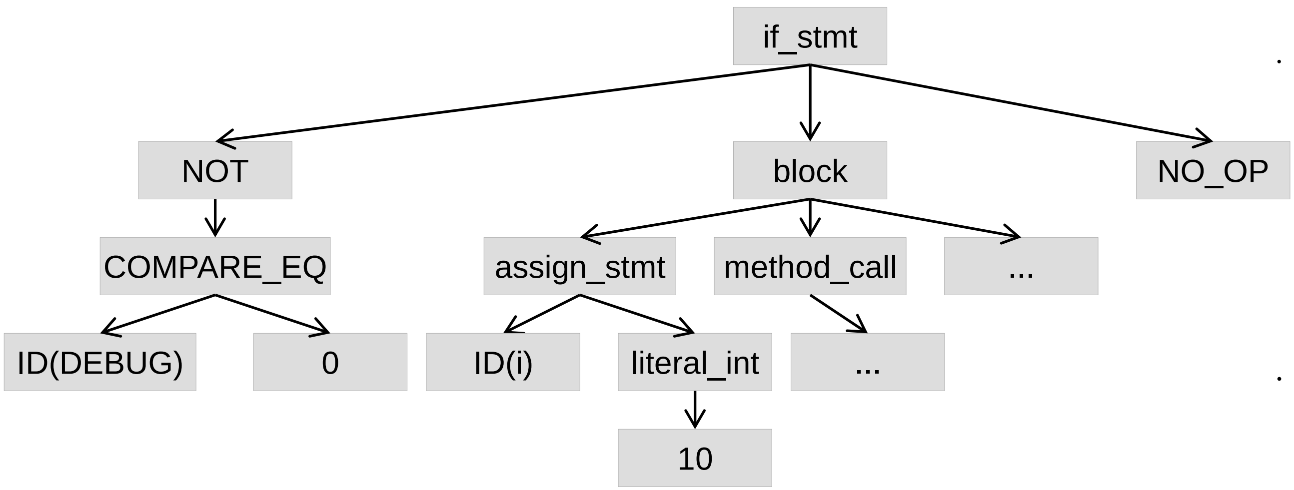
```
if ( $DEBUG ) // comment
{
        $i = 10;
        echo( $_GET[‘id’] );
        ...
}
```



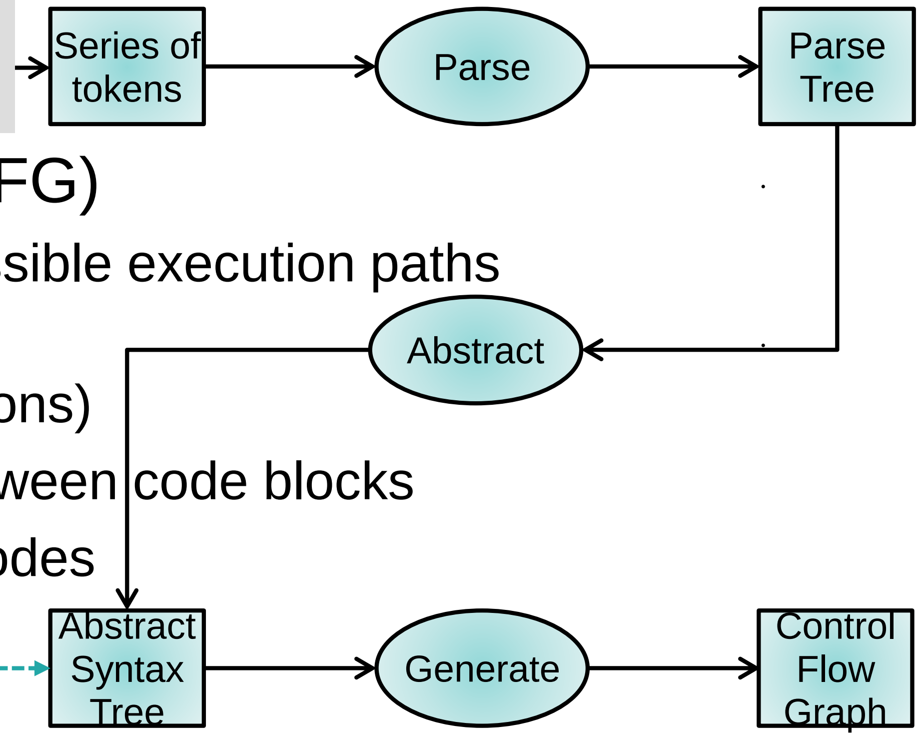
- Abstract syntax tree (AST)
  - Transform token string/parse tree
  - Unified abstract representation
  - Often file path, line number stored for each node

```
if ( $DEBUG ) // comment
{
        $i = 10;
        echo( $_GET['id'] );
        ...
}
```

– Abstract syntax tree (AST)



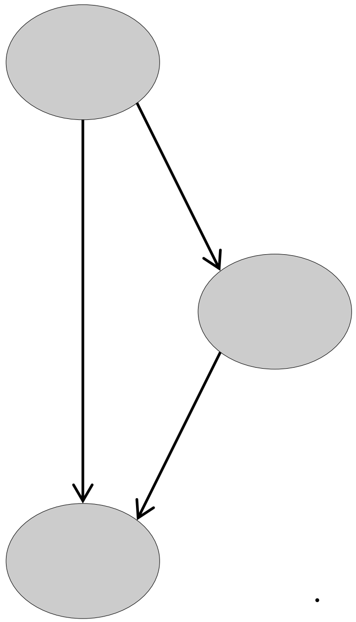
```
if ( $DEBUG ) // comment
{
        $i = 10;
        echo( $_GET['id'] );
        ...
}
```



- Control flow graph (CFG)
  - Representation of possible execution paths
  - Nodes: code blocks (sequence of instructions)
  - Edges: flow paths between code blocks
  - Trace: sequence of nodes

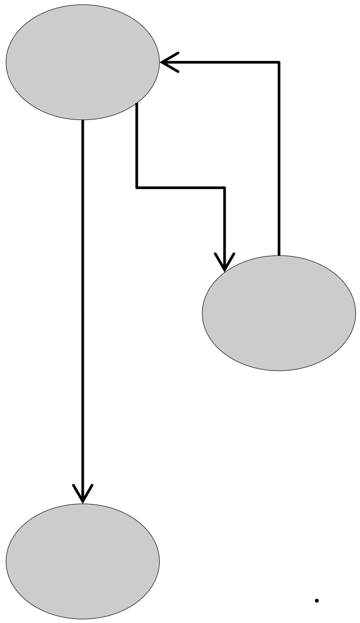
```
if ( $DEBUG ) // comment
{
        $i = 10;
        echo( $_GET['id'] );
        ...
}
```

– Control flow graph (CFG)

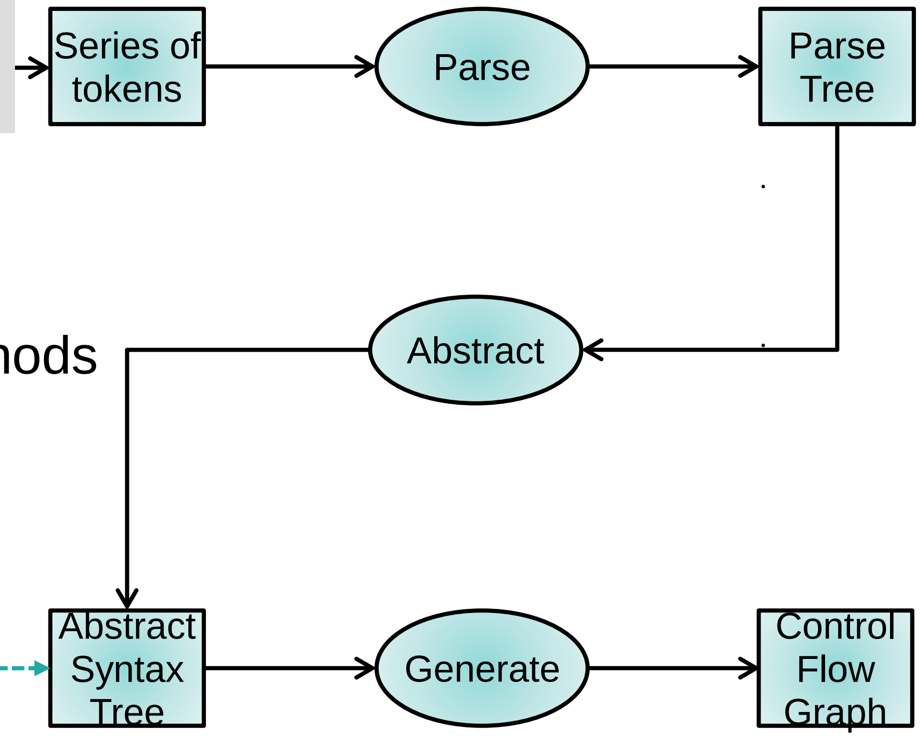


```
while ( $i<10 )
{
        $i = $i+1;
        ...
}
```

- Control flow graph (CFG) with loop



```
if ( $DEBUG ) // comment
{
        $i = 10;
        echo( $_GET[‘id’] );
        ...
}
```



- Call graph
- Special case of CFG
  - Nodes: functions/methods
  - Edges: Invocations of functions/methods (back edges are recursive functions)

Data flow analysis, taint analysis

- Commonly used to detect software vulnerabilities
- Tracking data flow
  - from untrusted input
  - to use of data in security-sensitive decisions/invocations

- Basis: CFG, call graph, AST
- Forward data flow analysis (input source => sink)
- Backward data flow analysis (sink => input source)

Data flow analysis, taint analysis

- "Tainted" data, i.e. controllable by adversary
- Source: function providing tainted data
  e.g. $_GET, std::cin
- Sink: function that should not receive tainted data
- Iterative
  - For each CFG node determine input, output variables
  - Order of CFG paths matters (loops, branches)
  - Is there a path from source to sink?
- Sanitization function: can filter data; after passing sanitization function, data is not tainted

Data flow analysis, taint analysis: example

Source

$a tainted

$b tainted

$c not tainted

```
$a = $_GET['id'];

$b = "..." . $a;

$c = rmvNonAlphabetic($b);

if(...)
    echo($c);
else
    echo($b);
```

Sanitization

$c not tainted
➔ no vulnerability

Sink

$b is tainted
➔ vulnerability

Data flow analysis, taint analysis

- Sources and sinks easy to define
  (basically just lists of functions)
- Sanitization functions might be hard to determine
  - Some widespread functions detected automatically
  - Manual marking by developers
    (potential for mistakes)
  - Sufficient to mitigate vulnerability?
  - Sanitization functions for one vulnerability might not be suitable for another vulnerability
    (e.g. XSS vs. SQLi)

Tools for static analysis

Static analysis tools

- Compiler warnings
- Test cases and tool comparison https://samate.nist.gov
- Long (and incomplete) lists of existing tools
  - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
  - http://ruthmalan.com/ArchitectureResourcesLinks/VisualizationInSoftware.htm#visualizing_and_managing_code_structure/dependencies
  - http://docs.codehaus.org/display/SONAR/Plugin+Library/
- Languages: C, C++, Java, .NET, PHP, ...
- Basis: Source code, byte code, binaries

Example: Coverity

- https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html
- **Looks for patterns**
  - Multiple programming languages supported
  - Misuse of APIs, resource handling, concurrency, efficiency, best practices
- (Not limited to security vulnerabilities)
- **No program execution**
- Claims false positive rate **~ 20%** (confirmed by own experimental results @HTWG)

Critical checks

- API usage errors
- Best practice coding errors
- Buffer overflows
- Build system issues
- Class hierarchy inconsistencies
- Code maintainability issues
- Concurrent data access violations
- Control flow issues
- Cross-site request forgery (CSRF)
- Cross-site scripting (XSS)
- Deadlocks
- Error handling issues
- Hard-coded credentials
- Incorrect expression
- Insecure data handling
- Integer handling issues
- Integer overflows
- Memory—corruptions
- Memory—illegal accesses
- Null pointer dereferences
- Path manipulation
- Performance inefficiencies
- Program hangs
- Race conditions
- Resource leaks
- Rule violations
- Security best practices violations
- Security misconfigurations
- SQL injection
- Uninitialized members

Example: Coverity

```
public PairOfSameType<HRegion> stepsBeforePONR(final Server server,
    final RegionServerServices services, boolean testing) throws IOException {
  // Set ephemeral SPLITTING znode up in zk.  Mocked servers sometimes don't
  // have zookeeper so don't do zk stuff if server or zookeeper is null
```

1. Condition `server != null`, taking true branch

2. **alloc_fn:** A new resource is returned from allocation method `getZooKeeper`. (The virtual call resolves to `org.apache.hadoop.hbase.master.cleaner.TestHFileCleaner.DummyServer.getZooKeeper`.)

3. Condition `server.getZooKeeper() != null`, taking true branch

CID 1090416 (#1 of 2): Resource leak (RESOURCE_LEAK)

4. **leaked_resource:** Failing to save or close resource created by `server.getZooKeeper()` leaks it.

```
  if (server != null && server.getZooKeeper() != null) {
    try {
      createNodeSplitting(server.getZooKeeper(),
        parent.getRegionInfo(), server.
    } catch (KeeperException e) {
      throw new IOException("Failed cre
        this.parent.getRegionNameAsStri
    }
  }
  this.journal.add(JournalEntry.SET_SPL
```

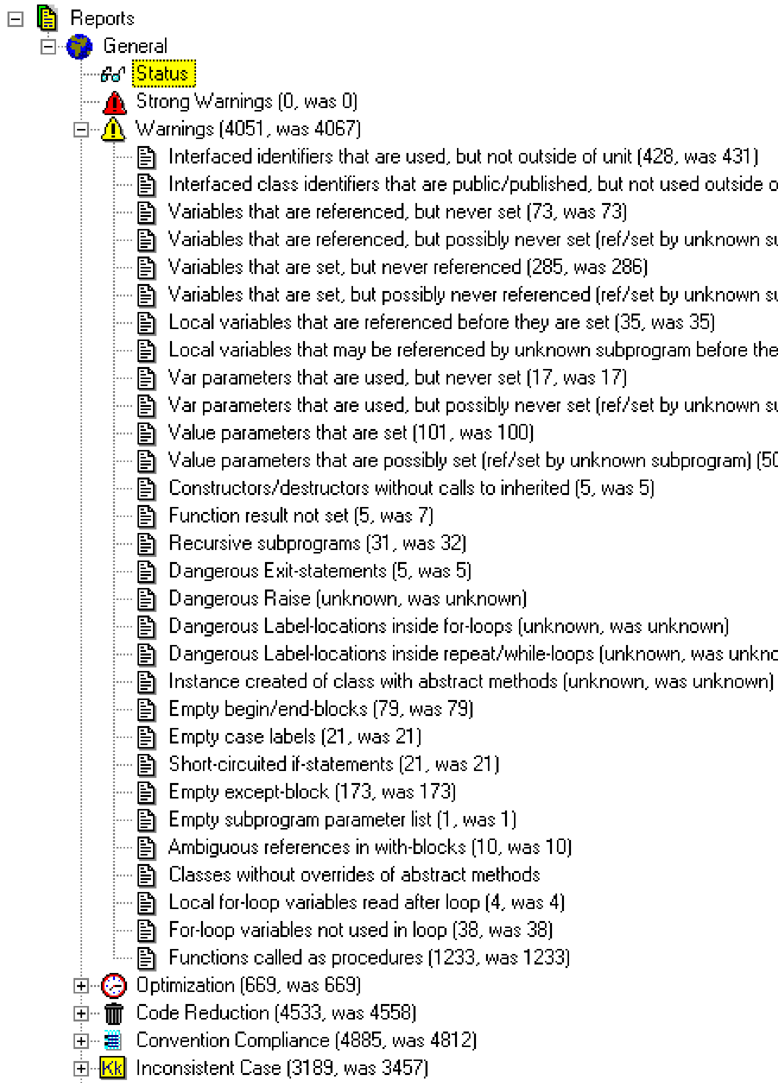
```
@Override
public ZooKeeperWatcher getZooKeeper() {
  try {
    return new ZooKeeperWatcher(getConfiguration(), "dummy server", this);
  } catch (IOException e) {
    e.printStackTrace();
  }
  return null;
}
```

Developer incentive

– Reduce number of warnings

– *"I felt that I really got done something today. I reduced the number of warnings from 600 to 400."*

– (Can be discouraging in existing projects, though)



Outlook: Proof-carrying code

- Avoid expensive review and run-time monitoring
  - Identify dangerous instructions in code
  - Have **certifying** compiler **prove** that instructions are always used in a **safe** way
  - Verify proof before execution
    - If successful, **execute** program **without constraints**

- **Hard to implement**
  - Coverage, scalability, performance, trust relationships
  - Still an active line of research, not used in production
  - Similar concept: managed app stores

Summary

- Trust and trustworthiness of supply chain elements
  - Internal/external staff, suppliers, alignment of incentives
  - **Dependencies**
- Software inspections
  - Familiarity with code, **hotspots**, focus on patterns
  - **Flaw hypotheses** for vulnerabilities
  - **Effort**, costs
- **Automation**
  - Static analysis
  - **Capabilities** and **limitations**