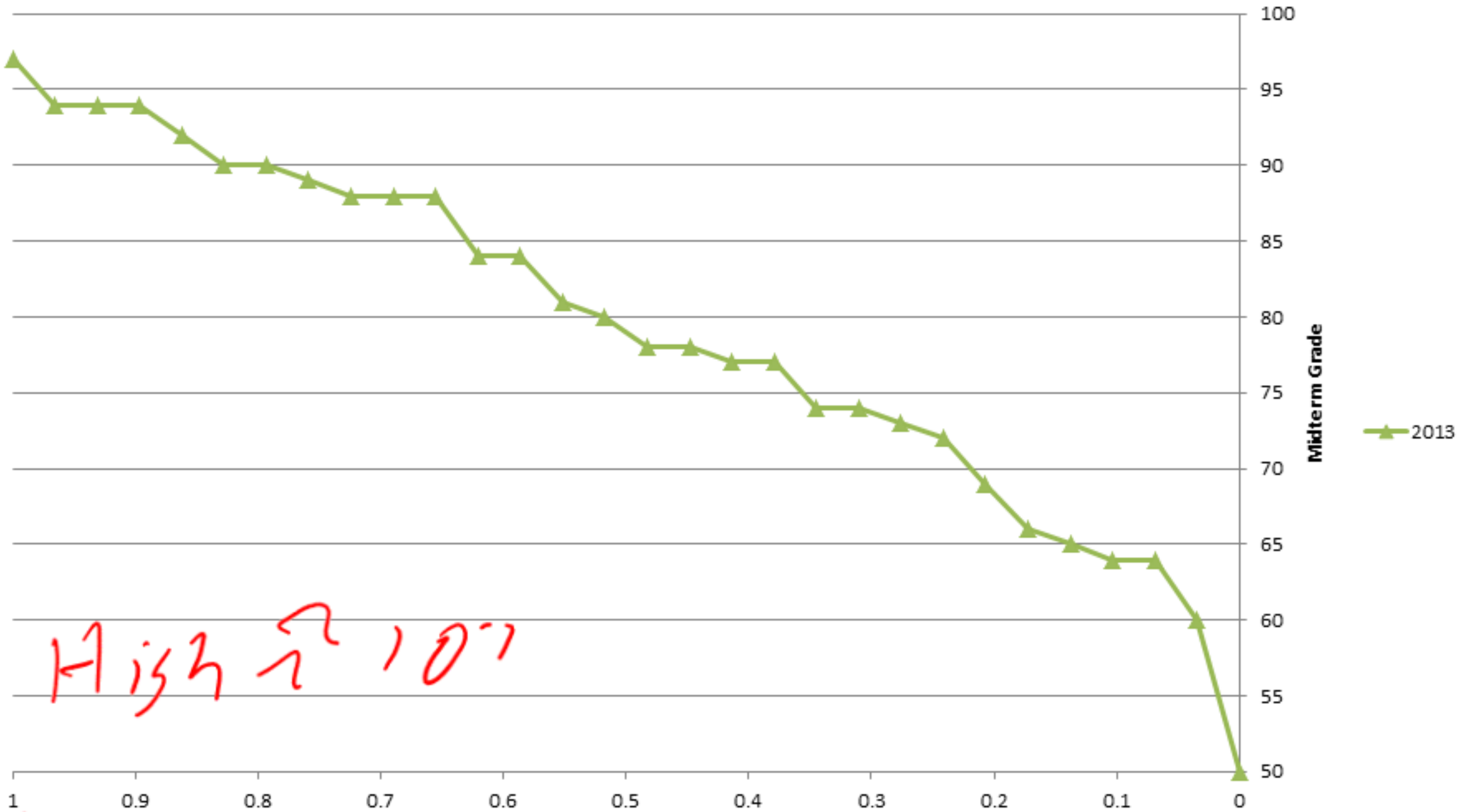
SE4831 Midterm Exam Distributions

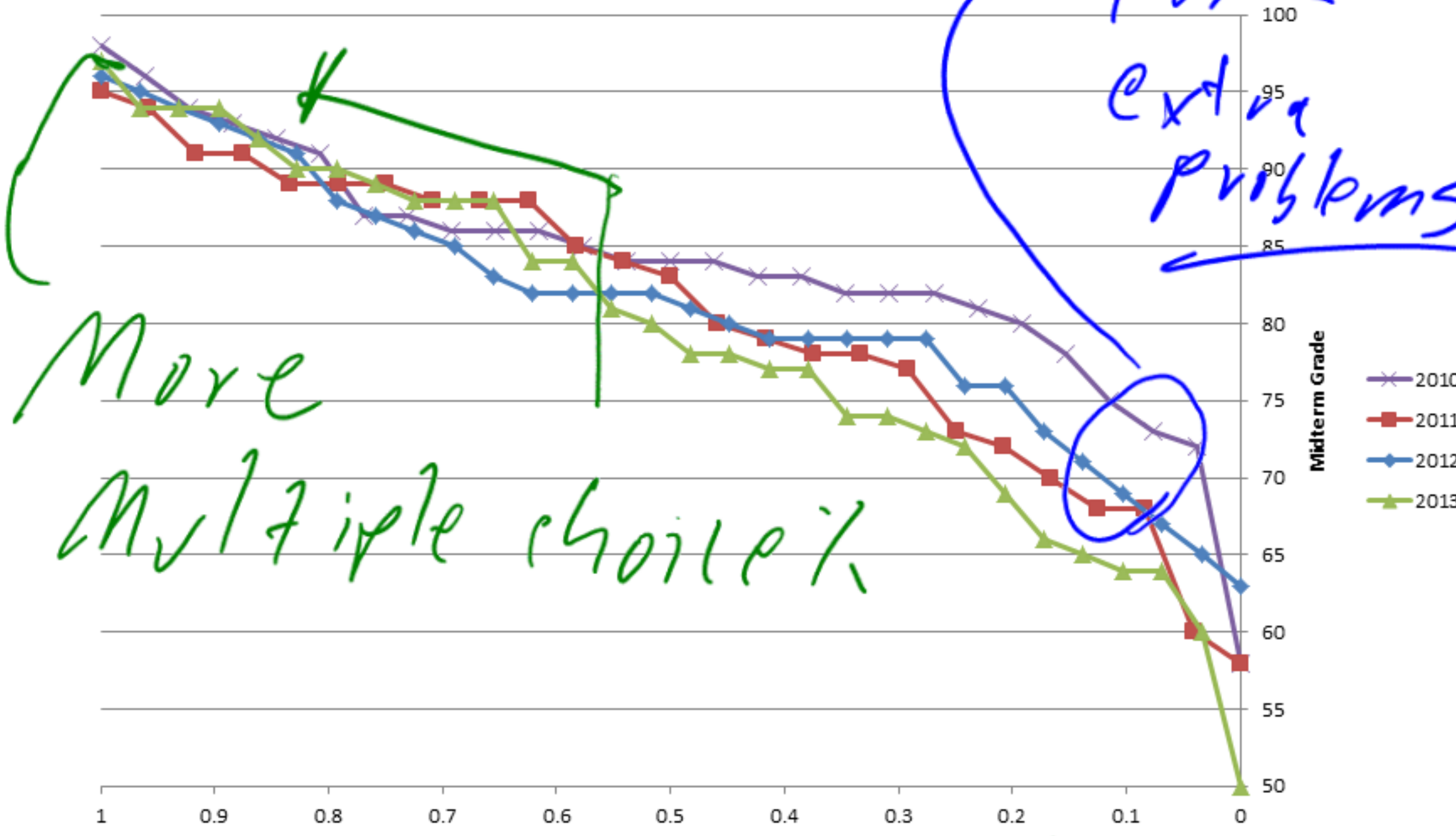


High ~ 10%

~ 80%

79%	Average	81%	88%	87%	92%	92%	65%	58%
79%	Median	84%	92%	94%	100%	100%	55%	100%
12%	STD	11%	14%	21%	14%	20%	28%	48%
97%	Max	100%	100%	100%	100%	100%	100%	100%
50%	Min	56%	58%	0%	50%	0%	15%	0%

SE4831 Midterm Exam Distributions



79%	Average	81%	88%	87%	92%	92%	65%	58%
79%	Median	84%	92%	94%	100%	100%	55%	100%
12%	STD	11%	14%	21%	14%	20%	28%	48%
97%	Max	100%	100%	100%	100%	100%	100%	100%
50%	Min	56%	58%	0%	50%	0%	15%	0%

SE4831 : Software Quality Assurance

QA Tools -> Static Analysis Tools

Dr. Walter W. Schilling, Jr.
Instructor

Objectives

- Understand the difference between static analysis and testing
- Define the halting problem
- Explain the difference between a false positive and a false negative
- Construct a primitive static analysis tool using grep
- Describe the impact of using static analysis tools over time
- Compare and contrast style guides and programming standards
- Explain the steps necessary to integrate static analysis into a development process
  - New code
  - Legacy code

What is testing?

- What do you do when you test a program?

See if it compile ?

Run unit tests

⇒ Executing a set of paths.

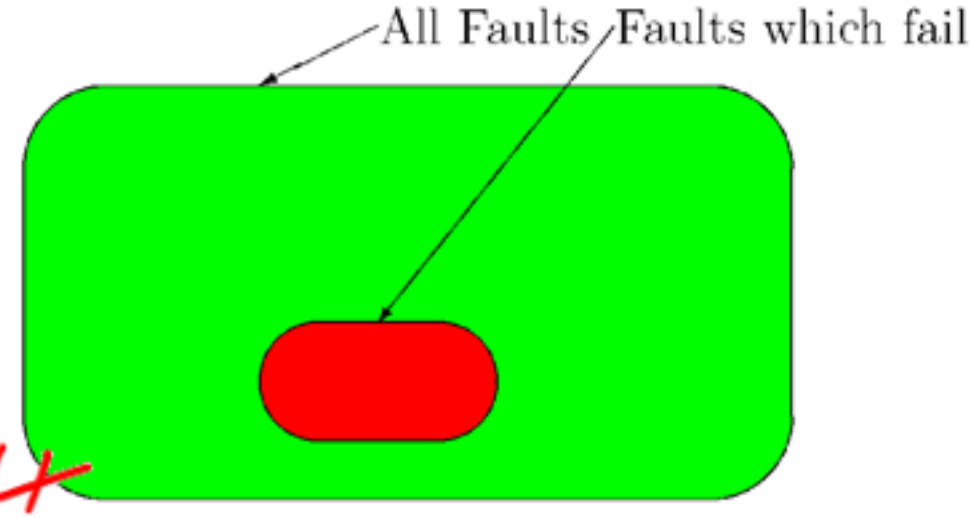
Introduction

- Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation [IEEE]
  - Does not involve the execution of the program
  - Software inspections are a form of static analysis

- "even well tested code written by experts contains a surprising number of obvious bugs" [Hovermeyer/Pugh]
- "Java has many language features and APIs which are prone to misuse." [Hovermeyer/Pugh]
- Static analysis tools "can serve an important role in raising the awareness of developers about subtle correctness issues. . . . prevent future bugs" [Hovermeyer/Pugh]

Static Analysis Overview

- Similar to a spell checker or grammar checker.
- Search through code to detect bug patterns
  - error prone coding practices that arise from the use of erroneous design patterns, misunderstanding of language semantics, or simple and common mistakes.
- Static Analysis tools detect faults
  - Not all faults will fail
    - 90% of downtime comes from 10% of the faults



- Can detect many different classifications of software faults
  - Coding standards violations
  - Buffer overflows (Viega et all)
  - Security vulnerabilities (Livshits and Lam)
  - Memory leaks (Rai)
  - Timing anomalies (race conditions, deadlocks, and livelocks) (Artho)

Static Analysis Overview (2)

- Required to claim compliance with MISRA C Standard
- Between 40% and 60% of statically detectable faults will eventually manifest themselves in the field (QA Systems)
- Has been shown to reduce software defects by a factor of six (Xiao and Pham)
- Can remove upwards of 91% of errors (R. Glass)
- Have been shown to have a 92% ROI_{time} (Schilling)
- NEW: Required by the state of New York for contracted SW

=> For financial Reasons

Static Analysis Overview (3)

- Used in safety critical software development
  - Avionics
  - Automotive
  - Rail Transit
- Security Analysis
  - ≈50% of security flaws can be caught with static analysis (McGraw)
- Shown to be cost effective for code reviews.

Static Analysis Overview (4)

- Impossible to prove a software program correct in the general case
  - Manifestation of the Halting Problem.

- Most static analysis tools are unsound and incomplete.

So we will have false negatives.

So we may have false positives.

Halting Problem

Define a procedure halts? that takes a procedure and an input evaluates to #t if the procedure would terminate on that input, and to #f if would not terminate.

(define (halts? procedure input) …)

Static Analysis Problems

- Static analysis tools aim for good, not perfect
  - False Positive rate can be very high
  - Greater than 50% for certain tools

- Tools can be influenced by programmer style.
  - Tools may need to be "tuned" based on constructs used

Static Analysis Classifications

- General Purpose Tools
  - General purpose Static Analysis tools are those geared for general developmental usage
  - Lint, QAC, Polyspace C, JLint, Findbugs
- Security Tools
  - Static Analysis tools targeting security issues within source code
  - RATS (Rought Auditing Tool for Security), SPLint, Flawfinder
- Style Checking Tools
  - Audit software code from a stylistic standpoint ensuring consistant implementation style
  - PMD, Checkstyle
- Teaching Tools
  - Developed to help students develop better software
  - Safer C Toolkit, Gauntlet

What is wrong with this code?
(Note: This is C)

```
 1: void example_routine( uint32_t x, uint32_t ok, BOOL wrong )
 2: {
 3:   if ( x = 1 )
 4:   {
 5:     printf( "X has a value of 1.\n" );
 6:   }
 7:
 8:   if ( ok == 2 )
 9:   {
10:     printf( "OK has a value of 2.\n" );
11:   }
12:
13:   if ( wrong = FALSE )
14:   {
15:     printf( "You are correct!\n" );
16:   }
17:
18:   /* if (commented=TRUE) Even though this code is commented out,
19:      the error is still shown. */
20: }
```

Assignment not eval.

Simple “Home Made” Static Analysis Tool using Grep

```
 1: void example_routine( uint32_t x, uint32_t ok, BOOL wrong )
 2: {
 3:   if ( x = 1 )
 4:   {
 5:     printf( "X has a value of 1.\n" );
 6:   }
 7:
 8:   if ( ok == 2 )
 9:   {
10:     printf( "OK has a value of 2.\n" );
11:   }
12:
13:   if ( wrong = FALSE )
14:   {
15:     printf( "You are correct!\n" );
16:   }
17:
18:   /* if (commented=TRUE) Even though this code is commented out,
19:      the error is still shown. */
20: }
```

```
$ grep "if ([[:space:]]*[[:alnum:]]*[[:space:]]*=[[:space:]]*
[[:alnum:]]" error_files.c
```

Simple "Home Made" Static Analysis Tool using Grep

```
 1: void example_routine( uint32_t x, uint32_t ok, BOOL wrong )
 2: {
 3:   if ( x = 1 )
 4:   {
 5:     printf( "X has a value of 1.\n" );
 6:   }
 7:
 8:   if ( ok == 2 )
 9:   {
10:     printf( "OK has a value of 2.\n" );
11:   }
12:
13:   if ( wrong = FALSE )
14:   {
15:     printf( "You are correct!\n" );
16:   }
17:
18:   /* if (commented=TRUE) Even though this code is commented out,
19:      the error is still shown. */
20: }
```

```
$ grep "if ([[:space:]]*[[:alnum:]]*[[:space:]]*=[[:space:]]*
[[:alnum:]]" error_files.c
  if ( x = 1 )
  if ( wrong = FALSE )
  /* if (commented=TRUE) Even though this code is commented out,
[wws@localhost wws]$
```

Lint

- One of the oldest and readily available static analysis tools
  - Developed initially by Bell Labs
    - C Language
    - UNIX development
    - Now available for Dos, Windows, Linux, OS/2
  - Commercial version available from Gimpel Software
    - Supports value tracking, MISRA C standard compliance verification, and Effective C++ Standards
    - Analyzes C and C++ code
    - ALOA metrics tool is available to collect quality metrics from Lint tool.
    - XML Output readily available

Sample Buffer Overflow Failure Source Code (C Language)

```
1:  typedef unsigned short uint16_t;
2:  void update_average(uint16_t current_value);
3:
4:  #define NUMBER_OF_VALUES_TO_AVERAGE (11u)
5:
6:  static uint16_t data_values[NUMBER_OF_VALUES_TO_AVERAGE];
7:  static uint16_t average = 0u;
8:
9:  void update_average(uint16_t current_value)
10: {
11:   static uint16_t array_offset = 0u;
12:   static uint16_t data_sums = 0u;
13:
14:   array_offset = ((array_offset++) % NUMBER_OF_VALUES_TO_AVERAGE);
15:   data_sums -= data_values[array_offset];
16:   data_sums += current_value;
17:   average = (data_sums / NUMBER_OF_VALUES_TO_AVERAGE);
18:   data_values[array_offset] = current_value;
19: }
```

Sample Buffer Overflow Failure Source Code (C Language)

```
1:  typedef unsigned short uint16_t;
2:  void update_average(uint16_t current_value);
3:
4:  #define NUMBER_OF_VALUES_TO_AVERAGE  (11u)
5:
6:  static uint16_t
        data_values[NUMBER_OF_VALUES_TO_AVERAGE];
7:  static uint16_t  average = 0u;
8:
9:  void update_average(uint16_t current_value)
10: {
11:   static uint16_t array_offset = 0u;
12:   static uint16_t  data_sums = 0u;
13:
14:   array_offset = ((array_offset++) %
        NUMBER_OF_VALUES_TO_AVERAGE);
15:   data_sums -= data_values[array_offset];
16:   data_sums += current_value;
17:   average = (data_sums /
        NUMBER_OF_VALUES_TO_AVERAGE);
18:   data_values[array_offset] = current_value;
19: }
```

Sample Buffer Overflow Failure
Lint Output

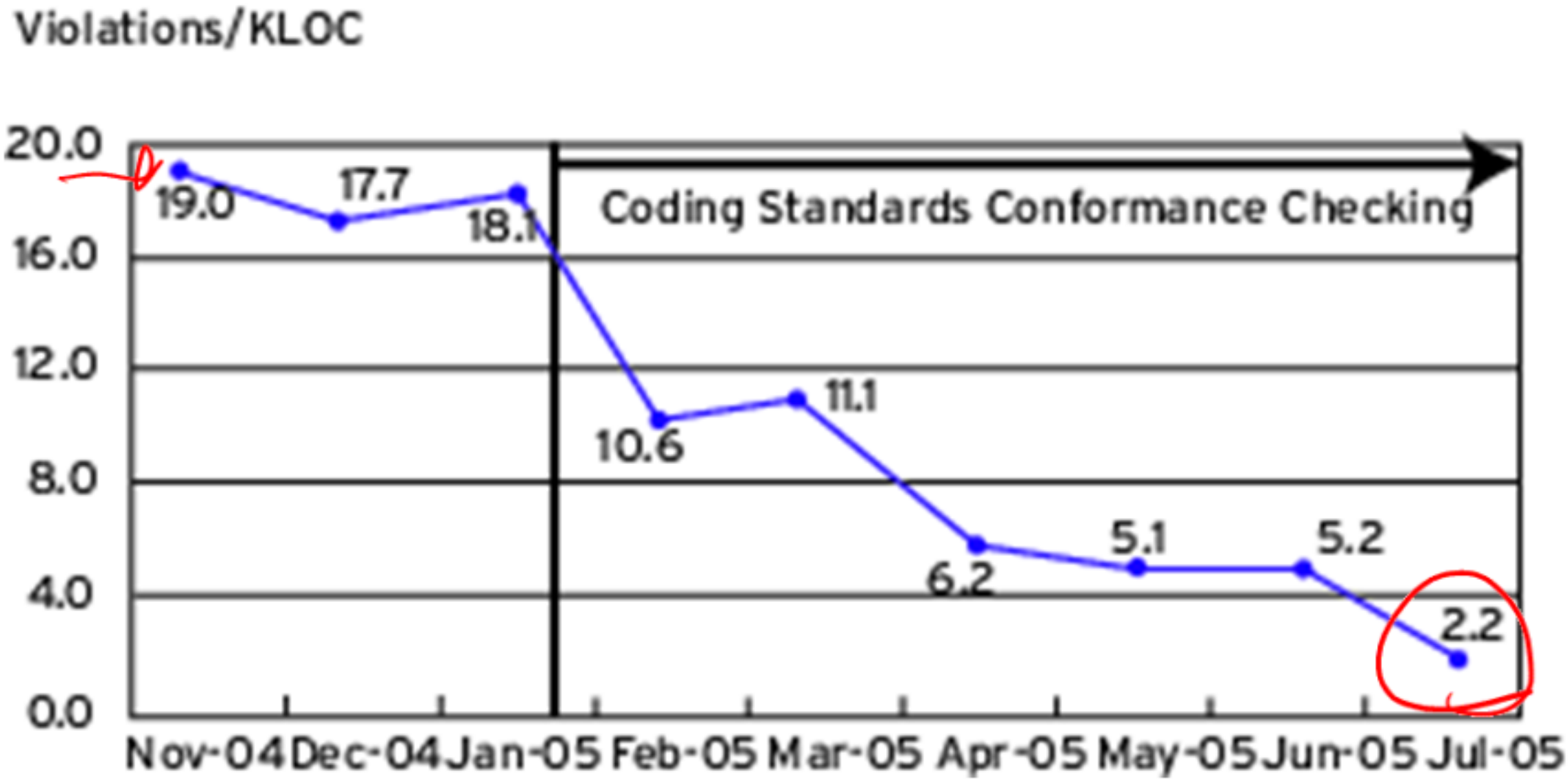
```
--- Module: buffer_overflow.c
array_offset = ((array_offset++) % NUMBER_OF_VALUES_TO_AVERAGE);

"*** \index{LINT}LINT: buffer_overflow.c(14) Warning 564:
   variable 'array_offset' depends on order of evaluation
   [\index{MISRA C}MISRA Rule 46]"
```

- Fault manifesting itself as a failure depends upon the compiler's handling of source code!
  - Some compilers may handle code properly.
  - Other compilers may cause failure to occur.
  - Compiler options may effect behavior.
    - Especially true of optimization flags

Impact of SA over Time

(Dr. Dobbs, *June 16, 2006* **Code Quality Improvement**
)



Polyspace C Verifier

- Developed as a result of the Ariane 5 launch failure
  - Initially developed as an Ada tool
  - Has been extended to analyze C, and C++ programs
  - Uses Abstract Interpretation technique
- Suffers from scalability issues > old reference
  - Only can analyze code 20 to 40 KLOC chunks (Venet and Brat)
  - 145 KLOC Sendmail failed after 4 days (Zitser et all)
- Detected 87% of buffer overflows seeded in an open source program (Zitser et all)
  - 50% false alarm rate.
  - Unable to discriminate between faulty code and patched code.

C Global Surveyor

- Developed by NASA Jet Propulsion Laboratory
  - Used to analyze
    - Mars Pathfinder software (135 KLOC)
    - Deep Space One Mission (280 KLOC)
  - Operated only in Linux
  - Supports Distributed Processing
  - Can analyze any C code
    - Currently being extended to handle C++.
    - Algorithms tuned for Mars programs

JLint

- Developed for NASA Ames Research Center
- Supports static analysis of Java programs
- Consists of 2 C programs
  - AntiC Syntax verifier
    - Checks for Java syntax problems due to C language heritage
  - JLInt Java analyzer
    - Looks for deadlocks, livelocks, race conditions, and other problems
  - Suffers from a high false positive rate.
- Has been applied to several large scale NASA projects and other programs.
- Source code is freely available.

Java Pathfinder

- Software program developed by the Robust Software Engineering group at NASA Ames Research Center
  - Available as an Open Source program on Sourceforge
  - Analyzes Java Bytecode for deadlocks, assertion violations, and other problems with temporal behavior
  - Uses a custom Java Virtual Machine to analyze source code
  - Allows customized extensions to the tool to be developed by software engineers

KlocWork K7

- Static analysis tool available from KlocWork
- Supports the analysis of C, C++ and Java
- K7 provides excellent coverage of a range of quality issues that can be found by static analysis.
  - Low False positive rate
- Provides
  - Defects & Vulnerabilities
  - Architecture & Header File Anomalies
  - Software Metrics
- Supports many different development IDES
  - Eclipsem Microsoft Visual Studio, IBM Rational Application Developerm Wind River Workbench, Gvim, Emacs, Visual SlickEdit, Platform Builder, KDevelop, Freescale CodeWarrior
- Flexible Reporting & Defect Management
  - Has built in report generation which can be customized to meet needs of developers

SofCheck Inspector

- Relatively new tool on the market

- Analyzes Java code
  - Future developments will support Ada, C#, C, and C++ code
- Operated by
  - Automatically creates assertions (preconditions and postconditions) characterizing each module.
  - Attempts to prove that the entire software system obeys all assertions.
  - Attempts to prove the absence of runtime errors, such as buffer overflows, which are responsible for many Internet security breaches.
- Performance
  - Averages about 1000 lines per minute, more or less depending on CPU speed, amount of RAM, and complexity of code.

Adding SA to Development Process

- 1. Develop a coding standard and style guides
  - Style guide is not a necessity to use SA effectively
  - There may be multiple style guides
- 2. Automate compliance checking with the standard
- 3. Add SA Compliance checking to review process

Style Guides

- Provides stylistic guidance for developing source-code modules.
- Items to define include:
  - Copyright notices
  - requisite commenting
  - Indentation
  - naming conventions
  - Any other stylistic items
- Can raise significant debate amongst software engineers
- Can be automated by providing templates to automatically format code in conformance with the style guide
  - Eclipse, JEdit, CodeWright all support style templates.

JavaDoc

Coding Standard

- Defines which coding constructs can and can not be used in a project.
  - Should predominantly be enforceable through static analysis methods
  - Should include general best practices as well as past experiences within the domain

- Example rule:
  - *"All variables shall be assigned a value before being used in any operation"*
  - Statically detectable
  - Can be easily understood by a programmer.

- Defined deviation procedure
  - With every coding standard, there will be a need for an occasional deviation.
  - All deviations should be reviewed in a formal setting (peer review, formal review, walkthrough, etc.)

- Standards Exist to use as a baseline
  - MISRA C
  - High Integrity C++

MSOE SDL Material

- Tabs should be used as the correct method of indentation. Tabs should be 4 spaces in width.
- Avoid lines longer than 80 characters as these cause problems on smaller displays and terminals.
- Lines that must wrap should be broken only at the following points:
  - After a comma
  - Before an operator
  - Prefer higher-level breaks over lower-level breaks
  - Align the new line with the beginning of the expression at the same level on the previous line.
  - If all else fails, use an indent of 8 spaces.

MSOE SDL Material (Part 2)

- Two blank lines should be added between:
  - Sections of a source file.
  - Class and interface definitions.
- One blank line should be added between:
  - Methods.
  - Local variables in a method and its first statement.
  - Before a block or single-line comment.
  - Between logical sections inside a method to improve readability.
- A blank space should be added between:
  - A keyword (such as if, while, for) and its opening parentheses. (Not method names!)
  - Comma-separated arguments in a list
  - All binary operators (except ".") ; Unary operators should never be separated.
  - The expressions in a "for" statement, including the for-each version.
  - A typecast and the variable name it affects.

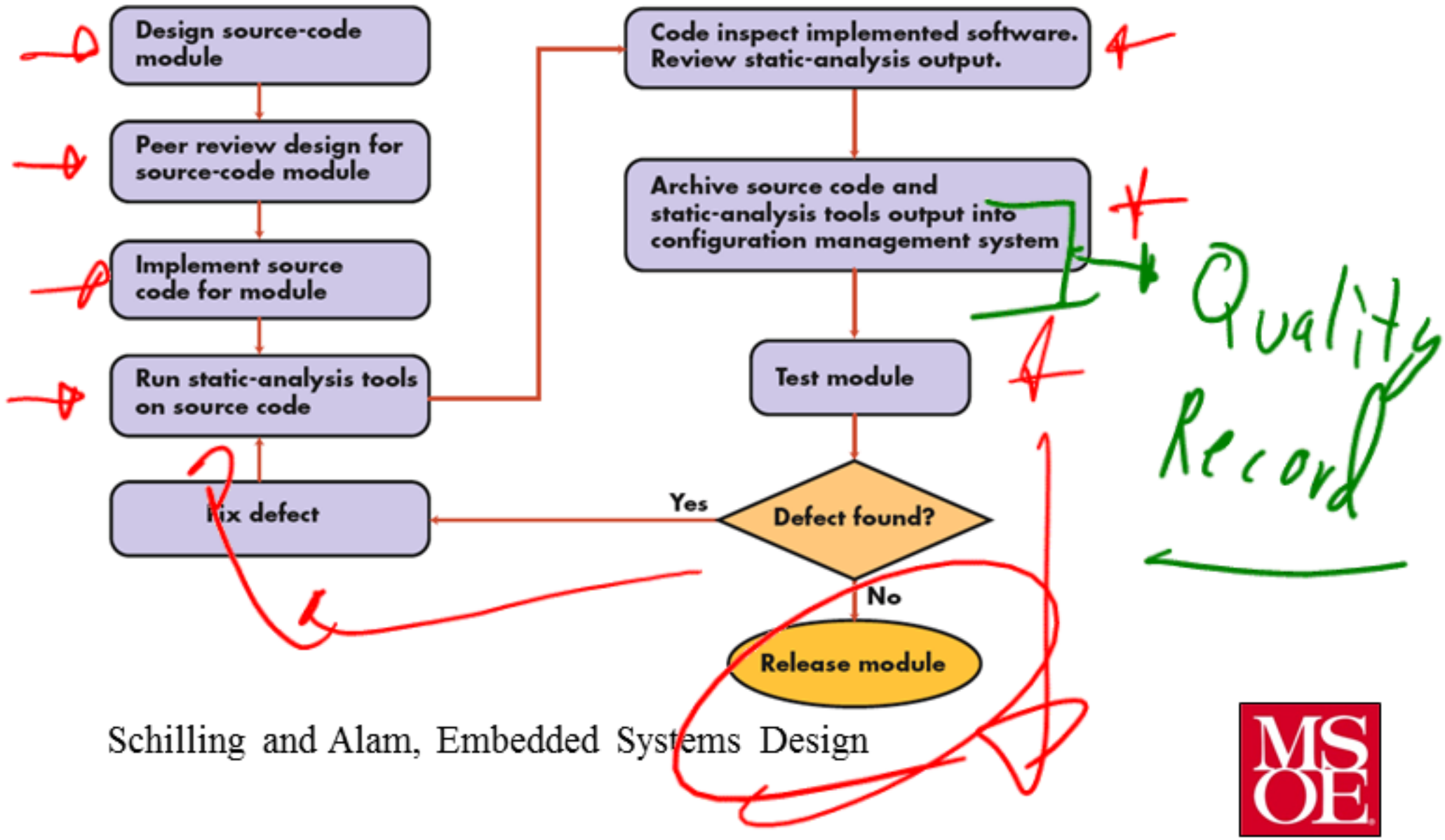
MISRA Coding Standard

- **Comments**
  - Comments shall not be nested
- **Functions**
  - Functions with a variable number of arguments shall not be used
  - Functions shall not call themselves directly or indirectly) Don't do recursion.
  - Functions shall always have prototype declarations and the prototype shall be visible at both the function definition and call
  - For each function parameter the type given and definition shall be identical, and the return type shall be identical

```
/* ~~~~~~~~~~~~~~~~
~~~~~~~~~~~~ */
/*
doit();
//        */
```

SW Development process incorporating Static Analysis

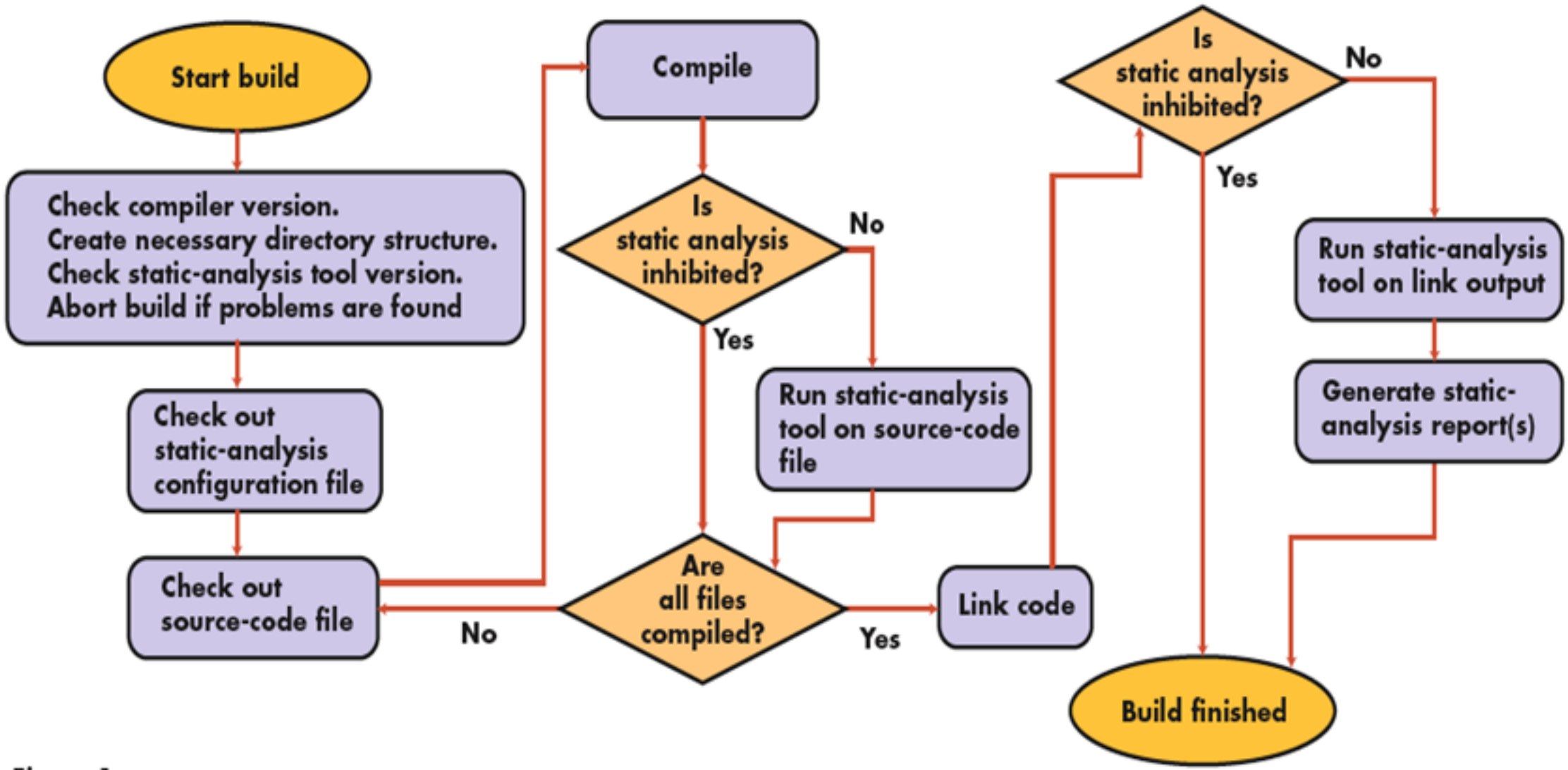
This software-development process segment incorporates static analysis



Schilling and Alam, Embedded Systems Design

Automate Coding Standard Compliance

Sample flowchart of an automated build script incorporating static analysis



Schilling and Alam, Embedded Systems Design

Legacy Code Integration

- Applying to existing code base can be challenging
- Success depends upon
  - age of the code
  - engineer's programming style
  - paradigms used
  - Diligence of engineers applying static analysis

- Many projects have abandoned SA when the first run of the tool generates 100,000 or more warnings.
- With legacy code, it's often not practical to remove all statically detectable faults.

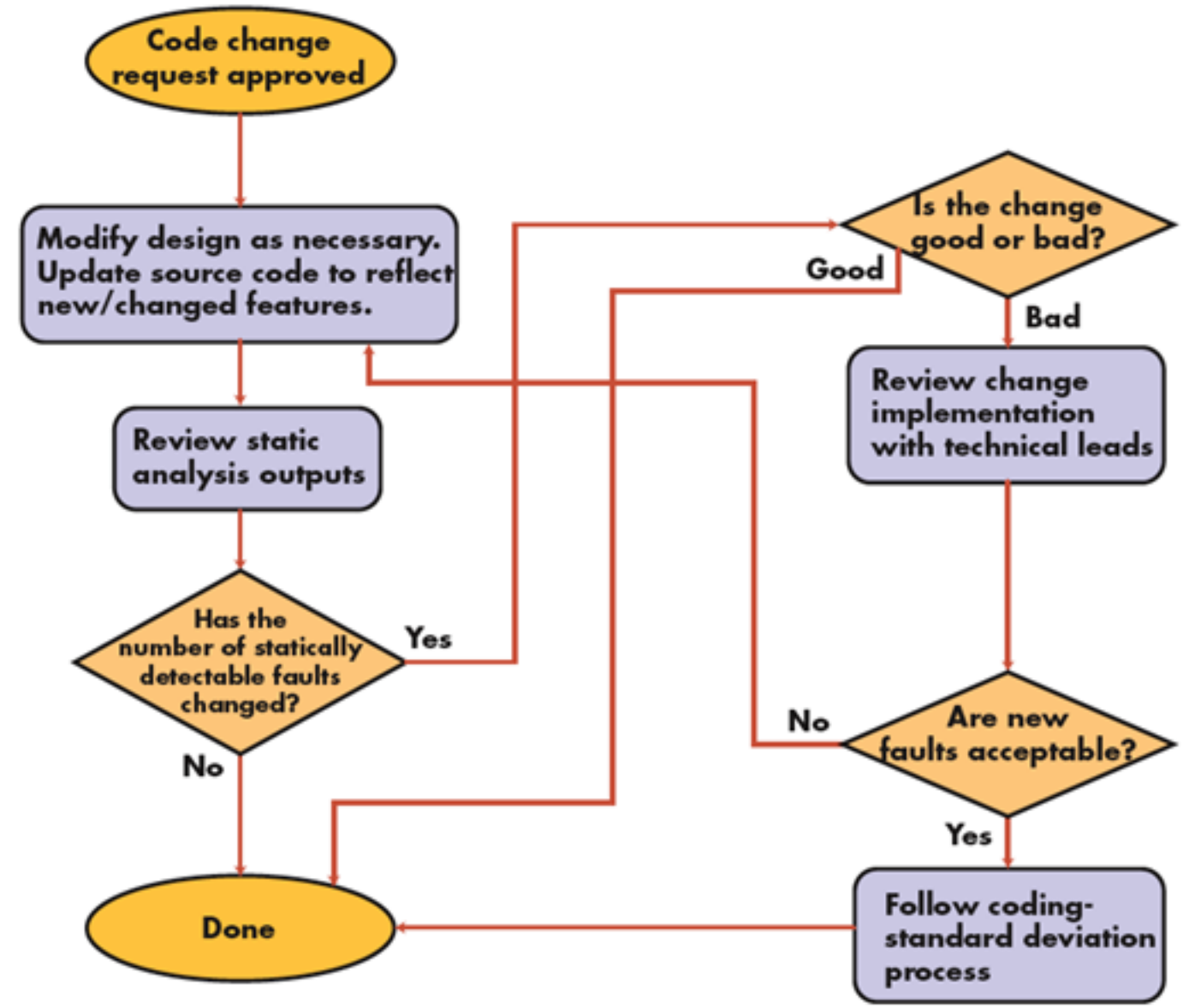
Legacy Software Methodology

- Treat each statically detectable fault as a bug fix.
  - Each time a fault is removed, there's the possibility of injecting a more serious fault into the module.
  - The worst thing would be to attempt to repair a false-positive that was statically detected as a fault and inject a failure.
  - This must be done diligently, as each Statically Detectable fault could be a catastronphic failure in the making
    - Ariane V

- With legacy code, the most important information to track isn't necessarily the presence of statically detectable faults, but the change in the number of faults as revisions are made to that code.
- Coding standard development follows the same behavior as that for traditional coding standard development.

Legacy Software Flowchart

Conceptual flow for analysis of legacy software



Schilling and Alam, Embedded Systems Design

Resources

- Integrate static analysis into a software development process
  - http://www.embedded.com/shared/printableArticle.jhtml?articleID=193500830
- NIST SAMATE - Software Assurance Metrics And Tool Evaluation Project
  - http://samate.nist.gov/index.php/Main_Page
- Static Source Code Analysis Tools for C
  - http://www.spinroot.com/static/