Static Analysis for Software Quality

Jonathan Aldrich

Associate Professor

Carnegie Mellon University

MSE / NIST Seminar
June 7, 2011

Find the Bug!

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        save_flags(flags);
        cli();                          <- disable interrupts
        if ((bh = sh->buffer_pool) == NULL)
                return NULL;
        sh->buffer_pool = bh->b_next;
        bh->b_size = b_size;
        restore_flags(flags);           <- re-enable interrupts
        return bh;
}
```

Find the Bug!

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
                int b_size) {
      struct buffer_head *bh;
      unsigned long flags;

      save_flags(flags);
      cli();
      if ((bh = sh->buffer_pool) == NULL)
            return NULL;
      sh->buffer_pool = bh->b_next;
      bh->b_size = b_size;
      restore_flags(flags);
      return bh;
}
```

disable interrupts

ERROR: returning with interrupts disabled

re-enable interrupts

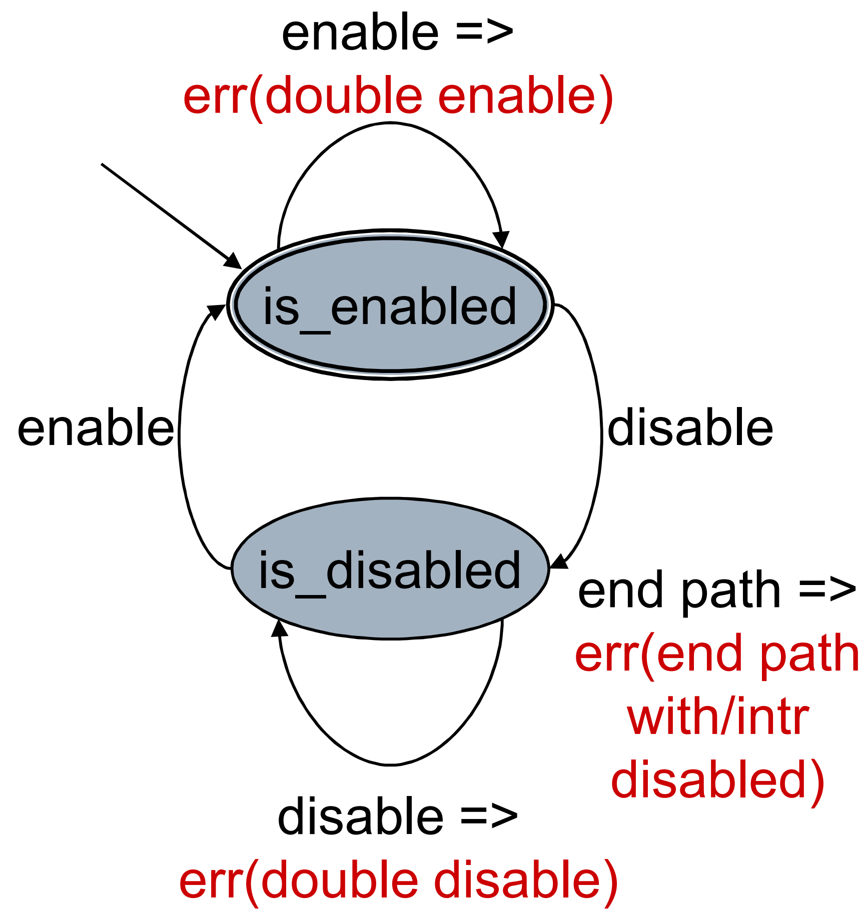
Metal Interrupt Analysis

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
{ #include "linux-includes.h" }
sm check_interrupts {
  // Variables
  // used in patterns
  decl { unsigned } flags;

  // Patterns
  // to specify enable/disable functions.
  pat enable = { sti(); }
             | { restore_flags(flags); } ;
  pat disable = { cli(); };

  // States
  // The first state is the initial state.
  is_enabled: disable ==> is_disabled
     | enable ==> { err("double enable"); }
     ;
  is_disabled: enable ==> is_enabled
     | disable ==> { err("double disable"); }
     // Special pattern that matches when the SM
     // hits the end of any path in this state.
     | $end_of_path$ ==>
        { err("exiting w/intr disabled!"); }
     ;
}
```



Applying the Analysis

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,       <- initial state is_enabled
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        save_flags(flags);
        cli();                                 <- transition to is_disabled
        if ((bh = sh->buffer_pool) == NULL)
                return NULL;                   <- final state is_disabled: ERROR!
        sh->buffer_pool = bh->b_next;
        bh->b_size = b_size;
        restore_flags(flags);                  <- transition to is_enabled
        return bh;                             <- final state is_enabled is OK
}
```

Session Objectives

After this session, attendees will be able to:

- Understand the benefits of analysis and how it complements techniques like testing or inspection.
- Grasp the basics of static analysis technology.
- Know some analysis tools that are available, and properties of others that are on the horizon
- Evaluate current and future commercial analysis tools for use in their organization
- Develop a plan for introducing analysis into their organization

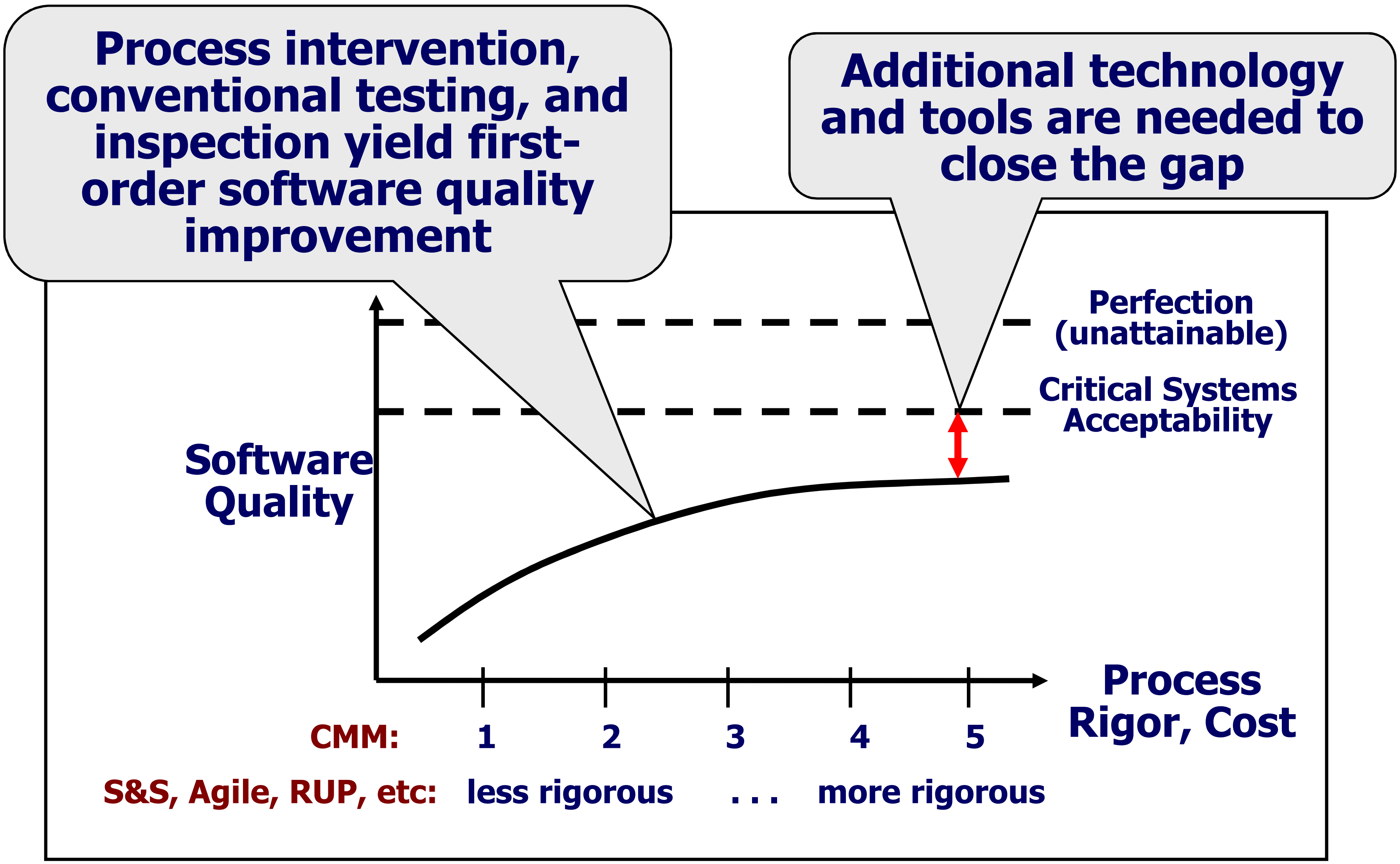
Outline

- Why static analysis?
  - The limits of testing and inspection
- What is static analysis?
- What are current tools like?
- What does the future hold?
- What tools are available?
- How does it fit into my organization?

Software Errors

Process, Cost, and Quality

Slide: William Scherlis



Existing Approaches

- Testing: *is the answer right?*
  - Verifies features work
  - Finds algorithmic problems
- Inspection: *is the quality there?*
  - Missing requirements
  - Design problems
  - Style issues
  - Application logic
- Limitations
  - Non-local interactions
  - Uncommon paths
  - Non-determinism
- Static analysis: *will I get an answer?*
  - Verifies non-local consistency
  - Checks all paths
  - Considers all non-deterministic choices

Errors Static Analysis can Find

- Security vulnerabilities
  - Buffer overruns, unvalidated input…
- Memory errors
  - Null dereference, uninitialized data…
- Resource leaks
  - Memory, OS resources…
- Violations of API or framework rules
  - e.g. Windows device drivers; real time libraries; GUI frameworks
- Exceptions
  - Arithmetic/library/user-defined
- Encapsulation violations
- Race conditions

Theme: consistently following rules throughout code

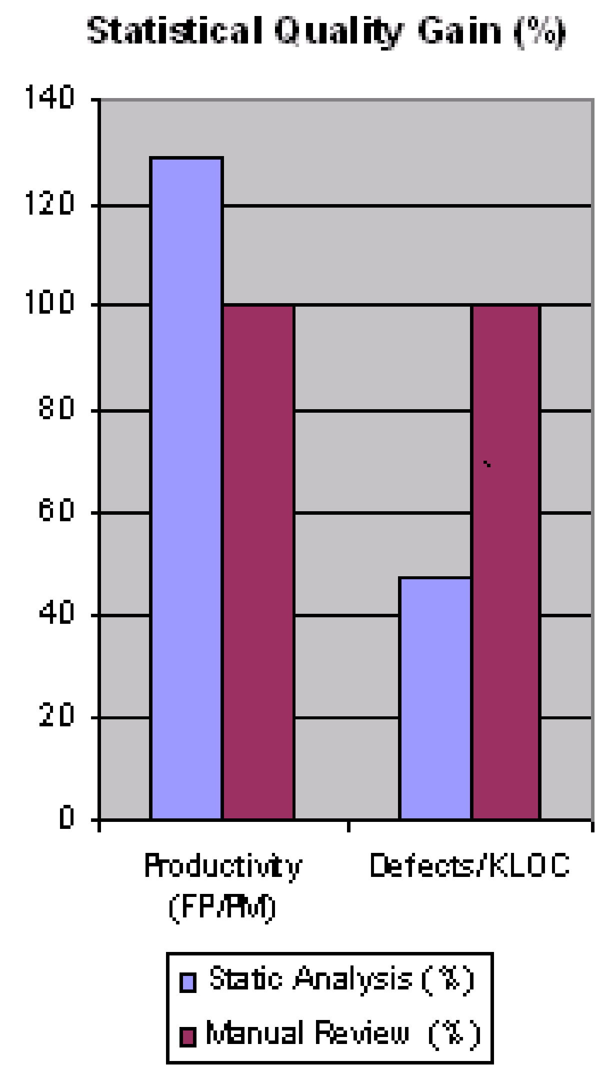
Empirical Results on Static Analysis

- Nortel study [Zheng et al. 2006]
  - 3 C/C++ projects
  - 3 million LOC total
  - Early generation static analysis tools
- Conclusions
  - Cost per fault of static analysis 61-72% compared to inspections
  - Effectively finds assignment, checking faults
  - Can be used to find potential security vulnerabilities

Empirical Results on Static Analysis

- InfoSys study [Chaturvedi 2005]
  - 5 projects
  - Average 700 function points each
  - Compare inspection with and without static analysis

- Conclusions
  - Fewer defects
  - Higher productivity



Adapted from [Chaturvedi 2005]

Quality Assurance at Microsoft (Part 1)

- Original process: manual code inspection
  - Effective when system and team are small
  - Too many paths to consider as system grew
- Early 1990s: add massive system and unit testing
  - Tests took weeks to run
    - Diversity of platforms and configurations
    - Sheer volume of tests
  - Inefficient detection of common patterns, security holes
    - Non-local, intermittent, uncommon path bugs
  - Was treading water in Longhorn/Vista release of Windows
    - Release still pending
- Early 2000s: add static analysis
  - More on this later

Outline

- Why static analysis?
- What is static analysis?
  - Abstract state space exploration
- What are current tools like?
- What does the future hold?
- What tools are available?
- How does it fit into my organization?

Static Analysis Definition

- Static program analysis is the systematic examination of an abstraction of a program’s state space

- Metal interrupt analysis
  - Abstraction
    - 2 states: enabled and disabled
      - All program information—variable values, heap contents—is abstracted by these two states, plus the program counter
  - Systematic
    - Examines all paths through a function
      - What about loops? More later…
    - Each path explored for each reachable state
      - Assume interrupts initially enabled (Linux practice)
      - Since the two states abstract all program information, the exploration is exhaustive

How can Analysis Search All Paths?

- Exponential # paths with if statements
- Infinite # paths with loops
- Secret weapon: Abstraction
  - Finite number of (abstract) states
  - If you come to a statement and you've already explored a state for that statement, stop.
    - The analysis depends only on the code and the current state
    - Continuing the analysis from this program point and state would yield the same results you got before
  - If the number of states isn't finite, too bad
    - Your analysis may not terminate

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 1 (before stmt): true/no loop

2: is_enabled

3: is_enabled

6: is_disabled

11: is_disabled

12: is_enabled

no errors

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 2 (before stmt): true/1 loop

2: is_enabled
3: is_enabled
6: is_disabled
7: is_disabled
8: is_enabled
9: is_enabled
11: is_disabled

already been here

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

<u>Path 3 (before stmt): true/2+ loops</u>

2: is_enabled
3: is_enabled
6: is_disabled
7: is_disabled
8: is_enabled
9: is_enabled
6: is_disabled

already been here

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 4 (before stmt): false

2: is_enabled

5: is_enabled

6: is_disabled

already been here

all of state space has been explored

Soundness and Completeness

- Soundness
  - If the analysis says the program is OK, there are no bugs
  - No *false negatives*
- Completeness
  - If the analysis gives a warning, it is real
  - No *false positives*
- Contrast: Testing is complete, but not sound
- No static analysis can be sound, complete, and terminating
  - Perfect static analysis is undecidable on nontrivial programs for even simple attributes
  - Thus, every analysis approximates (using abstraction)
- Many static analyses are useful nevertheless
  - E.g. a sound tool with few false positives in practice

Attribute-Specific Analysis

- Analysis is specific to
  - A quality attribute
    - Race condition
    - Buffer overflow
    - Use after free
  - A strategy for verifying that attribute
    - Protect each shared piece of data with a lock
    - Presburger arithmetic decision procedure for array indexes
    - Only one variable points to each memory location
- Analysis is inappropriate for some attributes
  - Approach to assurance is ad-hoc and follows no clear pattern
  - No known decision procedure for checking an assurance pattern that is followed

Outline

- Why static analysis?
- What is static analysis?
- What are current tools like?
  - Example: FindBugs
- What does the future hold?
- What tools are available?
- How does it fit into my organization?

FindBugs Demonstration

Outline

- Why static analysis?
- What is static analysis?
- What are current tools like?
- What does the future hold?
  - Design intent driven analysis
- What tools are available?
- How does it fit into my organization?

Example: java.util.logging.Logger

[Source: Aaron Greenhouse]

```
public class Logger { ...
  private Filter filter;

  public void setFilter(Filter newFilter) ... {
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }
}
```

Consider `setFilter()` in isolation

Example: java.util.logging.Logger

[Source: Aaron Greenhouse]

```
public class Logger { ...
  private Filter filter;








  public void log(LogRecord record) { ...
    synchronized (this) {
      if (filter != null
          && !filter.isLoggable(record)) return;
    } ...
  } ...
}
```

Consider `log()` in isolation

Example: java.util.logging.Logger

[Source: Aaron Greenhouse]

```
/** ... All methods on Logger are multi-thread safe. */
public class Logger { ...
  private Filter filter;

  /** ...
   * @param newFilter a filter object (may be null)
   */
  public void setFilter(Filter newFilter) ... {
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }

  public void log(LogRecord record) { ...
    synchronized (this) {
      if (filter != null
          && !filter.isLoggable(record)) return;
    } ...
  } ...
}
```

Consider class **`Logger`** in it's entirety!

Example: java.util.logging.Logger

[Source: Aaron Greenhouse]

```
/** ... All methods on Logger are multi-thread safe. */
public class Logger { ...
  private Filter filter;

  /** ...
   * @param newFilter a filter object (may be null)
   */
  public void setFilter(Filter newFilter)…{
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }

  public void log(LogRecord record) { ...
    synchronized (this) {
      if (filter != null
          && !filter.isLoggable(record)) return;
    } ...
  } ...
}
```

1

2

3

Class Logger has a ***race condition***.

Example: java.util.logging.Logger

[Source: Aaron Greenhouse]

```
/** ... All methods on Logger are multi-thread safe. */
public class Logger { ...
  private Filter filter;

   /** ...
    * @param newFilter a filter object (may be null)
    */
  public synchronized void setFilter(Filter newFilter)…{
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }

  public void log(LogRecord record) { ...
    synchronized (this) {
      if (filter != null
          && !filter.isLoggable(record)) return;
    } ...
  } ...
}
```

Correction: synchronize `setFilter()`

Tool Demonstration: JSure

Models are Missing

[Source: Aaron Greenhouse]

- **Programmer design intent is missing**
  - Not explicit in Java, C, C++, etc
    - *What lock protects this object*?
      - “This lock protects that state”
    - *What is the actual extent of shared state of this object*?
      - “This object is ‘part of’ that object”

- **Adoptability**
  - Programmers: “Too difficult to express this stuff.”
  - Annotations in tools like JSure: ***Minimal effort*** — concise expression
    - Capture what programmers are ***already thinking about***
    - No full specification

- **Incrementality**
  - Programmers: “I’m too busy; maybe after the deadline.”
  - Tool design (e.g. JSure): Payoffs early and often
    - Direct programmer utility — ***negative marginal cost***
    - Increments of payoff for increments of effort

- **Tooling benefits of design intent**
  - *Scaleability* because analysis is local
  - *Precision* (few false positives) due to avoiding incorrect assumptions

Reporting Code–Model Consistency

[Source: Aaron Greenhouse]

- Tool analyzes consistency
  - No annotations $\Rightarrow$ no assurance
  - Identify likely model sites

- Three classes of results

Code–model consistency

Code–model inconsistency

Informative — Request for annotation

Design Intent Case Study: Microsoft Standard Annotation Language

[Source: Manuvir Das]

- SAL: A language of contracts between functions
- Preconditions
  - Statements that hold at entry to the callee
  - What does a callee expect from its callers?
- Postconditions
  - Statements that hold at exit from the callee
  - What does a callee promise its callers?
- Usage example:

  a_0 **RT func(a_1 … a_n T par)**
- Buffer sizes, null pointers, memory usage, …

SAL Example

```
wchar_t wcsncpy ( __out_ecount(num) wchar_t *dest,
                  __in_ecount(num) wchar_t *src, size_t num );
```

_in	The function reads from the buffer. The caller provides the buffer and initializes it.
_out	The function writes to the buffer. If used on the return value, the function provides the buffer and initializes it. Otherwise, the caller provides the buffer and the function initializes it.
_bcount(size)	The buffer size is in bytes.
_ecount(size)	The buffer size is in elements.
_opt	This parameter / result can be NULL and must be checked for nullness before a dereference

Recommendations

- If you use Microsoft's tools…
  - Turn on /analyze
  - Annotate all functions that write to buffers
  - Annotate all library functions
  - Annotation other functions as possible

Available as part of Microsoft Visual Studio and Windows SDK

Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
- What are current tools like?
- What does the future hold?
- What tools are available?
- How does it fit into my organization?

Error Taxonomy (incomplete list)

- **Concurrency**
  - race conditions
  - deadlock
  - data protected by locks
  - non-lock concurrency (e.g. AWT)
- **Exceptional conditions**
  - integer over/underflow
  - division by zero
  - unexpected exceptions
  - not handling error cases
  - type conversion errors
- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - illegal free (double free, not allocated)
  - memory leak
  - use uninitialized data
- **Resource/protocol errors**
  - calling functions in incorrect order
  - failure to call initialization function
  - failure to free resources
- **Input validation**
  - command injection
  - cross-site scripting
  - format string
  - tainted data
- **Other security**
  - privilege escalation
  - denial of service
  - dynamic code
  - malicious trigger
  - insecure randomness
  - least privilege violations
- **Design and understanding**
  - dependency analysis
  - heap structure
  - call graph
- **Code quality**
  - metrics
  - unused variables

Microsoft Tools

- Static Driver Verifier (was SLAM)
  - http://www.microsoft.com/whdc/devtools/tools/sdv.mspx
  - Part of Windows Driver Kit
  - Uses model checking to catch misuse of Windows device driver APIs

- PREfast and the Standard Annotation Language
  - Ships with Visual Studio (premium edition) and Windows SDK
    - http://msdn.microsoft.com/en-us/windows/bb980924
  - Standard Annotation Language
    - Lightweight code specifications
    - Buffer size, memory management, return values, tainted data
  - PREfast
    - Symbolically executes paths to find memory errors
    - Lightweight version of PREfix analysis used internally at Microsoft
    - Verifies SAL specifications
  - Blogs on getting started with SAL
    - http://blogs.msdn.com/michael_howard/archive/2006/05/19/602077.aspx
    - http://blogs.msdn.com/michael_howard/archive/2006/05/23/604957.aspx
  - Microsoft docs
    - http://msdn2.microsoft.com/en-us/library/ms182025.aspx
    - http://msdn2.microsoft.com/en-us/library/y8hcsad3.aspx

- If you use Microsoft tools, use these!

FindBugs

- findbugs.sourceforge.net
- Focus: bug finding
- Language: Java
- Open source project
  - Free
  - Large community
  - Easy to adapt and customize
  - Many defect detectors
  - Eclipse plugin support
  - Mostly searches for localized bugs

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - double free
  - memory leak
  - use uninitialized data
- **Input validation**
  - command injection
  - tainted data
- **Concurrency**
  - race conditions
  - deadlock
  - data protected by locks
- **Resource/protocol errors**
  - failure to free resources
- **Exceptional conditions**
  - integer over/underflow
  - not handling error cases
  - type conversion errors
- **Code quality**
  - unused variables

Coverity Prevent/Extend

- www.coverity.com
- Focus: bugs and security
- Languages: C, C++, Java, C#
- OS: Windows, Linux, OS X, NetBSD, FreeBSD, Solaris, HPUX
- Builds on the Metal static analysis research project at Stanford
- Open source analysis project
  - http://scan.coverity.com
- Selling points
  - Low false positive rates
  - Scales to 10 MLOC+
  - Statistical bug finding approach
  - Extensibility with Extend
  - Seamless build integration

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - double free
  - memory leak
  - use uninitialized data
- **Input validation**
  - command injection
  - cross-site scripting
  - format string
  - tainted data
- **Concurrency**
  - race conditions
  - deadlock
- **Resource/protocol errors**
  - calling functions in incorrect order
  - BSTR library usage (Microsoft COM)
  - failure to free resources
- **Exceptional conditions**
  - not handling error cases

GrammaTech CodeSonar

- www.grammatech.com
- Focus: bug finding
- Languages: C, C++
- OS: Windows, Linux, Solaris, OS X
- Company founded by Tim Teitelbaum of Cornell and Tom Reps of U. Wisc. Mad.
- Selling points
  - Strong coverage of C/C++ errors
  - Minimize false negatives
  - Binary analysis support
  - Support for custom checks
  - Easy integration with build
  - CodeSurfer program understanding tool

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, freed)
  - illegal free (double free, not allocated)
  - memory leak
  - use uninitialized data
- **Input validation**
  - format string
  - tainted data
- **Concurrency**
  - race conditions
  - deadlock
- **Exceptional conditions**
  - integer over/underflow
  - not handling error cases
  - division by zero
  - type conversion errors
- **Design and understanding**
  - navigation
  - dependency analysis
  - ASTs, CFGs, pointer analysis
  - heap structure
  - call graph

Klocwork Insight

- www.klocwork.com
- Focus: security and bugs
- Languages: C, C++, Java
- OS: Windows, Linux, Solaris, AIX, OS X
- Selling points
  - Strong focus on both bugs and vulnerabilities
  - Built-in extensibility
  - Enterprise/process support
    - track quality over time
  - Architectural visualization support

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - illegal free (double free, not allocated)
  - memory leak
  - use uninitialized data
- **Input validation**
  - command injection
  - cross-site scripting
  - format string
  - tainted data
- **Concurrency**
  - race conditions
- **Resource/protocol errors**
  - calling functions in incorrect order
- **Exceptional conditions**
  - not handling error cases
- **Other security**
  - insecure randomness
  - least privilege violations
- **Design and understanding**
  - dependency analysis

Fortify 360 Source Code Analyzer

- www.fortify.com
- Focus: security
- Languages: C, C++, .NET family (C#, VB), Java, ColdFusion, TSQL, PLSQL, XML
  - OO support from the beginning
- Windows, Linux, OS X, Solaris, AIX, HP-UX, FreeBSD
- Sponsor of FindBugs, fully integrated FindBugs support
- Selling points
  - Strong focus on security
  - Built-in extensibility
  - Good coverage of security errors

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, freed)
  - double free
  - memory leak
  - use uninitialized data
- **Input validation**
  - command injection
  - cross-site scripting
  - format string
  - tainted data
- **Concurrency**
  - race conditions
  - deadlock
- **Resource/protocol errors**
  - calling functions in incorrect order
  - failure to call initialization function
  - failure to free resources
- **Exceptional conditions**
  - integer over/underflow
  - unexpected exceptions
  - not handling error cases
- **Code quality**
  - metrics (attack surface, etc.)

PolySpace

- www.polyspace.com
  - (now part of MathWorks)
- Focus: embedded system defects
- Languages: C, C++, Ada
  - UML Rhapsody, Simulink models
- OS: Windows, Linux, Solaris
- Selling points
  - Focus on embedded systems
  - Mathematically verifies code with proof engine
    - Assured code shown in green
    - Errors in checked classes cannot occur

- **Memory errors**
  - array bounds / buffer overrun
  - illegal dereference (null, integer, freed)
  - use uninitialized data
  - reference to non-initialized class members
- **Exceptional conditions**
  - integer over/underflow
  - division by zero
  - arithmetic exceptions
  - type conversion errors

SureLogic JSure

- www.surelogic.com
- Focus: concurrency, architecture, API usage
- Language: Java
- Selling points
  - Focus on Java concurrency
  - Immediate return on investment
  - Professional services
    - End-to-end support for FindBugs analysis
  - Sound analysis – shows assured code w/ green plus
    - Errors in checked classes cannot occur

- **Concurrency**
  - race conditions
  - data protected by locks
  - non-lock concurrency (e.g. AWT)
- **Architecture compliance**
  - module structure
- *Full disclosure: I have a stake in SureLogic as a consultant and potential technology provider*

Lattix LDM

- www.lattix.com
- Focus: architectural structure
- Languages: C, C++, Java, .NET
- OS: Windows, Linux, Mac OS X
- Published in OOPSLA 2005
- Selling points
  - Focus on architectural structure
  - Design Structure Matrix representation
    - Built automatically from code
    - Analysis extracts layered architecture
  - Checks design rules
  - Downloadable trial version

- **Design and understanding**
  - dependency analysis
  - impact analysis
  - architecture violations

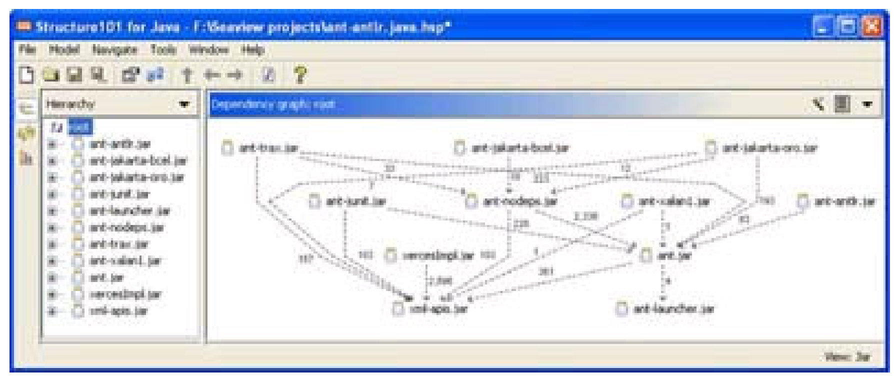
org.gjt.sp			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
jedit	print	1	.														
	proto.jed...	2		.													
	help	3			.						1						1
	options	4				.					2						1
	menu	5					.										4
	browser	6			1		7	.									3
	search	7						3	.								9
	gui	8			2	23	5	7	12	.		6		2		1	42
	pluginm...	9				2				1	.						1
	textarea	10					1		13	11		.	1				21
	buffer	11				1				1		4	.	1			13
	io	12			4	1	4	29	9	3	2		3	.			16
	syntax	13	3			4				1		6	1		.		16
	msg	14			1		2	4	3	4	2			3		.	25
	*	15	11	4	17	103	59	41	51	138	24	31	19	25	2	22	.

Source: OOPSLA 2005 paper

Headway Software Structure 101

- www.headwaysoftware.com
- Focus: architectural structure
- Languages: Java, .Net
- OS: Windows, Linux, OS X
- Selling points
  - Focus on architectural structure
    - Supports design structure matrices, other notations
  - Structural analysis
    - dependencies
    - impact of change
    - architectural evolution
  - Downloadable trial version

- **Design and understanding**
  - dependency analysis
  - impact analysis
  - architectural violations
  - complexity metrics



Source: Headway Software web site

Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
- What are current tools like?
- What does the future hold?
- What tools are available?
- How does it fit into my organization?
  - Lessons learned at Microsoft & eBay: Introduction, measurement, refinement, check in gates
    - Microsoft source: Manuvir Das
    - eBay source: Ciera Jaspan
      - OOPSLA 2007 Practitioner Report, “Understanding the Value of Program Analysis Tools”

Introducing Static Analysis

- Incremental approach
  - Begin with early adopters, small team
  - Use these as champions in organization
- Choose/build the tool right
  - Not too many false positives
  - Good error reporting
    - Show error context, trace
  - Focus on big issues
    - Something developers, company cares about
  - Ensure you can teach the tool
    - Suppress false positive warnings
    - Add design intent for assertions, assumptions
  - Bugs should be fixable [Manuvir Das]
    - Easy to fix, easy to verify, robust to small changes
- Support team
  - Answer questions, help with tool

Tool Customization

- Tools come with many analyses
  - Some relevant, some irrelevant
  - eBay example [Jaspan et al. 2007]
    - Dead store to local is a critical performance bug if the dead code is a database access
- Process
  - Turn on all defect detectors
  - Estimate value of reports, false positives
  - Assign each detector a priority
    - Tied to enforcement mechanism, e.g. prohibited on check-ins

Cost/Benefit Analysis

- Costs
  - Tool license
  - Engineers internally supporting tool
  - Peer reviews of defect reports
- Benefits
  - How many defects will it find, and what priority?
- Experience at eBay [Jaspan et al. 2007]
  - Evaluated FindBugs
  - Found less severe bugs than engineer equivalent
  - Clearly found more bugs than engineer equivalent
  - Ultimately incorporated tool into process
  - See OOPSLA 2007 practitioner report, Understanding the Cost of Program Analysis Tools

Enforcement

- Microsoft: check in gates
  - Cannot check in code unless analysis suite has been run and produced no errors
    - Test coverage, dependency violation, insufficient/bad design intent, integer overflow, allocation arithmetic, buffer overruns, memory errors, security issues
- eBay: dev/QA handoff
  - Developers run FindBugs on desktop
  - QA runs FindBugs on receipt of code, posts results
    - High-priority fixes required
- Requirements for success
  - Low false positives
  - A way to override false positive warnings
    - Typically through inspection
  - Developers must buy into static analysis first

Root Cause Analysis

- Deep analysis
  - More than cause of each bug
  - Identify patterns in defects
  - Understand why the defect was introduced
  - Understand why it was not caught earlier
- Opportunity to intervene
  - New static analyses
    - written by analysis support team
  - Other process interventions

Impact at Microsoft

- Thousands of bugs caught monthly
- Significant observed quality improvements
  - e.g. buffer overruns latent in codebaes
- Widespread developer acceptance
  - Check-in gates
  - Writing specifications

Analysis Maturity Model

Caveat: not yet enough experience to make strong claims

- Level 1: use typed languages, ad-hoc tool use
- Level 2: run off-the-shelf tools as part of process
  - pick and choose analyses which are most useful
- Level 3: integrate tools into process
  - check in quality gates, milestone quality gates
  - integrate into build process, developer environments
  - use annotations/settings to teach tool about internal libraries
- Level 4: customized analyses for company domain
  - extend analysis tools to catch observed problems
- Level 5: continual optimization of analysis infrastructure
  - mine patterns in bug reports for new analyses
  - gather data on analysis effectiveness
  - tune analysis based on observations

Analysis, Now and in the Future

- Static analysis is revolutionizing QA practices in leading companies today
- Exhibit A: Microsoft
  - Comprehensive analysis is centerpiece of QA for Windows
  - Now affects every part of the engineering process
- Static analysis enables organizations to:
  - increase quality while enhancing functionality
  - differentiate themselves from the competition

Questions?