

# Choosing the Right Software Assurance Tools

Paul E. Black

paul.black@nist.gov

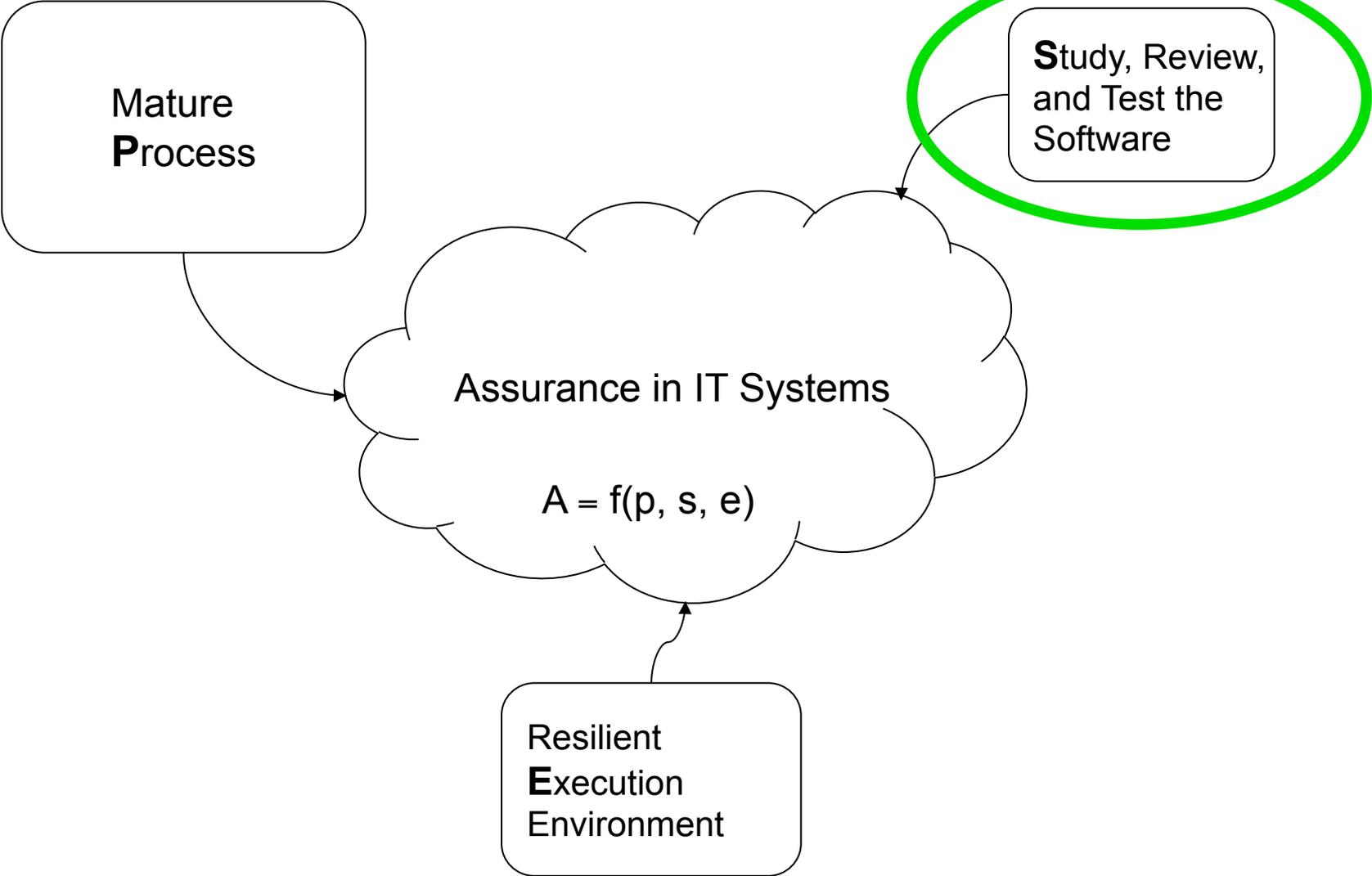
<http://samate.nist.gov/>



# Outline

- **Types of Software Assurance (SA) Tools**
- **Considerations and Variants**
- **Adding SA Tools to Your Process**
- **Using the CWE-121 Effectiveness Set**
- **More Resources from the SAMATE Reference Dataset (SRD)**

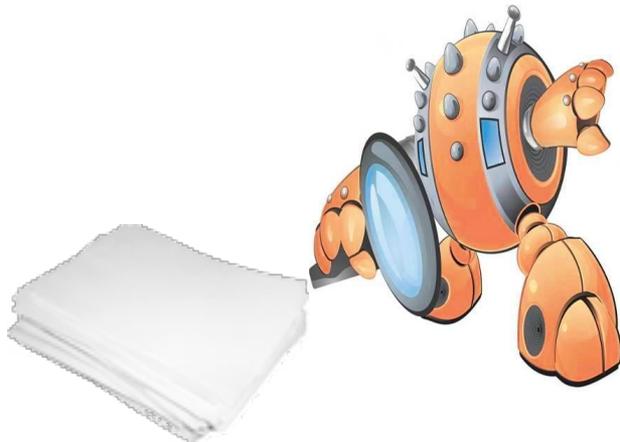
# What Goes Into Assurance?



# Two Kinds of Analysis: Static and Dynamic

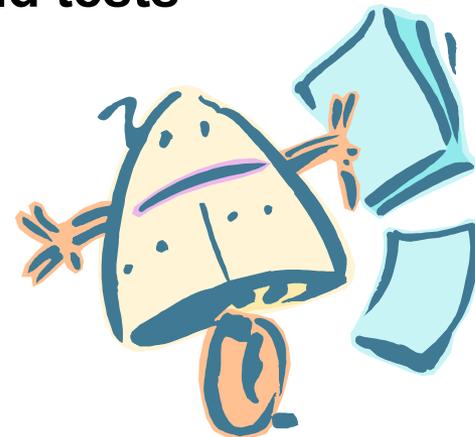
## Static Analysis

- Code review
- Binary, byte, or source code scanners
- Model checkers & property proofs
- Assurance case



## Dynamic Analysis

- Execute code
- Simulate design
- Fuzzing, coverage, MC/DC, use cases
- Penetration testing
- Field tests



# Static and Dynamic Analysis Complement Each Other

## Static Analysis

- **Handles unfinished code**
- **Higher level artifacts**
- **Can find backdoors, e.g., full access for user name “JoshuaCaleb”**
- **Potentially complete**

## Dynamic Analysis

- **Code not needed, e.g., embedded systems**
- **Has few(er) assumptions**
- **Covers end-to-end or system tests**
- **Assess as-installed**

# Different Static Analyzers Exist For Different Purposes

- To check intellectual property violation
- For developers to decide what needs to be fixed (and learn better practices)
- For auditors or reviewer to decide if it is good enough for use



# Outline

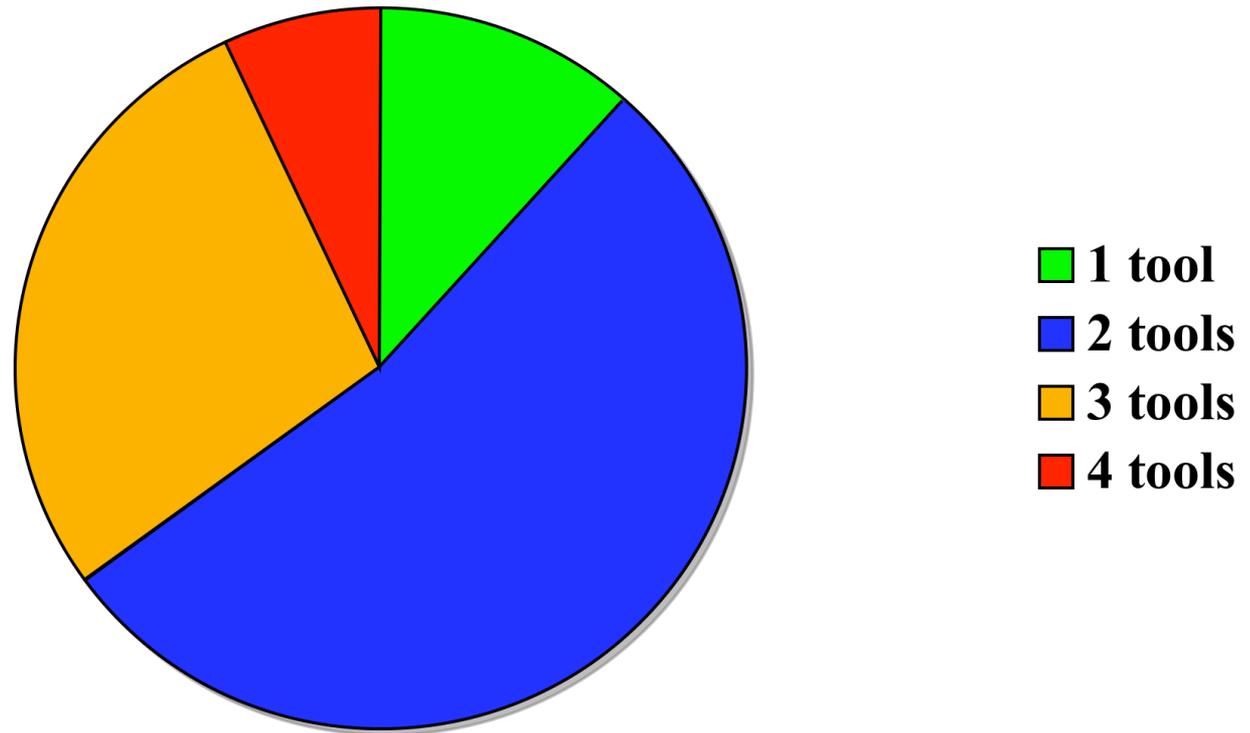
- Types of Software Assurance (SA) Tools
- **Considerations and Variants**
- Adding SA Tools to Your Process
- Using the CWE-121 Effectiveness Set
- More Resources from the SAMATE Reference Dataset (SRD)

# Consideration: Rates

- **False alarm rate**
- **Miss rate (recall)**
- **Precision**
- **Discrimination**

# Tools don't report the same flaws

## Overlap in Not-False Buffer Errors



# Consideration: Subject

- **What level?**
  - **Design, Requirements, Source code, Byte code, or Binary**
- **Language(s) handled**
- **Compiler extensions**
- **Platform**
- **Speed, scalability, max program size**

# Consideration: Properties

- **Analysis can look for anything from general or universal properties:**
  - don't crash
  - don't overflow buffers
- **to application-specific properties:**
  - log the date and source of every message
  - cleartext transmission
  - user cannot execute administrator functions
- ***Can I write my own "rules"?***

# Consideration: Level of Rigor

- **Syntactic**
  - flag every use of strcpy()
- **Heuristic**
  - every open() has a close(), every lock() has an unlock()
- **Analytic**
  - data flow, control flow, constraint propagation
- **Fully formal**
  - theorem proving

# Consideration: Human Involvement

- **analyst aides and tools**
  - call graphs
  - property prover
- **human-aided analysis**
  - annotations
- **completely automatic**
  - scanners

# Consideration: Output Format (1)

```
char sys[512] = "/usr/bin/cat ";  
25  gets(buff);  
    strcat(sys, buff);  
30  system(sys);
```

foo.c:30:Critical:Unvalidated string 'sys' is received from an external function through a call to 'gets' at line 25. This can be run as command line through call to 'system' at line 30. User input can be used to cause arbitrary **command execution** on the host system. Check strings for length and content when used for command execution.

# Consideration: Output Format (2)

Problem	Line	Source
		<i>/u1/paul/SATE/2010/c/irssi/irssi-0.8.14/src/core/rawlog.c</i>
		<i>Enter rawlog_save</i>
	140	<code>void rawlog_save(RAWLOG_REC *rawlog, const char *fname)</code>
	141	<code>{</code>
	142	<code>    char *path;</code>
	143	<code>    int f;</code>
	144	
	145	<code>    path = convert_home(fname);</code>
true	146	<code>    f = open(path, O_WRONLY   O_APPEND   O_CREAT, log_file_create_mode);</code>
	147	<code>    g_free(path);</code>
	148	
f <= -1	149	<code>    rawlog_dump(rawlog, f);</code>
		<i>Enter rawlog_save / rawlog_dump</i>
\$param_2 <= -1	102	<code>static void rawlog_dump(RAWLOG_REC *rawlog, int f)</code>
	103	<code>{</code>
	104	<code>    GSList *tmp;</code>
	105	
	106	<code>    for (tmp = rawlog-&gt;lines; tmp != NULL; tmp = tmp-&gt;next) { <i>/* Null Pointer Dereference</i></code>
f <= -1	107	<code>        write(f, tmp-&gt;data, strlen((char *) tmp-&gt;data)); <i>/* Negative file descriptor</i></code>
		<i>Exit rawlog_save / rawlog_dump</i>

# Consideration: Output Format (3)

- **Standard findings interchange format, e.g., SAFES or TOIF**

# Consideration: Tool Integration

- **Eclipse, Visual Studio, etc.**
- **Penetration testing**
- **Execution monitoring**
- **Bug tracking**

# Consideration: Non-Functional

- **Cost**
  - per seat, or per line of code
- **View issues by**
  - Category
  - File or Package
  - Priority
- **New issues since last scan**
- **Are issues increasing or decreasing?**
- **Which modules are hot spots?**



## CAS Static Analysis Tool Study - Methodology

Center for Assured Software  
National Security Agency  
9800 Savage Road  
Fort George G. Meade, MD 20755-6738  
cas@nsa.gov

December 2011

- **The report explains**
  - use of the Juliet test suite
  - the 14 weakness classes covered
  - automated run and scoring
  - measures: precision, recall, discrimination, etc.
  - graphs and tables to understand result
- **It does *not* evaluate specific tools.**

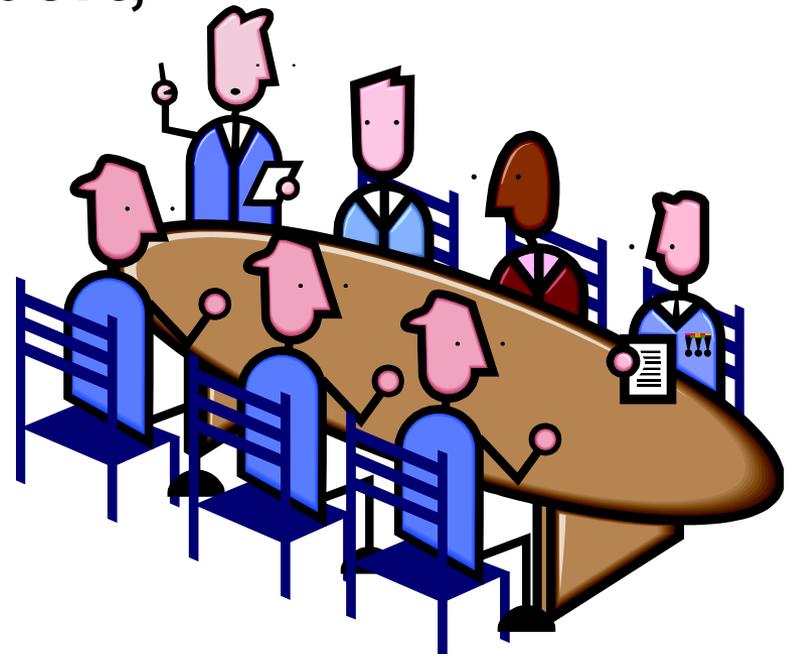
[//samate.nist.gov/docs/CAS 2011 Static Analysis Tool Study Methodology.pdf](https://samate.nist.gov/docs/CAS%202011%20Static%20Analysis%20Tool%20Study%20Methodology.pdf)

# Outline

- Types of Software Assurance (SA) Tools
- Considerations and Variants
- **Adding SA Tools to Your Process**
- Using the CWE-121 Effectiveness Set
- More Resources from the SAMATE Reference Dataset (SRD)

# Set up “consulting” group

- They have time to learn the tool, customize it for a project’s need, run it, and interpret results.
- Gradually withdraw support, e.g. longer turn around, less face-to-face effort – natural as consultants help other projects.



# Start with one class of flaw

- **Choose the class that is most critical or is easiest to catch.**
- **Add other flaw classes as value is demonstrated.**

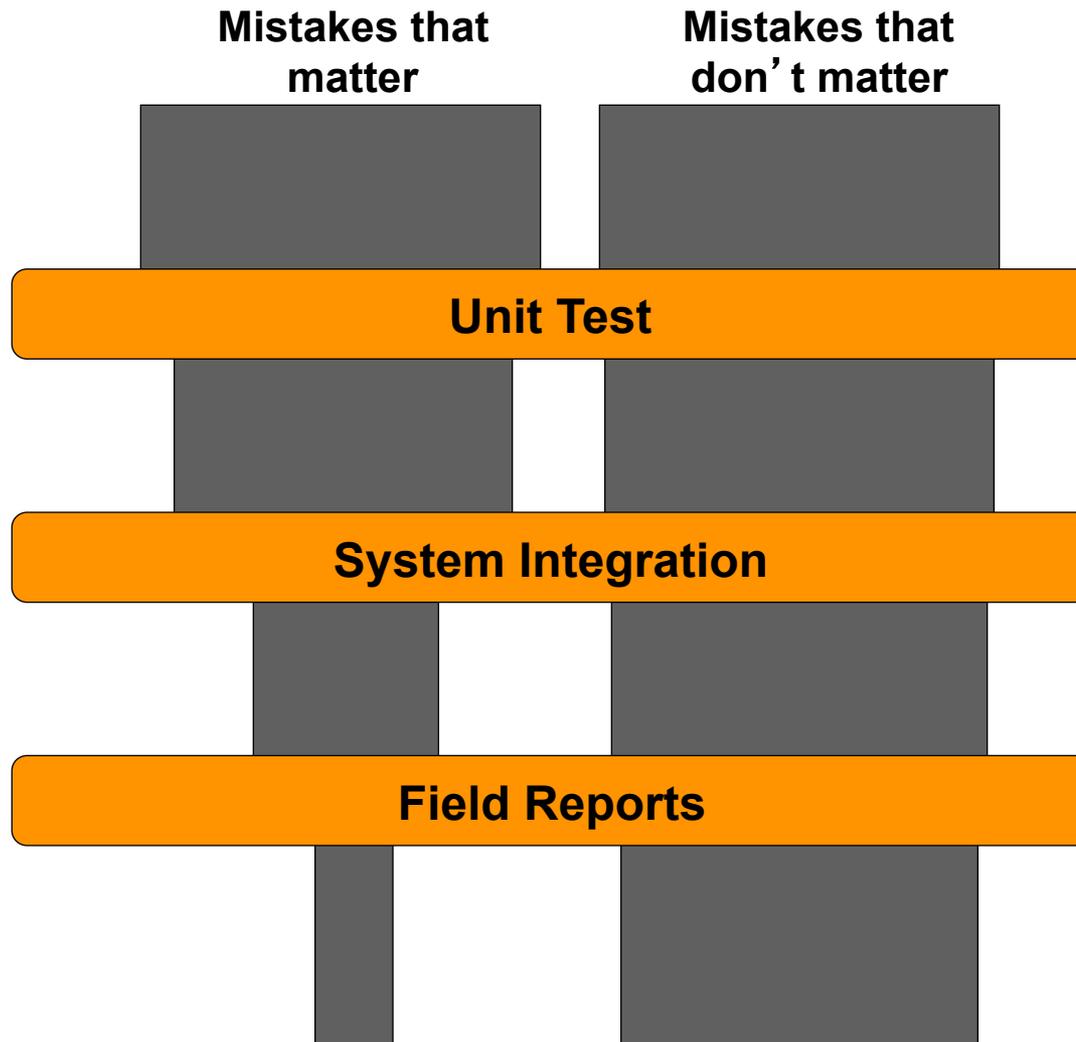
# Only Look at New Code

- **Ignore warnings from existing code**
  - it already runs, doesn't it?
- **Require that any brand new code needs to be “clean” – either code changed to avoid warnings or explicit justification.**
- **Then include code that is modified.**

# Increasingly Require Over Time

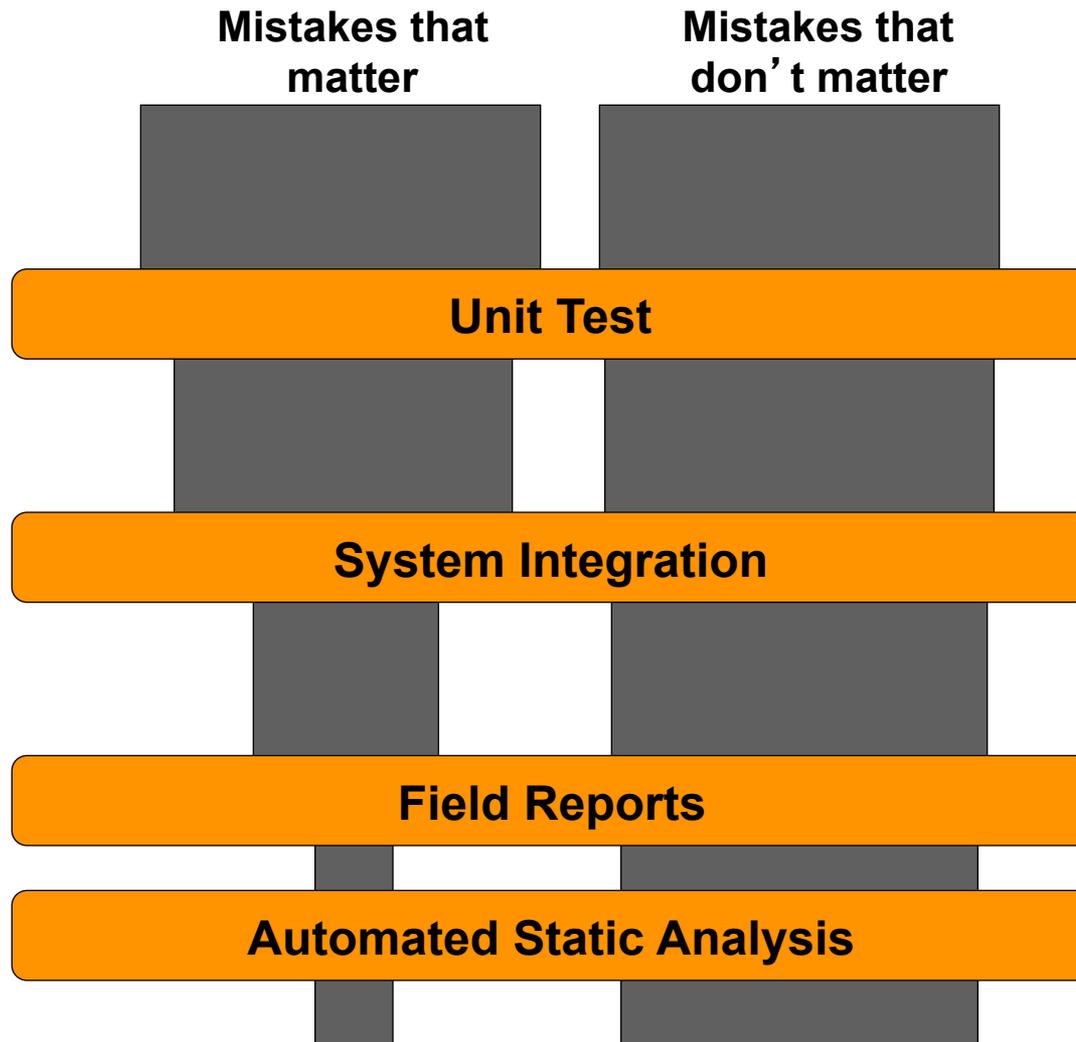
- **At first, the only requirement is that every developer *had* a static analyzer.**
- **Then required that it be run.**
- **Then standardize on one or two that developers found beneficial.**
- **Then require that warnings be reported.**
- **Then require that warnings be addressed (fixed or dismissed).**

# Survivor effect in software



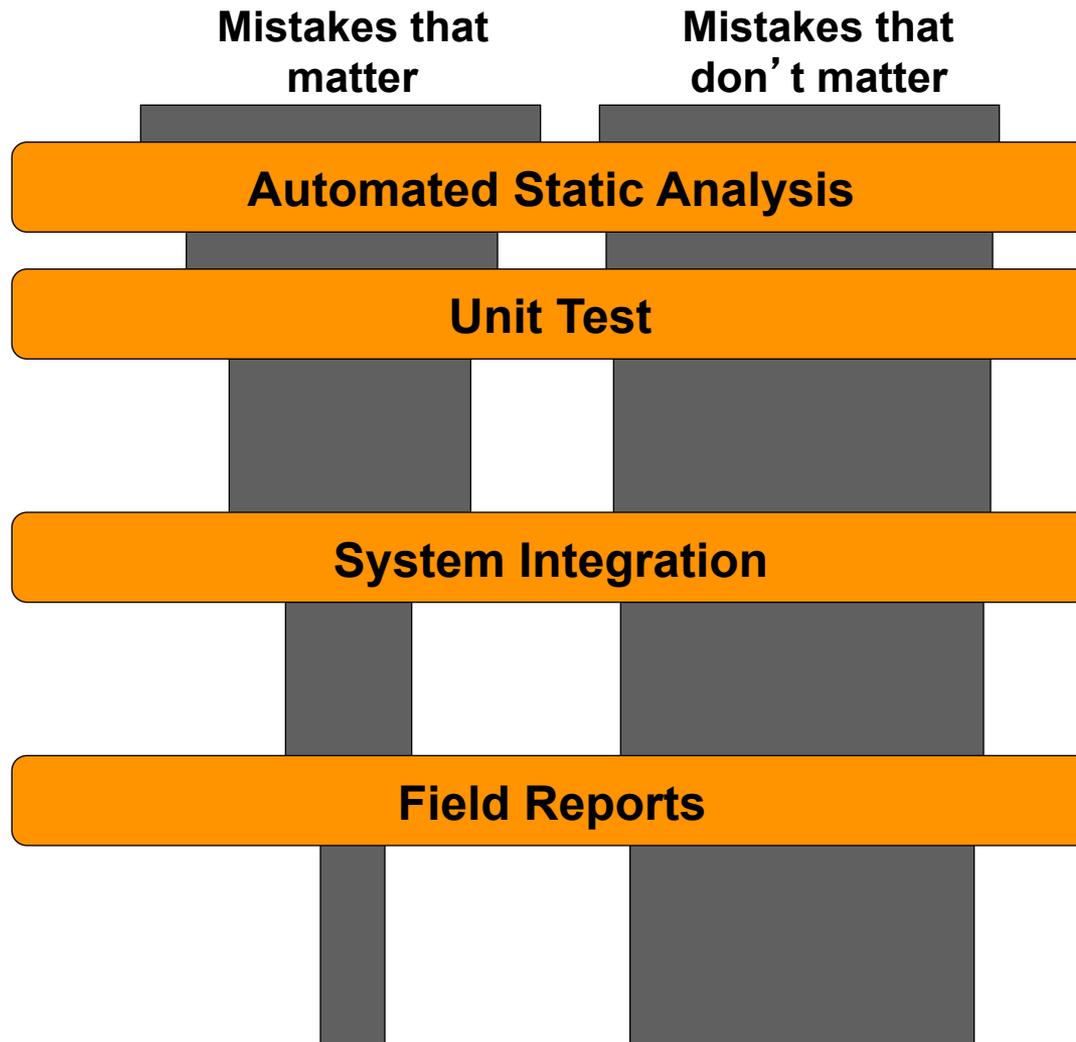
after Bill Pugh  
SATE workshop  
Nov 2009

# Late automated analysis is hard



after Bill Pugh  
SATE workshop  
Nov 2009

# Automated analysis best at start



after Bill Pugh  
SATE workshop  
Nov 2009

# When is survivor effect weak?

- **If testing or deployment isn't good at detecting problems**
  - True for many security and concurrency problems
- **If faults don't generate clear failures**
  - Also true for many security problems

after Bill Pugh  
SATE workshop  
Nov 2009

# Outline

- Types of Software Assurance (SA) Tools
- Considerations and Variants
- Adding SA Tools to Your Process
- **Using the CWE-121 Effectiveness Set**
- More Resources from the SAMATE Reference Dataset (SRD)

# MITRE's CWE Compatibility and Effectiveness Program

- **Phase 1 – Declare compatibility**
- **Phase 2 – Verify mapping to CWEs**
- **Phase 3 – Test cases show effectiveness**
  - tool effectively locates CWEs
  - tool deals with code complexities

<http://cwe.mitre.org/compatible/program.html>

# What is “Code Complexity”?

```
char data;
```

```
data = 'C';
```

```
data = 'Z';
```

```
printHexCharLine(data);
```

```
char data;
```

```
if (1) {
```

```
    data = 'C';
```

```
} else {
```

```
    data = 'C';
```

```
    printHexCharLine(data);
```

```
}
```

```
if (1) {
```

```
    data = 'Z';
```

```
    printHexCharLine(data);
```

```
} else {
```

```
    printHexCharLine(data);
```

```
}
```

CWE-563 Unused Variable, after SRD test cases 35455 and 35456

# What is content like?

- **Each CWE has one or more tests**
  - short (this is not about handling megacode)
  - code is vulnerable, i.e., exploitable
  - (usually) synthetic
  - fairly “clean”, but not necessarily pristine; meet SRD “accepted” standard
  - standard code; no language extensions
- **Test cases have corresponding “fixed” cases, to provide data on false positives**

# As a Proof-of-Concept

- **We started with CWE-121 Stack-based Buffer Overflow (in C language)**
  - **CWE-121 is a frequent, serious problem.**
  - **It is well-defined and easily understood.**
  - **We have thousands of examples.**
  - **It is addressed by static analysis, compile-time techniques, or run-time detection.**

# Background Work

- **Over the summer NIST researchers installed five static analyzers, then examined 7,338 in 9,962 files from**
  - Juliet (split into 5,892 good & bad cases)
  - Kratkiewicz (1,139 cases)
  - KDMA TCG (249 cases)
  - 2005 Fortify (41 cases)
  - other SRD (17 cases)

# Proposed CWE-121 Basic Set

- **It consists of five cases.**
- **The most basic case is basic-00001-min.c**

```
char buf[10];  
buf[10] = 'A';
```

– **This is so trivial it never occurs in real code.**

- **We added four more cases as simple variants.**

# Other Basic Cases

- **basic-00034-min.c**
  - access through a pointer
- **basic-00045-min.c**
  - use strcpy()
- **basic-00182-min.c**
  - fgets(): limited copy and external input
- **stack\_overflow\_loop.c**
  - loop initializes array, but bad bounds check

# Next Step – Complexity Cases

- **Cases related to SATE or Lippman**
- **Other fns: str(n)cpy/cat, memcpy/move, s(n)printf**
- **Separate files (caseA.c & caseB.c)**
- **Duplicate function names**
- **Dynamic allocation - alloca()**
- **Array indexing - see Kratkiewicz**
- **Data Types**
- **Buffer in struct**
- **Dead (infeasible) code**
- **Open coded or obfuscated str(n)cpy()**
- **Cases with a difference between I/J/M or min/med**

# What about tool “short cuts”?

- **Tool makers may build to a public, static set.**
  - A secret or dynamic set has other problems.
- **Change comments and identifier names for every download?**
- **Add innocuous statements?**
- **Transform code, like unroll loops?**

## ***Proposal:***

- **If concerns arise, privately corroborate results.**

# Outline

- Types of Software Assurance (SA) Tools
- Considerations and Variants
- Adding SA Tools to Your Process
- Using the CWE-121 Effectiveness Set
- **More Resources from the SAMATE Reference Dataset (SRD)**

# SAMATE Reference Dataset

The screenshot shows the SAMATE Reference Dataset website. At the top, there are logos for SAMATE, NIST (National Institute of Standards and Technology), and DHS National Cyber Security Division. Below the logos is a navigation bar with links: SRD Home, View / Download, Search / Download, More Downloads, Submit, and Test Suites. The main content area is divided into two sections: 'Extended Search' and 'Source Code Search'. The 'Source Code Search' section is active and contains a search form with the following fields and options:

- Number (Test case ID): [Text input]
- Description contains: [Text input]
- Contributor/Author: [Text input]
- Bad / Good: [Any... dropdown]
- Language: [Any... dropdown]
- Type of Artifact: [Any... dropdown]
- Status: Candidate  Approved
- Weakness: [Any... dropdown]
- Code complexity: [Any... dropdown]
- Date:  Any  Before  After  
(Format: M/d/Y) [Text input] [Calendar icon]

Below the search form is a 'Search Test Cases' button. To the right of the search form is a tree view of weaknesses and code complexity categories:

- Weakness
- Code Complexity
- Any...
- + CWE-485: Insufficient Encapsulation
- CWE-388: Error Handling
  - + CWE-389: Error Conditions, Return Values, Status Codes
- + CWE-254: Security Features
- + CWE-227: Failure to Fulfill API Contract (API Abuse)
- + CWE-019: Data Handling
- + CWE-361: Time and State
- CWE-398: Indicator of Poor Code Quality
  - CWE-470: Use of Externally-Controlled Input to Select Control Flow
  - + CWE-465: Pointer Issues
  - + CWE-411: Resource Locking Problems
  - CWE-401: Failure to Release Memory Before Removing Pointers
  - CWE-415: Double Free
  - CWE-416: Use After Free
  - + CWE-417: Channel and Path Errors

- Public repository for software assurance test cases
- Over 60,000 cases in C, C++, Java, C#, and Python
- Search and select by language, weakness, etc.
- Contributions from CAS, Fortify, Defence R&D Canada, Klocwork, MIT Lincoln Laboratory, Praxis, Secure Software, etc.

# Juliet 1.1 cases

- **23,957 cases in Java and 57,099 in C/C++ covering 181 weaknesses**
- **Each case is a page or two of code, sometimes crossing multiple files**
- **Most cases include similar unflawed code**
- **Organized by weakness, then variant, then complexity**
- **Described in IEEE Computer, Oct 2012**  
<http://samate.nist.gov/SRD/testsuite.php>

# STONESOUP cases

- **About 460 cases in Java and C, each a program typically 200-300 lines long**
- **Cover weaknesses in Number Handling (e.g. integer overflow), Tainted Data (e.g. input validation, Injection (e.g. command injection, Buffer Overflow, and Null Pointer**
- **Each case has inputs triggering the vulnerability, as well as “safe” inputs**
- **Available about November 2012**

# Kratkiewicz MIT cases

- **1164 cases in C for CWE-121 Stack-Based Buffer Overflow**
- **Created to investigate static analysis and dynamic detection methods**
- **Each case is one of four variants:**
  - **access within bounds (ok)**
  - **access just outside bound (min)**
  - **somewhat outside bound (med)**
  - **far outside bound (large)**
- **Code complexities: index, type, control, ...**

# Other SRD Content

- **Zitser, Lippmann, & Leek MIT cases**
  - 28 slices from BIND, Sendmail, WU-FTP, etc.
- **Fortify benchmark 112 C and Java cases**
- **Klocwork benchmark 40 C cases**
- **25 cases from Defence R&D Canada**
- **Robert Seacord, “Secure Coding in C and C++” 69 cases**
- **Comprehensive, Lightweight Application Security Process (CLASP) 25 cases**
- **329 cases from our static analyzer suite**