Static Analysis Tools in Industry: Dispatches From the Front Line

Dr. Andy Chou

Chief Scientist and Co-founder

Coverity, Inc.

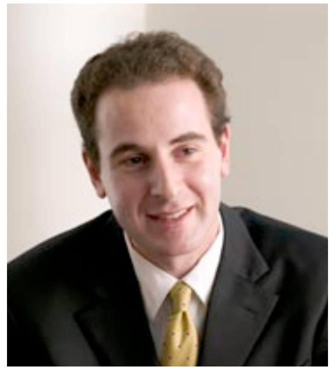
coverity®

Outline

- Things I know
  - A little bit about Coverity
  - Bug-Finding: Technology + Philosophy + Engineering
  - Beyond Bug-Finding: Fixing
- What I will show you
  - Demonstration of Coverity Static Analysis
- What I think I know
  - Making Money: Business model + Trials + Data
  - Socioeconomic aspects of developers and tools
  - A few specific problems that want to be solved
- Pure speculation

Coverity Founders

Andy Chou



Seth Hallem

Ben Chelf

Dawson Engler

Dave Park

coverity®

It Started with Research (1999-2003)

Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions, OSDI 2000

Using Meta-level Compilation to Check FLASH Protocol Code, ASPLOS 2000

An Empirical Study of Operating Systems Errors, SOSP 2001

A System and Language for Building System-Specific, Static Analyses, PLDI 2002

ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors, FSE 2003

… and more

About Coverity

- Founded in 2003
- Bootstrapped until 2007
- $22m venture funding in 2007 from Foundation and Benchmark Capital

As of mid-2011:

- 190+ employees
- 1100+ customers
- 100,000+ users worldwide
- Estimated 3-5 billion lines of code actively scanned
- Headquartered in San Francisco with offices in Boston, Calgary, Tokyo, and London

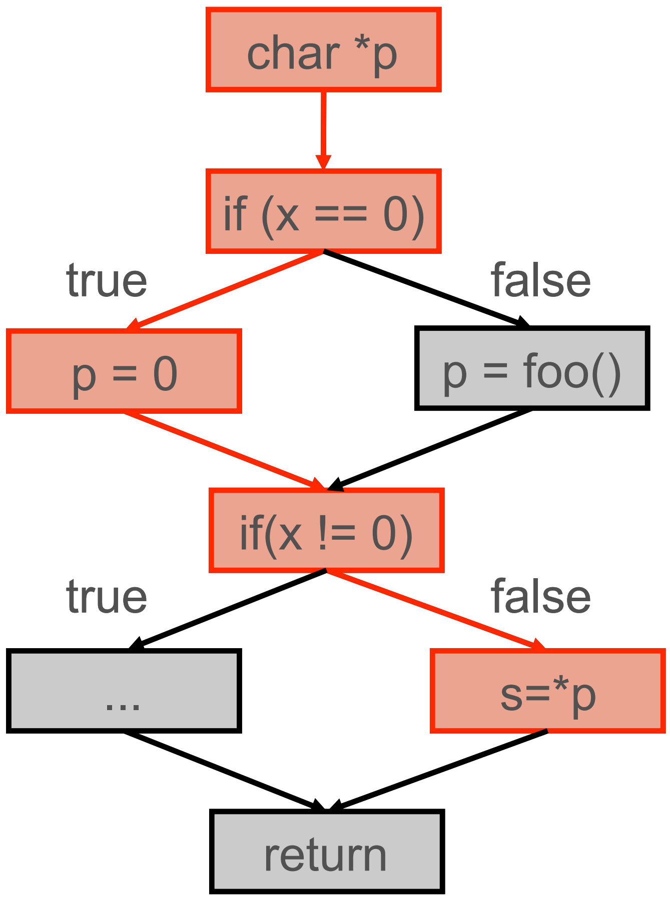
Static Analysis

Source Code

```
char *p;
if(x == 0)
  p = foo();
else
  p = 0;

if(x != 0)
  s=*p;
else
  ...;
return;
```

Symbolic CFG Analysis



Defects detected

Assigning: p=0

x!=0 taking true branch

Dereferencing null pointer p

Defective Sample Code

```c
#include <stdlib.h>

int process(char*, char*, char*, int);

int example(int size) {
    char *names;
    char *namesbuf;
    char *selection;

    names = (char*) malloc(size);
    namesbuf = (char*) malloc(size);
    selection = (char*) malloc(size);

    if(names == NULL || namesbuf == NULL || selection == NULL) {
        if(names != NULL) free(selection);
        if(namesbuf != NULL) free(namesbuf);
        if(selection != NULL) free(selection);
        return -1;
    }
    return process(names, namesbuf, selection, size);
}
```

Defects shown inline with the source code

```c
#include <stdlib.h>

int process(char*, char*, char*, int);

int example(int size) {
    char *names;
    char *namesbuf;
    char *selection;

```

CID 68629: Resource leak (RESOURCE_LEAK) [select defect]

```c
    names = (char*) malloc(size);
    namesbuf = (char*) malloc(size);
    selection = (char*) malloc(size);

    if(names == NULL || namesbuf == NULL || selection == NULL) {
```

CID 68630: Double free (USE_AFTER_FREE) [select defect]

```c
        if(names != NULL) free(selection);
        if(namesbuf != NULL) free(namesbuf);
        if(selection != NULL) free(selection);
        return -1;
    }
    return process(names, namesbuf, selection, size);
}
```

First Defect: Memory Leak

Allocated “names”

Checking for allocation failures for all variables

Freeing “selection” instead of “names”

“names” leaked

```
int example(int size) {
    char *names;
    char *namesbuf;
    char *selection;

CID 68629: Resource leak (RESOURCE_LEAK)
Calling allocation function "malloc".
Assigning: "names" = storage returned from "malloc(size)".
    names = (char*) malloc(size);
    namesbuf = (char*) malloc(size);
    selection = (char*) malloc(size);

At conditional (1): "names == NULL" taking the false branch.
At conditional (2): "namesbuf == NULL" taking the false branch.
At conditional (3): "selection == NULL" taking the true branch.
    if(names == NULL || namesbuf == NULL || selection == NULL) {
CID 68630: Double free (USE_AFTER_FREE) [select defect]
At conditional (4): "names != NULL" taking the true branch.
        if(names != NULL) free(selection);
At conditional (5): "namesbuf != NULL" taking the true branch.
        if(namesbuf != NULL) free(namesbuf);
At conditional (6): "selection != NULL" taking the false branch.
        if(selection != NULL) free(selection);
Variable "names" going out of scope leaks the storage it points to.
        return -1;
    }
    return process(names, namesbuf, selection, size);
}
```

Second Defect: Double Free

Freeing “selection” instead of “names”

Freeing “selection” again

```
int example(int size) {
    char *names;
    char *namesbuf;
    char *selection;

CID 68629: Resource leak (RESOURCE_LEAK) [select defect]
    names = (char*) malloc(size);
    namesbuf = (char*) malloc(size);
    selection = (char*) malloc(size);

    if(names == NULL || namesbuf == NULL || selection == NULL) {
CID 68630: Double free (USE_AFTER_FREE)
"free" frees "selection".
        if(names != NULL) free(selection);
At conditional (1): "namesbuf != NULL" taking the false branch.
        if(namesbuf != NULL) free(namesbuf);
At conditional (2): "selection != NULL" taking the true branch.
Calling "free" frees pointer "selection" which has already been freed.
        if(selection != NULL) free(selection);
        return -1;
    }
    return process(names, namesbuf, selection, size);
}
```

C/C++ Defects That Coverity Can Find

Part 1

Resource Leaks
- Memory leaks
- Resource leak in object
- Incomplete delete
- Microsoft COM BSTR memory leak

Uninitialized variables
- Missing return statement
- Uninitialized pointer/scalar/array read/write
- Uninitialized data member in class or structure

Concurrency Issues
- Deadlocks
- Race conditions
- Blocking call misuse

Integer handling issues
- Improper use of negative value
- Unintended sign extension

Improper Use of APIs
- Insecure chroot
- Using invalid iterator
- printf() argument mismatch

Memory-corruptions
- Out-of-bounds access
- String length miscalculations
- Copying to destination buffers too small
- Overflowed pointer write
- Negative array index write
- Allocation size error

Memory-illegal access
- Incorrect delete operator
- Overflowed pointer read
- Out-of-bounds read
- Returning pointer to local variable
- Negative array index read
- Use/read pointer after free

Control flow issues
- Logically dead code
- Missing break in switch
- Structurally dead code

Error handling issues
- Unchecked return value
- Uncaught exception
- Invalid use of negative variables

C/C++ Defects That Coverity Can Find

Part 2

Program hangs

- Infinite loop
- Double lock or missing unlock
- Negative loop bound
- Thread deadlock
- sleep() while holding a lock

Null pointer differences

- Dereference after a null check
- Dereference a null return value
- Dereference before a null check

Code maintainability issues

- Multiple return statements
- Unused pointer value

Insecure data handling

- Integer overflow
- Loop bound by untrusted source
- Write/read array/pointer with untrusted value
- Format string with untrusted source

Performance inefficiencies

- Big parameter passed by value
- Large stack use

Security best practices violations

- Possible buffer overflow
- Copy into a fixed size buffer
- Calling risky function
- Use of insecure temporary file
- Time of check different than time of use
- User pointer dereference

Java/C# Defects That Coverity Can Find

Resource Leaks

- Database connection leaks
- Resource leaks
- Socket & Stream leaks

API usage errors

- Using invalid iterator
- Unmodifiable collection error
- Use of freed resources

Concurrent data access violations

- Values not atomically updated
- Double checked locking
- Data race condition
- Volatile not atomically updated

Performance inefficiencies

- Use of inefficient method
- String concatenation in loop
- Unnecessary synchronization

Program hangs

- Thread deadlock

Class hierarchy inconsistencies

- Failure to call super.clone() or supler.finalize()
- Missing call to super class
- Virtual method in constructor

Control flow issues

- Return inside finally block
- Missing break in switch

Error handling issues

- Unchecked return value

Null pointer dereferences

- Dereference after null check
- Dereference before null check
- Dereference null return value

Code maintainability issues

- Calling a deprecated method
- Explicit garbage collection
- Static set in non-static method

coverity®

Philosophy + Engineering + Technology

- Focus on bug finding
- Focus on developer stickiness
  - Low false positive rate (typically <20% out of the box)
  - (more on next slide)
- Interprocedural analysis with bottom-up function summarization
  - Ensures bounded memory use: only one function + summaries for callees
  - Each function only analyzed once; recursive cycles are broken
  - Context sensitive
- Path sensitivity with false path pruning
  - Multiple independent false path pruners: integer interval solver, string logic, inequality, SAT-based
- Staged analysis
  - Cheaper analyses are run before more expensive ones – false path pruning only run if a candidate defect is found
- Parallel, incremental analysis
  - Android kernel: 700kLOC, 10 minutes with 8-way parallel analysis from scratch

Top reasons for low false positives

- Iterative checker design
  - Start with a defect example or idea
  - Implement a rough checker that casts a wide net
  - Run on open source
  - Sample first N results
  - Address idioms, refine heuristics, add options
  - Repeat until the checker has a low FP rate and still finds defects
    - Or, discard the checker altogether
- Evidence-based approach
  - Only report defects if enough evidence is available that it is likely to be real
  - This also helps developers understand the results
  - Evidence orientation is a good way to think about what analyses will be successful
- Perception: avoidance of stupid looking false positives is important
  - A single example of a dumb looking FP can result in loss of credibility
  - Credibility among a core individual / group is key to adoption

Technologies we don't use (much)

- Pointer alias analysis
  - Blobs cause FP explosions
  - Typical tricks for achieving scalability introduce inference steps that don't make sense to developers – e.g. field insensitivity, flow insensitivity, …
  - Checkers, derivers, and FPP do their own intraprocedural alias tracking with full understanding of what they do and don't care about
  - No single unified memory model – each checker can pick its own
  - E.g. No resource leak is detected in this code:

```
void example9(int x) {
    static struct S static_entry;
    struct S *p, *q;
    if(x)
        p = &static_entry;
    else
        p = malloc(sizeof(*p));
    q = p;
    if(q != &static_entry)
        free(q);
}
```

Other technologies we don't use much

- Heap structure analysis
- Complex string analysis
- Abstract interpretation (*)
- … many more

Beyond Bug-Finding: Fixing

coverity®

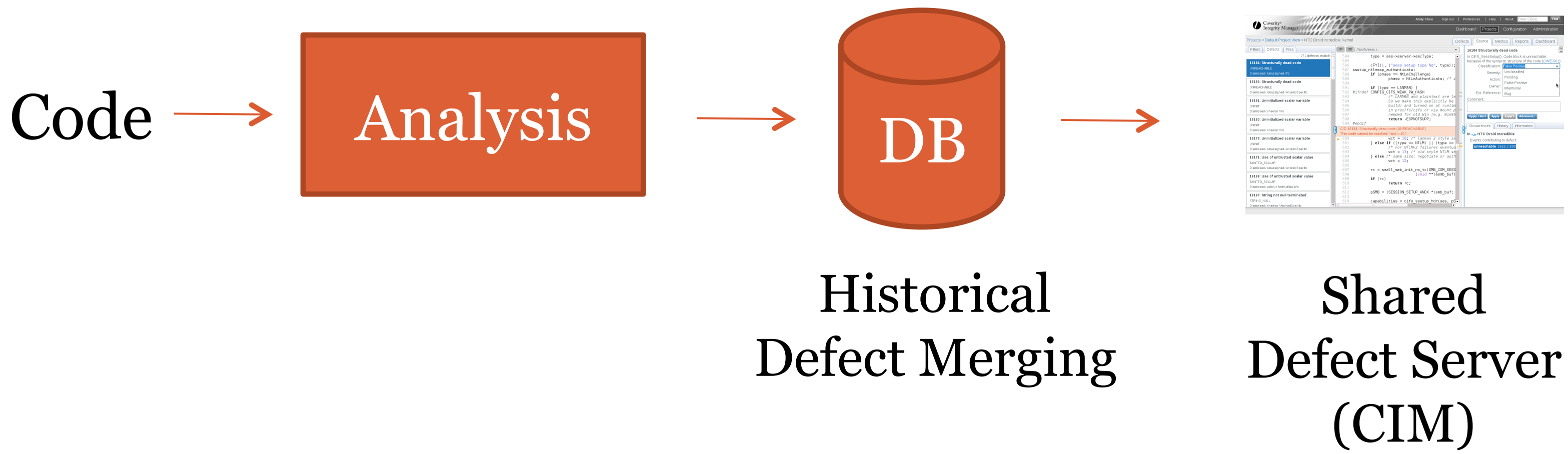
The importance of workflow

- What doesn't work:

- Why?
  - Bugs get fixed. False positives don't. Over time, FP rate approaches 100%.
  - Unclear what should be fixed; no prioritization
  - Unclear who should fix what; no ownership
- Workflow separates a static analysis *engine* from a static analysis *solution*.

Defect management and collaboration

- What works better:



- Track defects across time, even if the code changes (hashing/merging)
- Share triage information across developers
- Prioritize and assign ownership of defects
- Detect defect duplication across branches

coverity®

Deployment practices

- Clean before checkin
- Nightly build
- Continuous integration
- Incremental nightly build + weekend full analysis
- Code review integration
- Bug fix-it day
- Baselining

Baselining

- The first time static analysis runs, there may be thousands of errors
  - Typical: 1 defect/kLOC, 1MLOC code base = 1000 defects
  - Where to start?
- Analysis answer: rank
- Market's answer: baseline
  - Ignore all defects on existing code (the "baseline")
  - Fix defects in new code
  - "Someday" get around to fixing defects in old code
- Why is this so popular?
  - Old code is in the field. It works well enough. Risk is low.
  - New code is unproven. It might work, or it might not. Risk is high.

Demonstration

Business Model

We sell term-based project licenses sized by lines of code or team size.

Term-based:

- Customers purchase for a specific period of time, mostly 1 or 3 years.
- Customers renew every year based on then project size.

Project license:

- We license specific named projects (e.g. a code base).

Sizing:

- LOC is the most common metric (with special cases to handle OS and third party code).
- Team licenses are based on the total number of developers working on a project.

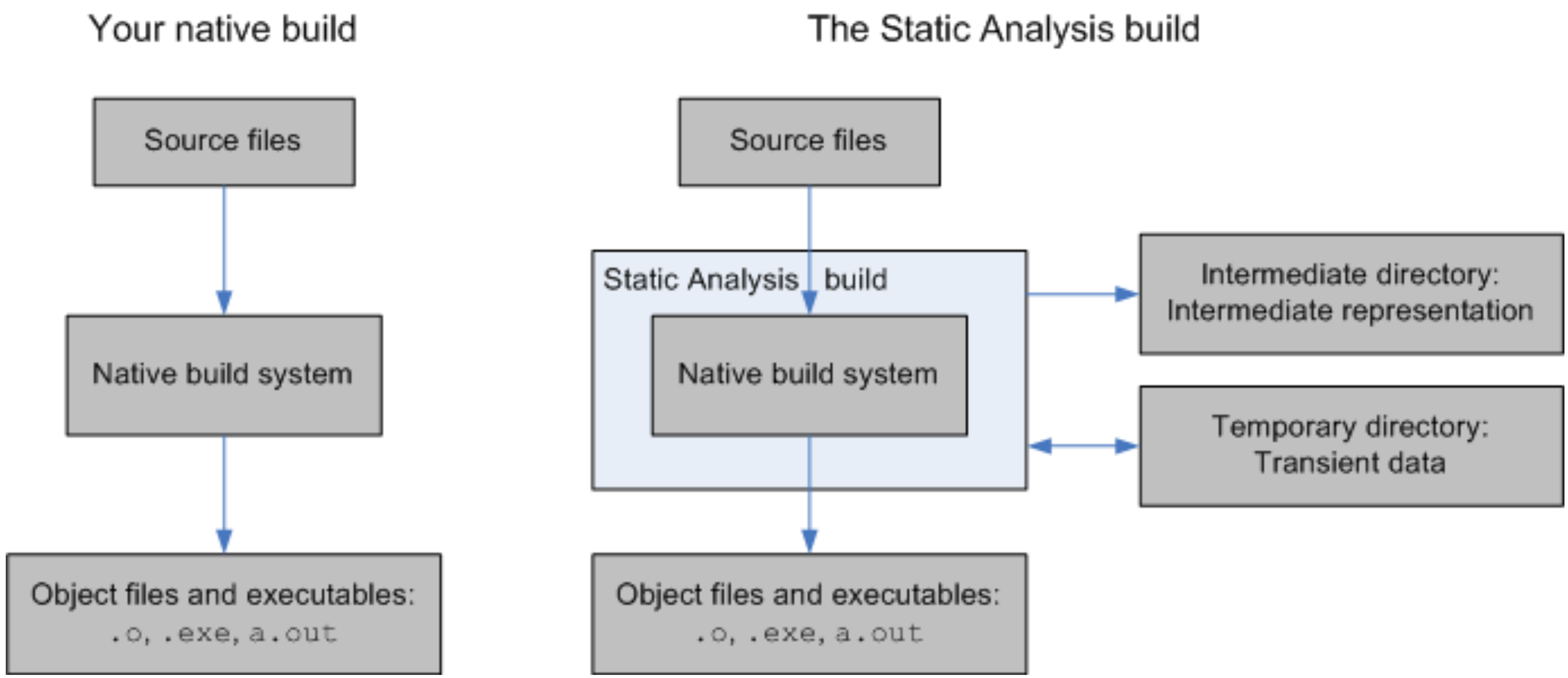
Enterprise licenses have custom terms.

Opportunity cost and urgency

- Favorite VC questions:
  - Where does the budget come from? What are they NOT going to spend on?
  - Why now?
- Decision maker is often a director of engineering or VP of engineering
  - ALWAYS strapped for resources
  - There are a multitude of problems to be solve to successfully deliver product
  - Is this use of money the most cost-effective use of these resources?
- "Why don't we instead…"
  - Hire 20 developers and QA engineers in low cost geography
  - Improve test coverage
  - Buy a collaborative code review tool
  - Developer training
- Quality is not a new problem.
  - Companies have already tried their best to optimize resources using many methods to try to lower costs and find defects early.
  - New technologies need to overcome all of these optimizations and deliver ROI of many multiples more

[Some slides omitted]

Build Integration - the code must be found and parsed to be analyzed

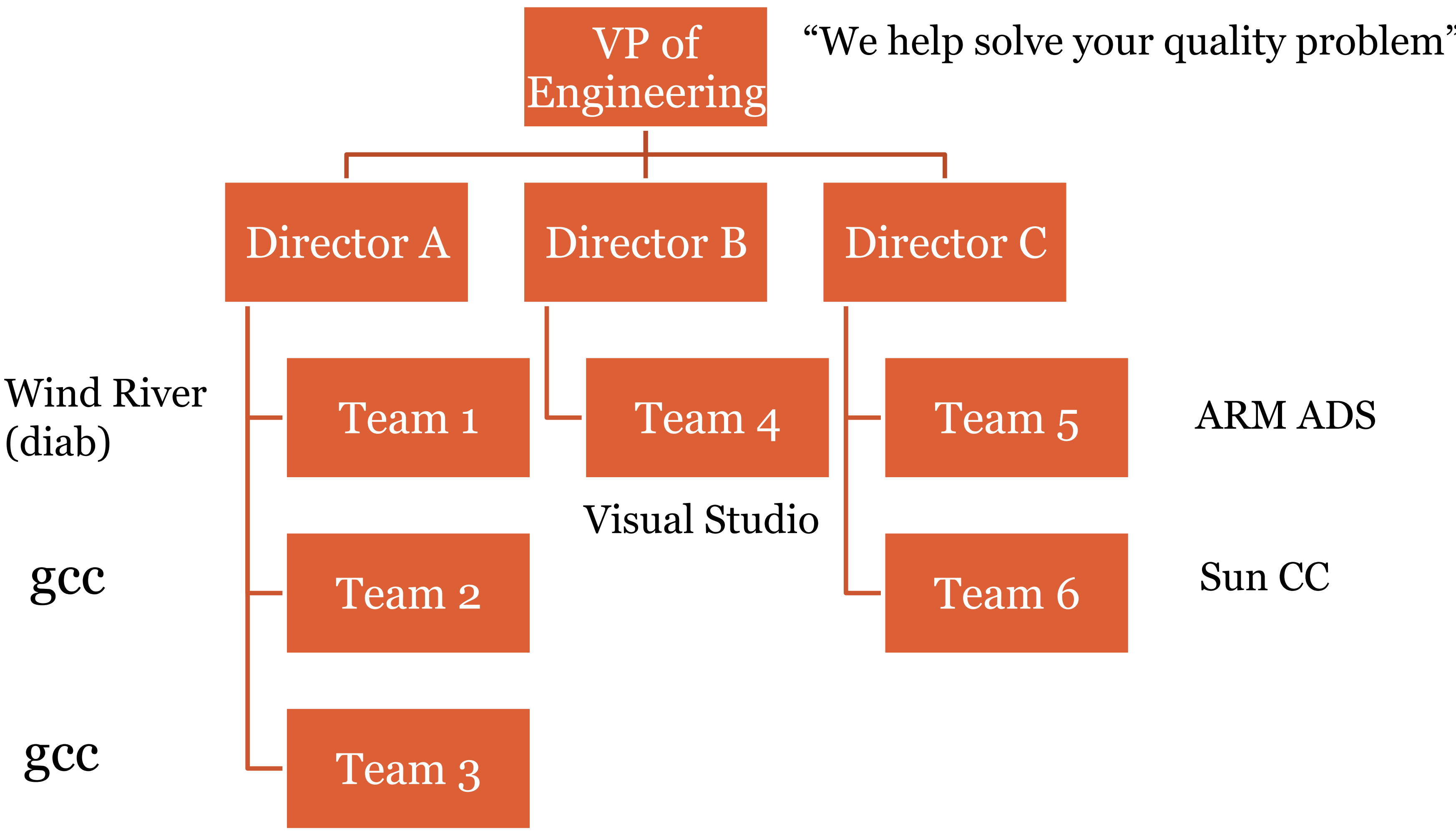


Support for Mimicking Dozens of Compilers

- Our build integration understands:
  - Compiler command line options
  - Built-in macro definitions
  - Compiler-specific language extensions
  - Compiler bugs that allow nonstandard code to parse

Analog Devices VisualDSP++	Nokia Codewarrior for Symbian
ARM C and C++	QNX C/C++
Borland C++	Renesas C/C++
Cosmic C Cross Compilers	Scratchbox
Freescale Codewarrior	SNC PPU C/C++
GNU GCC and G++	STMicroelectronics GNU C/C++
Green Hills C and C++/EC++	STMicroelectronics ST Micro C/C++
HI-TECH PICC	Sun (Oracle) CC and cc
HP aCC	Tensilica Xtensa xt-xcc and xt-xtc++
IAR Embedded Workbench C/C++	Texas Instruments Code Composer
Intel C++	TriMedia TCS
Keil Compilers	Visual Studio
Marvell MSA	Wind River (formerly Diab) C/C++

Why bother with the small compilers?



Organizational structure influences product requirements through buying behavior

- The higher you go in the org chart:
  - The more you can charge
  - The less they understand what you do
  - The more they want "coverage" of all of their code
  - The more they want a complete solution that meets more requirements
  - The fewer vendors they want to deal with
  - The more metrics you need to provide to prove value
- Hence:
  - MISRA
  - C/C++/Java/C# … Javascript, Ada, Cobol, Objective C, PHP, Actionscript/FLASH, PL/SQL, …
  - Reports, charts, pretty pictures

coverity®

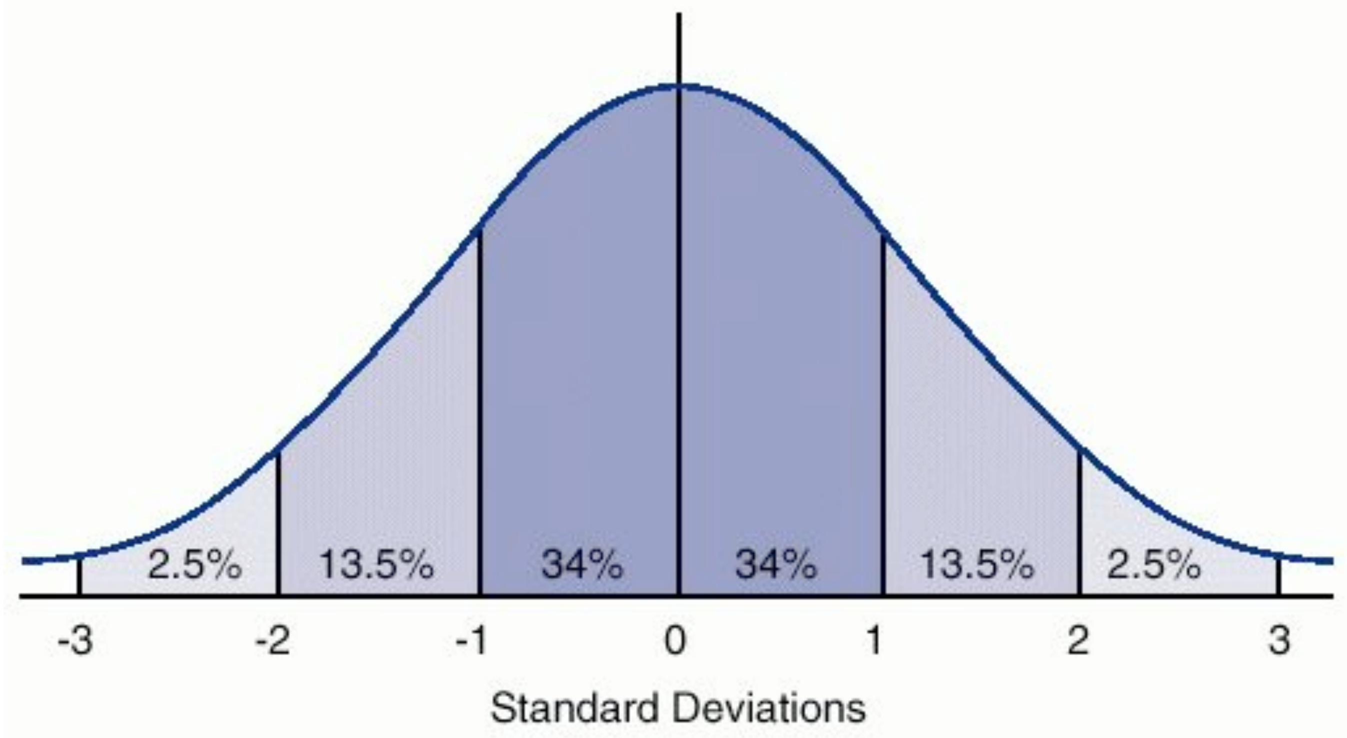
About Developers…

- The developer persona
  - Resistant to change
  - Impatient – “time to value” needs to be very short - think coffee break.
  - Quick to dismiss a tool that loses credibility – hence a focus on eliminating “stupid looking false positives”.
  - Instant gratification – Eclipse/VS highlight as you type; continuous integration happens every half hour
  - Hero complex
  - Artist complex
- “There’s no glory in fixing bugs”
- Firefighter by day, arsonist by night

Developers

- Like any large human population there is a normal distribution of talent and intelligence for developers

(This is getting worse for C/C++)



Yet… Developer Adoption is Key

- Developers need to adopt or there is no value to a tool
  - Priorities change like the wind howls – will the tool + process stick?
  - The term business model means a huge problem for renewal rate if adoption doesn't happen
- One possible solution:
  - Services to integrate everything
  - Automatic analysis "while you sleep" (or drink coffee)
  - Automatic assignment to the right developer
  - Proactive email notification
  - IDE integration
  - … and much more to make it smooth, seamless, and as painless as possible

coverity®

Problems that want to be solved

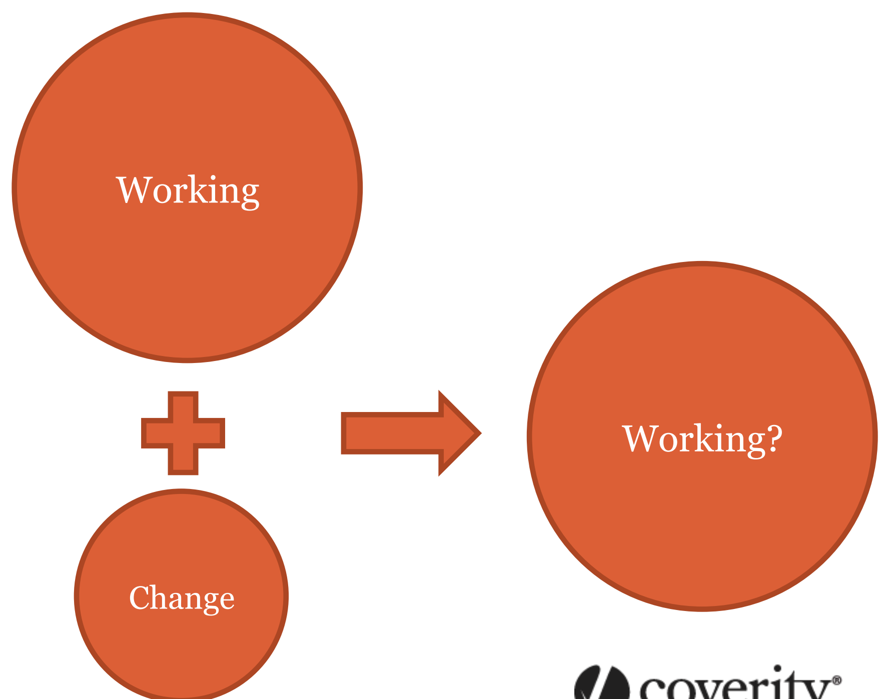
Most real-world problems are boring

- Maintaining a large legacy code base
  - Removing dead code
    - Large company: probably 60%+ of code is dead
    - This is an ongoing tax on understanding and modifying this code
    - Mindset: first eliminate code that doesn't matter, this lowers costs going forward
  - Visualizing code
- Standards compliance
  - MISRA, JSF++ / DO-178b / ISO 26262 / PCI
- Defect churn / instability
  - Normal bug: reproduce, fix, verify fix
  - Developers tend to want to work the same way on static analysis defects; this requires analysis to be very stable
- Tools that enable better productivity from the bottom 80% of developers
  - Tools are rarely put into the hands of the best people to use. They are too busy building product features.

coverity®

The non-boring real-world problems are hard

- Most static analysis considers the code as a monolithic input
- Development organizations don't see it that way at all.
- Their existing code works. They are changing it. They want to know:
  - Will this change introduce risk of customer issues?
  - What kind of customer issues should I expect?
  - Where should I expect them?
  - What should I test?
  - Am I on track to ship next month?
- Real life is a complex trade-off
  - They want help making this trade-off given business needs



[Some slides omitted]

Pure speculation

New languages do get adopted

Position Jul 2011	Position Jul 2010	Delta in Position	Programming Language	Ratings Jul 2011	Delta Jul 2010	Status	
1	1	=	Java	19.251%	+0.58%	A	1996
2	2	=	C	17.280%	-1.20%	A	1973
3	3	=	C++	9.017%	-1.45%	A	1983
4	5	↑	C#	6.221%	+0.49%	A	2001
5	4	↓	PHP	6.179%	-2.39%	A	1995
6	9	↑↑↑	Objective-C	5.181%	+2.68%	A	1986
7	6	↓	(Visual) Basic	5.106%	-0.41%	A	1991
8	7	↓	Python	3.583%	-0.63%	A	1991
9	8	↓	Perl	2.328%	-0.77%	A	1987
10	10	=	JavaScript	2.242%	-0.19%	A	1995
11	19	↑↑↑↑↑↑↑↑	Lua	1.572%	+1.04%	A	1993
12	12	=	Ruby	1.325%	-0.66%	A	1995
13	16	↑↑↑	Lisp	0.906%	+0.28%	A	1958
14	11	↓↓↓	Delphi/Object Pascal	0.887%	-1.44%	A	1995
15	24	↑↑↑↑↑↑↑↑↑	Transact-SQL	0.802%	+0.34%	A--	1974
16	15	↓	Pascal	0.668%	+0.03%	A-	1970
17	-	=	Assembly*	0.618%	-	B	
18	22	↑↑↑↑	RPG (OS/400)	0.559%	+0.09%	B	
19	28	↑↑↑↑↑↑↑↑↑	Ada	0.549%	+0.17%	B	1980
20	46	↑↑↑↑↑↑↑↑↑↑	C shell	0.545%	+0.33%	B	

← PLDI 2001
Snowbird, Utah

Getting the world to eat spinach

- It is a vital and important area of inquiry to understand how to make verification technologies more palatable
- Do we understand the traits that lead to language popularity, and how can we trojan horse the best ideas from modern research into something that will become popular?
  - Dynamic typing – less typing? Cleaner syntax? Error resilience?
  - Social aspects should not be underestimated
  - The web spawned Javascript, but nothing was ready to step in – a huge missed opportunity
- More than 50% of this is being ready at the right place and the right time – and mixing this with a larger trend

Or… be real about legacy code

- Be realistic about what can be expected
  - Restrict the scope to a segment of the market – and really understand that domain and how code is specialized for it
  - Realize that the market is already trying to optimize and might be "good enough" with proven technologies and processes
  - Change assumptions to better fit what can be realistically adopted

- "Everything described in the paper works. Everything else doesn't"
  - Why isn't that in the paper? That's the most important part.
  - An empirical approach with negative results is vital for legacy code problems

coverity®

Conclusion

Is there Hope?

- We are still taking baby steps… but many companies are starting to care
  - When there's a new quality initiative, someone speaks up: "Static analysis is one of the easiest things we can do…"
  - Companies are more ready to listen after a major incident
  - For any given company at any given time the chances are low, but eventually everyone gets burned
- The groundwork is being laid for lower barriers
  - Coverity and others are being deployed into build systems, processes, and management metrics
  - This will eventually lower the barrier to entry for new technologies on top of these platforms
- Exposure to real-world problems
  - Other academic disciplines have the notion of "field work"
  - Find ways to get out there and see what real development organizations are facing

Academic Program

- Get access to our static analysis product for a nominal fee (*)
- Teaching license
- Research license
- Some restrictions

http://www.coverity.com

Q & A

Andy Chou
andy@coverity.com