



# Static Analysis Use Case

## Samba and Coverity

- Launched, March 2006
- DHS sponsored “Open Source Hardening Project”
  - 2006-2009
- Using Coverity’s commercial static analysis product to identify bugs at the source code level
- 35 open source projects on day one
- Since grown to 300+ projects
- Over 15,000 bugs fixed

There is no single measure of the effectiveness of a tool on the software development process.

Since we can never run the same development effort twice, with identical teams, portions of this evaluation are highly subjective.

- Objective measures
  - Static Analysis produced defect counts
  - Numbers of Bug Reports
  - Defects confirmed as ‘real’ by the developers
- Subjective measures
  - Anecdotal comments by developers
  - Community feedback
  - ‘Support Load’ reduction

- Static Analysis produced defect counts
  - Good objective measure
    - Reproducible
    - Consistent
    - Low effort to collect
    - Automatable
    - “Static Analysis Tools as Early Indicators of Pre-Release Defect Density” - Microsoft Research Paper

- Numbers of Bug Reports
  - Potentially useful if all other factors are controlled
  - Not the case in our example
    - Multiple development branches
    - Concurrent new development during defect resolution
    - Userbase changes over time
    - Platform support changes over time

- Defects confirmed as ‘real’ by the developers
  - A high False Positive rate would bring the defect count metric into question
  - Would also affect future developer trust in the analysis tool

- Anecdotal comments by developers
  - Informative, but not comparable between projects
- Community feedback
  - Dependent on the nature of each project's community

- ‘Support Load’ reduction
  - Difficult to quantify in an open source environment, due to the variety of support channels

As in most engineering problems...

What do you want to minimize?

- Immediate Cost
- Long Term Cost
- Time
- Manpower
- Ongoing Support

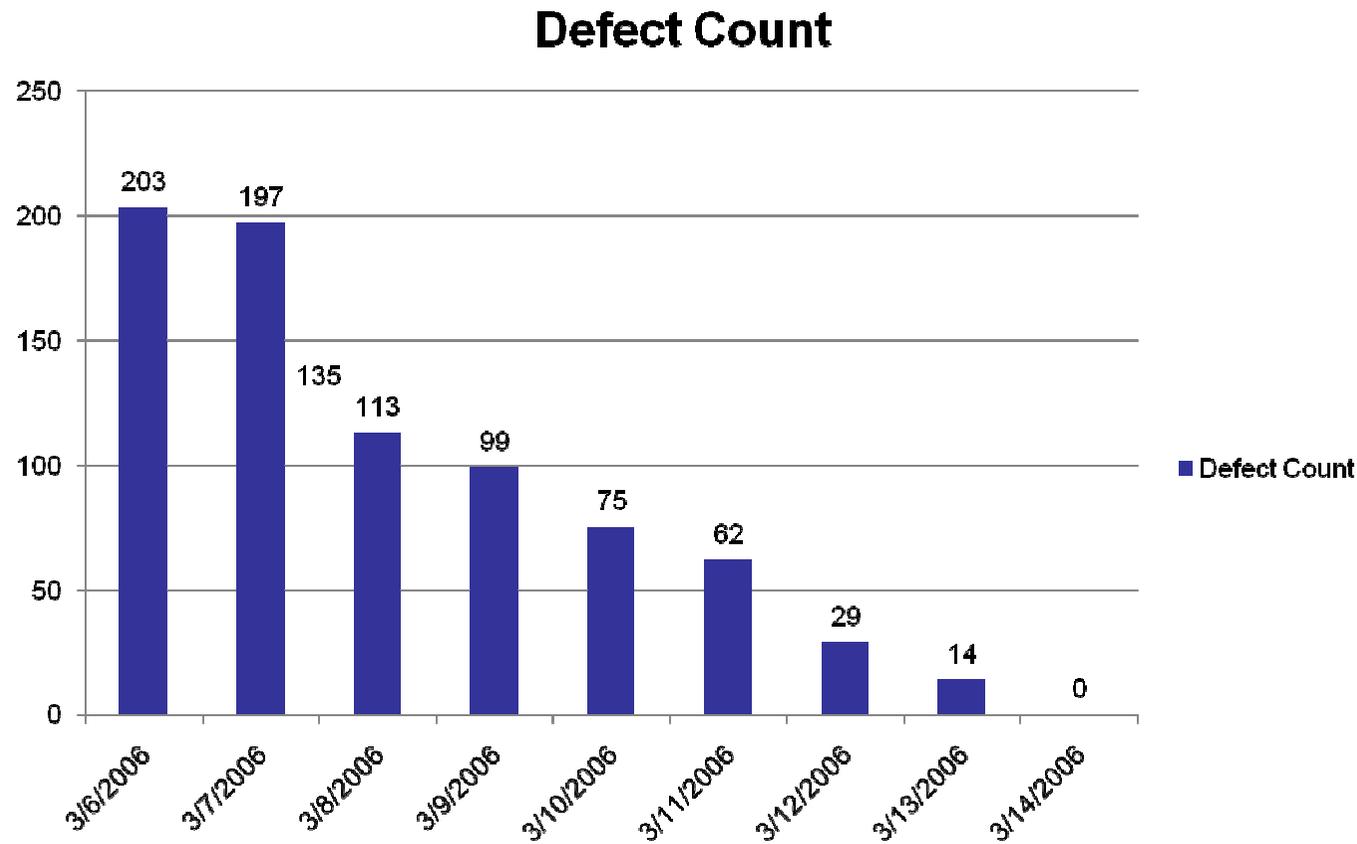


- Samba
  - Open source networking suite
  - Provides Microsoft protocol compatibility
  - International team, started in Australia
  - Project founded in 1992
  - ~300KLOC -> 850KLOC 2006-now



- Started regular scanning March 2006
- 14 Developers accessing the results
- Database available 24/7, SAAS
- New analysis every 2 days on average
  - (797 builds in database)

- Static Analysis defect counts, 310KLOC



# Use Case – Samba & Coverity

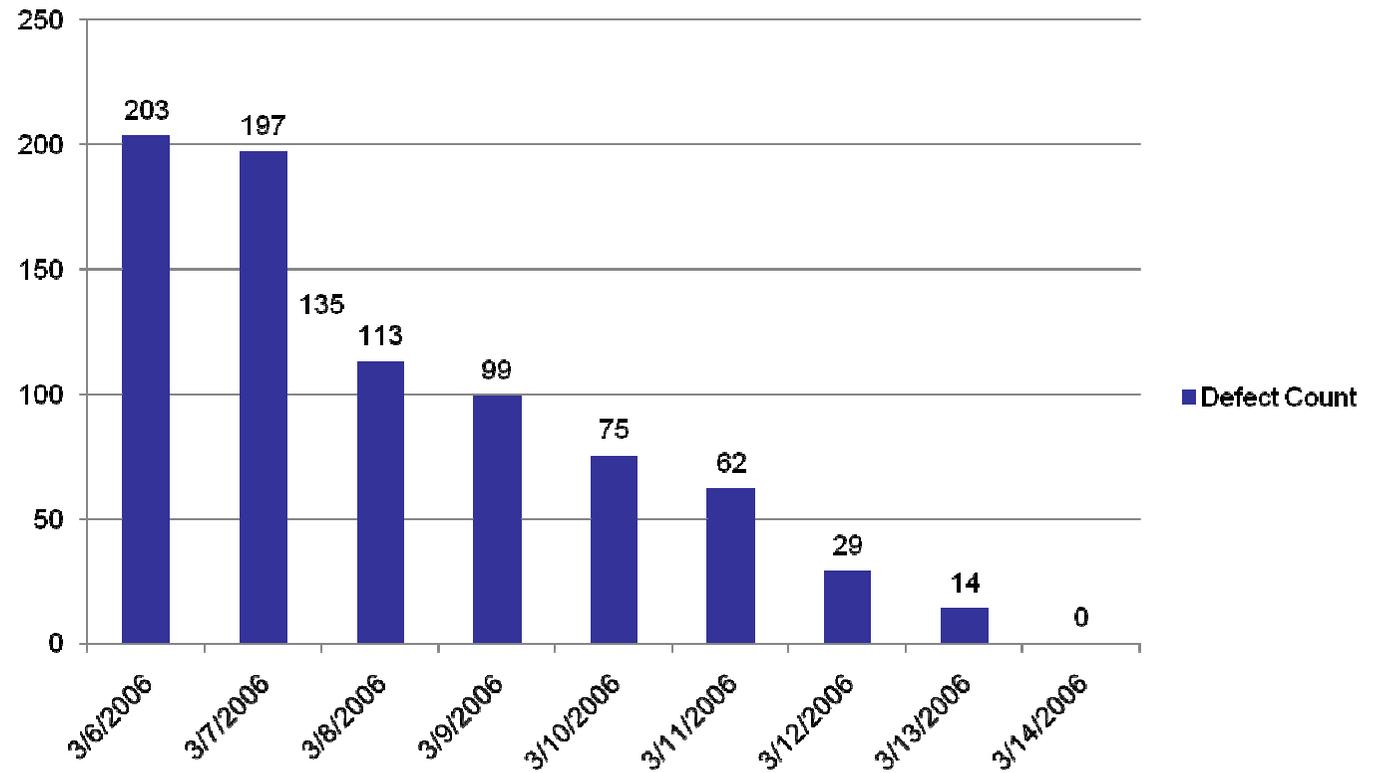


## Defect Count

Day 1: Fixed

4 NULL Pointer derefs  
10 Resource leaks  
1 Uninitialized data  
31 Use after free

But – other changes  
that day introduced  
new defects



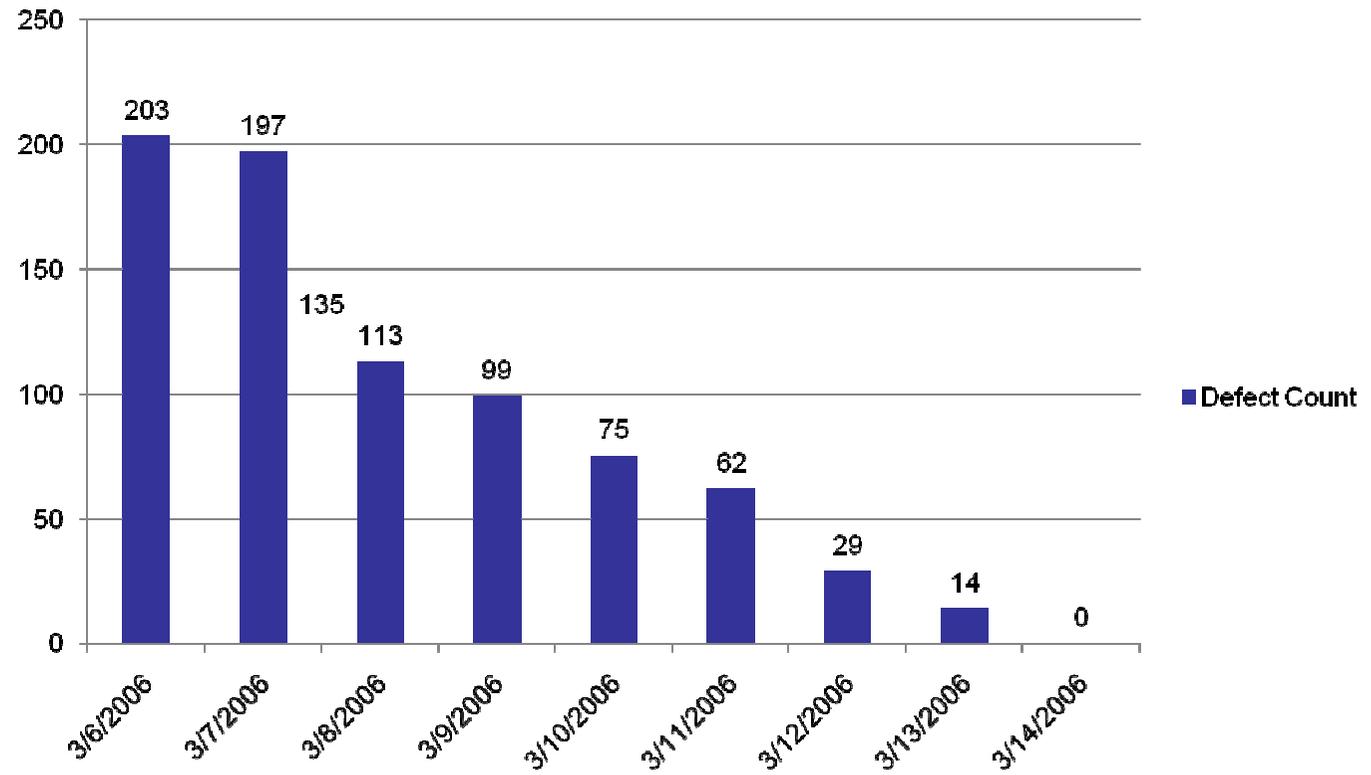
# Use Case – Samba & Coverity



## Defect Count

Day 2: Fixed

- 15 NULL Pointer derefs
- 4 Resource leaks
- 1 static buffer overrun
- 53 Use after free
- 3 returned NULL
- 2 bad comparison
- 1 Dead code



```
118  if (!brl_lock) {  
119          return False;  
120  }
```

Event **func\_conv**: Suspicious implicit conversion to function pointer:

```
"&brl_lock == 0";
```

did you intend to call the function?

```
118  if (!brl_lock) {  
119          return False;  
120  }
```

# Use Case – Samba & Coverity

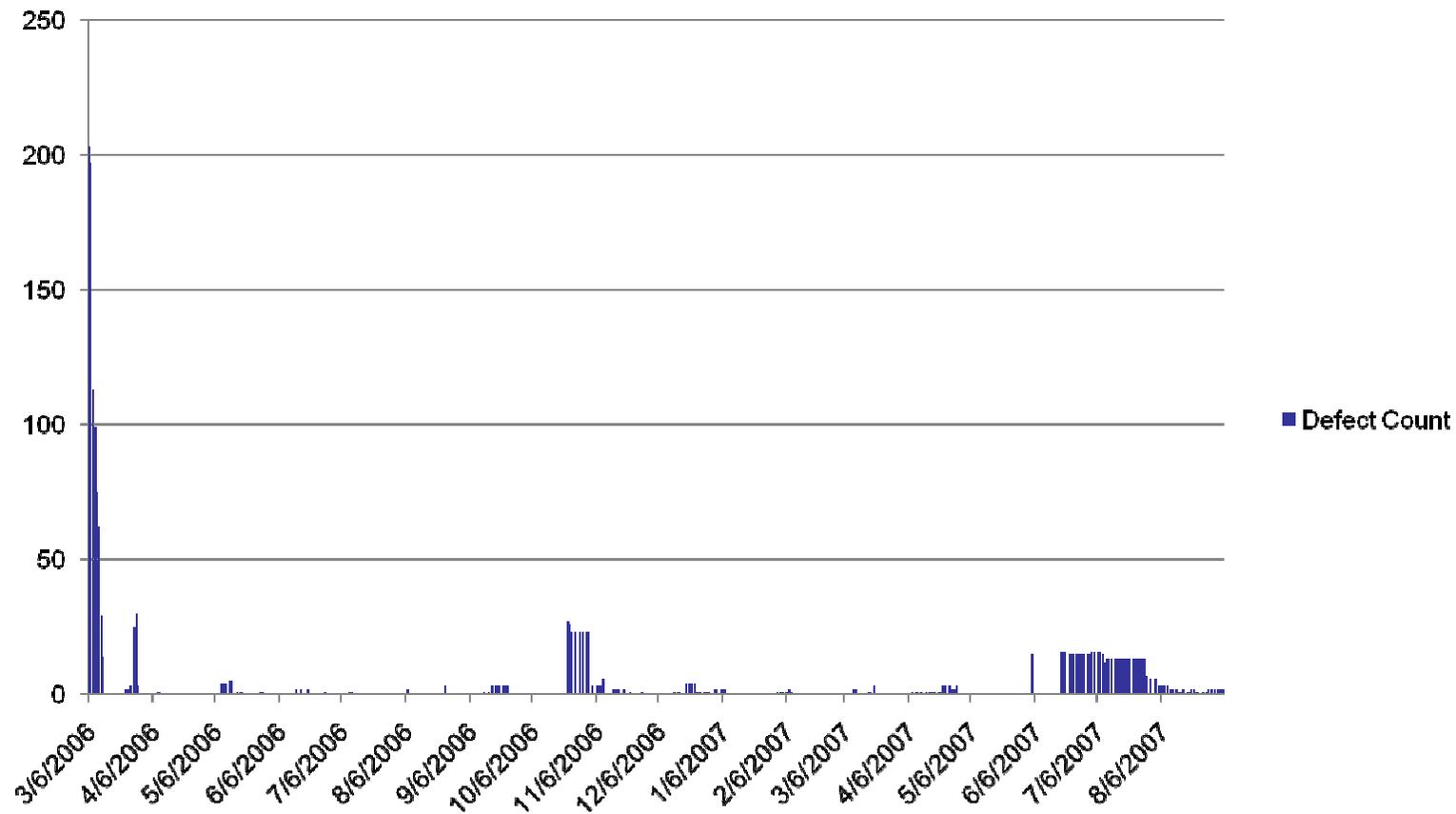


```
688 /*****
689 Lock a range of bytes.
690 *****/
691
692 NTSTATUS brl_lock(struct byte_range_lock *br_lck,
693                 uint16 smbpid,
694                 struct process_id pid,
695                 br_off start,
696                 br_off size,
697                 enum brl_type lock_type,
698                 enum brl_flavour lock_flav,
699                 BOOL *my_lock_ctx)
700 {
701     NTSTATUS ret;
702     struct lock_struct lock;
703
704     *my_lock_ctx = False;
```

# Use Case – Samba & Coverity



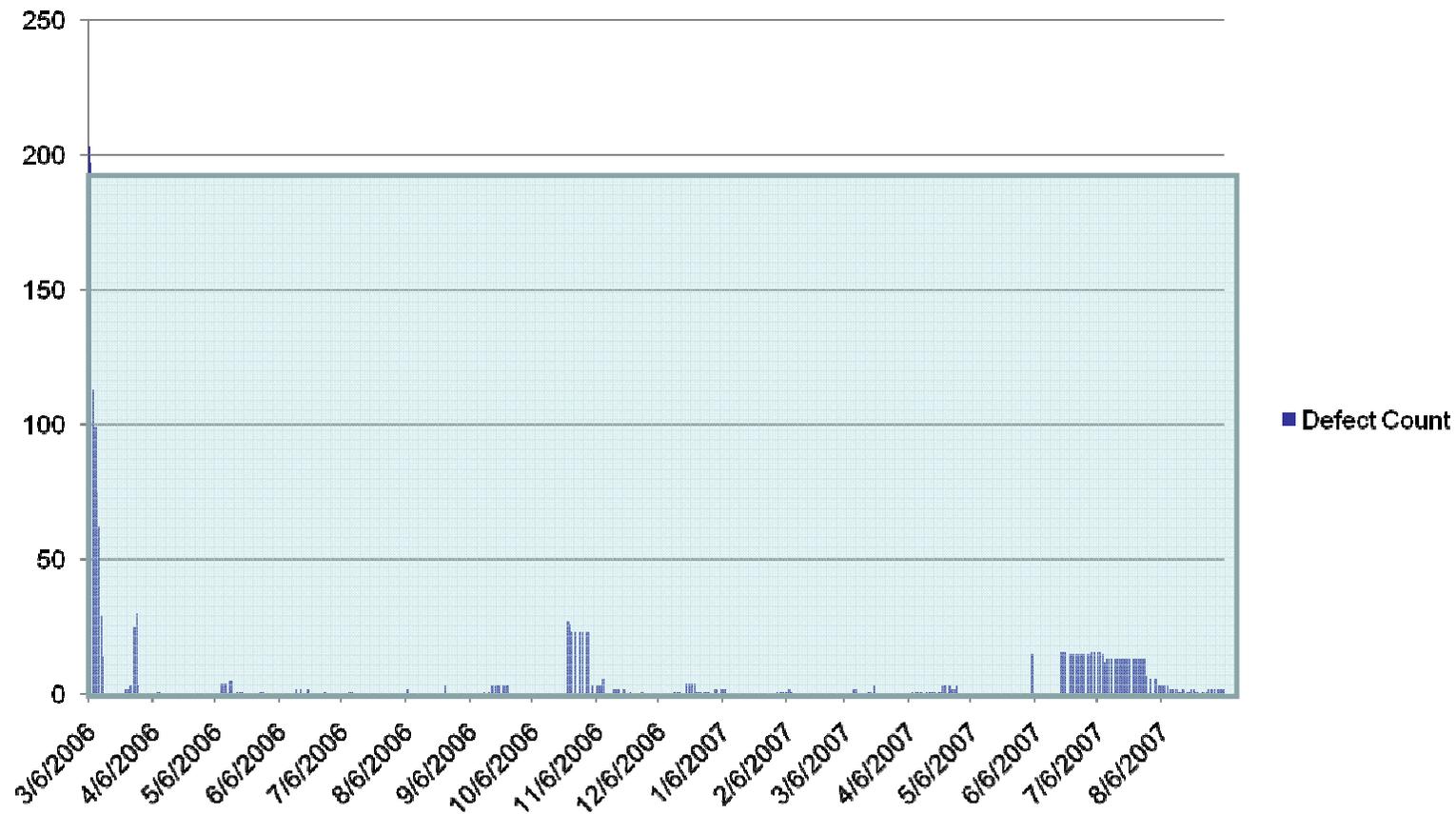
## Defect Count



# Use Case – Samba & Coverity



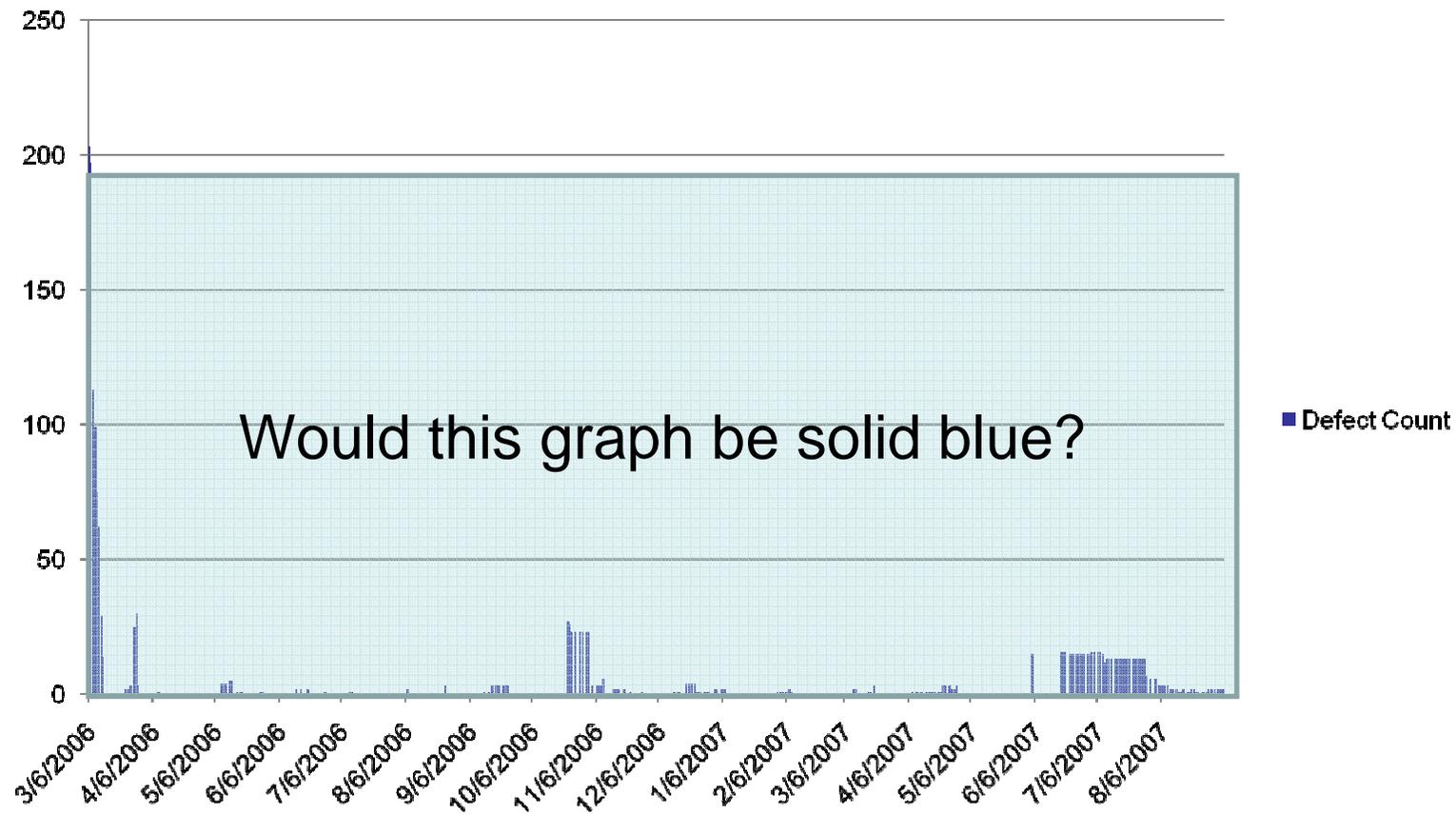
## Defect Count



# Use Case – Samba & Coverity



## Defect Count



- Defects confirmed as 'real' by the developers

13 defects marked False Positive

216 total defects

$$13 / 216 = 6\%$$

- Subjective measures
  - Anecdotal comments by developers

“This tool has become part of our process”

“Using [...] source code analysis technology is like having a developer on the team with an inhuman attention to detail, who points out all the corner cases and boundary conditions developers didn’t consider when they first wrote the code.”

“I code more carefully, because I know my laziness will be caught and embarrass me.”

- Community feedback

- **Community feedback**
  - Invited to give opening keynote at annual Samba conference in 2009

# Open Source Reports



- Whitepaper series - <http://scan.coverity.com/report/>
  - Open Source Report 2008
  - Open Source Report 2009

## Most Commonly Found Defects

Looking at the most commonly found defects in code can help formulate ideas about what kind of code constructs may cause developers to make more errors. A type of defect might be more frequently found because it involves code constructs that are harder to understand, more frequently used, or involve a hard-to-use programming interface. Programming defects at this level are often the root cause of crashes, security vulnerabilities, and other program misbehavior.

### Ranking Defect Types

From the inception of the Scan site in 2006 until May 2008, Scan discovered an aggregate of 27,752 defects among all the open source projects participating in Scan. That number increased to a total of 38,453 defects found by August 2009. In this section, we consolidated all defects across all participating open source projects and categorized them by defect types. The results are shown in the following table.

Defect Type	2008 Frequency	2009 Frequency	% Difference	Ranking Change
NULL Pointer Dereference	27.95%	27.81%	0.14% ↓	0
Resource Leak	25.73%	23.34%	2.39% ↓	0
Unintentional Ignored Expressions	9.76%	9.71%	0.05% ↓	0
Use Before Test (NULL)	8.09%	8.35%	0.25% ↑	-1
Use After Free	6.46%	5.91%	0.34% ↓	-1
Buffer Overflow (statically allocated)	6.14%	5.79%	0.55% ↓	-1
Unsafe Use of Returned NULL	5.85%	5.30%	0.55% ↓	-1
Uninitialized Values Read	5.50%	8.41%	2.91% ↑	+4
Unsafe Use of Returned Negative	3.72%	3.90%	0.18% ↑	0
Type and Allocation Size Mismatch	0.62%	1.10%	0.48% ↑	0
Buffer Overflow (dynamically allocated)	0.31%	0.21%	0.10% ↓	0
Use Before Test (negative)	0.21%	0.18%	0.03% ↓	0

Footnote: For more descriptive information on these defect types, see Appendix D.

The 2008 Scan results in the above table are generated using the 2006 version of Coverity Static Analysis. This 2006 version was also used for the 2009 analysis of open source projects on Rung 1. However, for projects on Rung 2, a newer version of Coverity Static Analysis was used to generate the 2009 results. This accounts for the two largest changes in the distribution of results from 2008 to 2009. Projects on Rung 2 had resolved all of their earlier identified resource leaks, so the latest runs have a smaller percentage of that type of defect. Improvements to the newer version of Coverity Static Analysis used for Rung 2 projects identify a large number of additional cases of uninitialized values, leading to the increase in that category between 2008 and 2009. Viewed alone, that one capability change raised uninitialized value errors to over 21% of the defect distribution for projects now on Rung 2.

## Frequency of Functions by Length

The graph is quite smooth until roughly the 200 LOC mark. At that point, it becomes rougher, and the data points become more sparse. This is an indication of where our source code data set begins to become more sparse. As the Scan data set continues to grow, we would expect to see a very gradual extension of the smooth area.

While the 2008 report showed numbers that were aggregated on a per-project basis, and happened to find that averages across all projects were in the neighborhood of a modern programmer's screen size, a close-up look at the function distribution shows no discontinuity near the screen-length functions. If programmers did have any significant tendency to break functions up, by refactoring, when the screen-length boundary is passed, then the graph should show one or more sudden drops at the common screen lengths as the functions over that size are broken up, and contribute to the distribution of shorter functions to the left of the limit.

Since the graph is clearly smooth, with no such drops, programmers either have no such tendency to break functions at screen length, or do so rarely enough for it to have no viable impact.

- Questions?