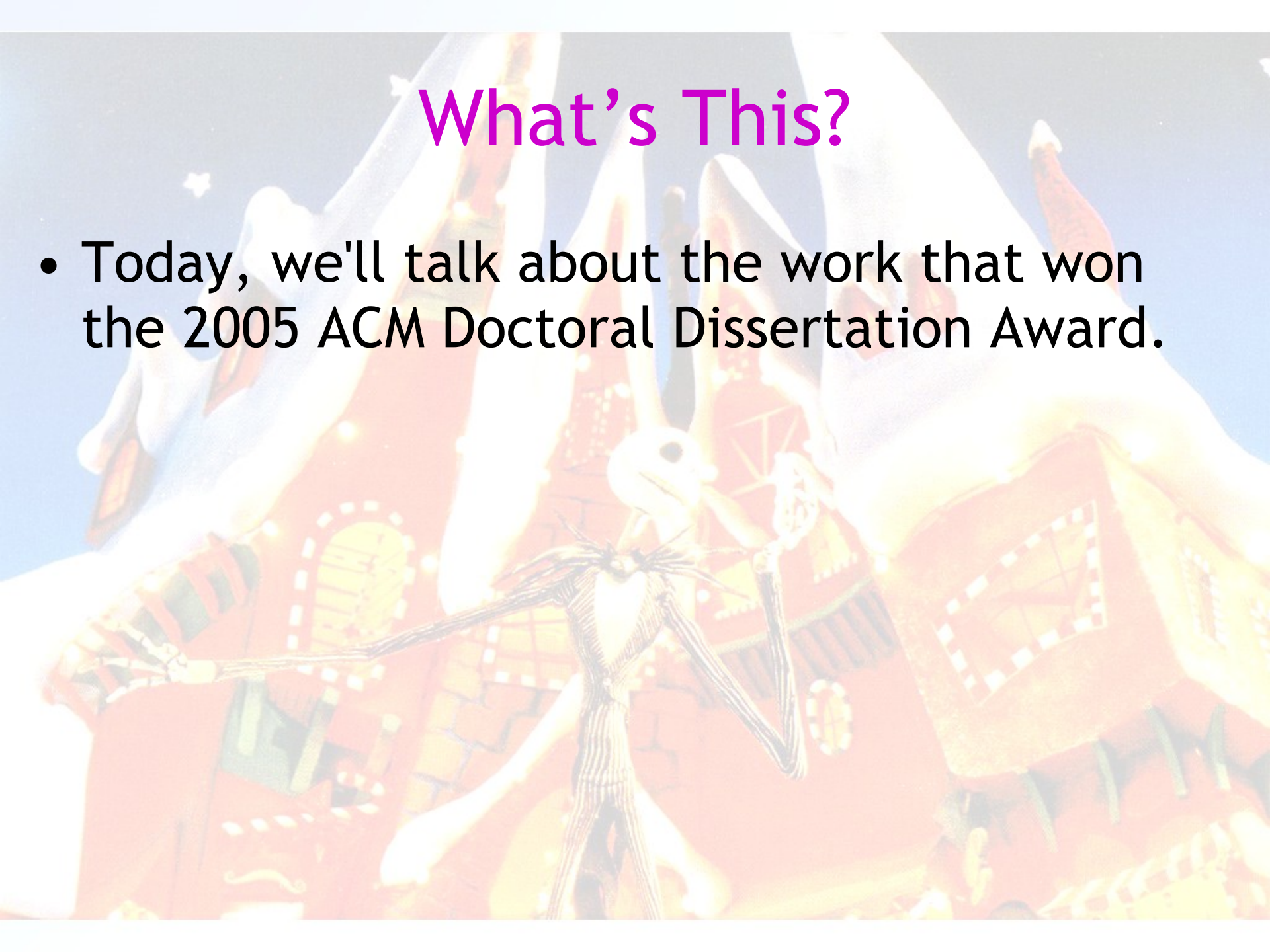




# What's This?

- Today, we'll talk about the work that won the 2005 ACM Doctoral Dissertation Award.



# But First, Set The Stage

- Tell me about Jones and Harrold's *Empirical evaluation of the Tarantula automatic fault-localization technique*
  - It was in the required reading ...
- Input? Static or Dynamic? Output? Spectrum?



# One-Slide Summary

- **Fault localization** reduces the costs associated with debugging by indicating suspicious portions of the code.
- **Cooperative bug isolation** is a distributed, statistical approach to fault localization in deployed software. It uses **sampling** to reduce the particular overhead on any one user.
- The deployment of this technology is a good case study in **technology transfer** of program analysis techniques.

# Sic Transit Gloria Raymondi

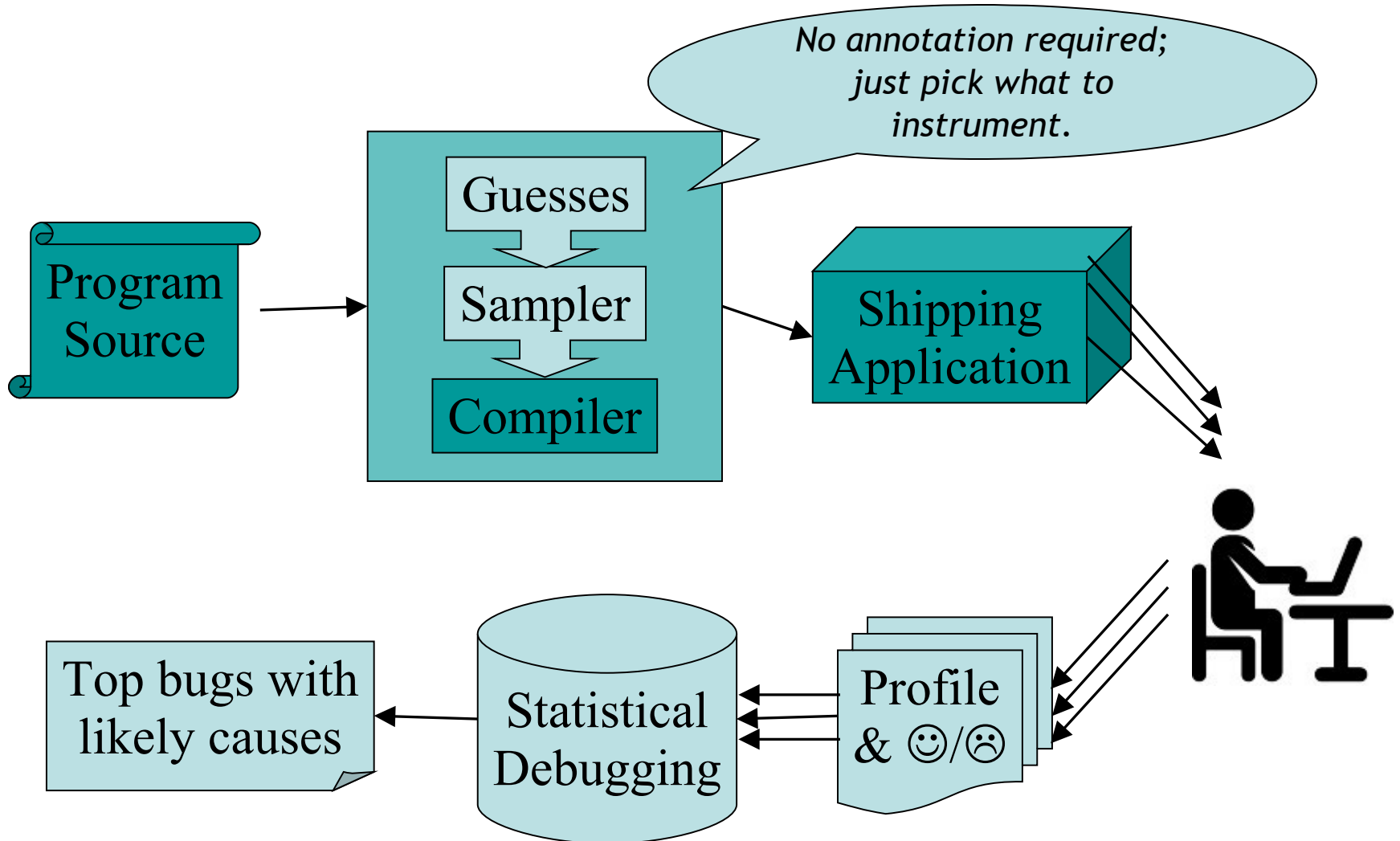
Eric S. Raymond: *“Given enough eyeballs, all bugs are shallow.”*

- Bugs experienced by users matter.
- We can use information from user runs of programs to find bugs.
- Random sampling keeps the overhead of doing this low.
- Large public deployments exist.

# Today's Goal: Measure Reality

- We measure bridges, airplanes, cars...
  - Where is flight data recorder for software?
- Users are a vast, untapped resource
  - 60 million XP licenses in first year; 2/second
  - +7 million Office 365 subscribers 2019 → 2020
  - Users know what matters most
    - Nay, users *define* what matters most!
- Opportunity for *reality-directed* debugging
  - Implicit bug triage for an imperfect world

# Bug Isolation Architecture



# Why Will This Work?

- Good News: Users can help!
- Important bugs happen often, to many users
  - User communities are big and growing fast
  - **User runs vastly exceed testing runs**
  - Users are networked
- *We can* do better, with help from users!
  - cf. crash/telemetry reports (Microsoft, Chrome)
  - Today: research efforts

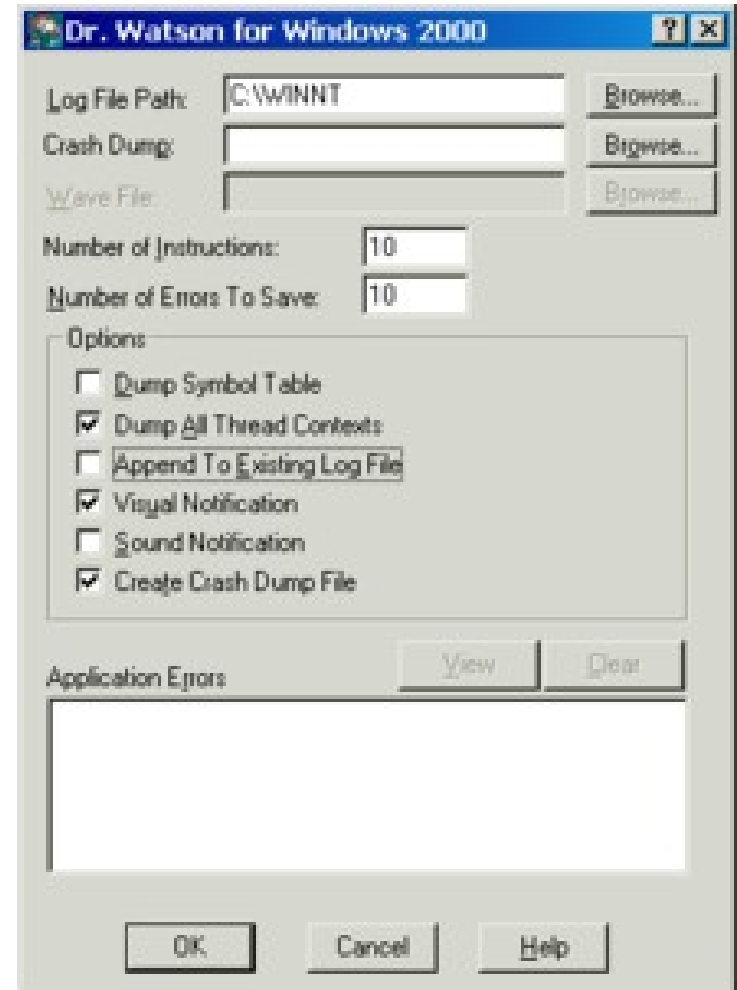


“There are no significant bugs in our released software that any significant number of users want fixed.”

-- *Bill Gates in 1995*

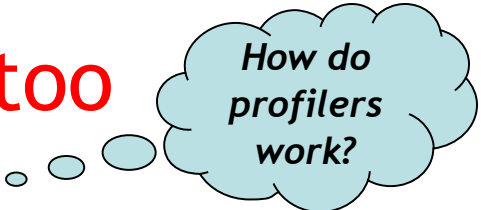
# Crash Reports

- In use since the mid 1990s
- Stack trace, memory address, details about host configuration, ...
- Advantage: fast and easy
- Limitations:
  - Crashes don't always occur “near” the defect
  - Hard to tell which crashes correspond to the same bug



# Let's Use Randomness

- Problem: recording everything is too expensive!

A light blue thought bubble with a black outline, containing the text "How do profilers work?". It is connected to the main text by three smaller circles of decreasing size.

How do profilers work?

- Idea: each user records 0.1% of everything
- Generic **sparse sampling framework**
  - Adaptation of Arnold & Ryder
- Suite of instrumentations / analyses
  - Sharing the cost of assertions
  - Isolating deterministic bugs
  - Isolating non-deterministic bugs

# Sampling Blocks

- Consider this code:

```
check(p != NULL);
```

```
p = p -> next;
```

```
check(i < max);
```

```
total += sizes[i];
```

- We want to sample 1/100 of these checks

# Global Counter

- Solution?
  - Maintain a global counter modulo 100

```
for (i=0; i<n; i++) {  
    check(p != NULL);  
    p = p -> next;  
    check(i < max);  
    total += sizes[i];  
}
```

# Global Counter

- Solution?
  - Maintain a global counter modulo 100

```
for (i=0; i<n; i++) {  
    check(p != NULL);  
    p = p -> next;  
    check(i < max);  
    total += sizes[i];  
}
```

**Records the first  
check 1/50 times.**

**Never records  
the other.**

# Random Number Generator

- Solution? Use random number generator.

```
if (rand(100)==0) check(p != NULL);
```

```
p = p -> next;
```

```
if (rand(100)==0) check(i < max);
```

```
total += sizes[i];
```

# Random Number Generator

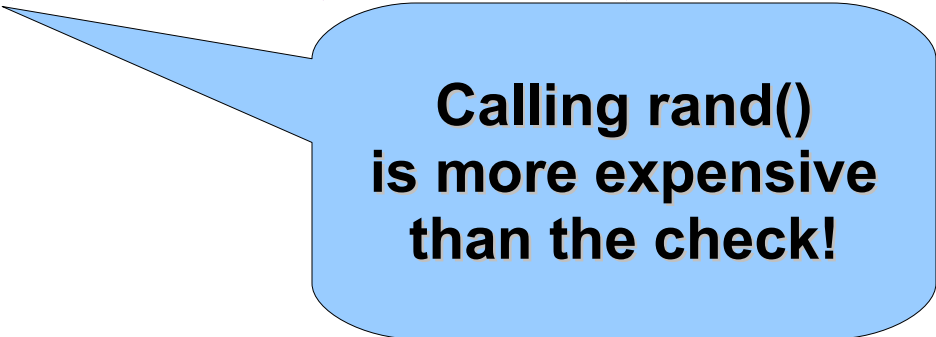
- Solution? Use random number generator.

```
if (rand(100)==0) check(p != NULL);
```

```
p = p -> next;
```

```
if (rand(100)==0) check(i < max);
```

```
total += sizes[i];
```



**Calling rand()  
is more expensive  
than the check!**




# Sampling the Bernoulli Way

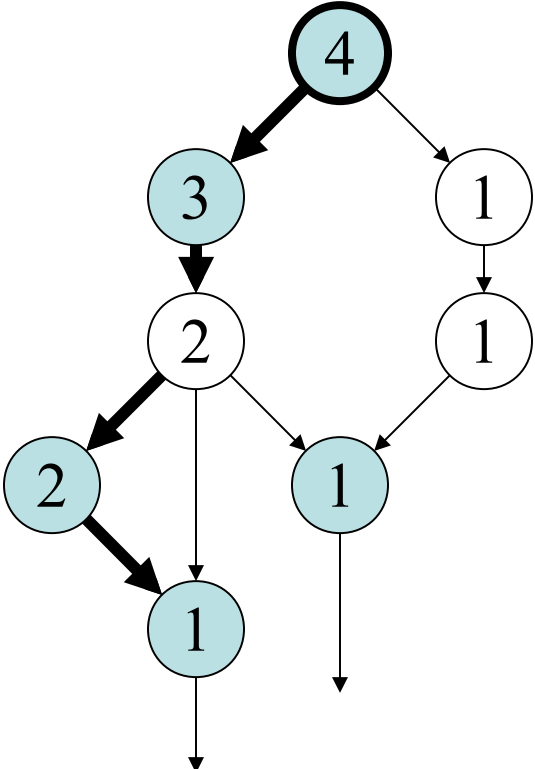
- Identify the points of interest
- Decide to examine or ignore each site...
  - Randomly
  - Independently
  - Dynamically
- ✘ Cannot use clock interrupt: no context
- ✘ Cannot be periodic: unfair
- ✘ Cannot toss coin at each site: too slow

# Anticipating the Next Sample

- Randomized global countdown
- Selected from *geometric distribution*
  - Inter-arrival time for biased coin toss
  - Stores: **How many tails before next head?**
    - i.e., how many sampling points to skip before we write down the next piece of data?
- Mean of distribution = expected sample rate

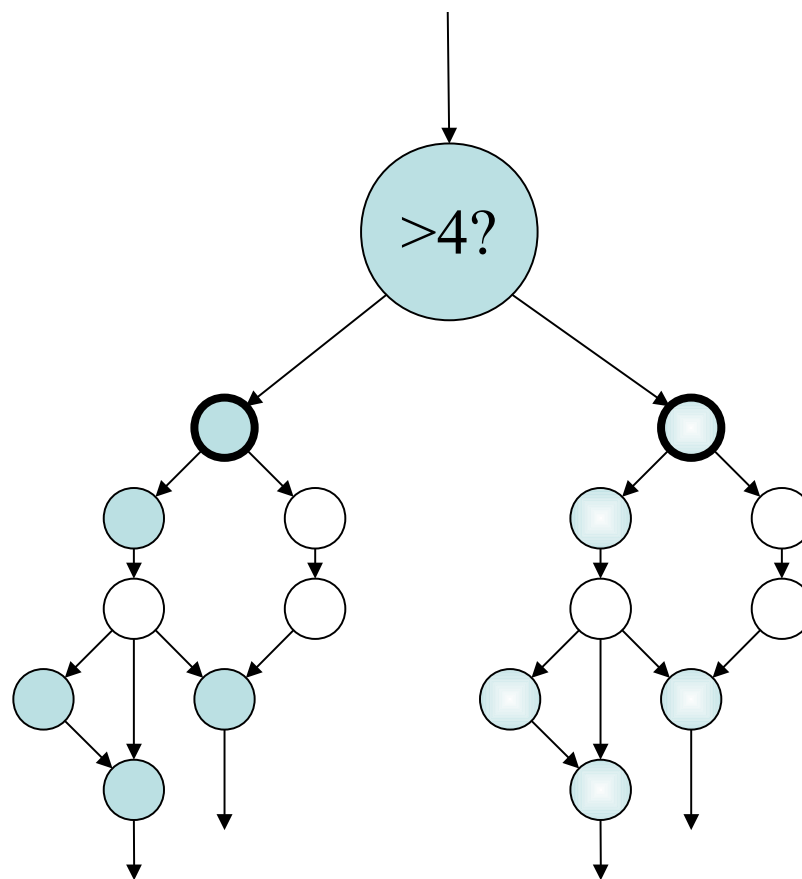
# Amortized Coin Tossing

- Each acyclic region:
  - Finite number of paths
  - Finite max number of instrumentation sites
  - Shaded nodes represent instrumentation sites 



# Amortized Coin Tossing

- Each acyclic region:
  - Finite number of paths
  - Finite max number of instrumentation sites
- Clone each region
  - “Fast” variant
  - “Slow” sampling variant
- Choose at run time



# Two Code Paths

- Fast Code Path

```
if (counter > 2) {  
    p = p -> next;  
    total += sizes[i];  
    counter -= 2;  
}
```

- Slow Code Path

```
if (counter-- == 0)  
    check(p != NULL);  
p = p -> next;  
if (counter-- == 0)  
    check(i < max);  
total += sizes[i];
```

# Optimizations

- Cache global countdown in local variable
  - Global → local at func entry & after each call
  - Local → global at func exit & before each call
- Identify and ignore “weightless” functions
- Avoid cloning
  - Instrumentation-free prefix or suffix
  - Weightless or singleton regions
- Static branch prediction at region heads
- Partition sites among several binaries
- Many additional possibilities ...

# Chinese Media



- This 2017 political drama received official support and funding as part of Xi Jinping's anti-corruption campaign and thus featured lighter censorship. It had the highest single-day ratings for a Chinese drama and was highly mentioned on social media, but has been criticized as propaganda. It features stories about power struggles and Chinese politics that were not previously talked about on mainstream television.

# Physics

- *This* is one of the four fundamental interactions. It holds most ordinary matter together because it confines quarks into protons and neutrons, as well as binding these neutrons and protons to create atomic nuclei. It operates over very small distances. After the discovery of the neutron in 1932 (Chadwick), it was proposed as early as 1935 (Yukawa, who predicted the meson, which was found in 1947 leading to the Nobel Prize in 1949) to explain why nuclei do not fly apart.



# Q. Computer Science

- Along with Shamir and Adleman, this American Turing-award winner is credited with revolutionizing public key cryptography. He is also responsible for the RC5 symmetric key encryption algorithm (“RC” stands for “*his* Cypher”) and the MD5 cryptographic hash function. He is also a co-author of *Introduction to Algorithms* (aka CLRS).

# Q. Philosophy and History

- This Greek (Macedonian) philosopher and scientist is known for writings on many subjects (physics, biology, logic, ethics, rhetoric, etc.). His works form the first comprehensive Western philosophy. Encyclopaedia Britannica claims he, “was the first genuine scientist in history ... [and] every scientist is in his debt.” He is credited with the earliest study of formal logic. Kant said that his theory of logic completely accounted for the core of deductive inference. His *Rhetoric* has been called “the most important single work on persuasion ever written.”

# Sharing the Cost of Assertions

- Now we know how to sample things.
- Does this work in practice?
  - Let's do a series of experiments.
- First: microbenchmark for sampling costs!
- What to sample: `assert()` statements
- Identify (for debugging) assertions that
  - *Sometimes fail* on bad runs
  - But *always succeed* on good runs

# Case Study: CCured Safety Checks

- Assertion-dense C code
- Worst-case scenario for us
  - Each assertion extremely fast
- No bugs here; purely performance study
  - Unconditional: 55% average overhead
  - $1/100$  sampling: 17% average overhead
  - $1/1000$  sampling: 10% average; half below 5%

# Isolating a Deterministic Bug

- Guess predicates on **scalar function returns**  
 $(f() < 0)$        $(f() == 0)$        $(f() > 0)$
- Count how often each predicate holds
  - Client-side reduction into counter triples
- Identify differences in good versus bad runs
  - Predicates *observed true* on some bad runs
  - Predicates *never observed true* on any good run

Triples about return values aren't the only things we can sample.

# Case Study: `ccrypt` Crashing Bug

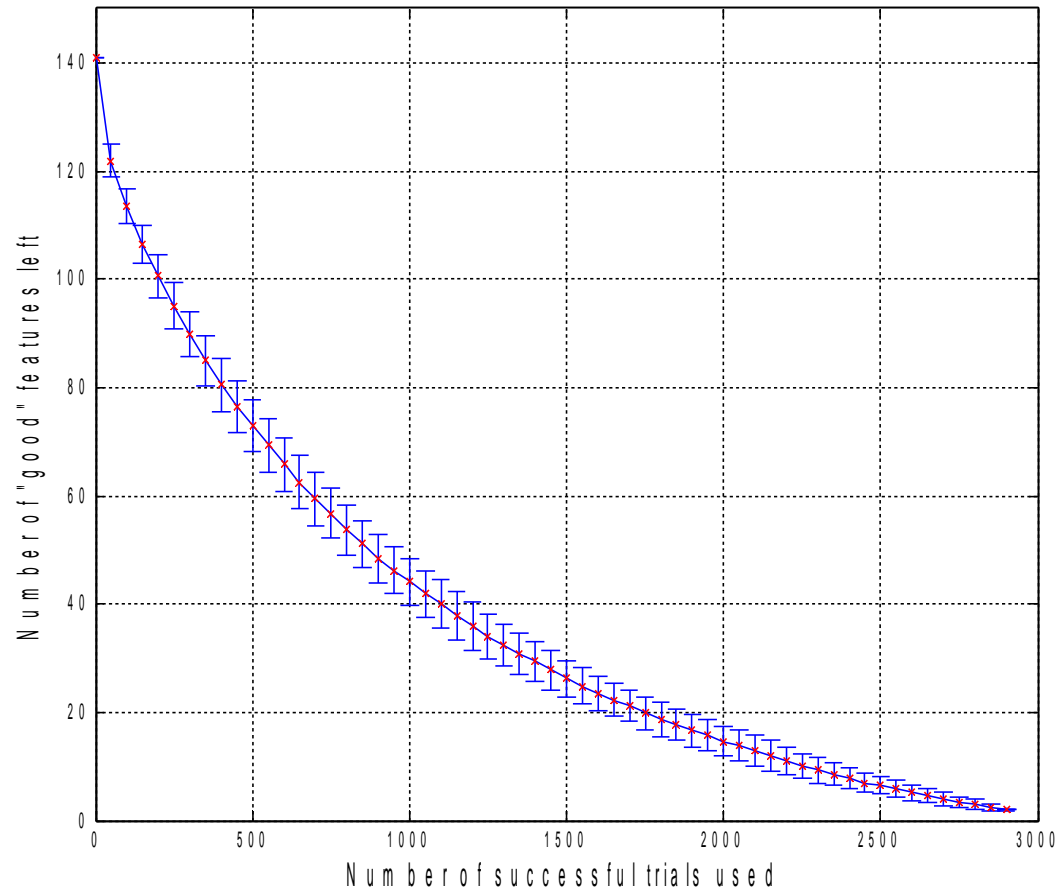
- 570 call sites
- $3 \times 570 = 1710$  counters
- Simulate large user community
  - 2990 randomized runs; 88 crashes
- Sampling density  $1/1000$ 
  - Less than 4% performance overhead
- Recall goal: sampled predicates should make it easier to debug the code ...

# Winnowing Down to the Culprits

- 1710 counters
- 1569 are always zero
  - 141 remain
- 139 are nonzero on some successful run
- Not much left!

```
file_exists() > 0  
xreadline() == 0
```

How do these pin down the bug? You'll see in a second.



# Isolating a Non-Deterministic Bug

- Guess: at each **direct scalar assignment**  
 $\mathbf{x} = \dots$
- For each same-typed in-scope variable  $\mathbf{y}$
- Guess predicates on  $\mathbf{x}$  and  $\mathbf{y}$   
 $(\mathbf{x} < \mathbf{y})$        $(\mathbf{x} == \mathbf{y})$        $(\mathbf{x} > \mathbf{y})$
- Compare: DIG, Daikon (invariant detection)
- Count how often each predicate holds
  - Client-side reduction into counter triples

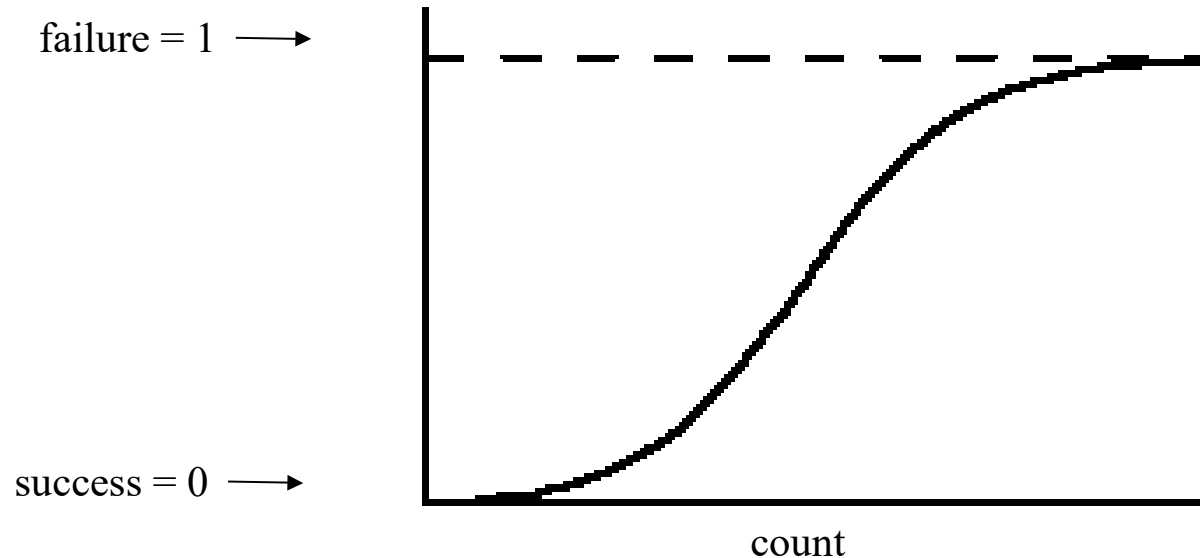


# Case Study: bc Crashing Bug

- Hunt for intermittent crash in `bc-1.06`
  - Stack traces suggest heap corruption
- 2729 runs with 9MB random inputs
- 30,150 predicates on 8910 lines of code
- Sampling key to performance
  - 13% overhead without sampling
  - 0.5% overhead with  $1/1000$  sampling



# Statistical Debugging via Regularized Logistic Regression



- S-shaped cousin to linear regression
- Predict success/failure as function of counters
- Penalty factor forces most coefficients to zero
  - Large coefficient  $\Rightarrow$  highly predictive of failure

# Top-Ranked Predictors

```
void more_arrays ()
{
    ...

    /* Copy the old arrays. */
    for (indx = 1; indx < old_count; indx++)
        arrays[indx] = old_ary[indx];

    /* Initialize the new elements. */
    for (; indx < v_count; indx++)
        arrays[indx] = NULL;

    ...
}
```

```
#1: indx > scale
#2: indx > use_math
```

# Top-Ranked Predictors

```
void more_arrays ()
{
    ...

    /* Copy the old arrays. */
    for (indx = 1; indx < old_count; indx++)
        arrays[indx] = old_ary[indx];

    /* Initialize the new elements. */
    for (; indx < v_count; indx++)
        arrays[indx] = NULL;

    ...
}
```

```
#1: indx > scale
#2: indx > use_math
#3: indx > opterr
#4: indx > next_func
#5: indx > i_base
```

# Bug Found: Buffer Overrun

```
void more_arrays ()
{
    ...

    /* Copy the old arrays. */
    for (indx = 1; indx < old_count; indx++)
        arrays[indx] = old_ary[indx];

    /* Initialize the new elements. */
    for (; indx < v_count; indx++)
        arrays[indx] = NULL;

    ...
}
```

# Moving To The Real World

- Pick instrumentation scheme
- Automatic tool instruments program
- Sampling yields low overhead
- Many users run program
- Many reports  $\Rightarrow$  find bug
- So let's do it!



# Multithreaded Programs

- Global next-sample countdown
  - High contention, small footprint
  - Want to use registers for performance
  - ⇒ Thread-local: one countdown per thread
- Global predicate counters
  - Low contention, large footprint
  - ⇒ Optimistic atomic increment

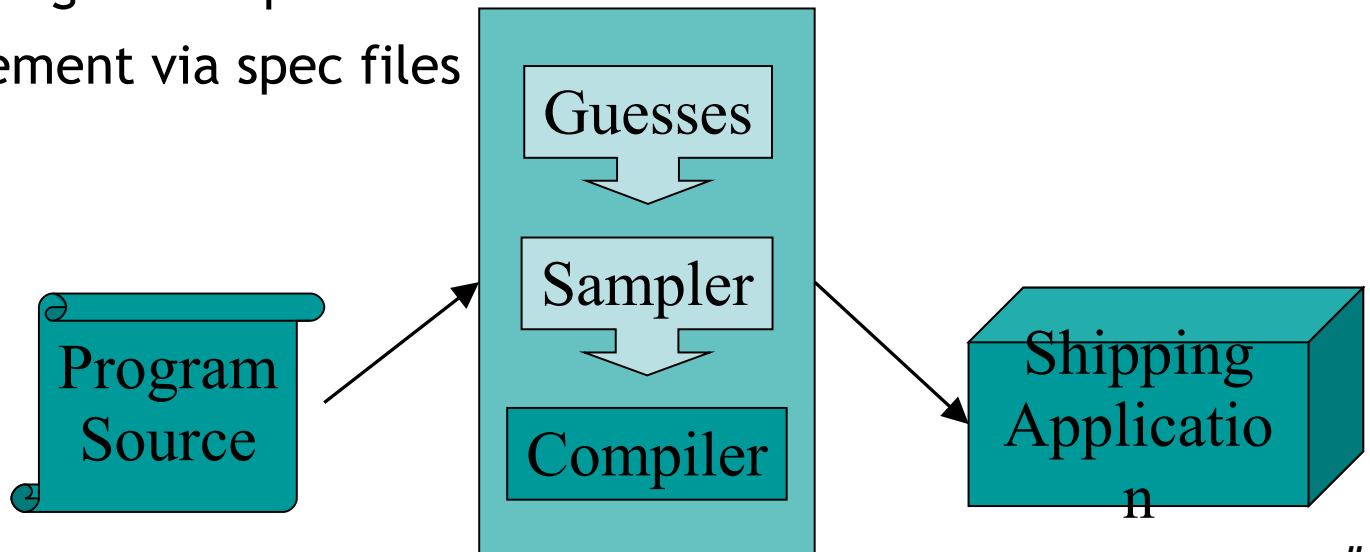
# Multi-Module Programs

- Forget about global static analysis
  - Plug-ins, shared libraries
  - Instrumented & uninstrumented code
- Self-management at compile time
  - Locally derive identifying object signature
  - Embed static site information within object file
- Self-management at run time
  - Report feedback state on normal object unload
  - Signal handlers walk global object registry

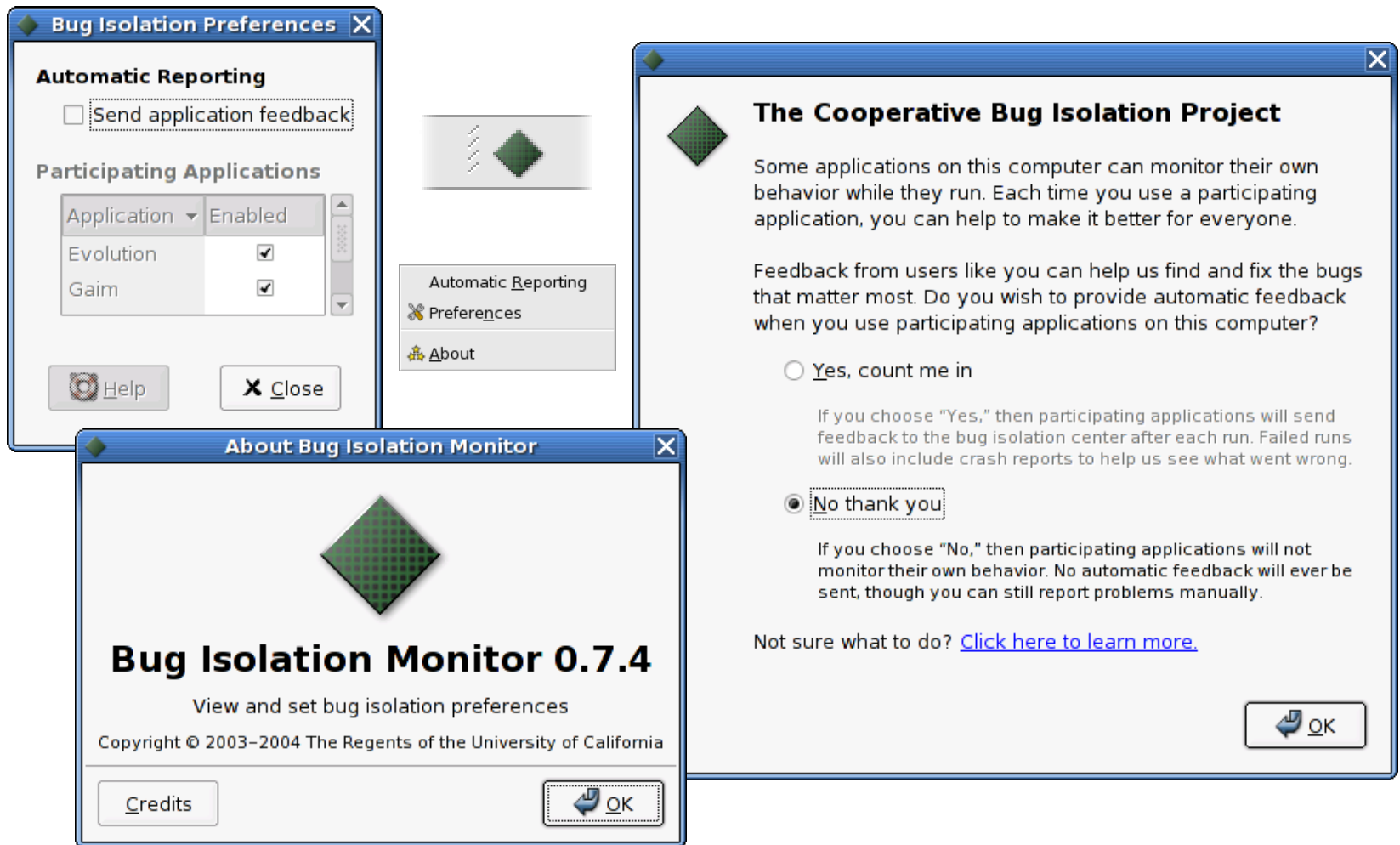


# Native Compiler Integration

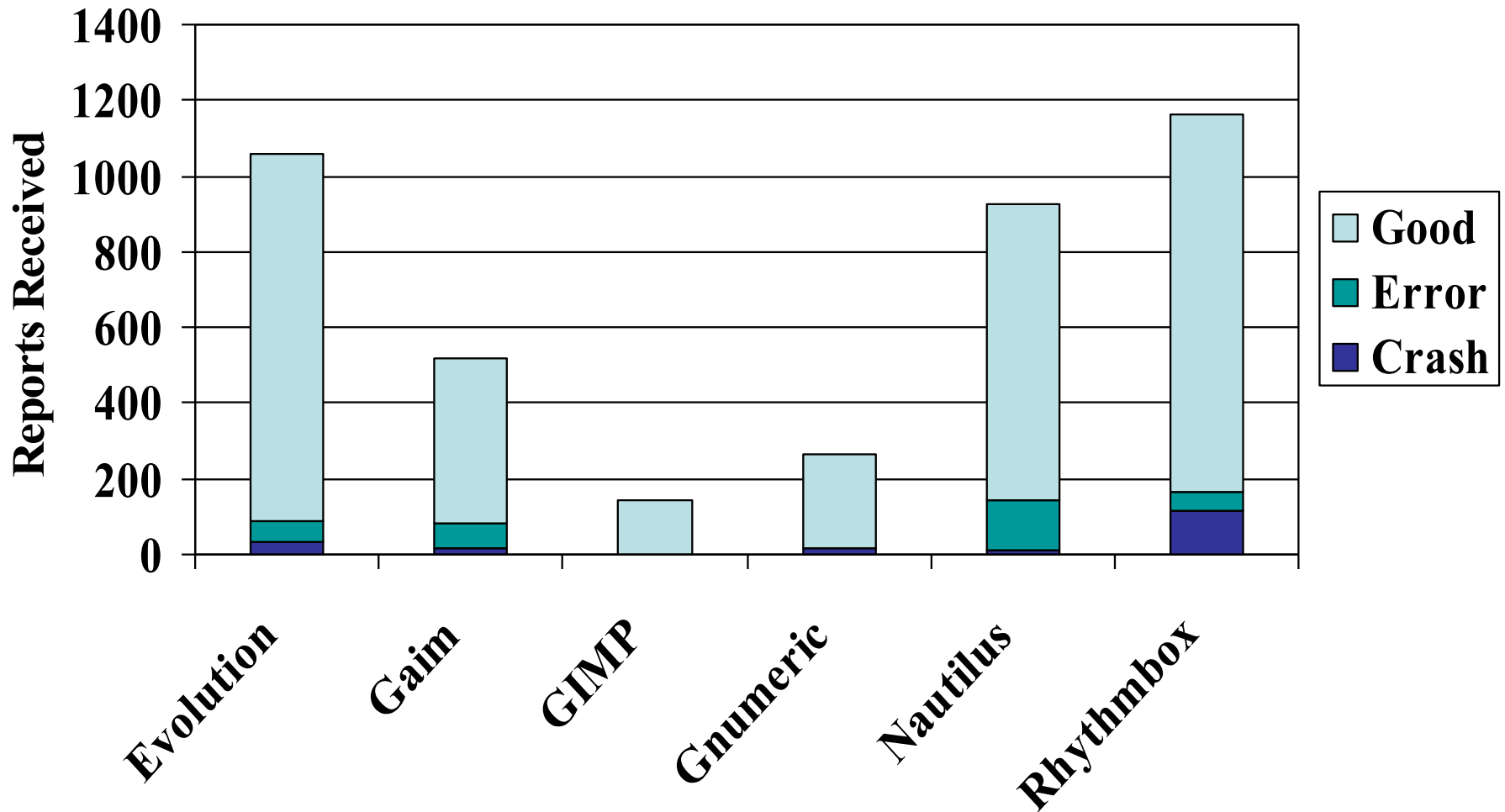
- Instrumentor must mimic native compiler
  - You don't have time to port & annotate by hand
- This approach: source-to-source, then native
- Hooks for GCC:
  - Stage wrapping via scripts
  - Flag management via spec files



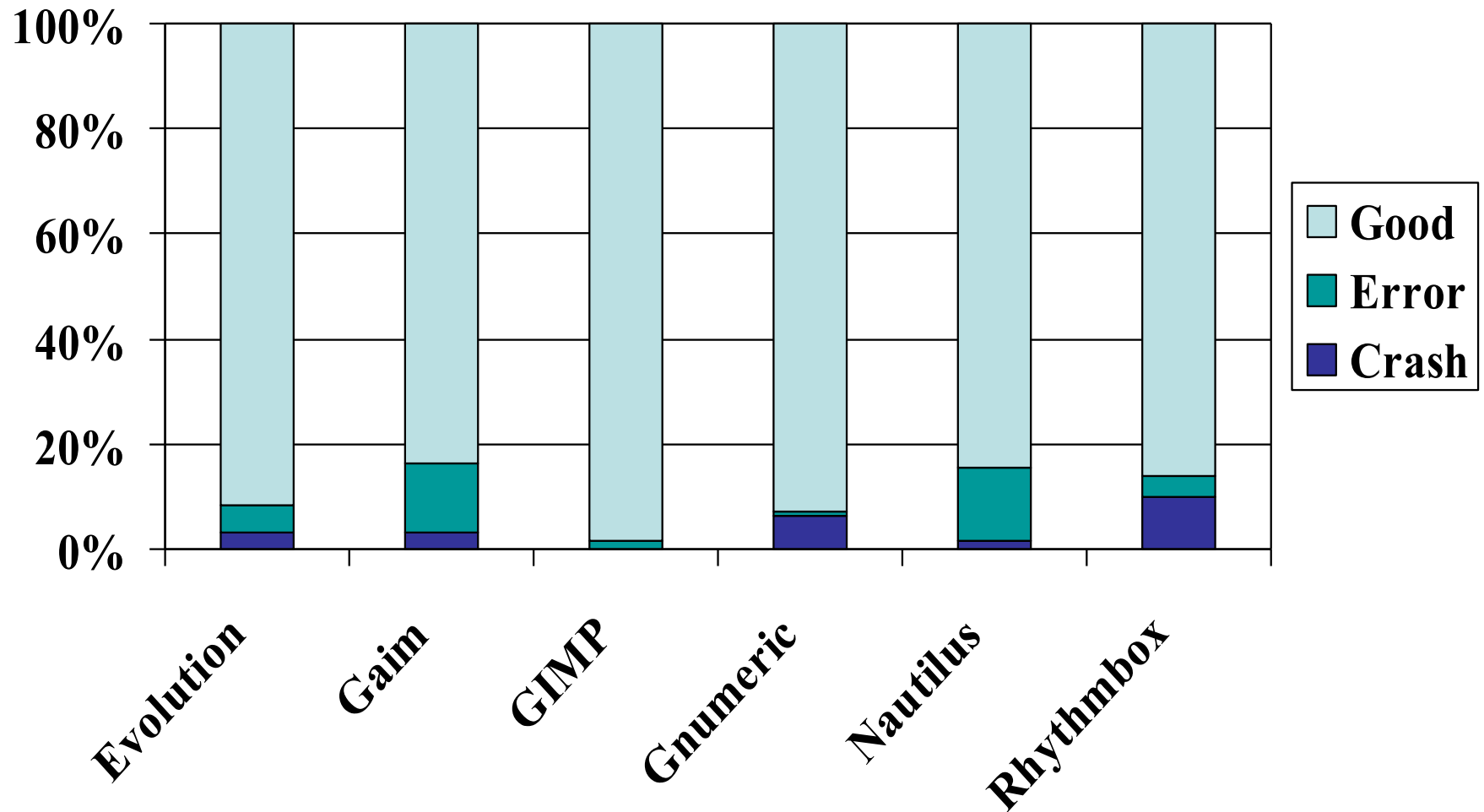
# Keeping the User In Control



# Public Deployment 2004



# Public Deployment 2004



# Sneak Peak: Data Exploration


C:\Documents and Settings\Ben Liblit\Desktop\Rhythmbox results\MR\_lb.html - Microsoft Internet Explorer











File Edit View Favorites Tools Help

Scheme: [\[branch\]](#) [\[return\]](#) [\[scalar\]](#) [\[all\]](#)

Sorted by: [\[lower bound of confidence interval\]](#) [\[increase score\]](#) [\[fail score\]](#) [\[true in # F runs\]](#)

Go to: [\[report summary\]](#) [\[CBI webpage\]](#)



	predicate	function	file:line
	monkey_media_player_get_uri = 0	info_available_cb	<a href="#">rb-shell-player.c:1774</a>
	monkey_media_player_get_uri = 0	info_available_cb	<a href="#">rb-shell-player.c:1765</a>
	rb_entry_view_get_entry_contained = 0	rb_shell_jump_to_entry_with_source	<a href="#">rb-shell.c:2118</a>
	g_source_remove > 0	cddb_disclosure_destroy	<a href="#">disclosure-widget.c:77</a>
	rhythmdb_tree_entry_insert = 0	rhythmdb_tree_parser_end_element	<a href="#">rhythmdb-tree.c:460</a>
	g_hash_table_lookup > 0	rhythmdb_tree_entry_insert	<a href="#">rhythmdb-tree.c:838</a>
	rhythmdb_query_model_entry_to_iter = 0	rb_entry_view_get_entry_contained	<a href="#">rb-entry-view.c:1902</a>
	g_hash_table_lookup = 0	rhythmdb_query_model_entry_to_iter	<a href="#">rhythmdb-query-model.c:870</a>
	remove_child = 0	remove_entry_from_album	<a href="#">rhythmdb-tree.c:1030</a>
	eel_gconf_handle_error > 0	eel_gconf_get_boolean	<a href="#">eel-gconf-extensions.c:107</a>

My Computer

# Summary: Putting it All Together

- Flexible, fair, low overhead sampling
- Predicates probe program behavior
  - Client-side reduction to counters
  - Most guesses are uninteresting or meaningless
- Seek behaviors that co-vary with outcome
  - Deterministic failures: process of elimination
  - Non-deterministic failures: statistical modeling

# Conclusions

- Bug triage that directly reflects *reality*
  - Learn the most, most quickly, about the bugs that happen most often
- Variability is a benefit rather than a problem
  - Results grow stronger over time
- Find bugs while you sleep!
- Public deployment is challenging
  - Real world code pushes tools to their limits
  - Large user communities take time to build

- But the results are worth it:

*“Thanks to Ben Liblit and the Cooperative Bug Isolation Project, this version of Rhythmbox should be the most stable yet.”*

# Homework

- HW6

