Statistical Debugging

Ben Liblit, University of Wisconsin–Madison

Reconstruction of Failing Paths

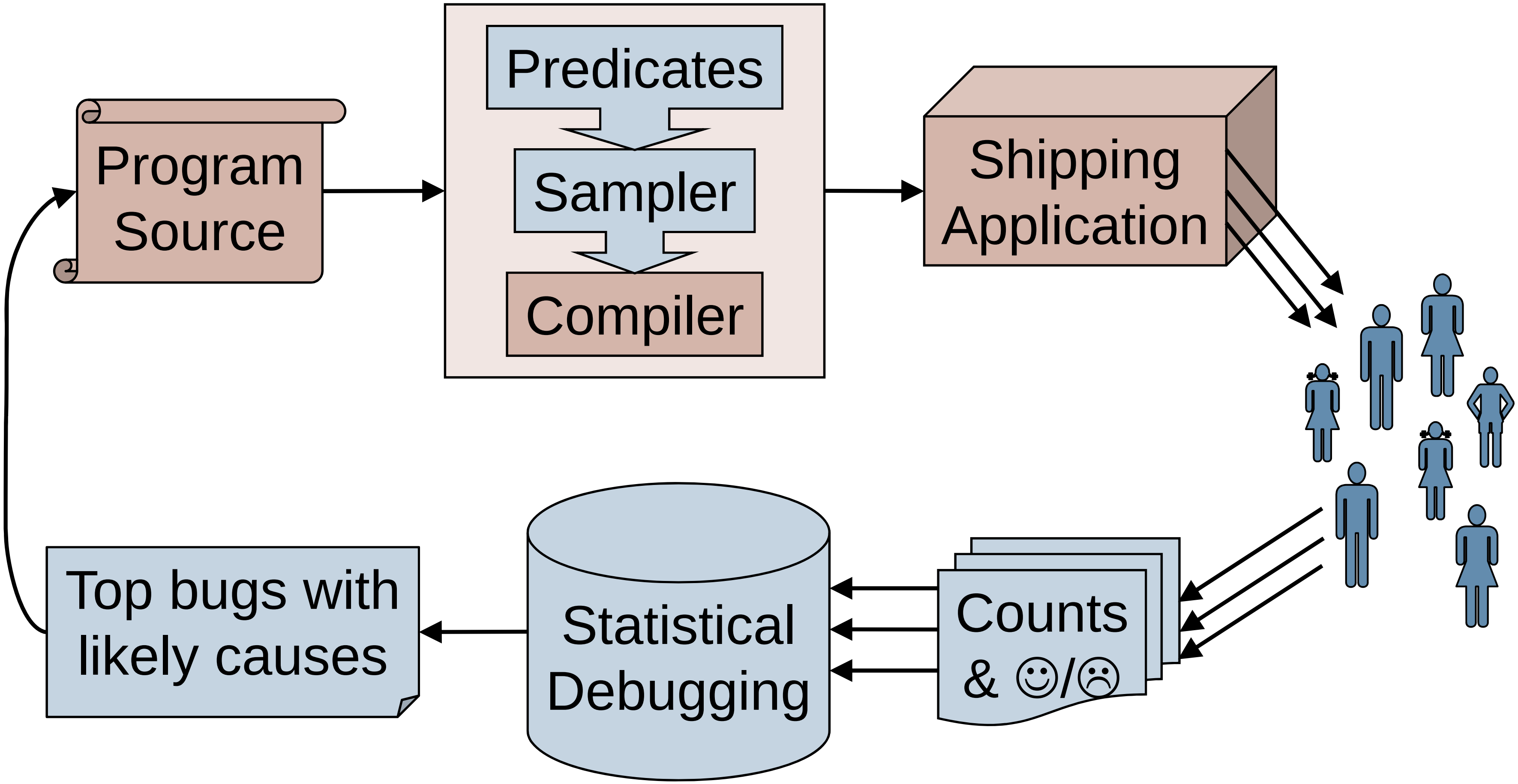
> “Just because it’s undecidable doesn’t mean we don’t need an answer.”

Alex Aiken, as roughly remembered by me

Practical Experiences With CBI

- Bug predictor is often the smoking gun, but not always
- “Redundant” predicates actually carry clues
  - Especially when spread across source code
- Bidirectional thinking can be very tricky
  - Debuggers only train us to think backwards

Putting Predictors in Context



Program Source

Predicates

Sampler

Compiler

Shipping Application

Counts & ☺/☹

Statistical Debugging

Top bugs with likely causes

A Debugging Scenario

```
int **a;

void main()
{
  ...
  process_input(a);
  ...
}

void clear_array(int **a)
{
  for (...)
    a[i] = NULL;
}
```

```
void process_input(int **a)
{
  cin >> input;
  switch (input) {
   case 'e':
       clear_array(a);
       break;
   case 'p':
       ...
   ...
  }
  ...
  a[i][j]++;
}
```

A Debugging Scenario

```
int **a;

void main()
{
  ...
  process_input(a);
  ...
}

void clear_array(int **a)
{
  for (...)
   a[i] = NULL;
}
```

```
void process_input(int **a)
{
  cin >> input;
  switch (input) {
   case 'e':
       clear_array(a);
       break;
   case 'p':
       ...
   ...
  }
  ...
  a[i][j]++;
}
```

A Debugging Scenario

```
int **a;

void main()
{
  ...
  process_input(a);
  ...
}

void clear_array(int **a)
{
  for (...)
    a[i] = NULL;
}
```

```
void process_input(int **a)
{
  cin >> input;
  switch (input) {
    case 'e':
        clear_array(a);
        ~~break~~; return
    case 'p':
        ...
    ...
  }
  ...
  a[i][j]++;
}
```

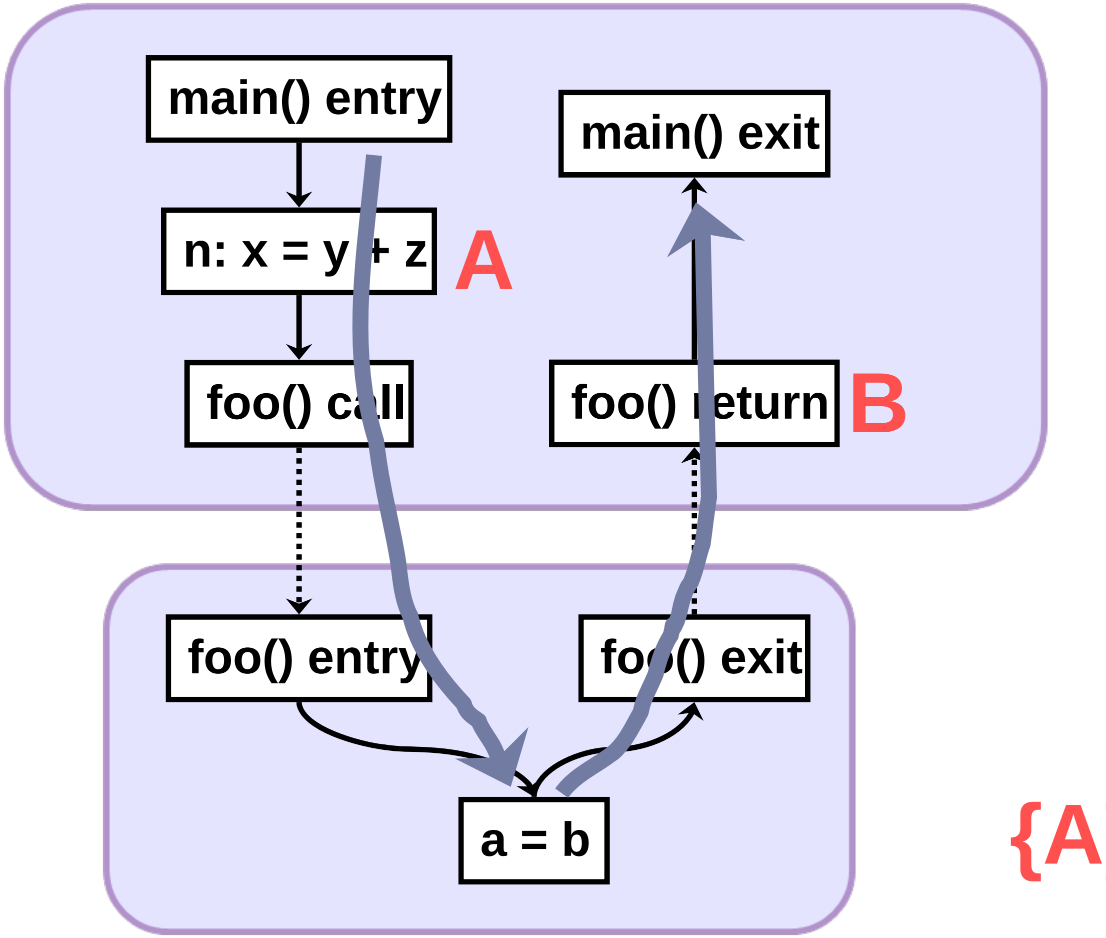
Goal: Find Minimal Failure Path

- Explore paths subject to constraints
  - Dynamic info (bug predictors, failure stack)
  - Static info (control flow, dataflow)
  - Interactive guidance from user
- Want short, feasible path that exhibits bug
  - Undecidable ☹
  - But still a very interesting problem!

Weighted Pushdown Systems

- PDS: finite automaton with stack
  - Describes control-feasible paths, including call/return
- WPDS: track dataflow “payload” along each path
  - Weight as transfer function on dataflow facts
- Instantiate WPDS by defining:
  - Initial weight associated with each PDS transition
  - Binary *extend* operator (⊗) for concatenating paths
  - Binary *combine* operator (⊕) for joining paths

Weight as Set of Bug Predictors



{A} ⊗ {B} = {A, B}

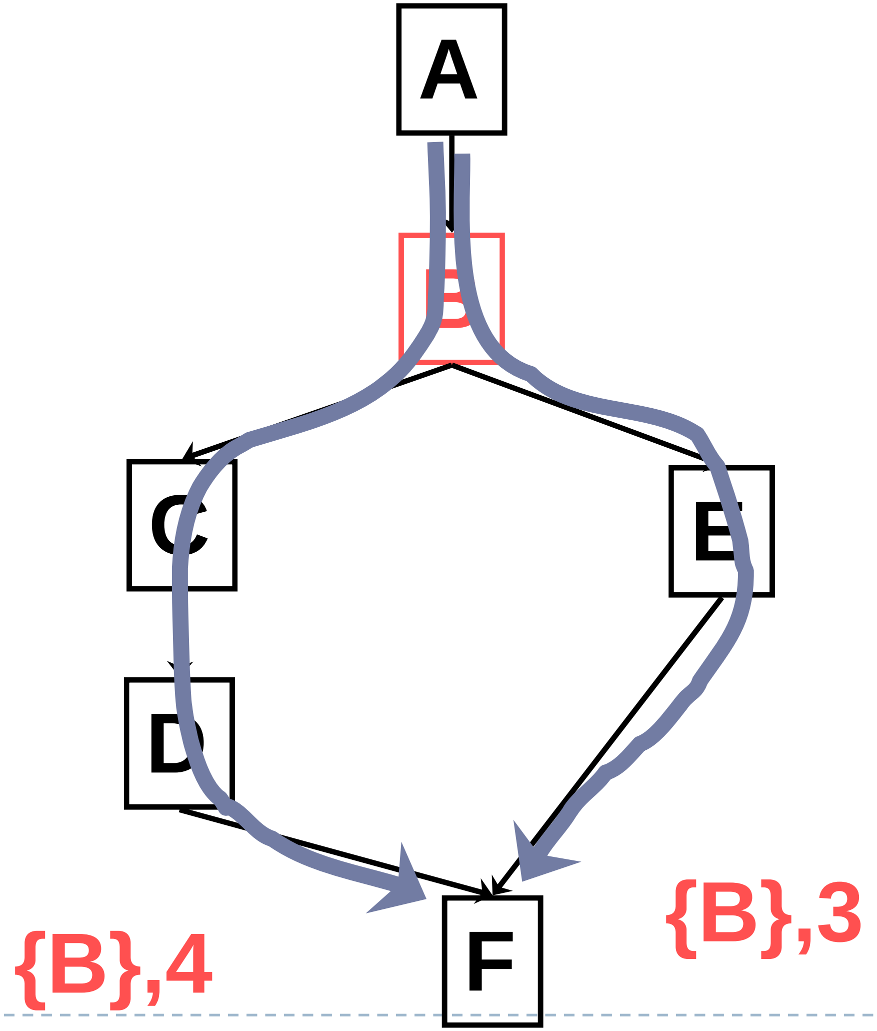
Weight as Set of Bug Predictors

- Path weight is set of predictors touched
- Singleton set at each bug predictor
  - Use “redundant” predictors suppressed earlier
  - Empty set at all other CFG nodes
- Path extension is set union
- Path merging: select path with biggest set?

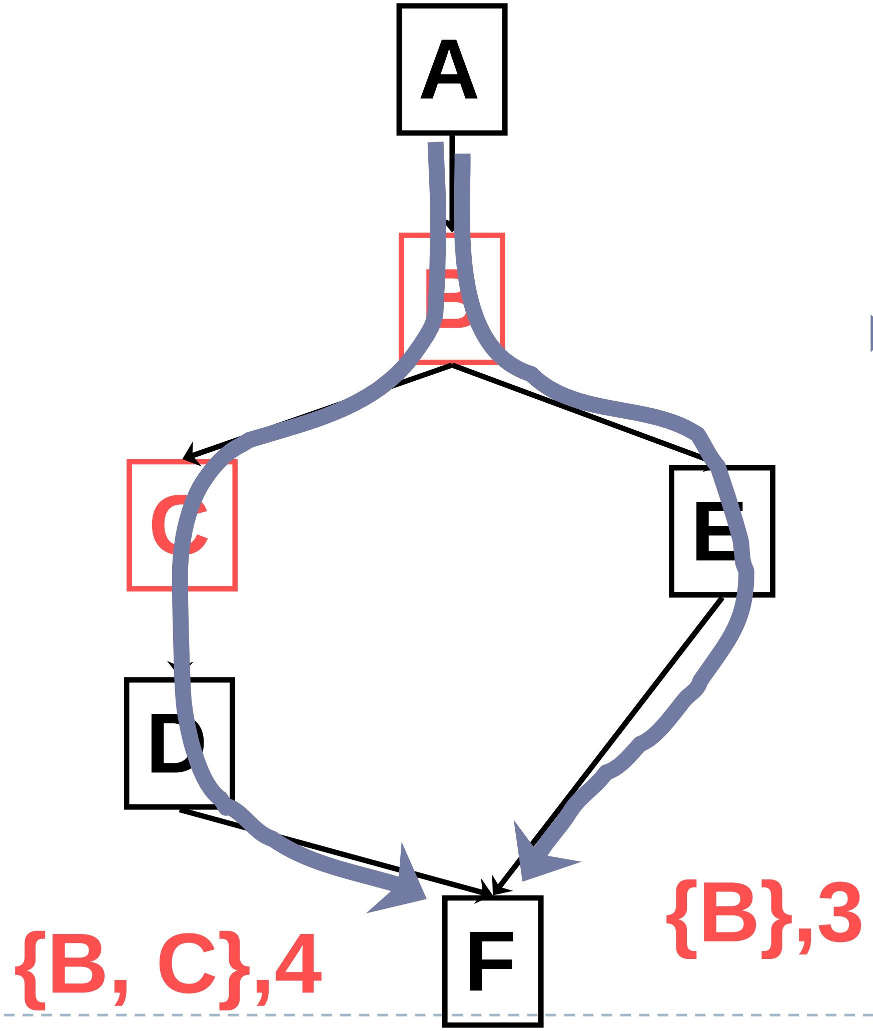
How Good is a Path?

- If two paths touch same bug predictors, which one do we want?
  - Shortest one!
- Need to reflect length in path weights
  - Weight = (set of bug predictors, path length)
  - Extend operator: union of sets, sum of lengths
  - Initial weights: length 1 for every transition

Path Weight Merging



Path Weight Merging



- One path per set of predictors touched
  - Exponential in # of predictors
  - Near linear in program size

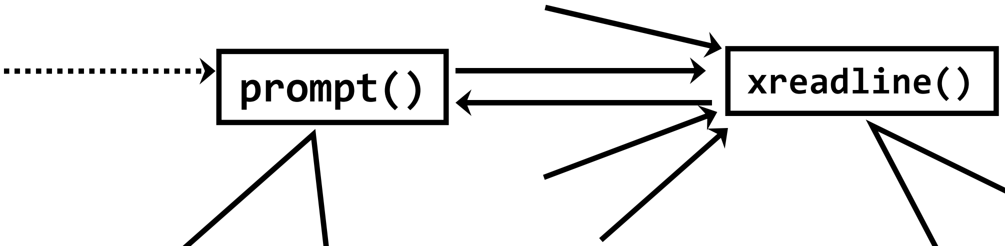
User Guidance & Interactivity

- Ordering constraints: A before B
  - $\{A\} \otimes \{B\} = \{A, B\}$
  - $\{B\} \otimes \{A\} = \perp$
  - Requires rebuild of solution automaton
- Steer path by changing scoring of nodes & paths
  - Assign scores based on statistical metrics
  - Avoid selected nodes (anti-predictors)
  - No rebuild of solution automaton
- Easy to mix in (most) dataflow analyses

Experiments: Siemens Test Suite

- Each program contains a single bug
  - Chose three programs where the bug predictors “miss” the true bug
- Reconstructed failure paths pass through the buggy lines of code

Experiments: ccrypt



prompt()

xreadline()

```
int prompt(void) {
    …
    line = xreadline(fin, cmd.name);
    return (!strcmp(line, 'y') ||
             !strcmp(line, "yes");
}
```

☠

```
char *
xreadline(FILE *fin, char *name) {
    int buflen = INITSIZE;

    char *buf = xalloc(buflen, name);
    char *res, *nl;

    res = fgets(buf, INITSIZE, fin);
    if (res == NULL) {
        free(buf);
        return NULL;
    }
    nl = strchr (buf, '\n');
    …
    return buf;
}
```

☹

Dataflow isolates call in **prompt()** as culprit

Experiments: bc

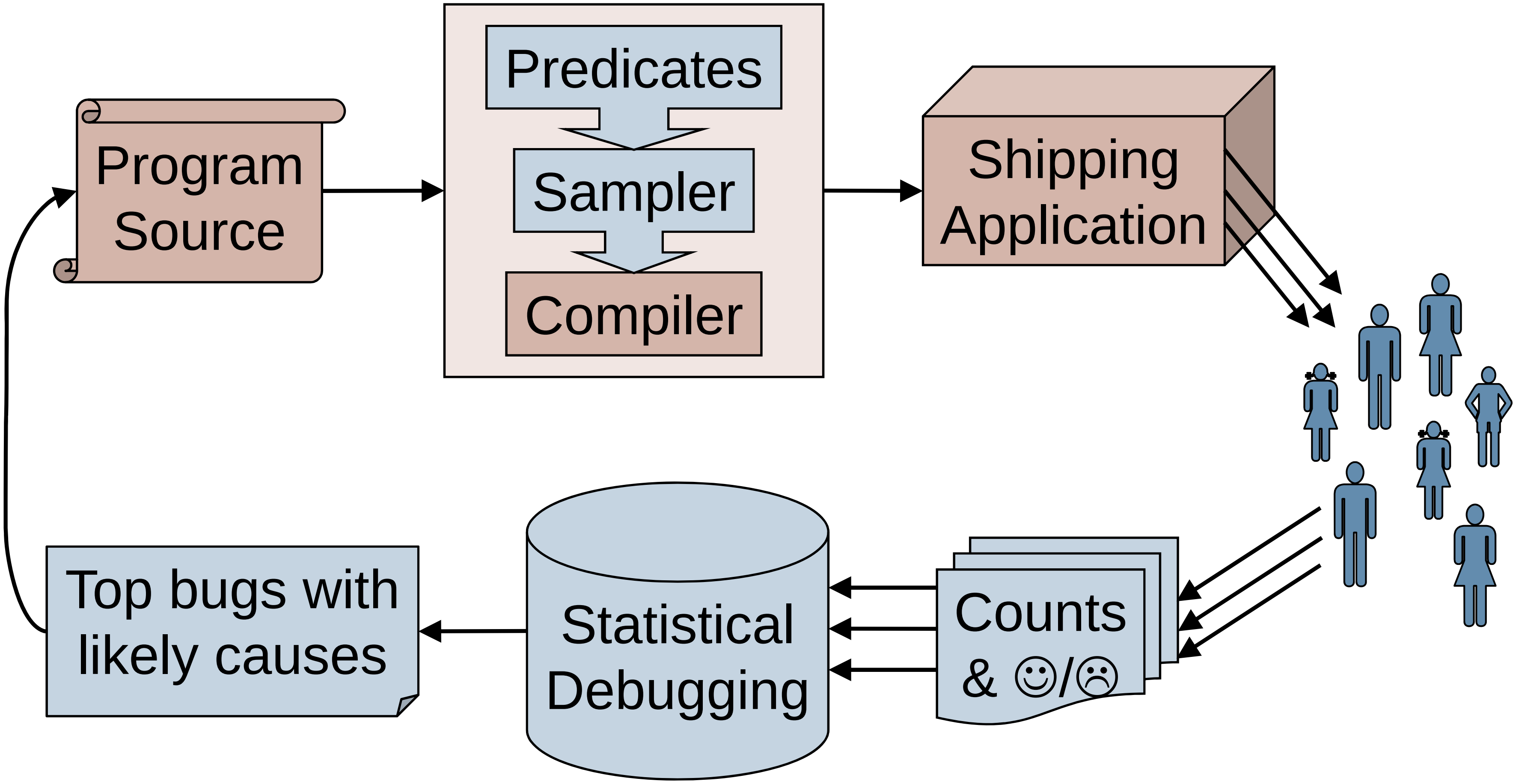
- Calculator tool with buffer overrun
- Statistical model: two bug predictor lists
  - Suggests two bugs in the program
- But reconstructed failure paths are identical!
  - Correctly reveals that only one bug is present

CBI in the Real World

“Beware of bugs in
the above code;
I have only proved it
correct, not tried it.”

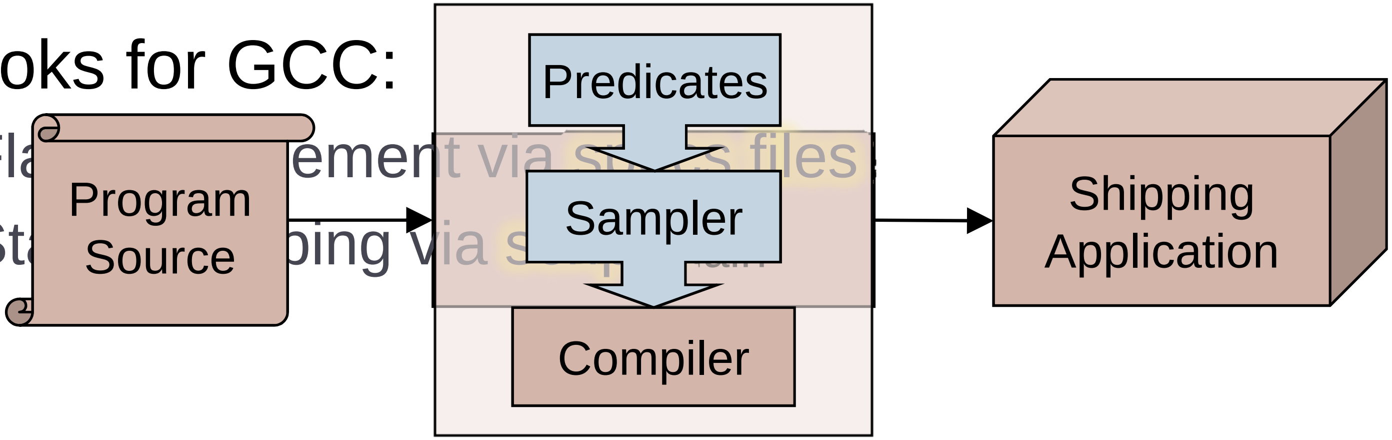
Donald Knuth, *Notes on the van Emde Boas construction of priority deques: An instructive use of recursion*

Bug Isolation Architecture Recap



Native Compiler Integration

- Instrumentor must mimic native compiler
  - You don’t have time to port & annotate by hand
- Our approach: source-to-source, then native
  - CIL: *highly* recommended, but for C only
- Hooks for GCC:
  - Fla[illegible]ement via s[illegible]s files
  - Sta[illegible]ping via s[illegible]



GCC Specs File

- Determines command-line flags to GCC stages
  - Used to be standalone file
  - Now built into `gcc` binary
  - View using “`gcc -dumpspecs`”
- Some fragments from the standard specs file:

  -
    ```
    *cpp:
    %{posix:-D_POSIX_SOURCE} %{pthread:-D_REENTRANT} …

    *lib:
    %{pthread:-lpthread} %{shared:-lc} …
    ```

Augmenting the Standard Flags

- Augment built-in specs with custom specs file:
  - `gcc -specs=myspecs …`
- Unrecognized "--*xyz*" flags prefixed with "`-f`*`xyz`*"
  - `--sampler-scheme=returns`
  - `-fsampler-scheme=returns`
- Pattern-match on custom flags in custom specs file
  - Can pattern-match on standard flags too, of course

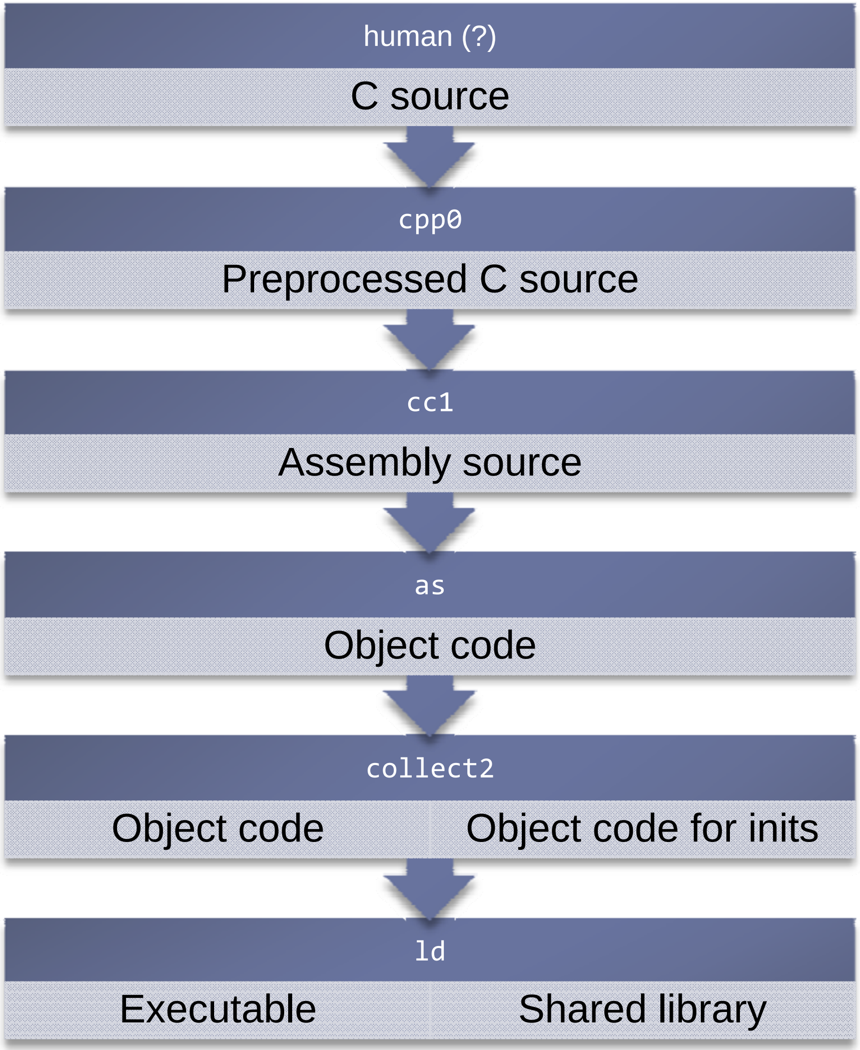
Specs Customization Example

```
*cpp:
+ -DCIL \
%{fsampler-scheme=returns:-include sampler/returns.h%s} \
%{fsampler-scheme=*:-include sampler/unit.h%s}

%rename libgcc old_libgcc

*libgcc:
--undefined=cbi_initialize \
%{fsampler-scheme=*:-lsampler-%*} \
%(old_libgcc)
```

Stages of GCC Compilation



human (?)

C source

`cpp0`

Preprocessed C source

`cc1`

Assembly source

`as`

Object code

`collect2`

Object code | Object code for inits

`ld`

Executable | Shared library

- Many formats & stages
  - Many hooks!
- Obvious injection point
  - Between `cpp0` and `cc1`
- Less obvious tweaks also needed to other stages
  - Tweak using specs only where possible
  - Tweak using specs + scripts for more complex tasks

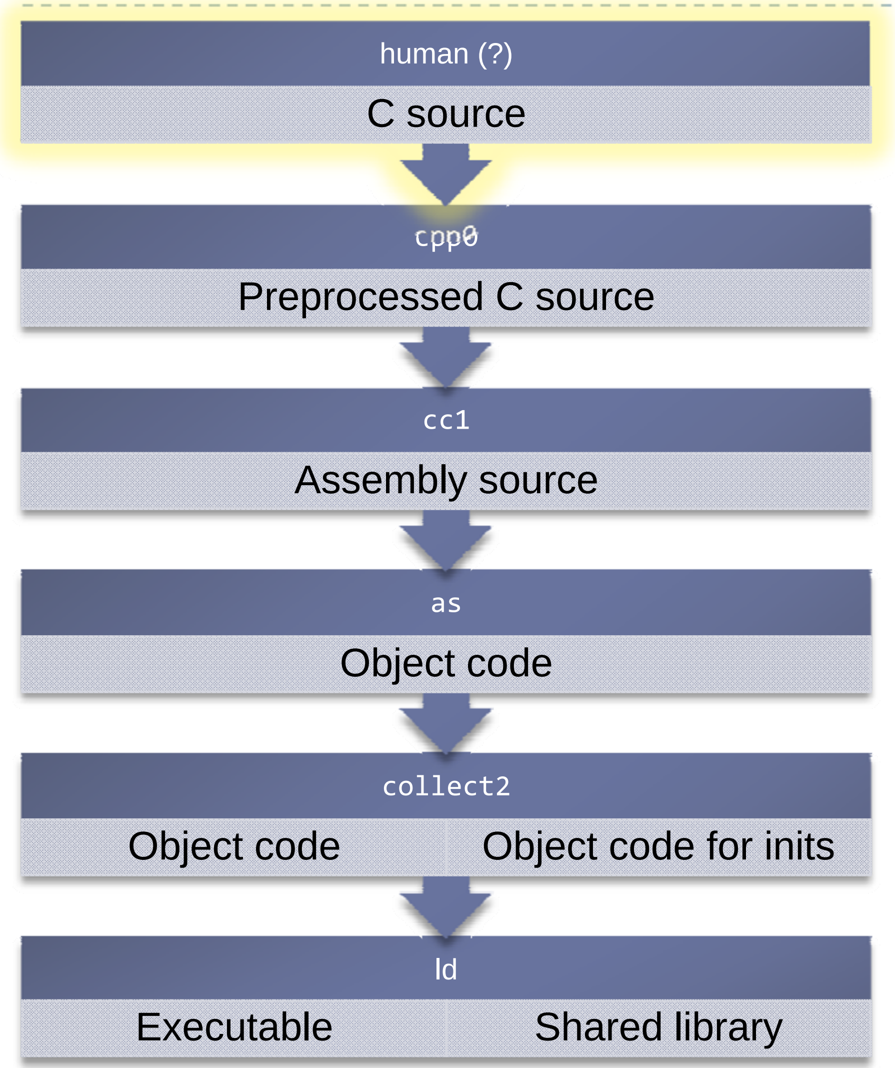
gcc -v -o main main.c

```
/usr/libexec/gcc/i686-pc-linux-gnu/4.2.0/cc1 -quiet -v -iprefix
/usr/lib/gcc/i686-pc-linux-gnu/4.2.0/ main.c -quiet -dumpbase main.c
-mtune=generic -auxbase main -version -o /tmp/cc8DBZxI.s

as -V -Qy -o /tmp/ccUvMQMf.o /tmp/cc8DBZxI.s

/usr/libexec/gcc/i686-pc-linux-gnu/4.2.0/collect2 --eh-frame-hdr -m
elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o main /usr/lib/crt1.o
/usr/lib/crti.o /usr/lib/gcc/i686-pc-linux-gnu/4.2.0/crtbegin.o
-L/usr/lib/gcc/i686-pc-linux-gnu/4.2.0 -L/usr/lib/gcc
-L/usr/lib/gcc/i686-pc-linux-gnu/4.2.0 -L/usr/lib/gcc/i686-pc-linux-
gnu/4.2.0/../../.. -L/usr/lib/gcc/i686-pc-linux-gnu/4.2.0/../../..
/tmp/ccUvMQMf.o -lgcc --as-needed -lgcc_s --no-as-needed -lc –lgcc
--as-needed -lgcc_s --no-as-needed /usr/lib/gcc/i686-pc-linux-
gnu/4.2.0/crtend.o /usr/lib/crtn.o
```

Machine-Generated C Code



human (?)

C source

`cpp0`

Preprocessed C source

`cc1`

Assembly source

`as`

Object code

`collect2`

Object code

Object code for inits

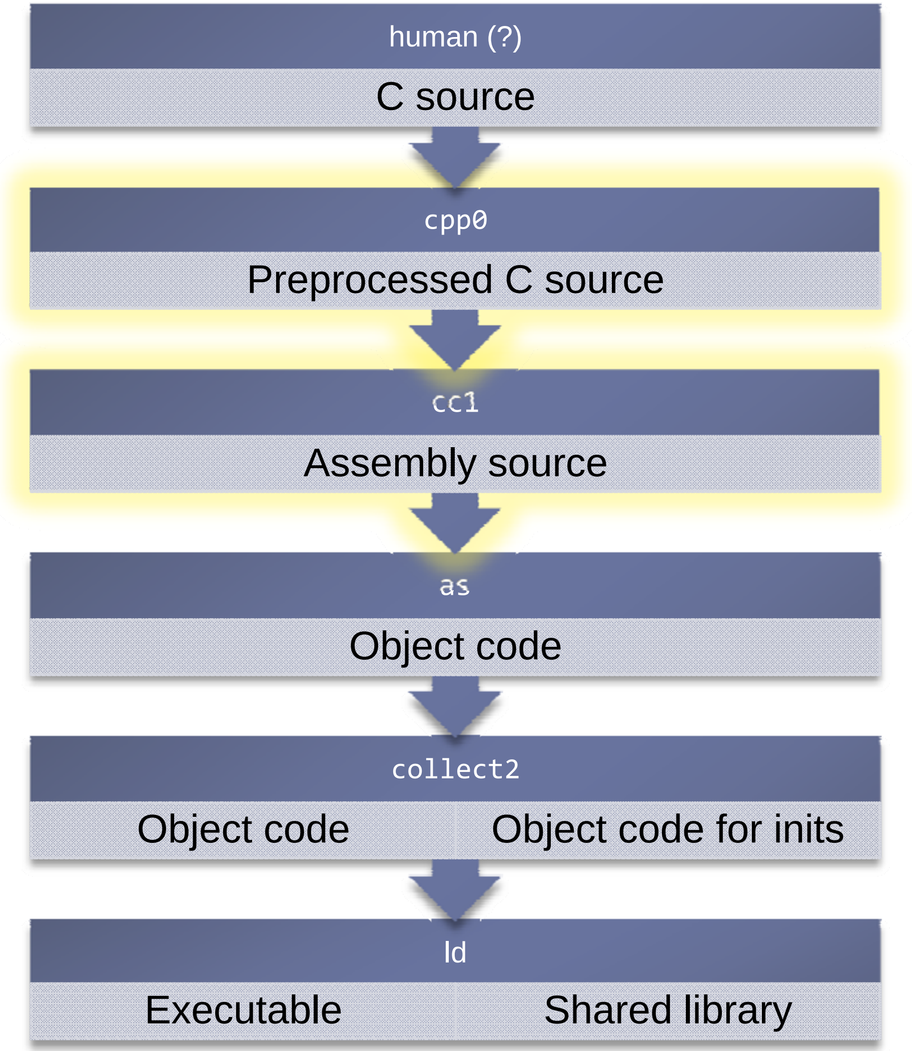
ld

Executable

Shared library

- Non-trivial projects contain non-human code
  - lex/flex, yacc/bison
  - Embedded icon data
- Breaks many tools
  - Big-O complexity matters!
- What to do about it?
  - Fix tools
  - Exclude by filename
  - Exclude by symbol name

“Obvious” Injection Point?



human (?)

C source

cpp0

Preprocessed C source

cc1

Assembly source

as

Object code

collect2

Object code

Object code for inits

ld

Executable

Shared library

- `cpp0` is gone!
  - Fused with `cc1`
  - Performance, debug info
  - `cc1` is the new `cpp0` ☺
- Steps for `cc1` script:
  1. Parse command line
  2. Run `cc1 -E`
  3. Transform
  4. Run `cc1`

Temporary File Management

- We need an extra temporary file
  - Output from preprocessor / input to our transformation
- Actually, make that *several* extra temporaries
  - Preprocessor output / transformation input
  - Transformation output / compiler input
  - A few more to come later…
- Could manage ourselves, but better to let GCC do it
  - Avoid reinventing the wheel
  - Retain expected behavior of "`-save-temps`"

GCC Specs Files to the Rescue!

- Magic “%u.*suffix*” directive
  - Can be used multiple times for multiple stages’ flags
  - Always expands to a unique file name for a given suffix
- Example:

  -
    ```
    *cc1:
    + \
    -finstrumentor-input %u.i \
    -finstrumentor-output %u.inst.i
    ```
- Replacement cc1 script can look for this flag
  - Automatically does the right thing for “`-save-temps`”

Embedding Extra Static Info

- Transformation produces several “outputs”:
  1. Modified C code (duh)
  2. Static information about instrumentation sites
  3. Static dump of control-flow graph
  4. Static dump of copy-constant data flow graph (default off)

- Want to keep these together
  - “Together” must survive `ar`, `mv`, and other `makefile` insanity
  - Must be physically embedded in object file, or not a chance

- Embedding massive literal strings doesn’t scale
  - Also, want to avoid intermixing static info with program data

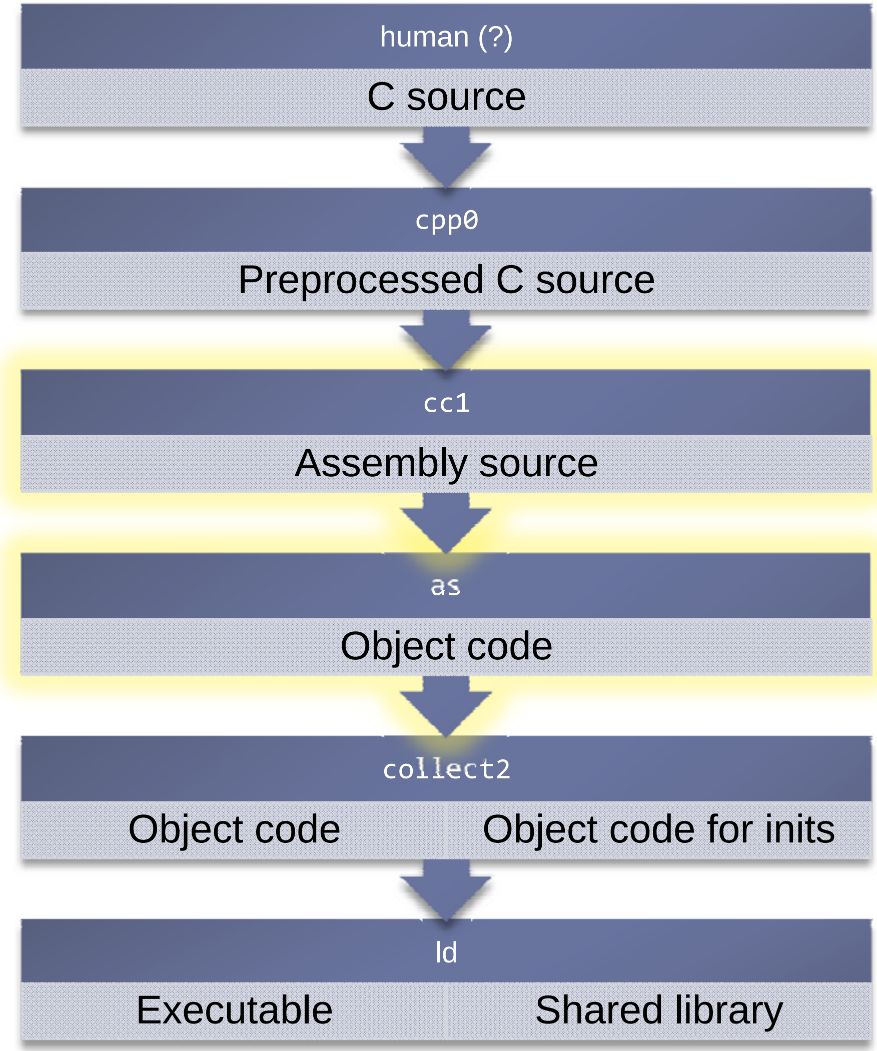
A Winning Strategy

1. Source-to-source transformation writes out several files
   - Extra static info sits around in temporaries
   - %u again!

2. Run real "`cc1`" and "`as`" to produce object file

3. Stash temporary file contents inside object file
   - ELF object files are collection of named sections
   - Several standard sections: `.text`, `.data`, `.bss`, …
   - Create new ELF sections with non-standard names
   - Hide our data inside!

Embedding Extra Static Information



human (?)

C source

`cpp0`

Preprocessed C source

`cc1`

Assembly source

`as`

Object code

`collect2`

Object code

Object code for inits

ld

Executable

Shared library

- `cc1` and `as` scripts
  - Must agree on temp names
- Specs files to the rescue!
  - -fsave-sites %u.sites \ -fsave-cfg %u.cfg
- Same “%u” suffix, same file
  - Even across stages

Custom as Script Steps

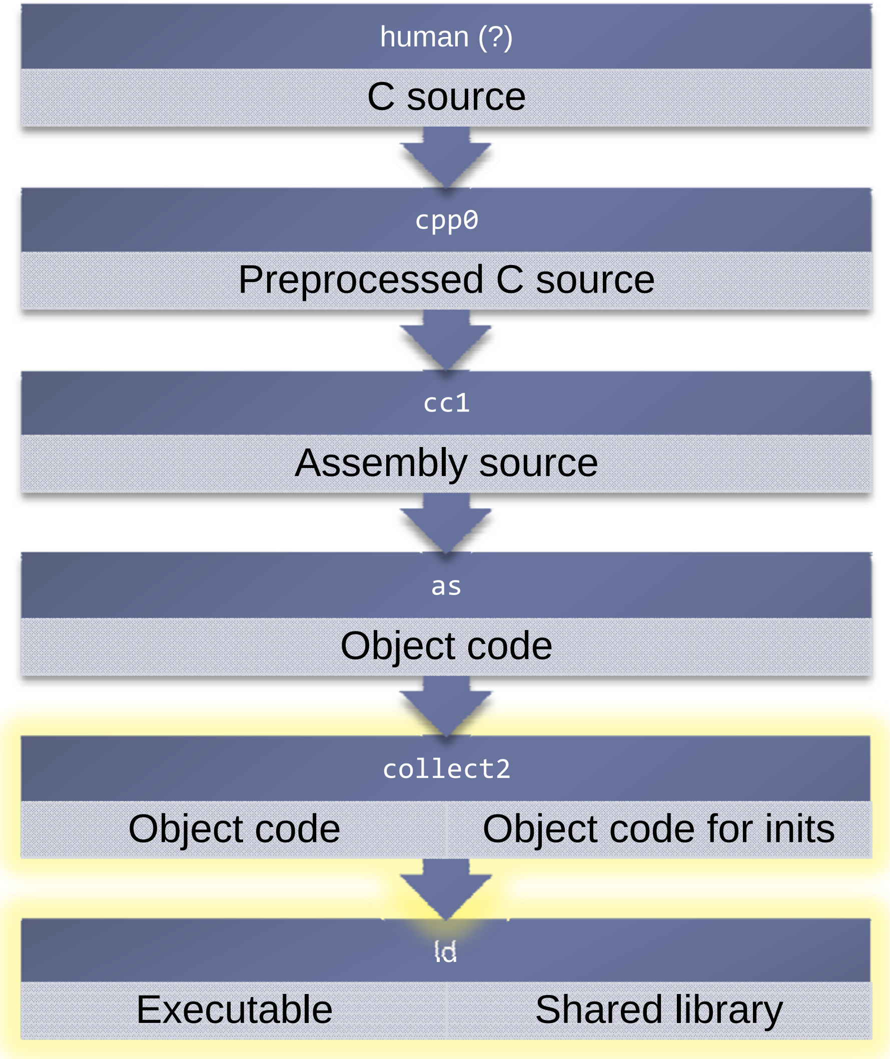
1. Parse command line
   - Make note of object file name
   - Make note of other temporary file names

2. Run real assembler to produce real object file
   - Remember, script starts with assembly *source* file

3. Run `objcopy` to add new section to object file

   -
     ```
     objcopy \
       --add-section .debug_site_info=$sitefile \
       $objectfile
     ```

Linker Tweaks



- Add support libraries using specs file
  - Saw example earlier
- `ld` combines non-standard ELF sections
  - Pad with null bytes
  - Concatenate in link order
  - Design format carefully!
- No replacement scripts
  - In my case, at least

Putting All the Pieces Together

- Simple top-level `gcc` wrapper script: `sampler-cc`

  -
    ```
    #!/bin/sh
    exec gcc -B stagedir -specs=specsfile "$@"
    ```
- Ready to hook into build systems
  - `make CC=sampler-cc …`
  - `./configure CC=sampler-cc …`
- We’ve done it!
  - Source-to-source transformation pretending to be `gcc`
  - Good enough to “fool” millions of lines of real code

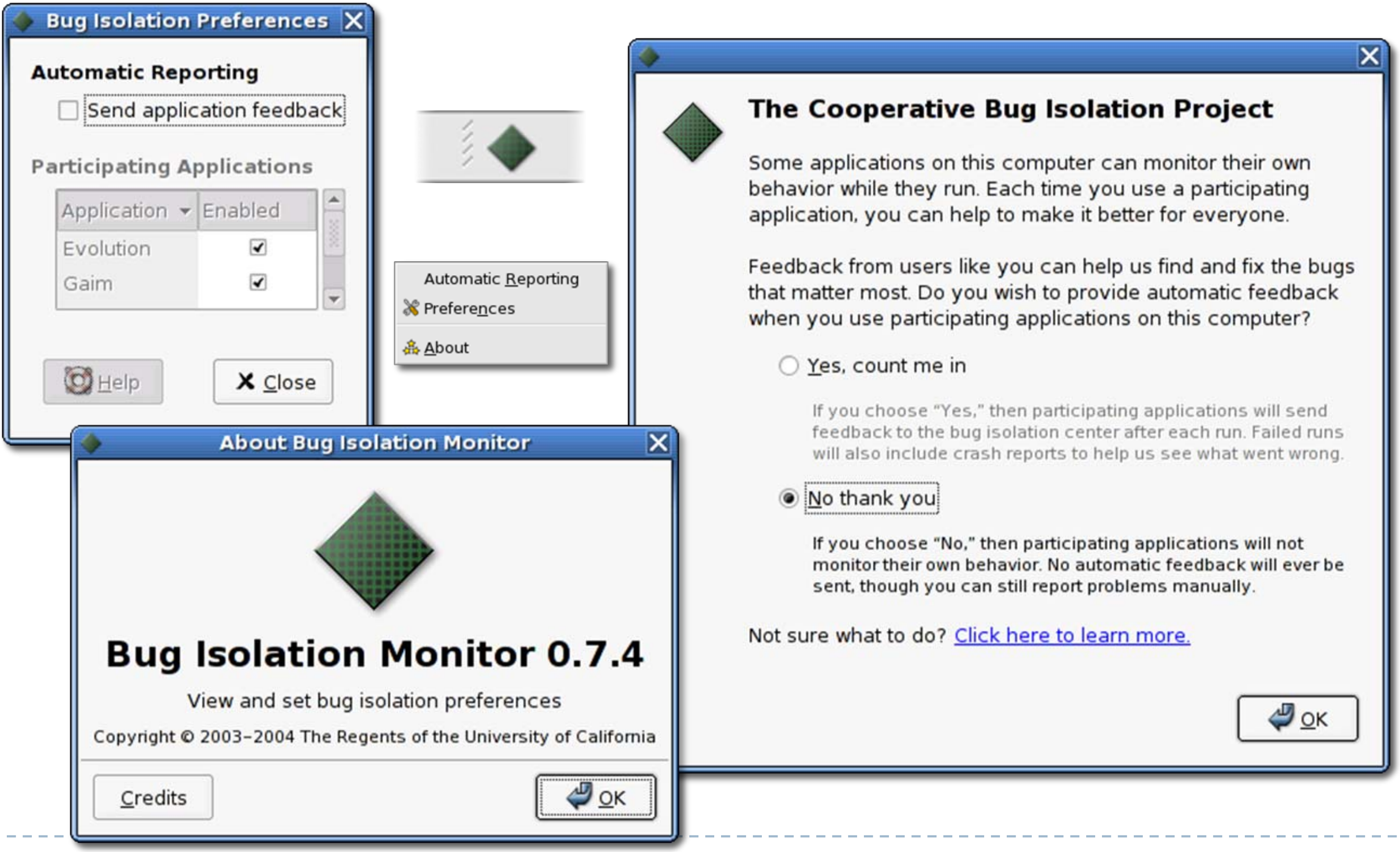
Multithreaded Programs

- Global next-sample countdown
  - High contention, small footprint
  - Want to use registers for performance
  - ⇒ Thread-local: one countdown per thread
- Global random number generator
  - High contention, small footprint
  - ⇒ Thread-local: one generator 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 & non-instrumented code
- Self-management at compile time
  - Locally derive identifying object signature
  - Embed static site information within object file
- Self-management at run time
  - On load, register self with global object registry
  - On normal unload, report feedback state and deregister
  - On fatal signal, walk global object registry

Keeping the User In Control



Bug Isolation Preferences

Automatic Reporting

□ Send application feedback

Participating Applications

Application	Enabled
Evolution	☑
Gaim	☑

The Cooperative Bug Isolation Project

Some applications on this computer can monitor their own behavior while they run. Each time you use a participating application, you can help to make it better for everyone.

Feedback from users like you can help us find and fix the bugs that matter most. Do you wish to provide automatic feedback when you use participating applications on this computer?

○ Yes, count me in

If you choose "Yes," then participating applications will send feedback to the bug isolation center after each run. Failed runs will also include crash reports to help us see what went wrong.

● No thank you

If you choose "No," then participating applications will not monitor their own behavior. No automatic feedback will ever be sent, though you can still report problems manually.

Not sure what to do? Click here to learn more.

About Bug Isolation Monitor

Bug Isolation Monitor 0.7.4

View and set bug isolation preferences

Copyright © 2003–2004 The Regents of the University of California

Database Poisoning

- Not (yet) observed in practice
  - Not intentionally, at least
- Methods are stable w.r.t. a few bad actors
- TCPA/Palladium for stronger guarantees
- Direct detection of bogus reports?

Privacy & Info Leakage

- Information leaks, but at slow rate
  - So does calling tech support
- Users’ interests align with developers’
  - You give me a little bit of information
  - I give you bug fixes that *you* care about

Privacy & Info Leakage

- Some code should not be instrumented
  - Don’t track branches in unrolled RSA code
- Attacker needs to aggregate reports
  - SSL makes eavesdropping harder
  - Database design to support safety in numbers

Lessons Learned

- Can learn a lot from actual executions
  - Users are running buggy code anyway
  - We should capture some of that information
- Great potential in hybrid approaches
  - Dynamic: reality-driven debugging
  - Statistical: best-effort with uncertainty
  - Static: use program structure to fill in the gaps

Vision for Statistical Debugging

- 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!

Join the Cause!

The Cooperative Bug Isolation Project

http://www.cs.wisc.edu/cbi/