LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

GCC’s -fanalyzer option

David Malcolm
<dmalcolm@redhat.com>

Overview

- What is it?
- Implementation details
- Current strengths and limitations
- Plans for GCC 11
- Ideas for future directions

The **-fanalyzer** option

- Added by me in GCC 10
- **-fanalyzer** enables a new interprocedural pass, implementing 15 new warnings
- Performs a much more expensive analysis of the code than traditional warnings

New Warnings

- -Wanalyzer-double-free
- -Wanalyzer-use-after-free
- -Wanalyzer-free-of-non-heap
- -Wanalyzer-malloc-leak

- -Wanalyzer-possible-null-argument
- -Wanalyzer-possible-null-dereference
- -Wanalyzer-null-argument
- -Wanalyzer-null-dereference

-Wanalyzer-double-fclose
-Wanalyzer-file-leak

-Wanalyzer-stale-setjmp-buffer
-Wanalyzer-use-of-pointer-in-stale-stack-frame

-Wanalyzer-unsafe-call-within-signal-handler

-Wanalyzer-tainted-array-index

-Wanalyzer-exposure-through-output-file

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

A simple example

```c
#include <stdlib.h>

void test (void)
{
  char *p = malloc (4096);
  char *q = malloc (4096);
  /* do stuff */
  free (p);
  free (p);
}
```

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

```
demo.c:9:3: warning: double-‘free’ of ‘p’ [CWE-415] [-Wanalyzer-double-free]
    9 |   free (p);
      |   ^~~~~~~~
  ‘test’: events 1-3
    |
    |    5 |   char *p = malloc (4096);
    |      |             ^~~~~~~~~~~~~
    |      |             |
    |      |             (1) allocated here
    |......
    |    8 |   free (p);
    |      |   ~~~~~~~~
    |      |   |
    |      |   (2) first ‘free’ here
    |    9 |   free (p);
    |      |   ~~~~~~~~
    |      |   |
    |      |   (3) second ‘free’ here; first ‘free’ was at (2)
    |
demo.c:10:1: warning: leak of ‘q’ [CWE-401] [-Wanalyzer-malloc-leak]
   10 | }
      | ^
  ‘test’: events 1-2
    |
    |    6 |   char *q = malloc (4096);
    |      |             ^~~~~~~~~~~~~
    |      |             |
    |      |             (1) allocated here
    |......
    |   10 | }
    |      | ~
    |      | |
    |      | (2) ‘q’ leaks here; was allocated at (1)
    |
```

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Why?

- The earlier a bug is found, the better
- Embedding checks in the compiler is the earliest possible point
  - Ideally, we will never hear about bugs found by this feature: they'd get fixed in the Edit-Compile-Debug cycle and never make it into published patches/repositories

Why? (2)

- The programmer can see the diagnostics as he or she works on the code, rather than at some later point.
  - Belief: if the analyzer is fast enough and has a good enough signal:noise ratio, many people would opt-in to deeper but more expensive warnings
  - I'm aiming for 2x compile time as my rough estimate of what's reasonable in exchange for being told up-front about various kinds of pointer snafu)

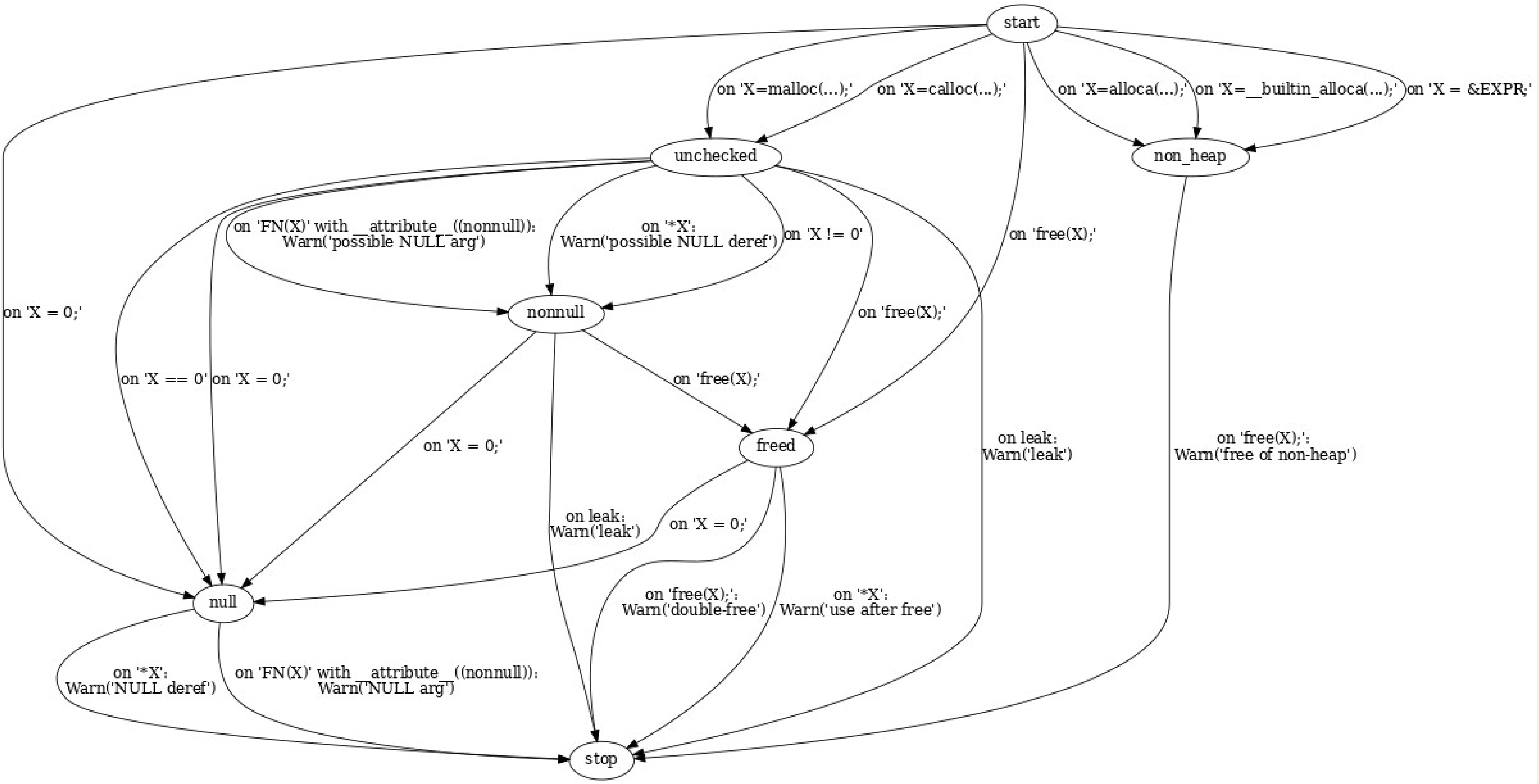
LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Why? (3)

- But clang-analyzer exists…
  - GCC and llvm both exist
  - We have two different FLOSS toolchains
    - Competition is good
    - GNU should have an analyzer in its toolchain

Implementation Details

- Using state machines to model APIs
- E.g.
  - “PTR = malloc(…);”: PTR → unchecked
  - “if (PTR)”:
    - True edge: PTR → nonnull
    - False edge: PTR→ null
  - “free(PTR);”: PTR → freed
  - “free(PTR);” when PTR is “freed”:
    - Warn about double-free of PTR


start
on 'X=malloc(...);'
on 'X=calloc(...);'
on 'X=alloca(...);'
on 'X=__builtin_alloca(...);'
on 'X = &EXPR;'
unchecked
non_heap
on 'X = 0;'
on 'FN(X)' with __attribute__((nonnull)):
Warn('possible NULL arg')
on '*X':
Warn('possible NULL deref')
on 'X != 0'
on 'free(X);'
nonnull
on 'free(X);'
on 'X == 0'
on 'X = 0;'
on 'free(X);'
on 'X = 0;'
freed
on leak:
Warn('leak')
on 'free(X);':
Warn('free of non-heap')
on leak:
Warn('leak')
on 'X = 0;'
null
on 'free(X);':
Warn('double-free')
on '*X':
Warn('use after free')
on '*X':
Warn('NULL deref')
on 'FN(X)' with __attribute__((nonnull)):
Warn('NULL arg')
stop

Initial Approach

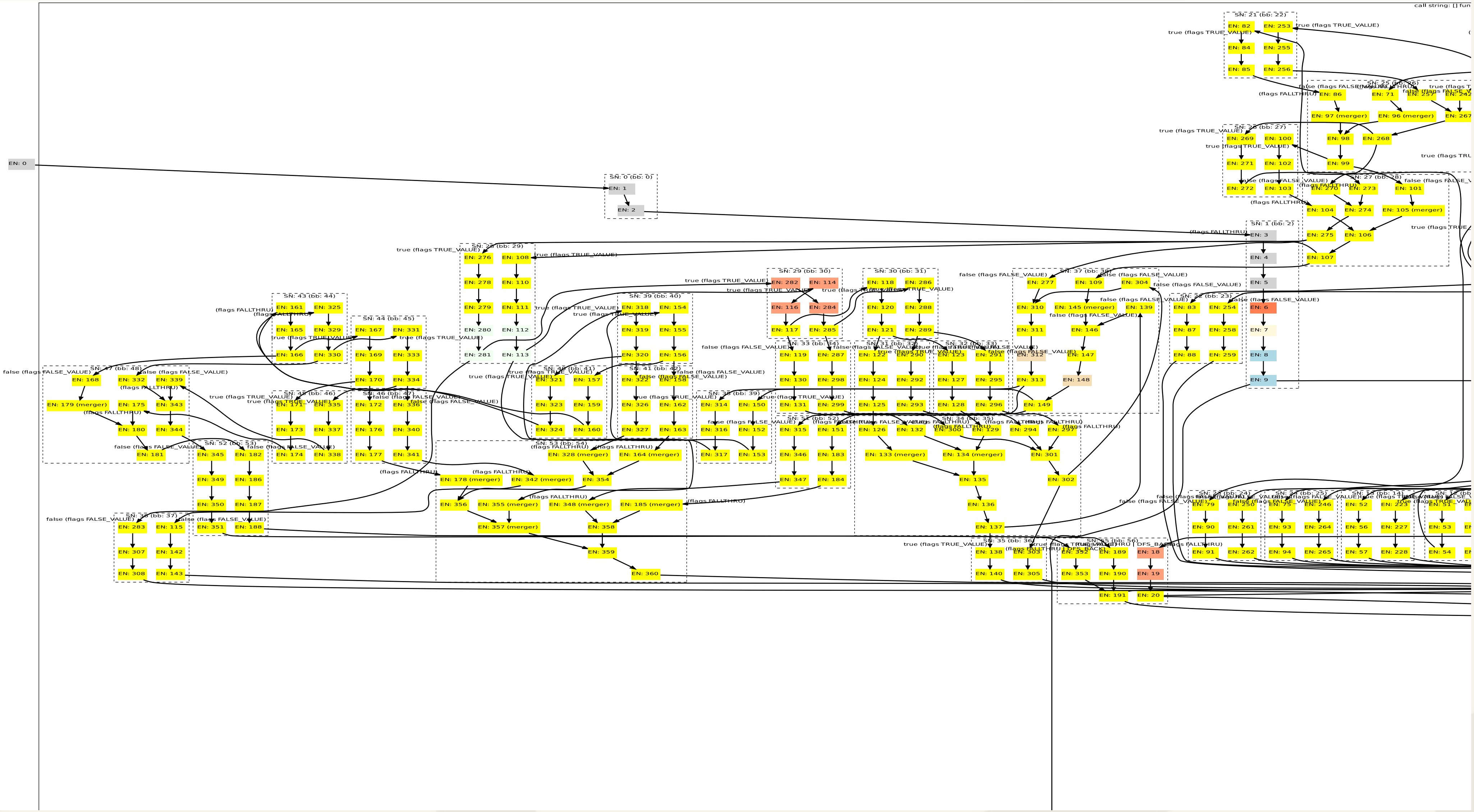
- What’s the minimum viable analyzer for detecting double-free bugs?
- Attempted to implement the approach from the Stanford Checker
- But the diagnostics from my implementation were inscrutable
  - Why is **-fanalyzer** warning about ***FOO*** ?
    - How can it happen?
    - If I can’t debug this, how is the user meant to figure it out?

Revised Implementation

- Emit control-flow paths to the user
  - Need to convince the user of the correctness of the problem
  - Without overwhelming them (not yet achieved)

Graph-based implementation

- Build an “exploded graph” combining both control-flow and state
  - A directed graph
  - Each node is a (point, state) pair
  - Terminology from *"Precise Interprocedural Dataflow Analysis via Graph Reachability"* (Thomas Reps, Susan Horwitz and Mooly Sagiv) 1995
- Detect problems when unexpected state occurs at a point
- Same approach as used by clang analyzer
- Tension between precision of state-modeling vs ensuring termination and not bloating memory



EN: 0
SN: 0 (bb: 0)
EN: 1
EN: 2
SN: 1 (bb: 2)
EN: 3
EN: 4
EN: 5
EN: 6
EN: 7
EN: 8
EN: 9
SN: 21 (bb: 22)
EN: 82
EN: 253
EN: 84
EN: 255
EN: 85
EN: 256
SN: 28 (bb: 29)
EN: 276
EN: 108
SN: 29 (bb: 30)
EN: 282
EN: 114
SN: 30 (bb: 31)
EN: 118
EN: 286
SN: 37 (bb: 38)
EN: 277
EN: 109
EN: 304
SN: 43 (bb: 44)
EN: 161
EN: 325
SN: 44 (bb: 45)
EN: 167
EN: 331
SN: 47 (bb: 48)
EN: 168
EN: 332
EN: 339
SN: 53 (bb: 54)
EN: 328 (merger)
EN: 164 (merger)
EN: 354
EN: 358
EN: 359
EN: 360
SN: 36 (bb: 37)
EN: 283
EN: 115
EN: 191
EN: 20
true (flags TRUE_VALUE)
false (flags FALSE_VALUE)
(flags FALLTHRU)

Soundness and completeness?

- Different communities have different definitions, one is:
  - “Soundness”: no false negatives
  - “Completeness”: no false positives
- **-fanalyzer** is **neither sound nor complete**
  - It attempts to explore "interesting" paths through the code and generate meaningful diagnostics
  - but it will merge states to try to keep the analysis tractable
  - and the states are abstract
    - over-approximations in some ways, under-approximations in others

Integration within GCC

- Works on GIMPLE SSA
  - I chose this in the hope of making it easier to support LTO
  - But puts it at the mercy of optimization options, perhaps should run earlier?
- The implementation is read-only: it doesn’t attempt to change anything, just emit warnings
- Assumes garbage-collector doesn’t run

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Algorithm

- First: build the exploded graph
  - Worklist of (point, state) nodes
    - Priority queue (e.g. group points together)
    - Merge nodes at a point when states are sufficiently similar
  - Prepopulate worklist with entrypoints to the public functions of the TU

Algorithm (2)

- First: build the exploded graph (continued)
  - Process (point, state) nodes in the worklist
  - Record diagnostics in node (e.g. “double free”)
  - Find successor nodes, add edges
    - Cache hits vs cache misses
    - Ideally converge on a solution where we’re hitting pre-existing nodes
  - Give up when limits are hit
    - Too many states at one point
    - Too many nodes overall

Algorithm (3)

- Having built the graph and saved diagnostics…
- Deduplicate diagnostics: partition them
  - find the shortest feasible path for each partition

Algorithm (4)

- Build a list of events along the path
- Apply peephole optimizer to try to only show the most pertinent events
- Emit the diagnostic
  - Precision-of-wording hooks:
    - **returning possibly-NULL pointer to 'make_obj' from 'allocator'**
    - **second 'free' here; first 'free' was at (1)**

C is hard

- Arbitrary pointers
- Casts and unions
- setjmp/longjmp
- Dynamic allocation
- etc...

Tracking state

- Abstraction of possible states of variables and memory
  - Hierarchy of “regions” e.g.:
    - The stack
      - Frame for current function
        - A local array
          - An element within the array

Tracking state (2)

- Symbolic values:
  - Constants
  - Initial value of a region
    - e.g. initial value of “**ptr→field**” at the start of the analysis path
  - Pointer to a region (e.g. “**&x**”)
  - Compound values (e.g. “**x + y**”)
  - “Conjured values” at a statement (e.g. when an escaped region could be clobbered by a call to an external function)
  - “Unknown” for when we need to give up
  - etc

Tracking state (3)

- Program state has 4 parts:
  - A “store”: bindings from regions to values
  - Constraints (e.g. “*INIT_VAL(p) != 0*”)
  - The list of function frames in the stack [*]
  - State-machine states [*]
    - e.g. malloc: “INIT_VAL (p_23)”: “unchecked”
    - e.g. signal: global state: “in signal handler”

    [*] == can prevent merging of states

LINUX PLUMBERS CONFERENCE
August 24-28, 2020

Current Status

- Experimental prototype for C, for early adopters only.
- But has found CVE-2020-1967 in OpenSSL, a NULL pointer dereference in error handling.
  - and various error-handling bugs in elfutils

Strengths and Limitations

- Interprocedural, with LTO support
  - ...but current implementation of call summaries is just a placeholder

Strengths and Limitations (2)

- Cute ASCII art showing control flow
  - ...but it’s too verbose, and can overwhelm the user

Strengths and Limitations (3)

- GCC 10 implementation of state had at least two major design flaws
  - Led to explosions of state where state should have been merged but wasn't

GCC 11 plans

- GCC development cycle is typically:
  - April → October: 7 months of feature development
  - November → March: 5 months of bugfixing/stabilization
  - So about two more months of feature work for GCC 11

GCC 11 plans (2): unbreaking the basics

- Big rewrite of state-tracking
  - Landed in trunk on 2020-08-13 (about 4 months work)
  - Fixed the two flaws mentioned earlier
  - State explosions still happen, but are much more tractable

GCC 11 plans (3): scaling up to work on real code

- Fixing scaling issues so that **-fanalyzer** can be used on real-world C code
  - State explosions
  - Ludicrously verbose diagnostics
    - e.g. for CVE-2005-1689 (krb5 double-free)
      - was 1187 lines of stderr
      - GCC 10: 170 lines
      - trunk: 57 lines
      - Ideal: even lower

GCC 11 plans (4): new features

- Generalizing the malloc/free and FILE checking to arbitrary acquire/release API pairs
  - An attribute for labeling function decls as acquire/release pairs
- Start on C++ support
  - new/delete needs the above
  - Exception-handling
  - ...etc
- Lots more ideas...

Future Plans

- Other state machines
  - Prototype of taint analysis
  - Prototype of information leakage
- Bounds-checking
- Plugin support
  - Kernel ideas?
    - User space vs kernel space pointers
    - Interrupts enabled vs disabled

Q&A

- Thanks for listening!
- Thanks to LPC for hosting us
- Project homepage: https://gcc.gnu.org/wiki/DavidMalcolm/StaticAnalyzer

Bonus Slides

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

Why do it in compiler?

- Make it easy to verify that all your source is being checked
  - Same source files, same preprocessor defines, etc
- Same parser
  - *"The C language does not exist; neither does Java, C++, and C#. While a language may exist as an abstract idea, and even have a pile of paper (a standard) purporting to define it, a standard is not a compiler. What language do people write code in? The character strings accepted by their compiler."* (Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, Dawson Engler)

    "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World"