Flow-Based Diagnostics In GCC
GNU Tools @ LPC 2020

Martin Sebor

Principal Software Engineer, Red Hat

August 2020

Agenda

Flow-Based Diagnostics In GCC

- Warnings in GCC
- Status of existing warnings
  - Implementation Strategy
  - Strengths and Weaknesses
- Solving weaknesses
  - Reducing false positives
  - Warnings in development
- Ideas For Future Work
- Overlap With Static Analyzer

Warnings In GCC

Flow-Based Diagnostics In GCC

- 328 distinct warning options in total (GCC 9, 8, and 7: 297, 278, 265)
- 270 common and shared C-family warning options (256, 241, 230)
- 78 warning options used in the middle end (74, 71, 65)
- 5 warning options used in back ends
- (Plus 16 analyzer warnings.)

This talk is about a flow based subset of the 78 middle end warning options:

- Warnings that work with Gimple/SSA representation.
- Traverse multiple statements to track control and data flow.
- Some run in dedicated passes (alloca, restrict, strlen, sprintf, VRP, etc.)
- Others run just before expansion (`builtins.c` or `calls.c`).

Partial Listing Of Middle End Warnings

Flow-Based Diagnostics In GCC

- `-Waggressive-loop-optimizations` (~30 LOC in 1 file)
- `-Walloc-size-larger-than` (~100 LOC in 1 file)
- `-Walloca/-vla/-larger-than` (588 LOC in 1 file)
- `-Warray-bounds` (643 LOC)
- `-Wformat-overflow/-truncation` (4053 LOC in 1 file)
- `-Wfree-nonheap-object` (15 LOC(!) in 1 file)
- `-Wrestrict` (1832 LOC in 1 file)
- `-Wreturn-local-addr` (837 LOC in 1 file)
- `-Wstring-compare` (~50 LOC in 1 file)
- `-Wstringop-overflow/-truncation` (~1000 LOC across ~4 files)
- `-Wuninitialized` (2581 LOC in 1 file)
- `-Wnonnull` (~100 LOC in 5 files)
- `-Wnull-dereference` (same as `-Wreturn-local-addr`)

Access Based Warnings

Flow-Based Diagnostics In GCC

- `-Warray-bounds`
- `-Wformat-overflow/-truncation`
- `-Wrestrict`
- `-Wstringop-overflow/-truncation`
- `-Wuninitialized`
- `-Wnonnull`
- `-Wnull-dereference`

Implementation Strategy

Flow-Based Diagnostics In GCC

- For each interesting access statement:
  - Traverse the IL looking for the target (decl or allocation call).
  - Determine cumulative **offset** along the way.
  - Determine the **size** of the target.
  - Issue a warning
    - if offset is out-of-bounds for the size (for overflow warnings), or
    - if access overlaps (`–Wrestrict`).

- Interesting statements include:
  - Array indexing (`ARRAY_REF` and `MEM_REF`).
  - Assignments to/from character types.
  - Calls to string/memory built-in functions.
  - Calls to annotated user-defined functions.

Strengths

Flow-Based Diagnostics In GCC

- Analysis of whole function bodies, including inlined functions.
- Basic support for cross-functional analysis (still early stages).
- Bounds checking/buffer overflow coverage for declared and dynamically allocated objects:
  - `alloca`, VLA
  - `calloc`, `malloc`, `realloc`
  - C++ `operator new`,
  - functions with attribute `alloc_size`.
- Handling of member subobjects and array of arrays.
- Handling of zero-length arrays and flexible array members.
- Support for ranges of both offsets and sizes (for allocated objects/VLAs).

Support For Ranges and Allocated Objects

Strengths of Flow-Based Diagnostics In GCC

```
void f (unsigned n, unsigned i)
{
  if (n > 4 || i < 4) return;
  char vla[n];    // n’s range is [0, 4)
  vla[i] = 0;     // i’s range is [4, UINT_MAX]
  …
}
```

```
warning: writing 1 byte into a region of size 0 [-Wstringop-overflow=]
     7 |   vla[i] = 0;
note: at offset [4, -1] to an object with size at most 4 declared here
     6 |   char vla[n];
```

Support For Cross-Functional Analysis

Strengths of Flow-Based Diagnostics In GCC

- Attribute `access` to annotate user-defined functions:

  ```
  __attribute__ ((access (write_only,  1, 2)))
  void init (int *, size_t);
  ```

- Implicit attribute access for VLAs and ordinary arrays (upcoming):

  ```
  void init (size_t n, int[n], int[32]);
  ```

- Used by:
  - `-Warray-bounds` (GCC 11)
  - `-Wformat-overflow`
  - `-Wrestrict` (since GCC 10)
  - `-Wstringop-overflow` (since GCC 10)
  - `-Wuninitialized` (GCC 11)
  - `-Wunused-variable` (GCC 11?)

Weaknesses

Flow-Based Diagnostics In GCC

- Weaknesses in existing access-based warnings
  - False negatives
  - False positives
  - Inconsistent approaches
  - Missing coverage

redhat.

False Negatives

Weaknesses of Flow-Based Diagnostics In GCC

- Diverse/inconsistent implementations (little code sharing).
- Missing support for multiple objects (PHI nodes).
- Overly conservative decisions:
  - Sometimes needed by optimization.
  - Sometime to accommodate hacks in system code.
- Poor value range information/support (should be improved by Ranger).
- No support for symbolic ranges (Ranger support?)
- No support for definite loops (with known number of iterations).
- Premature folding:
  - E.g., `strcpy` to `memcpy`, or `memcpy` to `MEM_REF`.
  - Past the end accesses to constant aggregates folded to zero.
- Very limited analysis across function boundaries.
- Poor/limited LTO integration (some warnings not enabled).

Example: PHI Nodes Not Handled

Weaknesses of Flow-Based Diagnostics In GCC

```
void f (unsigned i, bool c)
{
  if (i < 4) return;         // i’s range is [4, UINT_MAX]
  char a[4], b[4];
  char *p = c ? a : b;
  p[i] = 0;                  // past-the-end store not detected!
  …
}
```

What to do in cases like:

```
  char a[8], b[4];
  char *p = c ? a : b;       // a’s big enough but b is not
```

- `-Wmaybe-array-bounds`? `-Wmaybe-stringop-overflow`?
- Introduce new levels? (Both warnings already have “levels”).

Example: Permissiveness For “Special” Code

Weaknesses of Flow-Based Diagnostics In GCC

- Trailing arrays of any size treated as flexible array members.
- `memcpy` bounds checking doesn’t consider member boundaries.
- `strcpy` lowered to `memcpy`.

```
struct Account {
  char name[8], passwd[8];
};

void f (struct Account *p)
{
  strcpy (p->name, "***invalid account***");   // overflow not diagnosed!
}
```

Example: Incomplete Range Support

Weaknesses of Flow-Based Diagnostics In GCC

- Conversions from signed to unsigned integers result in anti-ranges.
- Anti-ranges are tricky, prone to bugs, and (for the most part) not handled.

```
char* f (int n)
{
  if (n > 8)
     n = 8;                         // n’s range [INT_MIN, 8) converted to size_t

                                    // yields anti-range ~[9, 0xfffffff7fffffff]

  char *p = malloc (n);             // object is at most 8 bytes big
  strcpy (p, "0123456789");         // buffer overflow not diagnosed!

  …
}
```

Example: Poor Support for Definite Loops

Weaknesses of Flow-Based Diagnostics In GCC

- Out of bounds accesses to trailing arrays in definite loops aren't diagnosed consistently.

```
struct A { int a[4]; };

void f (struct A *p)
{
  p->a[sizeof p->a - 1] = 0;      // -Warray-bounds (good)
}

void g (struct A *p)
{
  for (unsigned i = 0; i != sizeof p->a; ++i)
     p->a[i] = i;                 // buffer overflow not diagnosed!
}
```

False Positives

Weaknesses of Flow-Based Diagnostics In GCC

- Full/Partial Redundancy Elimination (FRE/PRE).
- Lack of support for pointer relationships.
- Imperfect loop unrolling.
- Interaction with sanitizers.
- And of course, bugs…

Example: Redundancy Elimination

False Positives of Flow-Based Diagnostics In GCC

Array bounds checking with `-Warray-parameter` (under review).

```
union U { char a3[3], a5[5]; };

void f3 (char[static 3]);    // requires at least 3 elements
void f5 (char[static 5]);    // … at least 5 elements

void g (union U *p)
{
    f3 (p->a3);              // okay
    f5 (p->a5);              // okay
}
```

Example: Redundancy Elimination

False Positives of Flow-Based Diagnostics In GCC

Output of `-fdump-tree-fre3-details=/dev/stdout` :

```
;; Function g (g, ...)
Value numbering stmt = _1 = &p_3(D)->a3;
…
Replaced &p_3(D)->a5 with _1 in all uses of _2 = &p_3(D)->a5;
Removing dead stmt _2 = &p_3(D)->a5;
…
```

Example: Redundancy Elimination

False Positives of Flow-Based Diagnostics In GCC

Output of `-fdump-tree-fre3-details=/dev/stdout` continued:

```
g (union U * p)
{
  char[3] * _1;
  <bb 2>:

  _1 = &p_3(D)->a3;
  f3 (_1);
  f5 (_1);
  return;
}
```

Example: Redundancy Elimination

False Positives of Flow-Based Diagnostics In GCC

```
union U { char a3[3], a5[5]; };

void f3 (char[static 3]);    // requires at least 3 elements
void f5 (char[static 5]);    // … at least 5 elements

void g (union U *p)
{
    f3 (p->a3);              // okay
    f5 (p->a5);              // okay, but a bogus warning!
}
warning: ‘f5’ accessing 5 bytes in a region of size 3 [-Wstringop-overflow=]

     9 |    f5 (p->a5);
```

Example: Loop Unrolling

False Positives of Flow-Based Diagnostics In GCC

```
struct S { int x, y, z; };

struct S a[1];

void g (int n)
{
  for (int i = 0; i < n; i++)
    {
       memset (&a[i], 0, sizeof *a);
       a[i].x = 1;
    }
}
```

Example: Loop Unrolling

False Positives of Flow-Based Diagnostics In GCC

```
g (int n)
{
  int i;
  struct s * _1;

  <bb 2>:
  if (n_5(D) > 0)
     goto <bb 3>; [50.00%]
  else
     goto <bb 5>; [50.00%]

  <bb 3>:
  __builtin_memset (&a, 0, 12);
  a[0].x = 1;

  …
```

Example: Loop Unrolling

False Positives of Flow-Based Diagnostics In GCC

```
…
if (n_5(D) > 1)
   goto <bb 4>; [50.00%]
else
   goto <bb 5>; [50.00%]

<bb 4>:
_1 = &a + 12;
__builtin_memset (&MEM <struct s[1]> [(void *)&a + 12B], 0, 12);
__builtin_unreachable ();


<bb 5>:
return;
}
```

Example: Loop Unrolling

False Positives of Flow-Based Diagnostics In GCC

```
struct S { int x; };

struct S a[1];

void g (int n)
{
  for (int i = 0; i < n; i++) {
     memset (&a[i], 0, sizeof *a);
     a[i].x = 1;
  }

}
```

```
warning: 'memset' offset [12, 23] is out of the bounds [0, 12] of object 'a' with type
'struct s[1]' [-Warray-bounds]
     6 |  memset (&a[i], 0, sizeof *a);
```

Inconsistent Approaches

Weaknesses of Flow-Based Diagnostics In GCC

- Most warnings implemented independently of others.
- Most perform the same IL traversal.
- Little code sharing.
- Implemented in separate passes (restrict, sprintf, VRP).
- Duplication of code, effort and bugs.
- Intricate interactions (`-Warray-bounds`, `-Wstringop-overflow`).
- Duplicate warnings (`TREE_NO_WARNING`).

Missing Coverage

Weaknesses of Flow-Based Diagnostics In GCC

- Modifying non-modifiable objects (subobjects, strings, or functions).
- Invalid accesses to atomic or volatile objects.
- Invalid pointer arithmetic or relational expressions.
- Freeing pointers not returned from `malloc` (or `operator new`).
- Accessing freed memory.
- Using freed or other indeterminate pointers.
- Invalid accesses in const and pure functions.
- Overlapping accesses to restrict-qualified pointers.

Solving Weaknesses

Weaknesses of Flow-Based Diagnostics In GCC

- Develop general infrastructure.
- Consolidate as many access warnings as possible under one (or fewer) passes:
  - `gimple-ssa-path-isolation.c`
- Provide two levels: definite and “maybe.”
- Tighten up checkers and provide warning options for “special” code to opt out.
- Introduce codegen options to control response to detected problems:
  - optimize away (with warning),
  - insert `__builtin_trap` (with warning),
  - insert `__builtin_unreachable` (with warning).
- Defer warnings until expansion, avoid warning for dead code, and eliminate duplicates:
  - `__builtin_warning`
- Change FRE/PRE to avoid substituting members.
- Reduce unnecessary instrumentation by sanitizers.

Why Path Isolation Pass?

Solving Weaknesses of Flow-Based Diagnostics

`Gimple-ssa-path-isolation.c` , the home of `-Wreturn-local-addr`:

- Model design
  - Supports PHIs (conditionals) by issuing “may be” warnings.
  - Tracks flow through built-in calls.
  - Implements path isolation.
  - Low rate of false positives (see bug 90556).
  - Controls response (`flag_isolate_erroneous_paths_xxx`).
- Future work
  - Detect escaping through indirection
  - Add attribute `returns_arg` to support `strcpy/stpcpy`-kind of functions
  - Detect returning through out-of-line functions defined in the same TU

New Warnings In Development

New Flow-Based Diagnostics In GCC

- `-Warray-parameter`, `-Wvla-parameter`: Bounds checking of array and VLA function parameters.
- `-Wwrite-const`: Diagnose modifying non-modifiable objects (strings, functions, etc.)
- `-Waccess-atomic`, `-Waccess-volatile`: Diagnose invalid accesses to atomic- and volatile-qualified objects.
- `-Waccess-free`: Diagnose accesses to freed objects and uses of freed pointers.
- `-Wfree-nonheap-object`: Enhance detection of calls to `free` with pointers not returned from `calloc`/`malloc`/`realloc` or C++ `operator new`.
- `-Wconst-function-access`, `-Wpure-function-access`: Diagnose invalid accesses by const and pure functions.
- Add attribute `free` (and/or `dealloc`) to annotate user-defined functions that free (or otherwise deallocate) memory.

Ideas For Future Work

Flow-Based Diagnostics In GCC

Analysis/state sharing across function calls:

1. For each function in a translation unit:
    - Record every call *F(args)* to another out-of-line function *F* with parameters *parms* and known definition.
2. For each call *F(args)*:
    - Substitute *args* into *F*'s *parms* and reanalyze *F*'s definition for accesses, considering *args* values.
3. Optimize to minimize compilation cost.

Is there an existing infrastructure to build the above on?

Overlap With Static Analyzer

Flow-Based Diagnostics In GCC

- Analyzer advantages:
  - Can work harder (runtime overhead is more acceptable).
  - Can analyze paths not interesting for optimization.
  - Not subject to inlining and other optimizer constraints.
  - Not affected by optimizing transformations/folding, etc.
  - Higher rates of false positives acceptable.
- Advantages of middle end warnings:
  - Reuse of existing optimizer infrastructure.
  - False positives/negatives often expose missing optimization.
  - Analysis opens up further optimization opportunities:
    - e.g., `sprintf`, `strlen`.
    - Path isolation.
  - Can modify generated code (fold code, inject traps, etc.)

Overlap With Static Analyzer

Flow-Based Diagnostics In GCC

Ideal goals:

- Present information in a consistent form (need conventions):
  - Same distinction between/control of definite vs “maybe” warnings.
  - Same notation for offsets, sizes, PHI nodes.
  - Same depiction of data/control flow?
- Avoid issuing duplicate diagnostics.
- Minimize duplication of code/logic with middle end warnings.
- Take advantage of existing middle end infrastructure?
- Share tests?

Open questions:

- Are any bugs/warnings ideally suited for middle end vs analyzer?
- How to decide where to invest resources?

Questions?

Feel free to email
msebor@gmail.com
or
gcc-help@gcc.gnu.org

THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHatNews

youtube.com/user/RedHatVideos