Static Analysis In the GCC Middle-End

Martin Sebor
Toolchain Team
Red Hat

GNU Tools Cauldron 2018

Brief Intro

My background:

- Red Hat GCC toolchain engineer.
- Focus on GCC static analysis/security/diagnostics improvements (e.g., `-Wrestrict`, `Wxxx-ovrerflow`, `-Wxxx-truncation`, …).
- Standards development and conformance.

Goals Of This Talk

- Share lessons from implementing warnings in the middle-end.
- Highlight both opportunities and challenges for future work.
- Drum up interest in future enhancements in this space.
- Long term: By eliminating bugs open up more optimization opportunities and help find even more (and more obscure) bugs.

- Not a proposal to turn GCC into a full-blown static analyzer…
- ...but rather an argument to make better use of existing infrastructure to aid in detecting and diagnosing certain or likely bugs/undefined behavior in user code…
- ...and to help improve the overall quality of the implementation…
- ...and drive further improvements to software.

Why Static Analysis In a Compiler?

- Natural extension of detecting language constraint violations.
- Increasing interest in “building security in” from the first stages of development — but not just security.
- The historical view to *trust the programmer* isn’t viable anymore.
- Increasing pressure to detect (and even mitigate) undefined behavior.
- The earlier a bug is detected the cheaper it is to fix.
- Increasing software complexity makes detecting bugs by traditional means less effective.
- Dedicated static analysis has a high process and “compile-time” overhead and doesn’t run early or often enough. May not be practical for huge code bases.
- Dynamic analysis has a runtime overhead and doesn’t run often enough, or at all, and exercises only executed code, and only with very limited data sets.
- The earliest time to detect bugs is during compilation, before they are committed.

Why Static Analysis In GCC?

- The historical view to *trust the programmer* isn’t viable anymore.
- Many useful diagnostics implemented in GCC front-ends.
- GCC is a feature-rich, mature optimizing compiler.
- Many deep flow analysis passes used for optimization.
- Some optimizations already detect undefined behavior and try to deal with it (in some way, though rarely by diagnosing it).
- Gaps and inconsistencies exist in detection/mitigation/diagnostic strategies.
- Many opportunities to improve the detection and expose it to users as warnings.
- Detection brings to the fore the question of appropriate mitigation.
- Increasing pressure from users, industry, and other compilers to do all three.
- It just makes sense! (We end up with a better, more user-friendly compiler.)

Next: Purpose of warnings.

Purpose Of Warnings

GCC definition:
Diagnostic messages that report constructions that are not inherently erroneous but that are risky or suggest there may have been an error.

"*Not inherently erroneous*" — not necessarily causing a compilation error (without `-Werror`).

"*Suggest there may have been an error*" — warnings may have false positives (and negatives), or may simply point out suspicious or *risky* code.

The goal is to help detect likely coding errors of all kinds without drowning users in too much noise: strike a reasonable balance between false positives and negatives, i.e., between *soundness* and *completeness*.

General Strategies

Pattern-based:

- “Glorified regular expressions” usually operating on declarations or expressions or their types.
- Theoretically both *sound* and *complete*: Zero rate of *false positives* and *negatives*.
- Suitable for a front-end implementation.

Flow-based:

- Depend on control- of data- flow analyses, ideally across the whole program.
- *False positives* and *negatives* unavoidable.
- Not suitable for a front-end alone.

(Not discussed: *Metric-based*)

True And False Positives And Negatives

- *True Positive*: Diagnostic issued according to design and documentation.
  Does not necessarily imply a bug.
  In practice/informally: a diagnostic most users find helpful.
- *False Negative*: Missing diagnostic where one should be issued according to design and documentation.
- *False Positive*: Diagnostic issued contrary to design and documentation.
  In practice/informally: a diagnostic users don’t agree with.
- *True Negative*: No diagnostic when none is expected.

Let’s see some examples…

Example True Positive

`-Wchar-subscripts`: *Warn if an array subscript has type char. This is a common cause of error, as programmers often forget that this type is signed on some machines.*

Note: Warning issued even with `-fno-signed-char`. (Does that make it a false positive?)

```
enum { Alpha = 1, Control = 2, Digit = 4, /* … */ };

static const char masks[256] = { /* … */ };

int is_digit (char c)
{
    return masks[c] & Digit;    // -Wchar-subscripts
}
```

Example False Negative

`-Wlogical-op`: *Warn about suspicious uses of logical operators in expressions. This includes using logical operators in contexts where a bit-wise operator is likely to be expected. Also warns when the operands of a logical operator are the same*.

```
int f (int i)
{
    if (i == 0 || i == 0 || i == 1)    // -Wlogical-op
        return 0;

    if (i == 0 || i == 1 || i == 0)    // PR 81320: missing warning
        return 0;                       // (or is it?)
    return 1;
}
```

Example False Positive

`-Wtype-limits`: *Warn if a comparison is always true or always false due to the limited range of the data type, but do not warn for constant expressions.*

```
int f (void)
{
    return (unsigned) -1 < 0;    // PR 86647: bogus -Wtype-limits
}
```

Constant expression.

Next: Where warnings are implemented.

Where Warnings Are Implemented

GCC issues warnings at distinct phases of compilation:

- Front-ends.
- Middle-end (GIMPLE). Usually only with optimization.
- During (or just before) expansion to RTL.
- Back-ends (mostly command line option issues and `-Wattributes`).
- While emitting assembly (rare).

Next: Advantages of front-end warnings.

Advantages Of Front-End Warnings

- Simple: Pattern-based heuristics often based on lexical or type system rules.
- Low noise: Relatively low rate of false positives (depends on rule definition).
- High precision: Theoretically no false negatives (though potentially at the cost of false positives).
- Stable/consistent: Not impacted by code transformations/optimization.
- Target independent (except for type system dependencies).
- Easy to understand and debug.
- Easily suppressed (also a downside).

Pattern-Based Front-End Warnings

Type-based.

Lexical pattern-based.

-Wattributes, -Wbad-function-cast, **-Wbool-operation**, -Wcast-qual, -Wcast-align, -Wcatch-value, **-Wdangling-else**, -Wdesignated-init, -Wdiscarded-qualifiers, -Wdiscarded-array-qualifiers, -Wdiv-by-zero, -Wignored-attributes, **-Wignored-qualifiers**, -Wimplicit-int, -Wimplicit-function-declaration, -Wincompatible-pointer-types, -Wint-conversion, **-Wlogical-op**, **-Wlogical-not-parentheses**, **-Wmissing-attributes**, **-Wmissing-braces**, -Woverflow, -Woverlength-strings, -Wpacked, -Wpacked-bitfield-compat, -Wpacked-not-aligned, -Wpadded, -Wpointer-arith, -Wpointer-compare, -Wpointer-sign, -Wsequence-point, **-Wshadow**, -Wswitch-default, -Wswitch-bool, -Wunused-but-set-parameter, -Wunused-but-set-variable, -Wunused-parameter, -Wunused-value, **-Wunknown-pragmas**, -Wunused-label, -Wwrite-strings

Other Front-End Warnings

Partly in the folder.

Prone To False Negatives and False Positives:

`-Wconversion` (PR 83112), `-Wconversion-null`, `-Wduplicated-branches`, `-Wduplicated-cond`, `-Wplacement-new`, `-Wshift-count-negative`, `-Wshift-count-overflow`, `-Wshift-negative-value`, `-Wshift-overflow`, `-Wbool-compare`, `-Wswitch-enum`, `-Wswitch-unreachable`, `-Wtautological-compare` (PR 80087) `-Wzero-as-null-pointer-constant`, `-Wsign-conversion`

Prone to False Negatives:

`-Wformat`, `-Wmemset-elt-size`, `-Wmemset-transposed-args`, `-Wstrict-aliasing`

Gray Area:

`-Wdiv-by-zero`, `-Wtautological-compare`, `-Woverflow`

Downsides Of Front-End Warnings

- Some noise: In practice non-zero rate of false positives (see last bullet).
- Simplistic: Relatively low efficacy (good at finding “superficial” bugs).
- Limited insight: No data or control flow analysis available.
- In theory, both *sound* and *complete* but...
- ...in practice, not always chosen appropriately (and thus suffer from false positives and negatives).

Let’s see some examples...

`-Wformat` False Negative

`-Wformat`: *Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense.*

Case #1 detected — workaround for front-end limitation. Cause: No constant propagation.

```
int f (char *d, int i, int x, double y)
{
  sprintf (d, i ? "%i" : "%e", i ? x : y);    //#1: -Wformat

  const char *f = i ? "%i" : "%e";
  sprintf (d, f, i ? x : y);                  //#2: false negative
}
```

`-Wsign-compare` False Positive

`-Wsign-compare`: *Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned.*

PR 38470. Root cause: No value range propagation.

```
int f (int i)
{
    return i >= 0 && i < sizeof (int) ? 1 : 2; //bogus -Wsign-compare
}
```

`-Wswitch` False Positive

`-Wswitch`: *Warn whenever a switch statement has an index of enumerated type and lacks a case for one or more of the named codes of that enumeration.*

PR 23577, PR 50422. Root cause: No control flow/value-range analysis.

```
enum E { E0, E1, E2 };
void f (enum E e, int *p)
{
    if (e == E0) return;
    switch (e) {   // enumeration value 'E0' not handled
    case E1: *p = 1; break;
    case E2: *p = 2; break;
    }
```

`-Wshift-count-negative` False Negative

`-Wshift-count-negative`: *Warn if shift count is negative.*

Root cause: No constant propagation.

```
int f (int i)
{
    if (i < 123)
        return i << -1;     // -Wshift-count-negative
    int n = -1;             // not a compile-time constant
    return i << n;          // false negative
}
```

`-Wshift-count-overflow` False Positive

`-Wshift-count-overflow`: *Warn if shift count >= width of type.*

Root cause: No dead code elimination.

```
enum { N = 33 };

int f (long x)
{
    if (sizeof (x) > 4)    // not eliminated early enough
        return x << N;     // false positive
    return -1;
}
```

Next: When to implement a front-end warning.

When To Implement Front End Warning

- Purely lexical or type-based rules.
- Independent of operand values: consider compile-time constants vs others.
- Independent of control flow: consider unreachable code.

For everything else, consider either a middle-end implementation or a combined approach.

But beware of middle-end challenges (discussed later).

Next: Middle-end warnings.

Middle-End Warnings

Both front- and middle-end.

Control or data-flow based warnings:

`-Walloc-zero`, `-Walloca`, `-Walloca-larger-than`, `-Warray-bounds`, `-Wframe-address`, `-Waggressive-loop-optimizations`, `-Wformat-overflow`, `-Wformat-truncation`, `-Winline`, `-Wlarger-than`, `-Wnonnull`, `-Wimplicit-fallthrough`, `-Wmaybe-uninitialized`, `-Wnull-dereference`, `-Wreturn-local-addr`, `-Wreturn-type`, `-Wstrict-overflow`, `-Wstringop-overflow`, `-Wstringop-truncation`, `-Wsuggest-attribute`, `-Wframe-larger-than`, `-Wrestrict`, `-Wtype-limits`, `-Wstack-usage`, `-Wunsafe-loop-optimizations`, `-Wclobbered`, `-Wuninitialized`, `-Wunused-function`, `-Wunused-result`, `-Wunused-variable`, `-Wunused-const-variable`, `-Wuninitialized`, `-Wvla-larger-than`

What Is the Middle-End?

- All analysis and transformation stages between parsing and RTL generation.
- Starting with GENERIC to GIMPLE transformation (gimplification).
- Includes constant propagation, inlining, value-range propagation, pointer alias analysis, dead code elimination, dead store elimination, object size analysis, string length optimization, sprintf analysis, path isolation, and dozens of others.
- Ends with RTL expansion.

Advantages Of Middle-End Warnings

- Access to (most) GCC’s middle-end analysis passes.
- Can detect non-trivial bugs involving complex computation
- Inlining exposes data flow across function call boundaries.
- LTO exposes flow across translation units.
- Complex heuristics possible.

Next: Adding a middle-end warning.

Adding a Middle-End Warning

- Prefer a dedicated warning pass unless integrating with an existing warning.
- Consider when to run it. Earlier is less effective but subject to fewer interactions.
- Does it make sense to also run without optimization?
- Does it need to trigger before/during folding? (Prefer to avoid warning from the folder.)
- Is it subject to “false positives?” (Consider a two-level option.)
- Consider interaction with LTO and sanitizers.
- Avoid issuing duplicate warnings (Use `TREE_NO_WARNING`/`gimple_no_warning`.)
- Consider interaction with other warnings. (Same as above.)
- Does it need to trigger for expansion of macros defined in system headers? (Use `expansion_point_location_if_in_system_header()`)
- Should it be enabled by `-Wall`, or `-Wextra`, or off by default?
- Consider suppression mechanisms.
- Consider false positives and negatives when documenting.

Information to Include in Warnings

- Read Diagnostic Guidelines on the Wiki.
- Provide sufficient detail to describe the problem. Avoid generalizations like “access is undefined” or “argument is invalid.”
- Avoid guessing at programmer’s intent or suggesting a solution — if it isn’t right it will only confuse the user.
- Include relevant operand types, values, or ranges. Do not assume they are apparent from the source code (they may be obscured by macros, constants, typedefs, or template arguments).
- Use `inform()` to reference the relevant declarations if they are distant from the locus of the diagnostic (in a different function or even file).
- Consider localization (avoid `warning(i == 1 ? “%i byte” : “%i bytes”);`).
- Include the inlining context (`%G` and `%K`).
- Compare message to other compilers.

Insufficient Detail

Which subscript, what's its value, what array, what is the bound, what caller?

```
void f (int (*p)[][5], int i, int j) {
    (*p)[i][j] = 2;
}

void g (int i, int j) {
    extern int a[3][5];
    f (&a, i, j < 5 ? 5 : j);
}
warning: array subscript is above array bounds [-Warray-bounds]
    (*p)[i][j] = 2;
    ~~~~~~~~^~~~~~~
```

Right Amount of Detail?

Balance insufficient vs excessive detail.

```
const char a[] = "test string";
char d[8];

void f (int i)
{
  if (i < -99 || 999 < i)
     i = -99;
  sprintf (d, "%i: %s", i, a);
}
```

Right Amount of Detail? (cont’d)

```
In function ‘f’:
warning: ‘%s’ directive writing 11 bytes into a region of size between
3 and 5 [-Wformat-overflow=]

9 |   sprintf (d, "%i: %s", i, a);
  |                    ^~      ~
note: ‘sprintf’ output between 15 and 17 bytes into a destination of
size 8
9 |   sprintf (d, "%i: %s", i, a);
  |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Too Much Detail or Not Enough?

```
void g (int i, int j, int n)
{
    extern char a[9];
    if (i < 1 || 3 < i) i = 1;
    if (j < 2 || 3 < j) j = 2;
    if (n < 3 || 4 < n) n = 3;
    memcpy (a + i, a + j, n);
 }
```

Offsets from what?

```
warning: 'memcpy' accessing between 3 and 4 bytes at offsets [1, 3]
and [2, 3] overlaps between 1 and 4 bytes at offset [2, 3]
[-Wrestrict]
  8 |   memcpy (a + i, a + j, n);
```

Testing a Middle-End Warning

- Verify that inlining context is included (`%G` and `%K`). Use `{ dg-message "function .foo..*inlined from .bar." "" { target *-*-* } 0 }`.
  (See existing tests for examples.)
- Verify expression operands are underlined as expected. Use `dg-begin-multiline-output`and `dg-end-multiline-output`.
- Use `profiledbootstrap`to find false positives in GCC builds.
- Build a few packages, such as Glibc, GDB, and the Linux kernel and look for unexpected warnings.
- Be on the lookout for known problems (discussed later).
- Verify that suppression mechanisms work.

Next: Ideas for middle-end warning projects.

Ideas For Middle-End Warnings

- `-Wstringop-overflow`for user-defined functions (WIP).
- Invalid `malloc`/`free` and `new`/`delete` calls, including double free, leaks, and mismatched calls. (Includes optimization opportunities.)
- Invalid pointer arithmetic (including forming out-of-bounds pointers or using pointers to different objects in the same subtraction or inequality expression).
- Common C++ exception-safety bugs (includes optimization opportunities).
- Passing unterminated arrays to string functions (WIP).
- Accessing objects via incompatible or misaligned lvalues (including `const`/`volatile`).
- Missing error handling for C/POSIX library calls, including misuses of `errno`.
- Invalid or suspicious signal handlers.
- Concurrency errors (Clang `-Wthread-safety`).

Next: Downsides of middle-end warnings.

Downsides Of Middle-End Warnings

- Very limited or unavailable without optimization.
- Often affected by optimization levels and choice of target.
- Influenced by/incompatible(?) with sanitizers.
- False positives can be difficult to suppress.
- Can be tricky to implement (see next slide).

Next: Challenges for middle-end warnings.

Challenges For Middle-End Warnings

- Dependency on optimization.
- Lack of data sharing between passes (e.g., `objsize`, `strlen`, `sprintf`).
- Interference from early folding/simplification of expressions.
- Lack of distinction between explicit and implicit casts (`NOP_EXPR`).
- Lost information about structure of data due to prior transformations (`MEM_REF`).
- Unreliability of pointer types and pointer provenance.
- Adverse interactions with other passes (jump threading, sanitizers).
- Tension between code analysis for optimization vs warning.
- Poor selective suppression support.
- Philosophical differences.

Let’s look at some examples...

Early `strlen` Folding To Integer

`strlen (&a[i])` folded to `(3 - i)` during gimplification:

```
const char a[] = "123";

void f (int i)
{
    if (i < 7)
        i = 7;
    // missing -Warray-bounds, folded to true(!)
    if (strlen (&a[i]) > 3)
        puts ("impossible!");
}
```

Early `strlen` Folding - the Code

GCC internal `c_strlen` function:

```
if (string_length (ptr, eltsize, maxelts) < maxelts)
  /* Return when an embedded null character is found.  */
  return NULL_TREE;

…
   /* We don't know the starting offset, but we do know that the string
 has no internal zero bytes.  We can assume that the offset falls
 within the bounds of the string; otherwise, the programmer deserves
 what he gets.  Subtract the offset from the length of the string,
 and return that.  This would perhaps not be valid if we were dealing
 with named arrays in addition to literal string constants.  */

return size_diffop_loc (loc, size_int (maxelts * eltsize), byteoff);
```

Legacy code.
Not the current thinking!

Early Folding To `memcpy`

`strcpy/strncpy` to `memcpy` folding prevents `-Wstringop-overflow.`
(`memcpy` is allowed to write across multiple struct members, thus no warning).

```
struct S { char a[4]; void (*pf)(void); };

void f (struct S *p)
{
    // folded to memcpy (p->a, “1234567”, 8)
    // missing -Wstringop-overflow
    strcpy (p->a, "1234567");
}
```

Unreliable Pointer Provenance

Missing `-Wstringop-overflow` (except with `_FORTIFY_SOURCE=2`):

```
struct S { char a[3], b[5], c; } s[3];

void f (int i, int j)
{
    char *p = i ? s[i].a : s[j].b;
    strcpy (p, "1234567");   // buffer overflow!
}
```

Unreliable Pointer Provenance (cont’d)

```
// GIMPLE before laddress:

<bb 2> [local count: 1073741825]:
if (i_2(D) != 0)
    goto <bb 3>; [50.00%]
else
    goto <bb 4>; [50.00%]

<bb 3> [local count: 536870912]:
iftmp.0_3 = &s[i_2(D)].a;
goto <bb 5>; [100.00%]
```

```
// GIMPLE before laddress (cont’d):


<bb 4> [local count: 536870912]:
iftmp.0_5 = &s[j_4(D)].b;

<bb 5> [local count: 1073741825]:
# iftmp.0_1 = PHI <iftmp.0_3(3),
iftmp.0_5(4)>
memcpy (iftmp.0_1, "1234567", 8);
```

Strcpy replaced during gimplification.

Unreliable Pointer Provenance (cont’d)

```
// GIMPLE after laddress:

<bb 2> [local count: 1073741825]:
if (i_2(D) != 0)
    goto <bb 3>; [50.00%]
else
    goto <bb 4>; [50.00%]

<bb 3> [local count: 536870912]:
_8 = (sizetype) i_2(D);
_9 = _8 * 9;
iftmp.0_3 = &s + _9;
goto <bb 5>; [100.00%]
```

```
// GIMPLE after laddress (cont’d):


<bb 4> [local count: 536870912]:
_10 = (sizetype) j_4(D);
_11 = _10 * 9;
_12 = _11 + 3;
iftmp.0_5 = &s + _12;

<bb 5> [local count: 1073741825]:
# iftmp.0_1 = PHI <iftmp.0_3(3),
iftmp.0_5(4)>
memcpy (iftmp.0_1, "1234567", 8);
```

Interaction With Sanitizers

PR 82076. Compile with `gcc -DN=3 -O2 -Wall -fsanitize=undefined`

```
int main (void)
{
    char a[4];
    const char *s = "1";
    memcpy (a, s, N);   // -Wstringop-overflow (read past end)
    puts (a);
}
```

...but runs with no error.

GIMPLE for -DN=3

`memcpy` of an odd size not folded to `MEM_REF`, overflow detected, but no error at runtime.

```
main ()
{
  char a[4];
  <bb 2> [local count: 1073741825]:
  memcpy (&a, "1", 3);    // -Wstringop-overflow
  puts (&a);
  a ={v} {CLOBBER};
  return 0;
}
```

Interaction With Sanitizers (cont’d)

Compile with `gcc -DN=4 -O2 -Wall -fsanitize=undefined`

```
int main (void)
{
    char a[4];
    const char *s = "1";
    memcpy (a, s, N);   // missing -Wstringop-overflow
    puts (a);
}
```

...but error at runtime:

```
runtime error: load of address 0x000000400724 with insufficient space
for an object of type 'char'
```

GIMPLE for -DN=4

`memcpy` with a small power of 2 folded to `MEM_REF`, sanitizer detects overflow but no warning.

```
main ()
{
  char a[4];
  unsigned long _8 = (unsigned long) "1";
  __builtin___ubsan_handle_type_mismatch_v1 (&*.Lubsan_data2, _8);
  unsigned int _2 = MEM[(char * {ref-all})"1"];
  MEM[(char * {ref-all})&a] = _2;
  puts (&a);
  a ={v} {CLOBBER};
}
```

Tension Between Optimization And Warning

- Control and data flow analyses serve optimizations; warning is secondary.
- Flow-based warnings are only as good as the analyses they depend on.
- Optimizers tend to simplify code early to enable more simplification later on.
- Warnings depend on useful information that gets lost in some transformations.
- Optimization focuses on making common coding patterns efficient.
- Bugs are often (not always) in “unusual code.”
- Optimizers must be conservative and assume questionable/difficult to analyze code is correct and avoid optimizing (and give up further analysis).
- It’s acceptable, even useful, for warnings to assume questionable code could be the source of bugs (at the cost of some false positives).

Philosophical Differences

- Driven by the tension between optimization and warning.
- Common view: GCC is an optimizing compiler — first and foremost.
- Okay to diagnose undefined behavior when detected but not to go out of our way to look for it.
- Warning code “pollutes” (or distracts from) the main purpose — optimization.
- Different tolerance for rates of false positives, among users and GCC developers.
- Ease vs. difficulty of suppression is subjective.
- Main suppression mechanism doesn’t work well for middle-end warnings.

Mitigation Of Undefined Behavior

What to do when analysis uncovers undefined behavior?

Discussed in detail in *Future Directions for Treatment of Undefined Behavior in C*.

Approaches:

- Fold to zero or some safe value (e.g., out-of-bounds read from a constant).
- Emit bad code (e.g., most out-of-bounds writes).
- Fold to a safe value or range (e.g., `strlen()` of arrays of known size).
- Replace with `__builtin_trap()` (e.g., returning from a non-void function in C++).
- Insert `__builtin_unreachable()`.
- One size probably does not fit all.