Memcheck vs Optimising Compilers: keeping the false positive rate under control

Julian Seward, jseward@acm.org

4 February 2018. FOSDEM. Brussels.
Motivation

Memcheck checks

Whether memory accesses are to allowable locations
(Relatively) easy

Whether branches depend on undefined data
(Relatively) difficult

Low false positive rates are very important
Circa 2005  Everything under control
Circa 2015  Increasingly problematic – clang 3+, gcc 5+

Overview
Some definedness tracking basics; some building blocks
Some problems to which we have solutions
Some problems with no solutions (so far)
Some basics

For every bit of process state, Memcheck maintains a shadow ("V") bit
    all registers and memory locations are shadowed

1 means Undefined.  0 means Defined.

When program computes a result from operands ..
    \[ r = x + y \]
.. Memcheck computes definedness of result from definedness of operands
    \[ r# = \ldots x# \ldots y# \ldots \]

When program does a conditional branch, Memcheck checks definedness of the condition
    and emits an error if undefined

As described in our Usenix 2004 paper (Seward & Nethercote)
Some building blocks

**UifU** -- “Undefined if either is Undefined”
- eg $\text{UifU}(\text{DDDU}, \text{DDUD}) = \text{DDUU}$
- implementation: $\text{UifU}(x\#, y\#) = x\# | y\#$

**DifD** -- “Defined if either is Defined”
- dual to UifU, implemented with $\&$
- eg $\text{DifD}(\text{DDDU}, \text{DDUD}) = \text{DDDD}$

**Left** -- propagate undefinedness leftwards in word
- eg $\text{Left}(\text{DUDUDD}) = \text{UUUUDD}$
- implementation: $\text{Left}(x\#) = x\# | (- x\#)$

**PCast** -- pessimistic cast. Changes size. Any Us in input cause all output to be Us.
- eg $\text{PCast\_to\_4bits}(\text{DDDDUDDD}) = \text{UUUU}$
- $\text{PCast\_to\_4bits}(\text{DDDDDDDD}) = \text{DDDD}$
- Most important case is PCasting down to a single bit
- implementation: $\text{PCast}(x\#) = (x\# | (- x\#)) >> \text{signed} \ (\text{dest\_size}-1)$
Instrumenting addition

\[ r = x + y \]

if any input bit is \( U \) then the corresponding output bit is \( U \)

Hence

\[ r# = UifU(x#, y#) \]

Ignores carry propagation. Assume worst case

\[ r# = \text{Left}( \text{UifU}(x#, y#) ) \]

cheap: \text{mov}, \text{or}, \text{mov}, \text{neg}, \text{or}

Is overly conservative

defined zeroes stop carry propagation

and LLVM knows that :-(

Instrumenting AND and OR

\[ r = x \& y \]

as with addition -- start with bitwise propagation to output

\[ r# = \text{UifU}(x#, y#) \]

But .. AND with defined zero is defined.
Too pessimistic. And it matters.

Fold in “improvement” terms for defined zero bits in input

\[ r# = \text{DifD}( \text{UifU}(x#, y#), \text{ImproveAND}(x, x#), \text{ImproveAND}(y, y#) ) \]

where \( \text{ImproveAND}(q, q#) = q \mid q# \)

Exact same story (modulo De Morgan) for OR.

This is exact! Yay.
bool r = (x == y)
Takes 2 (eg) 32 bit ints and produces a 1-bit result

Use old friend $\text{UifU}$ and new friend $\text{PCast}$, to turn result into a single bit
$r\# = \text{PCast}( \text{UifU}(x\#, y\#) )$

Result is undefined if any input bit is undefined. Sounds reasonable?
bool r = (x == y)
Takes 2 (eg) 32 bit ints and produces a 1-bit result

Use old friend UifU and new friend PCast, to turn result into a single bit
r# = PCast( UifU(x#, y#) )

Result is undefined if any input bit is undefined. Sounds reasonable?

struct { short x; short y; }
if (p->x == 0x1234 && p->y == 0x5678)
becomes
  cmpl $0x12345678, (p)

If p->x is not 0x1234 and p->y is undefined, the C source is fine
  but the machine code contains a comparison on partially uninitialised data

Thanks gcc5! (or is it clang?)
Observation:
Result is defined if we can find two corresponding input bits, which are defined but different

\[ XX0XX == XX1XX \text{ defined!} \]
\[ XX0XX == XX0XX \text{ we don't know} \]

We can fix up our scheme..

If \( r = (x == y) \)
then \( r# = \text{PCast}( \text{DifD( UifU(x#, y#), OCast(improver) )}) \)
where \( \text{improver} = x# | y# | \neg(x \land y) \)
\[ \text{OCast(vec)} = (\text{vec} - (\text{vec} \gg \text{unsigned 1})) \]
\[ \gg \text{signed (word_size-1)} \]

memcheck/mc_translate.c, function expensiveCmpEQorNE
Hard to understand, verify, prove right

Is totally Not-Obvious! \text{OCast} was “discovered” by the GNU superoptimizer.
Can we do better?
Kinda.

We know exact instrumentation schemes for AND, OR, NOT, XOR on individual bits
We can write any combinatorial logic function using any 3 of AND, OR, NOT, XOR

So we can mechanically derive V bit rules
eg \([x_2, x_1, x_0] == [y_2, y_1, y_0]\)
   \(-\rightarrow (x_2 == y_2) & (x_1 == y_1) & (x_0 == y_0)\)
   \(-\rightarrow ~(x_2 \oplus y_2) & ~(x_1 \oplus y_1) & ~(x_0 \oplus y_0)\)

I proved the “informal” exact integer equality case to be correct

Open question: can we prove the exact integer ADD/SUB cases to be correct?
memcheck/mc_translate.c, function expensiveAddSub
Current status (V git repo)

New in 3.14 (unreleased):
   Integer ADD/SUB: exact where needed, cheap when not
   Integer EQ/NE: exact by default

Long since implemented
   AND, OR, XOR, NOT, shifts, widening, narrowing: exact
   Most other stuff -- approximated

Works fairly well for gcc 7, clang 5, rustc compiled code

Open-ish questions
   POWER 3-way comparisons (bug 386945) fixable, but very expensive
   Can we do ADD/SUB, EQ/NE faster?
   Can we be cleverer about the instrumentation?
      Abstract interpretation of monotonic functions on 2 x..x 2 lattices?
XOR falls outside the framework
   produces defined result (bitwise) if the same bit is given for both args
Identity of values matters
MSVC bitfield assignment \( a ^ \left( (a ^ b) \& c \right) \) causes problems
We sometimes rewrite it to the GCC form: \( (a \& \sim c) \mid (b \& c) \)
Really open questions 2

XOR falls outside the framework
produces defined result (bitwise) if the same bit is given for both args
Identity of values matters
MSVC bitfield assignment \( a ^ ( (a ^ b) \& c) \) causes problems
We sometimes rewrite it to the GCC form: \( (a \& \sim c) \mid (b \& c) \)

More serious
Memcheck assumes all conditional branches matter, but:
gcc 6+/clang 4+:
\[
\text{if } (A \&\& B) \ldots \quad \rightarrow \quad \text{if } (B \&\& A) \ldots
\]
if A is always false whenever B is undefined

int result;
bool ok = fn(&result, ...)
if (ok \&\& result == 42) ... 

Don't know what to do about this
End of the road?
Need new instrumentation framework/scheme?
So, in conclusion..

We saw..

.. some simple examples of definedness tracking
.. some cases where improved precision is needed
.. lots of complexity and expense in implementation and validation

We need..

.. people with mathematical skills and enthusiasm, to try and improve this

Thank you for listening!

Questions?