Undefined Behavior in 2017

John Regehr
University of Utah, USA

http://regehr.org/cppcon17.pdf

- What is undefined behavior (UB)?
- Why does it exist?
- What are consequences of UB in C and C++?
- Modern UB detection and mitigation

sqrt(-1) = ?

- i
- NaN
- Throw an exception
- Abort the program
- Return an arbitrary value
- Undefined behavior

- **Undefined behavior** (UB) is a design choice
  - But a somewhat extreme one
- UB is the most efficient alternative because it imposes the fewest requirements on the compiler
  - "… behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements"

$$a = b$$

$$a^2 = ab$$

$$a^2 - b^2 = ab - b^2$$

$$(a + b)(a - b) = b(a - b)$$

$$a + b = b$$

$$b + b = b$$

$$2 = 1$$

NO

```
#include <iostream>

int main(void) {
  int a[] = {1, 2, 3};
  int *p = (int *)(1 + (char *)a);
  std::cout << *p << "\n";
  return 0;
}
```

You might say:

"This program is fine, because x86 and x86-64 support unaligned data accesses"

```
#in

int
   i
   i
   s
   r
}
```

You
"This
support unaligned data accesses

This is like saying:

"Somebody once told me that in basketball you can't hold the ball and run. I got a basketball and tried it and it worked just fine. He obviously didn't understand basketball."

http://www.eskimo.com/~scs/readings/undef.950311.html

This is like saying:

"Somebody once told me that in

t hold the ball

ketball and

ed just fine.

understand

basketball."

http://www.eskimo.com/~scs/readings/undef.950311.html

And anyhow, unaligned accesses can cause crashes on x86-64 via e.g. movdqa

You

"This

support unaligned data accesses

C and C++ have lots of UB

- To avoid overhead
- To avoid compiler complexity
- To provide maximal compatibility across implementations and targets

Does UB really avoid overhead?

- Can only answer this case by case
- Consider a shift: `x << y`
- What happens when y is too large?
  - At least 3 different behaviors are seen in CPUs
  - The language standard could say anything, but what behavior do we really want?
  - Masking the shift amount by the number of bits in the left-hand operand would cost one instruction on all architectures I can think of
  - So yes, this UB avoids overhead

According to Appendix J of the standard, C11 has 199 undefined behaviors

- But this list isn't complete
- I don't know of a comparable list for C++
  - Tricky since not "undefined behavior" does not appear in the description of all UBs
  - And new UBs are being added

8) Integer, floating-point, or enumeration type can be converted to any complete enumeration type. The result is unspecified (until C++17)undefined behavior (since C++17) if the value of *expression*, converted to the enumeration's underlying type, is out of range (if the underlying type is fixed, the range is the range of the type. If the underlying type is not fixed, the range is all values possible for the smallest bit field large enough to hold all enumerators of the target enumeration)

From: http://en.cppreference.com/w/cpp/language/static_cast

What happens when your program executes undefined behavior?

- Case 1: Program breaks immediately
  - Segfault, math exception, compiler error, …
- Case 2: Program continues, but will fail later
  - Corrupted RAM, …
- Case 3: Program works as expected
  - This UB is a "time bomb" − a latent problem that might go off when compiler, compiler version, or optimization level is changed
- Also, of course, sometimes we can't trigger the UB (but an attacker can…)

Important trends over the last 25 years:

- UB detection tools have been getting better
  - Starting perhaps with Purify in the early 1990s
- Compilers have been getting cleverer at exploiting UB to improve code generation
  - The time bombs have been regularly going off since about 2000
- Security has become a primary concern in software development

- **Q:** What happens when you tell a compiler developer that changing optimization level broke your program that executes UB?
- **A:** The compiler developer…
  - recommends reading the standard
  - recommends being more careful in the future
  - wishes you good luck in your future endeavors

```
lias.c, (1896:25)> : Op: +, Reason : Signed Addition Overflow, BINARY OPERATION: left (int32): -2147483647 right (int32): -4
lias.c, (322:44)> : Op: *, Reason : Signed Multiplication Overflow, BINARY OPERATION: left (int32): 399999996 right (int32): 8
uiltins.c, (7681:57)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
uiltins.c, (7699:57)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
uiltins.c, (7709:25)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
uiltins.c, (7717:25)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
ombine.c, (10620:62)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
ombine.c, (10655:62)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
ombine.c, (11350:54)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
ombine.c, (7047:63)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
ombine.c, (7205:54)> : Op: +, Reason : Signed Addition Overflow, BINARY OPERATION: left (int32): 2147483647 right (int32): 1
ombine.c, (7838:22)> : Op: <<, Reason : Unsigned Left Shift Error: Right operand is negative or is greater than or equal to the wi
onfig/i386/i386.c, (10253:10)> : Reason : The current index is greater than array size!
onfig/i386/i386.c, (16316:17)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): 0 right (int32): -21
onfig/i386/i386.c, (16362:18)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): 0 right (int32): -21
onfig/i386/i386.c, (16473:11)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
oxout.c, (674:14)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
ouble-int.c, (115:13)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
ouble-int.c, (158:21)> : Op: *, Reason : Signed Multiplication Overflow, BINARY OPERATION: left (int32): 65535 right (int32): 6553
se.c, (1636:28)> : Op: +, Reason : Signed Addition Overflow, BINARY OPERATION: left (int32): 2147483647 right (int32): 1
warf2out.c, (4753:46)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
nit-rtl.c, (261:44)> : Op: *, Reason : Signed Multiplication Overflow, BINARY OPERATION: left (int32): 134217728 right (int32): 50
nit-rtl.c, (262:40)> : Op: *, Reason : Signed Multiplication Overflow, BINARY OPERATION: left (int32): 786432 right (int32): 25000
xpmed.c, (1092:13)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
xpmed.c, (2928:15)> : Op: -=, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
xpmed.c, (3107:8)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
xpmed.c, (3707:52)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
xpmed.c, (3813:23)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
xpmed.c, (4151:12)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
xpmed.c, (949:38)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
xpmed.c, (954:42)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
old-const.c, (7127:33)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
old-const.c, (7128:28)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
cc.c, (8558:21)> : Op: *, Reason : Signed Multiplication Overflow, BINARY OPERATION: left (int32): 1280858519 right (int32): 1000
imple.c, (3304:33)> : Op: <<, Reason : Unsigned Left Shift Error: Right operand is negative or is greater than or equal to the wid
imple.c, (3304:7)> : Op: <<, Reason : Unsigned Left Shift Error: Right operand is negative or is greater than or equal to the widt
ra-color.c, (1867:12)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): 2147483632 right (int32): -1
ra-color.c, (1868:17)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): 2147483632 right (int32): -1
ra-color.c, (394:29)> : Op: *, Reason : Signed Multiplication Overflow, BINARY OPERATION: left (int32): -69630937 right (int32): 1
ra-costs.c, (1056:15)> : Op: +=, Reason : Signed Addition Overflow, BINARY OPERATION: left (int32): 2147000000 right (int32): 1000
ra-costs.c, (1301:6)> : Op: +=, Reason : Signed Addition Overflow, BINARY OPERATION: left (int32): 1901515000 right (int32): 11796
ra-costs.c, (1306:27)> : Op: +=, Reason : Signed Addition Overflow, BINARY OPERATION: left (int32): 1179630000 right (int32): 1179
ra-costs.c, (919:8)> : Op: +=, Reason : Signed Addition Overflow, BINARY OPERATION: left (int32): 2098120000 right (int32): 655350
oop-iv.c, (2218:24)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): 2147483647 right (int32): -214
to-streamer-out.c, (418:26)> : Op: <<, Reason : Unsigned Left Shift Error: Right operand is negative or is greater than or equal t
ostreload.c, (1628:44)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): 1374389535 right (int32): -
eal.c, (1440:13)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
eal.c, (1442:12)> : Op: -, Reason : Signed Subtraction Overflow, UNARY OPERATION: right (int32): -2147483648
eal.c, (2159:11)> : Op: +=, Reason : Signed Addition Overflow, BINARY OPERATION: left (int32): 2147483647 right (int32): 1
tlanal.c, (3988:51)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
tlanal.c, (3991:51)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
tlanal.c, (4096:42)> : Op: -, Reason : Signed Subtraction Overflow, BINARY OPERATION: left (int32): -2147483648 right (int32): 1
```

- But of course nearly all old programs contain copious UB
- What are we going to do about that?
  1. Go back and fix all the old code
  2. Stop setting off time bombs by reining in the optimizers
  3. Keep making optimizers more aggressive while also not necessarily going back and fixing all of the old code

```
int foo (int x) {
  return (x + 1) > x;
}
int main() {
  cout << ((INT_MAX + 1) > INT_MAX) << "\n";
  cout << foo(INT_MAX) << "\n";
  return 0;
}
```

```
$ clang++ -O foo.cpp ; ./a.out
0
1
```

This code in Google Native Client:

```
aligned_tramp_ret =
  tramp_ret & ~(nap->align_boundary - 1);
```

Was changed to:

```
return addr & ~(uintptr_t)
  ((1 << nap->align_boundary) - 1);
```

But **`nap->align_boundary`** was 32

Shift-past-bitwidth is UB, and as a result the compiler turned a sandboxing safety check into a nop

Code reading uninitialized data can also be removed.

This code in Google Native Client:

```
aligned_tramp_ret =
  tramp_ret & ~(nap->align_boundary - 1);
```

Was changed to:

But **`nap->align_boundary`** was 32

Shift-past-bitwidth is UB, and as a result the compiler turned a sandboxing safety check into a nop

Code reading uninitialized data can also be removed.

```
int main() {
    int *p = (int*)malloc(sizeof(int));
    int *q = (int*)realloc(p, sizeof(int));
    *p = 1;
    *q = 2;
    if (p == q)
        printf("%d %d\n", *p, *q);
}
```

```
$ clang -O foo.c ; ./a.out
1 2
```

Without -DDEBUG

```
void foo(char *p) {
#ifdef DEBUG
  printf("%s\n", p);
#endif
  if (p)
    bar(p);
}
```

```
foo:
    testq   %rdi, %rdi
    je      L1
    jmp     _bar
L1: ret
```

With -DDEBUG

```
void foo(char *p) {
#ifdef DEBUG
  printf("%s\n", p);
#endif
  if (p)
    bar(p);
}
```

```
foo:
    pushq   %rbx
    movq    %rdi, %rbx
    call    _puts
    movq    %rbx, %rdi
    popq    %rbx
    jmp     _bar
```

```
void foo(int *p,
         int *q,
         size_t n) {
    memcpy(p, q, n);
    if (!q)
      abort();
}
```

```
foo:
        jmp     memcpy
```

Optimization is valid even when n == 0

```
int check(int *h,
          int *k) {
  *h = 5;
  *k = 6;
  return *h;
}
```

```
check:
  movl $5, (%rdi)
  movl $6, (%rsi)
  movl (%rdi), %eax
  retq
```

```
int check(int *h,
          long *k) {
  *h = 5;
  *k = 6;
  return *h;
}
```

```
check:
  movl $5, (%rdi)
  movq $6, (%rsi)
  movl $5, %eax
  retq
```

Optimization is valid even when sizeof(int) == sizeof(long)

Optimization is not valid for int * and unsigned *

```
void bar();

int foo(int z) {
  bar();
  return 100 % z;
}
```

```
foo:
   pushq   %rbx
   movl    $100, %eax
   xorl    %edx, %edx
   idivl   %edi
   movl    %edx, %ebx
   callq   bar()
   movl    %ebx, %eax
   popq    %rbx
   ret
```

Externally visible operations can't be reordered

But undefined behavior isn't externally visible!

```
#include <iostream>

void bar() {
  std::cerr << "HELLO\n";
}

int foo(int);

int main() {
  foo(0);
  return 0;
}
```

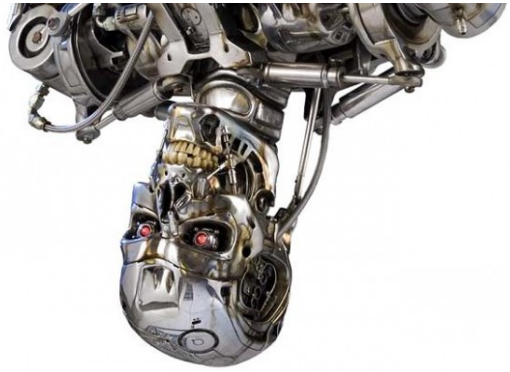
```
#include <iostream>

voi
   st
}

int

int
   fo
   re
}
```

```
$ clang++ tt1.cpp tt2.cpp
$./a.out
HELLO
Floating point exception: 8
$
$ clang++ tt1.cpp tt2.cpp -O
$ ./a.out
Floating point exception: 8
$
```



- UB can travel back in time!
- This is explicitly permitted by C++

5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

```
int fermat() {
  const int MAX = 1000;
  int a = 1, b = 1, c = 1;
  while (1) {
    if (((a * a * a) == ((b * b * b) + (c * c * c))))
      return 1;
    a++;
    if (a > MAX) {
      a = 1;
      b++;
    }
    if (b > MAX) {
      b = 1;
      c++;
    }
    if (c > MAX) {
      c = 1;
    }
  }
  return 0;
}
```

```
#include <iostream>
using namespace std;

int fermat();

int main() {
  if (fermat())
    cout << "Fermat's Last Theorem disproved!\n";
  else
    cout << "Nope.\n";
  return 0;
}
$ clang++ -O fermat.cpp
$ ./a.out
Fermat's Last Theorem disproved!
$
```

The compiler can terminate infinite loops that don't contain side-effecting operations

- C11 doesn't allow this when the controlling expression is a constant expression
  - So then we can at least rely on **`while (1)`** …
  - C++ doesn't have this loophole
- Hans Boehm elaborates on this issue: **`http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1528.htm`**

As a developer:

- You have to obey all of these 200+ rules, all of the time
- By default, nobody tells you when you break a rule

This recipe has lead to serious problems

What can we, the developers, do about undefined behavior in C and C++?

- Code review
- Static analysis
- Dynamic analysis
- Mitigation

- Code reviewers need to think about UB
- Many operations have an implicit precondition to avoid UB

  Precondition: **0 ≤ y < width(x)**

  Precondition: **y high bits of x are clear**

```
int foo(int x, int y) {
  return x << y;
}
```

- Explicit reasoning about preconditions is recommended
  - Ask people to prove that the preconditions hold
  - This is hard, especially around loops

Static analysis: Find bugs without running code

Unsound tools: Not intended to catch all bugs

- Enable and heed compiler warnings
  - Use **`-Werror`** when appropraite
- Coverity
- Klocwork
- Clang static analyzer
- …

Static analysis: Find bugs without running code

Sound tools: Designed to catch every instance of a given kind of bug

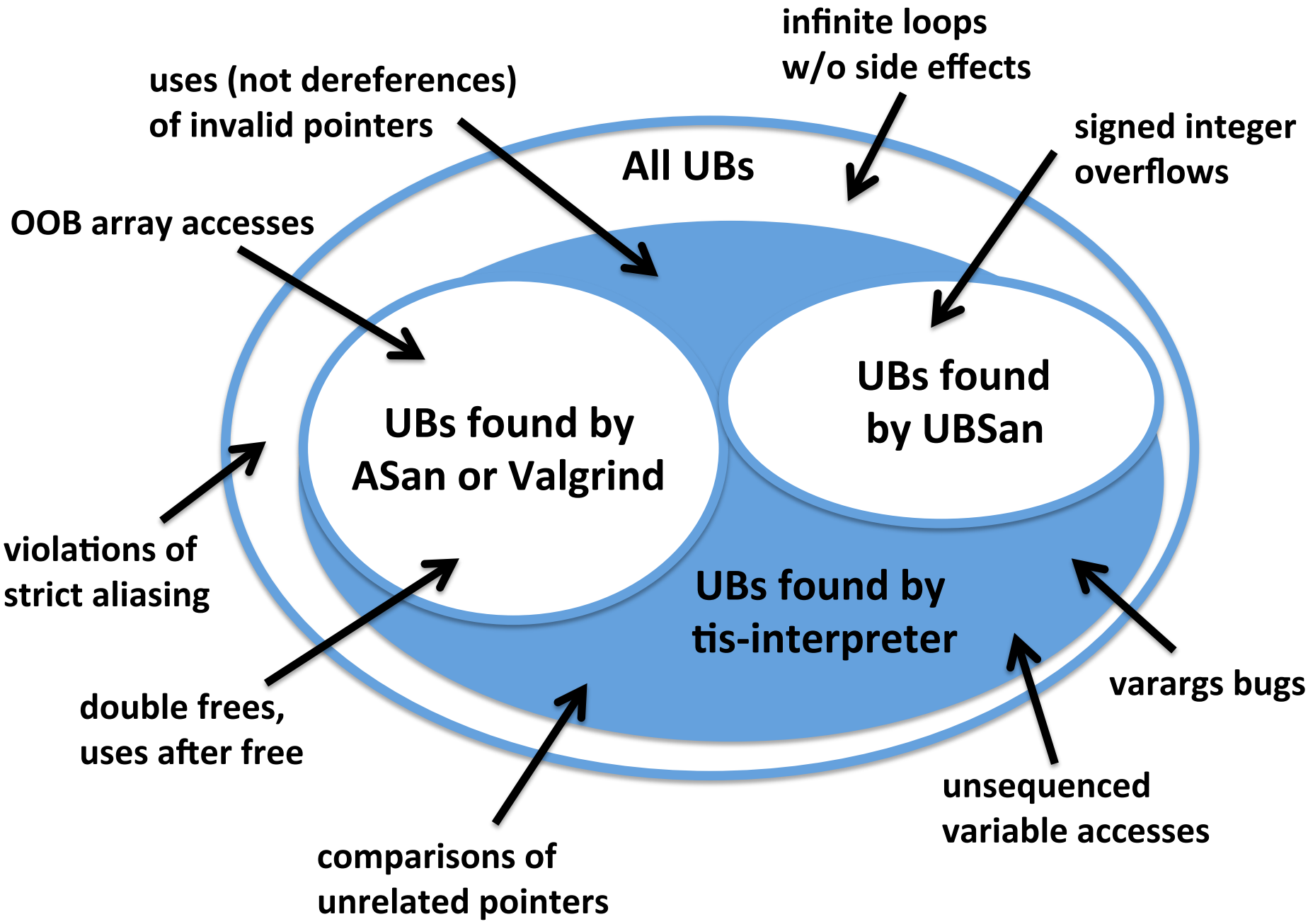
- E.g. No array OOB found == provable absence of array OOB errors
- Require significant expertise and effort
- Far fewer tools handle C++ than C
- Polyspace Code Prover
- Trust-in-Soft Analyzer

Dynamic analysis: Find bugs while running code a.k.a. Sanitizers

- I'll focus on the LLVM-based tools
- Address Sanitizer (ASan)
  - Spatial and temporal memory safety errors
- Undefined Behavior Sanitizer (UBSan)
  - Shift errors, signed integer overflow, pointer overflow, alignment errors, missing return statements, etc.
  - Also unsigned integer overflow!

- Memory Sanitizer (MSan)
  - Use of uninitialized storage
- Thread Sanitizer (TSan)
  - Data races, deadlocks
- Type Sanitizer (TySan)
  - Strict aliasing violations
  - (Under development)

- Advantages of dynamic analysis
  - Finds bugs on popular code paths
  - No false positives (typically)
  - Doesn't kill you with minutiae (typically)
- Drawback of dynamic analysis
  - Results are limited by how well we can test
    - Thorough testing is extremely hard…
  - Fuzzers are a big help, but aren't a panacea


infinite loops
w/o side effects
uses (not dereferences)
of invalid pointers
signed integer
overflows
All UBs
OOB array accesses
UBs found
by UBSan
UBs found by
ASan or Valgrind
violations of
strict aliasing
UBs found by
tis-interpreter
varargs bugs
double frees,
uses after free
unsequenced
variable accesses
comparisons of
unrelated pointers

Mitigation: Render UBs harmless in deployed code

- Linux compiles with **`-fno-delete-null-pointer-checks`**
- MySQL compiles with **`-fwrapv`**
- Many programs compile with **`-fno-strict-aliasing`**
- Parts of Android build with parts of UBSan
  - Want to use alternate runtime for deployment
- Chrome on Linux is built with control flow integrity (CFI) enabled
  - Provided by recent LLVMs
  - Very low overhead

Issues with UB mitigation

- Solutions aren't very standardized or portable
- There's no mitigation for concurrency errors
- Memory safety error mitigation tends to be expensive and may break code
  - ASan is not a mitigation tool
- UBSan can be configured as a hardening tool
  - Software developers need to decide if they want to turn a potential exploit into a crash

Where We Need to Go

- For every UB in C++ (yes, all 200+ of them) we should do one of the following:
  - Define the runtime behavior
  - Reliably diagnose it with a fatal compiler error
  - Provide a reliable runtime sanitizer

How are we doing?

- Not too bad, overall
- Remaining big problems
  - Production-grade memory bounds checking
  - Strict aliasing
- Minor issues
  - Non-terminating loops
  - Unsequenced side effects
    - In a function argument list
    - In an expression

Recommendations for C++ Devs

- Be educated about UB
- Explicitly consider UB during code reviews
- Test like hell
  - Use both coverage tools and fuzzers
- Use sanitizers and easy static analysis tools
  - Fix the bugs they find
- Be familiar with the hard tools

Thanks!