Undefined Behavior

What happened to my code?

Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia,
Nickolai Zeldovich, Frans Kaashoek

MIT CSAIL Tsinghua IIIS

Undefined behavior (UB)

- Many languages have UB
  - C, C++, Haskell, Java, Scheme, …
- UB definition in the C standard

  "behavior, … for which this International Standard imposes no requirements"
  - Compilers are allowed to generate any code
- UB example: integer division by zero
  - 1 / 0 → trap (gcc/x86)
  - 1 / 0 → no trap (gcc/ppc, clang/any)

UB allows generating efficient code

- x / y: on ppc (no hardware div trap)
- If division by zero is defined to trap

```
if (y == 0)
    raise(SIGFPE);
… = x / y;
```

- Compiler assumes no UB: `y != 0`

```
… = x / y;
```

UB leads to broken code

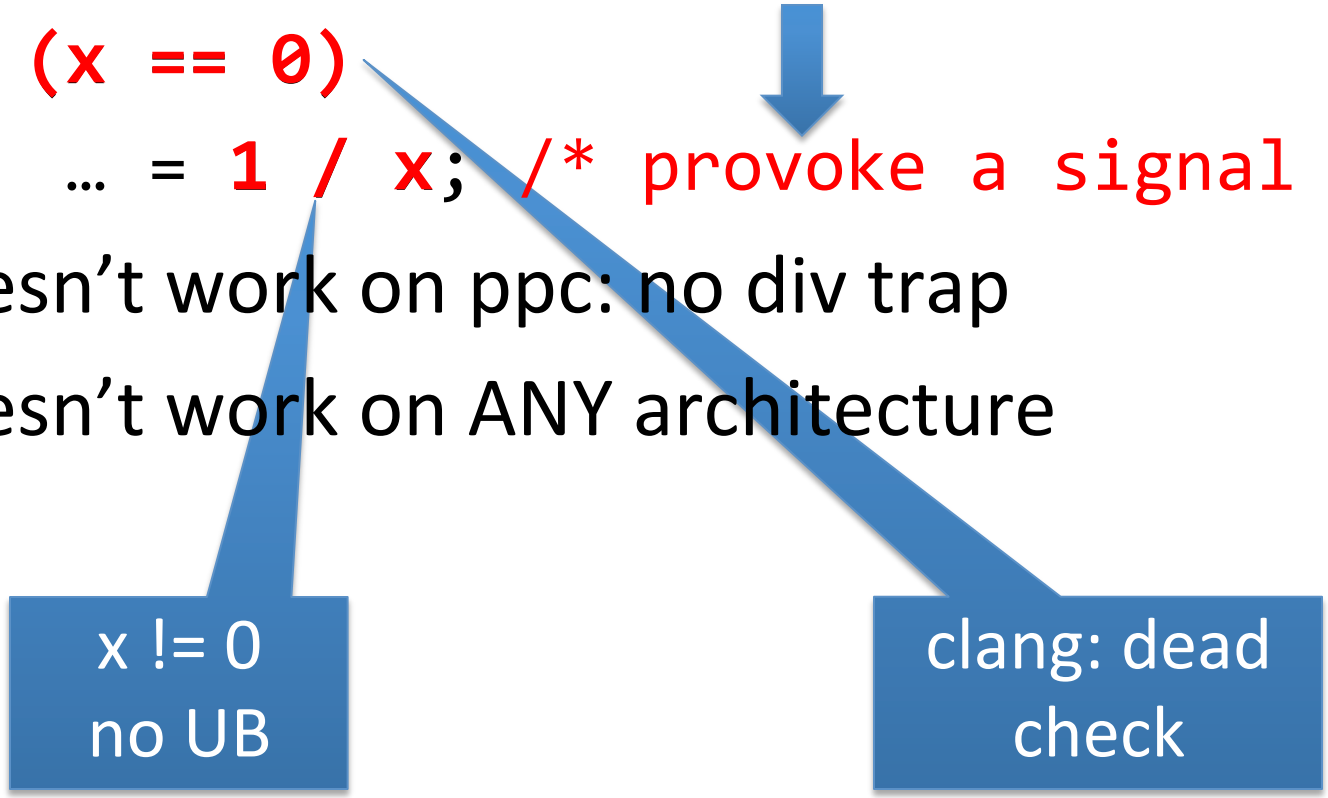
- Programmers invoke UB on purpose
  - Break compiler's no-UB assumption
- Programmers are unaware of UB
  - Optimizations exploit UB unexpectedly

Problem 1: programmers invoke UB

- Libgcrypt & Python: division by zero

```
if (x == 0)
    … = 1 / x; /* provoke a signal */
```

- Doesn't work on ppc: no div trap
- Doesn't work on ANY architecture



Problem 2: innocent UB consequence

- PostgreSQL & Ruby

```
if (y == 0)
    my_raise(); /* call longjmp()
                   never return */
… = x / y;
```

gcc: division is always reachable

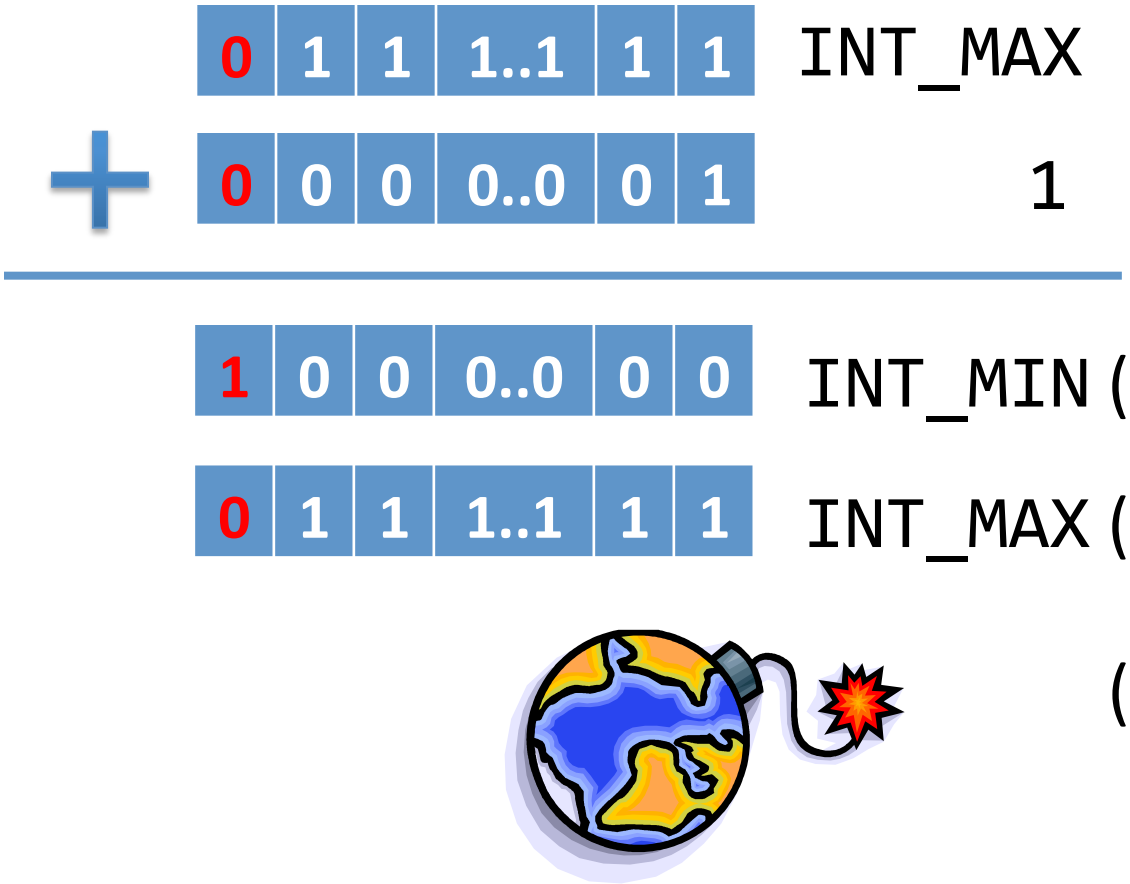
Contributions

- Survey & identify 7 UB bug patterns in systems
  - Sanity checks gone
  - Sanity checks reordered (after uses)
  - Expressions rewritten & broken
- Happen to major C compilers
  - gcc, clang, icc, …
  - With just `-O2` (even `-O0`)

Outline

- 7 UB bug patterns
  - Division by zero
  - Oversized shift
  - Signed integer overflow
  - Out-of-bounds pointer
  - Null pointer dereference
  - Type-punned pointer dereference
  - Uninitialized read
- Finding UB is difficult
- Research opportunities
  - Better language & tools

Bug 1: signed integer overflow



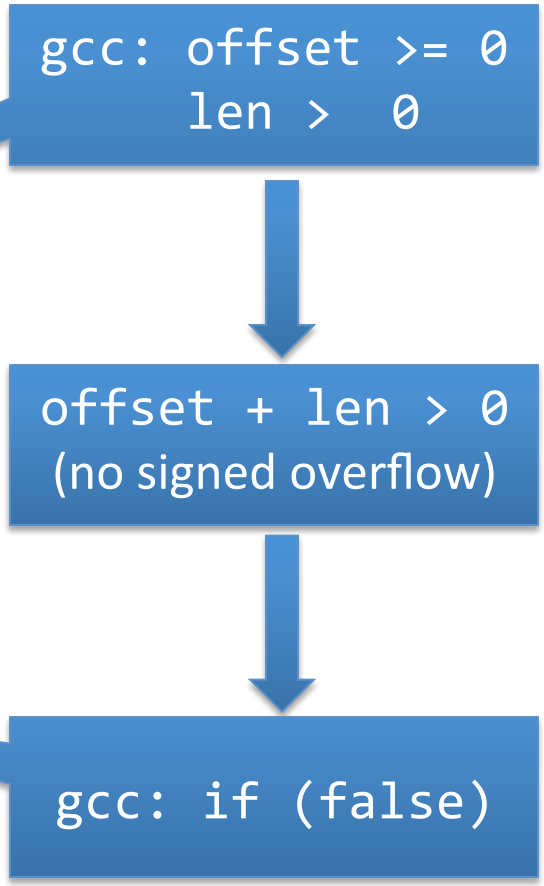
Signed integer overflow in C

- Undefined behavior
- Compilers assume no signed integer overflow
- Post-overflow check $x + 1 < x \rightarrow$ false
  - Common idiom for unsigned integers
  - Doesn't work for signed integers

Broken overflow check in Linux kernel

```
signed long offset = <userspace>;
signed long len = <userspace>;

    /* Reject negative values */
①   if (offset < 0 || len <= 0)
            return -EINVAL;
    /* Sum too large? */
②   if (offset + len > s_maxbytes)
            return -EFBIG;
    /* Check for wrap through zero too */
③   if (offset + len < 0)
            return -EFBIG;
    /* Allocate offset + len bytes */
```



Signed overflow is widely misused

- A lot of systems got bitten by signed overflow
  - glibc, MySQL, PostgreSQL, …
  - IntegerLib & SafeInt from security experts
- Many analysis tools get this wrong
  - KLEE & clang static analyzer
  - Conclude x + 1 < x when x = INT_MAX!

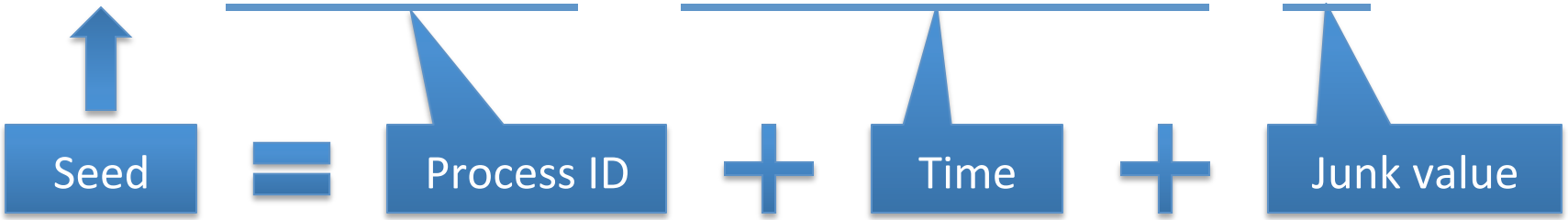
Bug 2: uninitialized read

- A local variable in C is uninitialized
  - Hold a random value?
- Undefined behavior
  - Assign arbitrary value to uninitialized variable
  - Assign arbitrary value to derived expression

Seeding random numbers in BSD libc

```
struct timeval tv;
unsigned long junk; /* XXX left uninitialized on purpose */

gettimeofday(&tv, NULL);
srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```



Seeding random numbers in BSD libc

```
struct timeval tv;
unsigned long junk; /* XXX left uninitialized on purpose */

gettimeofday(&tv, NULL);
srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

clang: expression undefined

Uninitialized read for randomness

- More predictable random numbers
- Also appeared in:
  - FreeBSD (DragonFly/OS X), Varnish, Android app

Finding bugs due to UB is challenging

- Slip through code review
  - Even with comments to explain why “correct”
- Hard to confirm broken checks
  - Inspect assembly code
- How to help programmers?

Existing solutions & ideas

- Disable optimizations
- Warn against UB bugs
- Revise the C language

Approach 1: disable optimizations

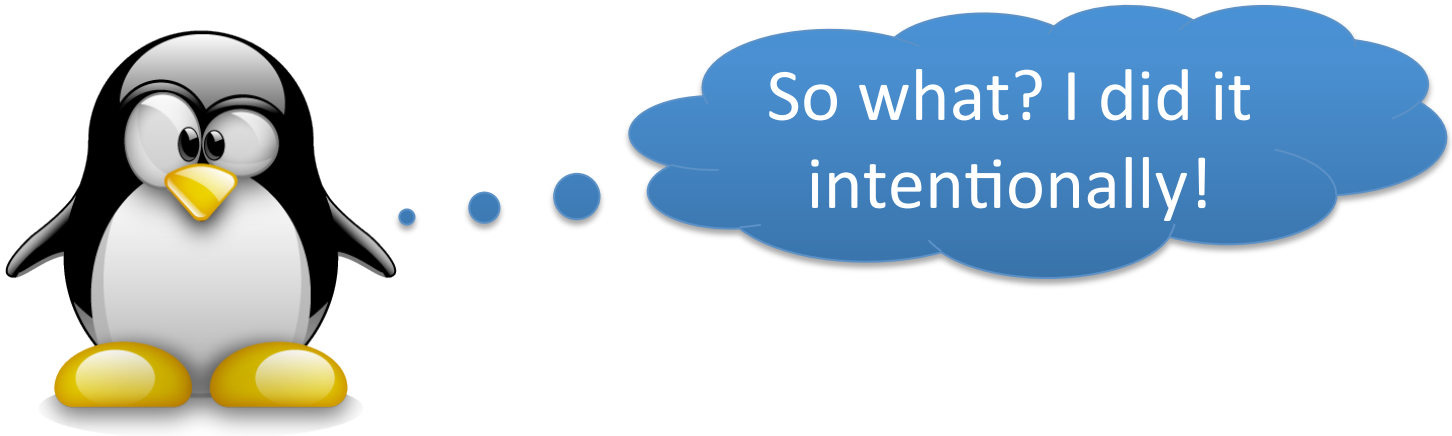
- GCC & Clang: incomplete workarounds
  - Force to wrap for signed integers: `-fwrapv`
  - No workarounds for division by zero, etc.
- Workarounds don't work for other compilers
  - Intel's icc doesn't have `-fwrapv`
  - Fix code if programmers want portability
- Hurt performance?

Performance impact is inconclusive

- Our results on SPECint 2006
  - Only on 2 out of 12 programs
  - Up to 7.2% (GCC) and 11.8% (Clang) slowdown
  - Fix slowdown by minor code changes
- Mixed reports from others
  - 0 ~ 50% slowdown
  - Noticeable on Android

Approach 2: generate good warnings

```
if (x == 0)
    x = 1 / x; /* provoke a signal */
             ^
Warning: division by zero
```



A better warning

```
if (x == 0)
    ^^^^^^
    x = 1 / x; /* provoke a signal */
        ~~~~^
```

```
Warning: eliminating check (x == 0),
under the assumption that x, used as a
divisor, cannot be zero.
```

Approach 3: safer C

- Disallow UB
  - Any violation → trap
  - Compilers insert checks for UB
- Define UB from common impression (e.g., Java)
  - Division by zero → trap
  - Signed integer overflow → wrap
  - Uninitialized variable → zero
- Will future compilers exploit other UB?
  - Data race: hard to redefine

Research questions

- How does disabling UB-based optimizations hurt performance?
- How to generate informative warnings on uses of UB?
- To what degree can we reduce UB in the language?

Conclusion

- We identified 7 UB bug patterns
  - Real-world systems with major compilers
  - Subtle to notice and understand
- Existing solutions are insufficient
  - Performance impact evaluation
  - Better compiler warnings
  - Revise UB in C