Integer Overflow

"Understanding Integer Overflow in C/C++"

W. Dietz, P. Li, J. Regehr, V. Adve

International Conference on Software Engineering (ICSE), 2012

Integer Overflows Can Create Problems

These errors are also a source of serious vulnerabilities, such as integer overflow errors in OpenSSH [1] and Firefox [2], both of which allow attackers to execute arbitrary code. In their 2011 report MITRE places integer overflows in the "Top 25 Most Dangerous Software Errors" [3].

Ranges for Values

Examples from usr/include/limits.h

/* Minimum and maximum values a `signed short int' can hold. */

define **SHRT_MIN** (-32768)

define **SHRT_MAX** 32767

/* Maximum value an `unsigned short int' can hold. (Minimum is 0.) */

define **USHRT_MAX** 65535

/* Minimum and maximum values a `signed int' can hold. */

define **INT_MIN** (-INT_MAX - 1)

define **INT_MAX** 2147483647

/* Maximum value an `unsigned int' can hold. (Minimum is 0.) */

define **UINT_MAX** 4294967295U

Examples

Table I

EXAMPLES OF C/C++ INTEGER OPERATIONS AND THEIR RESULTS

Expression	Result
`UINT_MAX+1`	0
`LONG_MAX+1`	undefined
`INT_MAX+1`	undefined
`SHRT_MAX+1`	`SHRT_MAX+1` if `INT_MAX>SHRT_MAX`, otherwise undefined
`char c = CHAR_MAX; c++`	varies[1]
`-INT_MIN`	undefined[2]
`(char)INT_MAX`	commonly `-1`
`1<<-1`	undefined
`1<<0`	1
`1<<31`	commonly `INT_MIN` in ANSI C and C++98; undefined in C99 and C++11[2,3]
`1<<32`	undefined[3]
`1/0`	undefined
`INT_MIN%-1`	undefined in C11, otherwise undefined in practice

[1] The question is: Does `c` get "promoted" to `int` before being incremented? If so, the behavior is well-defined. We found disagreement between compiler vendors' implementations of this construct.
[2] Assuming that the `int` type uses a two's complement representation
[3] Assuming that the `int` type is 32 bits long

Well-Defined Behaviors

Some kinds of unsigned integer arithmetic uses well-defined and portable wraparound behavior, with two's complement semantics [6]. Thus, as Table I indicates, `UINT_MAX+1` must evaluate to zero in every conforming C and C++ implementation. nOf course, even well-defined semantics can lead to logic errors, for example if a developer naïvely assumes that $x + 1$ is larger than x.

Many unsigned integer overflows in C and C++ are well-defined, but *non-portable*. For example `0U-1` is well-defined and evaluates to `UINT_MAX`, but the actual value of that constant is *implementation defined*: it can be relied upon, but only within the context of a particular compiler and platform. Similarly, the `int` type in C99 is not required to hold values in excess of 32,767, nor does it have to be based on a two's complement representation.

Undefined Behaviors

Some kinds of integer overflow are undefined, and these kinds of behavior are especially problematic. According to the C99 standard, undefined behavior is

> "behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements."

In Internet parlance:[3]

> "When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose."

Undefined Behaviors: Silent Breakage

A C or C++ compiler may exploit undefined behavior in optimizations that silently break a program. For example, a routine refactoring of Google's Native Client software accidentally caused `1<<32` to be evaluated in a security check.[4] The compiler—at this point under no particular obligation—simply turned the safety check into a nop. Four reviewers failed to notice the resulting vulnerability.

Undefined Behaviors: Silent Breakage

when programs have undefined operations, optimizing compilers may silently break them in non-obvious and not necessarily consistent ways

```
1 int foo (int x) {
2   return (x+1) > x;
3 }
4
5 int main (void) {
6   printf ("%d\n", (INT_MAX+1) > INT_MAX);
7   printf ("%d\n", foo(INT_MAX));
8   return 0;
9 }
```

Recent versions of GCC, LLVM, and Intel's C compiler, invoked at the `-O2` optimization level, all print a 0 for the first value (line 6) and a 1 for the second (line 7). In other words, each of these compilers considers `INT_MAX+1` to be both larger than `INT_MAX` and also not larger, at the same optimization level, depending on incidental structural features of the code.

Undefined Behaviors: Time Bombs

Undefined behavior also leads to *time bombs*: code that works under today's compilers, but breaks unpredictably in the future as optimization technology improves. The Internet is rife with stories about problems caused by GCC's ever-increasing power to exploit signed overflows. For example, in 2005 a principal PostgreSQL developer was annoyed that his code was broken by a recent version of GCC:[5]

> It seems that gcc is up to some creative reinterpretation of basic C semantics again; specifically, you can no longer trust that traditional C semantics of integer overflow hold ...

This highlights a fundamental and pervasive misunderstanding: the compiler was not "reinterpreting" the semantics but rather was beginning to take advantage of leeway explicitly provided by the C standard.

Undefined Behaviors: Illusion of Predictability

Some compilers, at some optimization levels, have predictable behavior for some undefined operations. For example, C and C++ compilers typically give two's complement semantics to signed overflow when aggressive optimizations are disabled. It is, however, unwise to rely on this behavior, because it is not portable across compilers or indeed across different versions of the same compiler.

Undefined Behaviors: Changing Standards

Some kinds of overflow have changed meaning across different versions of the standards. For example, `1<<31` is implementation-defined in ANSI C and C++98, while being explicitly undefined by C99 and C11 (assuming 32-bit ints). Our experience is that awareness of this particular rule among C and C++ programmers is low.

A second kind of non-standardization occurs with constructs such as `INT_MIN%-1` which is—by our reading—well defined in ANSI C, C99, C++98, and C++11. However, we are not aware of a C or C++ compiler that reliably returns the correct result, zero, for this expression. The problem is that on architectures including x86 and x86-64, correctly handling this case requires an explicit check in front of every % operation. The C standards committee has recognized the problem and C11 explicitly makes this case undefined.

Interesting Experiment in the Paper

To find the time bombs, we altered IOC's overflow handler to return a random value from any integer operation whose behavior is undefined by the C or C++ standard. This creates a high probability that the application will break in an observable way if its execution actually depends on the results of an undefined operation. Perhaps amusingly, when operating in this mode, IOC is still a standards-conforming C or C++ compiler—the standard places no requirements on what happens to a program following the execution of an operation with undefined behavior.

Interesting Experiment in the Paper

SPEC CINT is an ideal testbed for this experiment because it has an unambiguous success criterion: for a given test input, a benchmark's output must match the expected output. The results appear in Table IV. In summary, the strict shift rules in C99 and C++11 are routinely violated in SPEC 2006. A compiler that manages to exploit these behaviors would be a conforming implementation of C or C++, but nevertheless would create SPEC executables that do not work.

Benchmark	ANSI C / C++98	C99 / C++11
400.perlbench	✓	✓
401.bzip2	✓	✗
403.gcc	✗	✗
445.gobmk	✓	✓
464.h264ref	✓	✗
433.milc	✗	✗
482.sphix3	✓	✗
435.gromacs	✓	✓
436.cactusADM	✓	✗