Chapter 11
Numeric Errors: Floating Point Numbers

Revision 1.1, December 2025.

Objectives

- Understand the potential pitfalls associated with floating point numbers.
- Understand the causes of those problems.
- Learn how to mitigate integer problems.

11.1 Motivation and a Bit of History

In the previous chapter, we saw a variety of cases where using integers incorrectly could have significant impact on the security of a program. In this chapter, we will extend that discussion to using floating point numbers.

Floating point numbers typically use a fixed number of bits and are able to express non-integers, even ones that have very large or very small values. Floating point numbers are of the form $\pm man \times b^{exp}$, where ± is the sign bit, *man* is the mantissa, *b* is the base, and *exp* is the exponent. The particular floating point representation that you use will decide the number of bits available for the mantissa and exponent, and the selection of base, which is typically 2 or 10. The representation is typically based on what is supported by your computer's hardware. Note that almost all modern computers support base 2 floating point representations. Figure 1 shows a summary of these values for some historically notable computers.

For many years, each company chose its own floating point format, along with the rules the control such things as rounding, overflow, and underflow. This diversity of representations made writing portable numeric code difficult as you could easily get different answers on different processors, even for representations that had the same total number of bits.

In 1987, Prof. William Kahan[1] of the University of California, Berkeley lead Intel in its design of the floating point coprocessor for the Intel 8086 processor. This co-processor, the Intel 8087, was based on improved floating point representation and algorithms using a solid mathematical foundation. While a discussion of this new floating point representation is beyond the scope of this chapter, it is important to note that it contained many new and important changes that affected the accuracy and stability of floating point

[1] `https://en.wikipedia.org/wiki/William_Kahan`

calculations. Kahan went on to take these ideas, refine them, and lead the IEEE floating point standard effort, called IEEE 754[2]. The first draft of that standard was produced in 1985, with an expanded version of the standard in 2008. The last rows of Figure 1 show the parameters for the IEEE 754 (which look quite similar to the Intel 8087) and Figure *2* shows how the bits are laid out for a 32 bit number.

When this standard was being created, it produced an enormous amount of debate and protest among the computer manufacturers, who all had a stake in keeping their own representations and, more importantly, computational algorithms. Ultimately, the IEEE 754 standard prevailed because it was the mathematically and computationally right approach, and because of the tireless efforts of Kahan, his students and his colleagues. More recently, this standards effort was taken over by an international standards organization and is now known as ISO/IEC 60559.

	Format Name	**Total Bits**	**Exponent Bits**	**Mantissa Bits**	**Exponent Base**
IBM 370	Short	32	7	24	16
	Long	64	7	56	16
	Extended	128	7	112	16
CDC 7600	Standard	60	11	48	2
Cray 1	Standard	64	15	49	2
DEC VAX	F_floating	32	8	*24	2
	D_floating	64	8	*56	2
	G_floating	64	11	*53	2
	H_floating	128	15	*113	2
Intel 8087	Single	32	8	*24	2
	Double	64	11	*53	2
	Extended	80	15	*64	2
IEEE 754	Single	32	8	*23	2
	Double	64	11	*53	2
	Quad	128	15	*113	2

Figure 1: Floating Point Number Characteristics for Notable Historical Computers
** means that there is one* implied *bit. Since the most significant bit of the mantissa of any normalized floating number is one, you can get an extra bit by not storing this leading bit.*

[2] `https://en.wikipedia.org/wiki/IEEE_754`

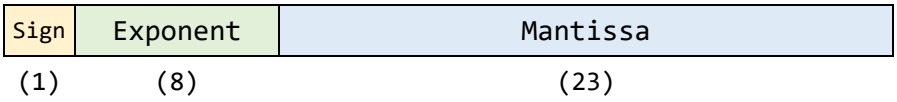

Figure 2: Memory Layout for IEEE 754 32-bit Floating Point Number

11.2 Floating Point Representation and Calculation Errors

In this section, we discuss a variety of ways that floating point numbers can cause unexpected results. The cases described in this section are concerns for security because code that looks like it obviously works may not work over a wide range of input values. An attacker may craft an input such that it violates the (perhaps implicit) assumptions that the programmer made but does not trigger any of the error checking code.

11.2.1 Precision Errors

Floating point representations are powerful because they can both represent very large and very small numbers. However, no matter whether you have a large or small number, the number of bits of accuracy in the mantissa remains the same. This means that if you mix numbers of significantly different magnitude in the same expression, you might get surprising results.

Consider the code in Figure 3. The `while` loop is controlled by a variable of type `float` that gets incremented each time through the loop. Also, each time through the loop, the previous value of the counter is saved. The loop terminates and the `printf` executes when a seemingly impossible condition happens: when the previous value equals the current value of the counter. If `count` and `prev` were integers, this loop would never terminate. So, what happens in the case where these variables are of type `float`? Try to figure out what will be the output. You can also try to compile this program and run it.[3]

```
float count = 0.0f, prev = 1.0f;
while (count != prev) {
   prev = count;
   count++;
}
printf("count == prev == %f\n", count);
```

Figure 3: Loop with Comparison of Potentially Different Magnitude Numbers

The answer is that this loop *will* terminate and execute the `printf`. At some point, `count` will get big enough that its value can no longer be fully

[3] If you compile this code, you will need to add a `#include <stdio.h>`

represented by the mantissa and larger exponents are needed to handle the magnitude of the number. As floating point numbers increase in value, their exponents increase. The result is that the smallest increment in value increases by a power of two each time the exponent increases.

This situation in our example program where we can no longer track increases by one happens when `count` reaches 16,777,216, which is `1000000000000000000000000` in binary.

The same thing can happen for very small positive values. Consider the code in Figure 4. We start with a very small number, 10^{-30}, and add one to it and then subtract one from it. Algebraically, you would expect that adding one and subtracting one would keep the value the same. So, what will the output of this small program?

```
float small, same;
small = 1E-30f;
same = small + 1.0f - 1.0f;
printf ("small = %e, same = %e\n", small, same);
```

Figure 4: Increment Very Small Positive Number by 1

Since there are not enough bits of precision to simultaneously represent the 1 and the 10^{-30}, the very small number (10^{-30}) gets truncated once `small` is incremented by one, leaving only the one. Then the subtract brings the result, which is assigned to `same`, to zero. The output of the program is:

```
small = 1.000000e-30, same = 0.000000e+00
```

11.2.2 Comparison Errors

Figure 1 shows a simple program that allocates floating point numbers, two of type float and two of type double. Look carefully at this program and try to figure out what will be the output. You can also try to compile this program and run it.

```
int main() {
  float ftenth = 0.1f, fquarter = 0.25f;
  double dtenth = 0.1, dquarter = 0.25;
  if (ftenth == dtenth)
    printf("Tenths equal\n");
  if (fquarter == dquarter)
    printf("Quarters equal\n");
}
```

Figure 5: Simple Program using Floating Point Numbers

This program prints "`Quarters equal`" but does *not* print "`Tenths equal`". The question is why? The clue was given in Section 11.1, where we described modern floating point representations. Since almost all computers use IEEE 754, that is a good starting assumption. The key point for this example is that the mantissa is encoded in binary.

Figure 6 shows various forms of the numbers used in the simple program. Most importantly, we see that while 0.25 can be represented exactly in binary, 0.1 is a repeating pattern in binary. When `float` and `double` are compared, the `float` is converted to a `double`, which pads the number with zero bits on the end of the mantissa.

Number	Decimal	Binary
$^1/_{10}$	`0.1`	`0.0001100110011 ...`
$^1/_4$	`0.25`	`0.01`

Figure 6: Three Different Representations of the Numbers used in Figure 5

When the 23 bit mantissa for the `float` is converted to a `double`, the resulting 53 bit mantissa has 30 bits of zero at the end. Which, of course, does not match the 53 bits of repeating pattern of the `double` variable.

11.2.3 Exceptional Values

The IEEE 754 standard supports some interesting non-numeric values to help ensure correct computation. The values include:

NaN (not a number): This value occurs as a result of a mathematically undefine operation, such as dividing by zero (when the numerator is non-zero), taking the logarithm of zero, or taking the square root of a negative number. Having such a value ensures that you do not propagate an invalid intermediate result through a calculation.

Infinity: This value occurs when a computation produces a positive result larger than the largest possible value for the particular floating point type, or when dividing by positive zero (when the numerator is also zero).

-*Infinity*: This value occurs when a computation produces a negative result whose magnitude is larger than the largest possible value for the particular floating point type, or when dividing by negative zero (when the numerator is also zero).

Interestingly enough, many of the input routines that process floating point numbers in various languages can take inputs such as "NaN", "Infinity", and "-Infinity". Note that some languages are case sensitive for these values and

some are not. And some languages, such as C (using something like `strtod`[4]) will accept other variations such as "inf".

When a computation produces one of these exceptional values, it should result in an exception being raised. The IEEE 754 standards specifies that there are five different exception conditions (1) invalid operation, (2) division by zero, (3) overflow, (4) underflow, and (5) inexact calculation. It is a critical part of the mathematical soundness of the IEEE 754 standard to properly handle these exceptions. Unfortunately, the behavior of exceptions for floating point calculations is inconsistent and weak across languages. Most languages ignore these exceptions and require explicit checking for the exceptional values in the code, and possibly raising a user-defined exception. This lack of attention to floating point exceptions often stems from the fact that modern processes do not generate exceptions for these operations. From personal conversations with Kahan – the inspiration behind the IEEE 754 standard and lead author of the standard – he was quite disappointed with this state of affairs.

11.3 Exploit Examples

Now, we will example a couple of examples where careless use of floating point numbers caused security vulnerabilities. These examples are based on vulnerabilities found in real systems.

11.3.1 Buffer Overflow from Conversion of Floating Point to a String

There have been several vulnerabilities caused by conversion of a floating point number to a string. This kind of vulnerability comes from not constraining the number of output digits when making the conversion. For example, programs that are using floating point numbers for representing monetary amounts often expect two digits behind the decimal point (or comma). 64 bit IEEE 754 floating point numbers can have up to approximately 16 digits of precision, plus the decimal point and sign. Unchecked inputs can be constructed to have more than this number, allowing overflow of the character string buffer.

Such a buffer overflow occurred in the popular curl utility[5] and was reported in CVE-2016-9586[6]. Such overflows can be prevented by checking the length of the destination buffer each time a string or new character is added

[4] `https://litux.nl/man/htmlman3/strtod.3.html`
[5] Curl is a command line tool and library to allow scripts and programs to make Web requests based on URLs. `https://curl.se/`
[6] `https://nvd.nist.gov/vuln/detail/cve-2016-9586`

to the buffer. Conversion functions like the UNIX `sprintf`[7] will default to a maximum of 6 digits when converting a double to a string.

This is an unfortunately common type of error. We see a similar vulnerability in PostgresSQL, when a user specifies a floating point type format string conversion with more than 500 digits. This vulnerability was reports in CVE-2015-0242[8].

11.3.2 Buffer Overflow from Conversion of String to Floating Point

Conversion errors can occur in the opposite direction, when converting string input to a floating point number. When the Ruby interpreter was converting string input to a floating point number, it used a locally written arbitrary precision integer representation. This representation, called `BigInt` in their code[9], is based on a structure that has (among other fields) a length and array of numbers that make up the arbitrary length integer. As the size of the value contained in the `BigInt` grows, more memory is allocated from the heap for the structure.

An exceptionally long input value could cause continuous heap allocation until memory is used up. The results of this overflow were potentially crashing or executing arbitrary code. This vulnerability was reported in CVE-2013-4164[10].

There are a couple of checks that might be used to prevent such memory exhaustion. First, in general, the length of the input string should have a fixed bound, after which further characters are rejected. Second, for this implementation, the use of the intermediate arbitrary integer representation caused (in our opinion) unnecessary complexity and risk. Such an implantation was attractive because it allowed the conversion to proceed one input digit at a time. A conversion that read in the entire input string (with the length restriction check), then selected the maximum number of digits to convert, might be based on direct conversion to double floating point numbers without an intermediate representation.

11.3.3 Buffer Overrun Caused by Mixing Doubles and Integers

In JavaScript, array classes often have an `indexOf` method that finds the position of a given value in an array. The `indexOf` method always has a parameter that is the search value. The search through the array starts, by default, at element zero. The `indexOf` method optionally takes a second

---

[7] `https://man7.org/linux/man-pages/man3/fprintf.3p.html`
[8] `https://nvd.nist.gov/vuln/detail/cve-2015-0242`
[9] `https://github.com/ruby/ruby/commit/5cb83d9dab13e14e6146f455ffd9fed4254d238f`
[10] `https://nvd.nist.gov/vuln/detail/CVE-2013-4164`

parameter that is the offset into the array. If this offset (called the `fromIndex`) is negative, it means that the starting position is relative to the *end* of the array; so the (negative) `fromIndex` value is added the length of the array to calculate the starting position for the search. When the `fromIndex` is negative, the computation for the search starting position is:

```
1:  fromIndex += len;
2:  if (fromIndex < 0)
3:     goto done;
4:  k = fromIndex;
```

The QuickJS JavaScript engine[11] had a vulnerability when a negative floating point number was passed as the `fromIndex` parameter. If the value passed was a very small negative number (1×10^{-20}, in this case), the value effectively became zero when it was added to `len` on line 1. As a result, the integer value used for the starting position is the array length, which is one position past the end of the array because valid indices into an array go from `0` to `len-1`. This vulnerability was reported in CVE-2025-62492[12].

11.3.4 Exception Caused by Not Checking for NaN

In TensorFlow, the widely used machine learning library from Google, in certain cases if you provide input that contains a Not a Number (NaN) value, TensorFlow does not check for the presence of a NaN, then tries to recast the value to an `int32`, causing an uncaught floating point exception[13]. If TensorFlow is implemented as part of a critical service, this could crash the entire service causing a denial of service (DoS) exploit.

In general, it is important and powerful to be able to represent non-valid numeric values in floating point arithmetic. However, the possibility of having such a value leads to an extra burden, the need to check for the presence of these values and handle the situation when one appears. The response to finding a NaN when not expected will depend on the structure of the code. Some possible responses include:

Return an error: If it is acceptable that the function/method that contains the floating point operation can fail, then simply returning an error from that might is the easiest fix.

[11] `https://github.com/quickjs-ng/quickjs`
[12] `https://nvd.nist.gov/vuln/detail/CVE-2025-62492`
[13] `https://nvd.nist.gov/vuln/detail/CVE-2022-29211`

Using a default value: If the function cannot fail, then it might be acceptable to use a default value – such as zero or the maximum floating point value – to substitute for the NaN.

Ignore the value: Sometimes bad data can just be ignored. Of course, this is dependent on the computation being implemented.

Abort the program: While this is the least desirable option, the program may be structured in a way that it is not be able handle invalid numbers. In this case, it is important to print out an informative error message before exiting the program.

11.4 Language Specific Issues

There are a lot of languages in common use today and there is a disconcerting and sometimes confusing variation in the way that these languages handle floating point numbers. Where confusion resides, security mistakes are not far behind.

In this section, we discuss a few interesting variations. It is important that you study and understand how floating point (and other related representations, like fixed point) numbers work in the language that you using. And understand what type of number representation makes sense for the program that you are writing or analyzing.

11.4.1 C and C++

C and C++ support 32, 64, and 128 bit floating point numbers with the `float`, `double`, and `long double` types. These are based on what the hardware supports, so use the IEEE 754 representation.

11.4.2 Go

Go support 32 and 64 bit representation, called `f32` and `f64`, based on what the hardware supports, so uses the IEEE 754 representation.

Go also supports arbitrary precision floating point with the big.Float and arbitrary precision rational numbers (stored as a pair of arbitrary precision big.Int numbers). Of course, these arbitrary precision calculations are done in software so much slower than calculations done with the `f32` and `f64` types.

11.4.3 Rust

Rust support 32 and 64 bit representation, called `float32` and `float64`, based on what the hardware supports, so uses the IEEE 754 representation.

Rust also support the `Decimal` number type from the `rust_decimal` crate (package). A `Decimal` number consists of a 96-bit integer, a scaling factor to define the decimal fraction, and a one bit sign. This allows for about 28 base significant decimal digits.

11.4.4 JavaScript

JavaScript is an especially interesting example when studying floating point numbers, because JavaScript had only one number type, `Number`. And `Number` is a 64 bit floating point number, using the IEEE 754 standard of course.

Because of the way that numbers are represented in IEEE floating point (see Figure 1), the biggest integer value you can safely represent is 2^{53}-1. You can represent larger integer values here, but they could generate truncation, so you cannot operate on them as you would with integers.

`Number` is still the most commonly used type of numeric variable in JavaScript, though they recently added `BigInt` for arbitrary precision integer arithmetic. While this is quite useful, calculations with `BigInt` can be quite a bit slower than those with `Number` because `Number` calculations are done in hardware and `BigInt` operations are done in software.

11.4.5 Python

Python supports floating point numbers with the float type by either assigning a number with a decimal point or exponential notation to a variable or using the `float` operation. `floats` are 64 bits and typically done in hardware (so based on IEEE 754).

Python also supports an arbitrary precision fixed-point decimal representation of numbers. As we discussed in earlier sections of this chapter, representing monetary values in a binary floating point representation can lead to roundoff and truncation errors, so decimal representations are useful in the context. The Python `Decimal` type defaults to 28 significant decimal digits, though that can be configured by setting the global context variable, `getcontext().prec`. While computation with the arbitrary precision numbers is slower than with the float type, it is attractive for financial calculations.

11.4.6 C#

Basic floating point operations in C#, for variables declared as float or double, work as we have described in the earlier sections. They are based on IEEE 754 and done in hardware.

C# also has support to help with financial calculations with the `decimal` type, which implements the IEEE 754 decimal-base floating point standard. Of course, since modern processors do not support decimal floating point, these operations in C# are implemented in software, so are slower than the comparable `float` or `double` operations.

The exception for decimal support appears in processors from IBM, such as the Power11, which support IEEE 754 decimal floating point in hardware. Such a feature makes these processors especially attractive for large financial computations. Support for decimal floating point support in the Power architecture dates back to 2007 with the introduction of the Power7. IBM has a long history of decimal support going back to the IBM 360 architecture from 1964. The IBM 360 supported decimal numbers with fixed decimal points.

11.4.7 Perl

In Perl, you do not declare variables so type is inferred from their use. Because Perl is an interpreted language, different interpreters may treat numbers differently.

Commonly, if you write whole numbers, then they will stored as integers and if you write numbers with a decimal point or exponential notation, they will be stored as IEEE 754 floating point numbers. During calculation, the format might change, such as when you divide an integer.

In addition, Perl supports an arbitrary precision floating point representation call `Math:BigFloat`. While this can be useful, it is implemented in software so can be quite a bit slower than the regular number types. Perl also supports arbitrary precision integers with the `Math:BigInt` type. And, unusually, they have arbitrary precision rational numbers, call Math:BigRat, stored as a pair of `BigInt`'s.

11.5 Summary

Floating point arithmetic is a mathematically complex subject. Programmers who write programs that perform serious floating point computation, such as scientific simulations, would benefit from taking a course in numerical analysis to help prevent unexpected results and inaccuracies.

For financial computations, floating point numbers are a tempting and easy approach to handle currencies that are broken down into smaller parts such as cents. However, these monetary amounts are better done with a decimal representation.

However, for the systems programmer and security analyst, floating point numbers appear in a variety of places, sometimes unexpectedly.

Understanding the basics of floating point numbers helps to prevent unexpected behaviors in these unexpected places.

11.6 Exercises

1. Implement the example from Figure *3* in other languages that you know. Do you get results consistent with the version that we showed in C when you execute the program?
2. Implement the example from Figure *4* in other languages that you know. Do you get results consistent with the version that we showed in C when you execute the program?
3. For the example from Figure 5:
   a. Implement it in Python, Rust, or C#. Do you get the same results that we showed with C ?
   b. For the calculation that was done with the `float` and `double`, implement it in Python or Rust with the `Decimal` type or in C# with the `decimal` type. Do the results differ when you execute this version?
4. In languages with which you are familiar, write small test programs that cause a variety of floating point number errors such as overflow, underflow, and divide by zero. When you execute these programs, are there any exceptions or detected runtime errors?