Big Integer Design

Note: this page uses MathJax (https://www.mathjax.org/) in order to display mathematic formulas. If you disabled JavaScript processing, then formulas will appear in raw LaTeX notation.

This page explains the design and implementation of operations on big (modular) integers, used for RSA and generic elliptic curve computations. There are actually four such implementations in BearSSL, with the following code names:

- "i32" is historically the first one implemented. It splits integers into sequences of 32-bit words. It is in semi-deprecated state, because "i31" is faster. Some of the newer operations (optimized modular exponentiation, and modular divisions) have not been implemented in i32.
- "i31" splits integers into sequences of 31-bit words (each 31-bit word is stored in a 32-bit type `uint32_t`). As will be explained below, there are performance and portability reasons to do so.
- "i15" splits integers into sequences of 15-bit words. Contrary to "i31", it does not use any 32→64 multiplication opcode, so it is recommended on platforms on which such an opcode is not constant-time (e.g. ARM Cortex M3) or even missing (e.g. ARM Cortex M0+). On the ARM Cortex M0+, it is also quite faster than the i31 implementation.
- "i62" is identical to "i31", except for a dedicated routine for modular exponentations that leverages 64→128 multiplication opcodes on platforms that offers them (this uses non-standard extensions on either GCC/Clang on 64-bit architectures, or MSVC on x86 in 64-bit mode). On compatible platforms, use of i62 considerably speeds up RSA computations (public and private key operations, and key pair generation).

All of this code is in the `src/int` directory (https://www.bearssl.org/gitweb/?p=BearSSL;a=tree;f=src/int;h=2fa2ff106f0cf006b83e705855c2f85bf7d76ece;hb=8ef7680081c61b486622f2d983c0d3d21e83caad).

Note that some elliptic curve implementations (called "m31" and "m15" in BearSSL) use dedicated functions for computations in the specific fields used by some curves, leveraging the special format for the field modulus for faster computations (e.g. Curve25519 works with integers modulo $p = 2^{255} - 19$). These implementations are not described here.

The designs described herein are influenced by BearSSL goals: portability, security (in particular constant-time code (constanttime.html)), and economy of both RAM and ROM.

Math Basics

This section describes the mathematical foundations of the algorithms implemented in BearSSL. This is in no way an exhaustive treaty of computations on big integers; there are many representations and algorithms that are not shown here. These algorithms are the ones that are used in BearSSL (except Karatsuba multiplication, which is briefly exposed below, but not used in BearSSL right now).

Readers who want to learn more on such subjects are invited to have a look at the Handbook of Applied Cryptography (http://cacr.uwaterloo.ca/hac/), especially chapter 14 (the "handbook" chapters are free to download). Another source is the Guide to Elliptic Curve Cryptography (http://cacr.uwaterloo.ca/ecc/), which has a chapter on finite field arithmetic (and this is even the "free sample chapter" that can be downloaded from the site).

Representation

While C offers some integer types that can store integer values and perform computations on them, the integer values we are interested in far exceed the range of values that fit in a single `uint32_t` or `uint64_t`. Big integers must use a more complicated representation. There are several possibilities; the one used in BearSSL is a simple split into digits in some adequate basis.

Specifically, for a given integer W (the basis), a big integer a is represented as a sequence of integers a_i, such that $0 \leq a_i < W$ for all i, and:

$$a = \sum_{i=0}^{\infty} a_i W^i$$

The sum above is on an infinity of terms, but, in practice, all a_i are equal to zero beyond a certain index, so the sum is actually finite.

This representation is exactly what we are doing when writing integers in decimal: the integer “324” consists in three *digits*, which are 4, 2 and 3; the basis is $W = 10$; and digits beyond index 2 are zero for this integer. In a computer, though, the basis “10” is not very convenient; computers work in binary and are more comfortable with bases that are equal to powers of two. We will see that, in BearSSL, the i32, i31 and i15 implementations use bases 2^{32}, 2^{31} and 2^{15}, respectively. For non-decimal bases, it is traditional to call *limbs* the individual a_i values, intead of “digits”.

Other possible representations of big integers include Residue Number Systems (https://en.wikipedia.org/wiki/Residue_number_system) and Fourier transforms (for FFT multiplication (https://www.aimath.org/news/congruentnumbers/howtomultiply.html)). However, these alternate representations are cumbersome and relatively inefficient to use for the operations we are interested in, especially modular multiplications and exponentiations with moduli up to a few thousands of bits.

Additions And Subtractions

Suppose that a and b are two nonnegative integers that use N limbs in basis W, i.e. $a_i = 0$ and $b_i = 0$ for $i \geq N$. This can also be expressed as: $0 \leq a < W^N$ and $0 \leq b < W^N$.

The sum $d = a + b$ can be computed with the following algorithm:

```
1. Set: $c \leftarrow 0$     (it is the “carry”)
2. For $i$ from $0$ to $N - 1$, do:
     ◦ Set: $z \leftarrow a_i + b_i + c$
     ◦ Set: $d_i \leftarrow z \bmod W$
     ◦ Set: $c \leftarrow \lfloor z/W \rfloor$
```

The following must be noted:

- Even if both a and b fit on N limbs each, the sum d might need and extra limb. In the algorithm above, this will manifest itself as a non-zero carry c on output of the algorithm.

- The intermediate value z may exceed W. Indeed, since both a_i and b_i can range up to $W - 1$, then their sum can be up to $2W - 2$. If the carry c, upon entry of one iteration, can be 0 or 1, then the maximum value for z is $2W - 1$, which implies that the new carry on exit cannot be larger than 1. This shows that the carry c can only be 0 or 1 throughout the algorithm.

- If W is a power of two, then the modular reduction for the computation of d_i, and the division for the new carry c, are simple and do *not* involve the expensive division opcodes of the CPU. Indeed, if $W = 2^{15}$ (for instance), then reduction modulo W means keeping the low 15 bits, which can be done with a simple bitwise AND with `0x7FFF`, while division by W, rounded low, really is right-shifting the value by 15 bits.

Subtraction is done in a similar way. To compute $d = a - b$, use the following:

```
1. Set: $c \leftarrow 0$     (it is the “carry”)
2. For $i$ from $0$ to $N - 1$, do:
     ◦ Set: $z \leftarrow a_i - b_i - c$
     ◦ If $z < 0$, set $d_i \leftarrow z + W$ and $c \leftarrow 1$; otherwise, set $d_i \leftarrow z$ and set $c \leftarrow 0$.
```

In this case, we expressed the carry management with a comparison. The intermediate value z may range from $-W$ to $W - 1$ (inclusive), and the carry c will be 0 or 1, as in the case of the addition. Again, if W is a power of two, and z is represented as a signed integer type with two’s complement representation, then d_i can be obtained as a bitwise AND, and the new carry with some shifting/masking.

A subtraction result may be negative, in case $b > a$. In the algorithm above, this will show up as a non-zero carry c at the end of the algorithm: the final carry is 1 if and only if $d < 0$. This means that a subtraction can also be used as a comparison function.

Negative Integers And Two's Complement

So far we used only nonnegative integers. We are not especially interested in negative integers, because what we really want is *modular* integers, i.e. integers in the 0 to $m-1$ range for some positive modulus m, so we never really go into the "negative" range. However, for intermediate values, we may occasionally have to handle negative values.

A representation as limbs, each of value 0 to $W-1$, really is a *modular* represention: for an integer a (possibly negative), we can define for all $i \geq 0$ the limb a_i as:

$$a_i = \left\lfloor \frac{a}{W^i} \right\rfloor \bmod W$$

Note that in the notation above, $\lfloor . \rfloor$ denotes rounding towards $-\infty$, not necessarily towards 0. In particular, $\lfloor -1/W \rfloor = -1$, not 0. However, the modulus operation "$\bmod W$" returns a value in the 0 to $W-1$ range.

If a is negative, that definition yields an infinity of non-zero limbs; the "upper limbs", beyond some index i (that depends on the integer a) will all have value $W-1$. Of course, having an infinity of non-zero limbs means that the summation that yields a back is ill-defined. However, this does not matter much because we never really recompute the complete integer; we just want to use an adequate *representation* that allows us to keep on with operations.

In practice, we will use a given number N of limbs. These N limbs then really represent the nonnegative integer $a \bmod W^N$. When W is a power of two, this is called **two's complement**.

It is useful at this point to recall some properties of binary representations. "Binary" means using $W = 2$. However, using a larger basis W as a power of two really means using binary and handling "digits" by groups. A decimal equivalent would be the following: the integer 77805, represented in basis 10, consists in the digits 5, 0, 8, 7 and 7, followed by an infinity of zeros (that we normally choose not to write down). The same integer, in basis 100, would use the "digits" (limbs) 05, 78 and 7. Here, we see that each limb in basis 100 is really a group of two digits in basis 10. More generally, if $W' = W^k$ for some integer $k > 1$, each limb in basis W' consists in a group of k consecutive limbs in basis W.

This means that, when we use limbs in basis 2^{31} (for instance), we will really be using a binary representation, such that binary digits are grouped into sequences of 31 consecutive bits, and we process these bits by whole groups at a time because it happens to be more convenient.

Now consider the integer $a = 332$. We will represent it in binary, over 16 bits. There is nothing special about "16" here; it is just a number of bits which is large enough to demonstrate things, but not so large that it would overwhelm readers with too many zeros and ones. Everything I write below can be applied to other lengths such as 32 or 75 or whatever. Expressed in binary, over 16 bits, integer a is:

```
    a = 0000000101001100
```

Here, I wrote the bits in the conventional "big-endian" order, that we use in languages that are written with the Latin alphabet, such as English: the "least significant" digit a_0 is on the rightmost place, while the "most significant" digit a_{15} is on the leftmost place.

There is a convenient notation called **hexadecimal**. This really is basis 16, and the limbs (of value 0 to 15) are written down as either normal decimal digits (`0` to `9`) or letters (`A` to `F`, for values 10 to 15, respectively). 16 is a power of two: $16 = 2^4$. Thus, to convert from binary to hexadecimal, we just have to group binary digits into packs of four:

```
    a = 0000 0001 0100 1100
          0    1    4    C
```

The rightmost group has value 12, which is represented as the letter `C` in hexadecimal. Since hexadecimal uses decimal digits, it can lead to confusion. In the C programming language, and many other inspired from its syntax, an hexadecimal literal integer is indicated by a `0x` prefix[1]. We would then write `0x014C` for the integer 332. The ease of conversion between binary and hexadecimal makes it a convenient representation to use.

Now for two's complement. It is usually defined in the following way: to negate an integer, first invert all bits, then add 1. For instance, for 332, expressed over 16 bits, this yields the following:

```
       a = 0000000101001100     (0x014C)
      ~a = 1111111010110011     (0xFEB3)
    ~a+1 = 1111111010110100     (0xFEB4)
```

The ~ notation is here the C operator that invert all bits, i.e. replaces each zero with a one, and each one with a zero. This definition seems a bit magical if presented without justification, but what it does is the following: since $1-0=1$ and $1-1=0$, inverting all bits really means subtracting each bit from 1, i.e. subtracting the value a from an "all-ones" value, 0xFFFF in our example. The all-ones value over n bits is equal to 2^n-1; thus, ~a is really the representation of $2^{16}-1-a$. By adding 1, we then get $2^{16}-a$. This last value is indeed equal to $-a$ modulo 2^{16}; this is why two's complement is used for negation.

Another view is the following: each sequence of 16 bits can be *interpreted* into an integer with two conventions, called *signed* and *unsigned*. The *unsigned* convention is the one we started this whole page with: bit i has numerical value 2^i. In the unsigned convention, the most significant bit (leftmost in our written sequences) has value $2^{15}=32768$. The two's complement of a, as computed above, has value 65204 (in hexadecimal, 0xFEB4) when interpreted in unsigned convention. The *signed* convention, on the other hand, posits that the sequence is really meant to represent integers that can be negative; to that effect, it declares that the most significant bit is the "sign bit" and it has value $-2^{15}=-32768$; all other bits have the same value as in the unsigned convention. If you interpret the 0xFEB4 sequence in signed convention, i.e. with that special meaning for the top bit, then you will find that the resulting numerical value is -332, which is what we wanted.

Take note that 2^{15} and 2^{-15} are identical modulo 2^{16}; and, indeed, the two interpretation results 65204 and -332 are also identical modulo 2^{16}. This is why we talk of *interpreting* the result. From the point of view of the computer, where operations are really done modulo 2^{16}, this changes nothing. A developer writing code may have in his head the convention that a given value is signed or unsigned; usually, the compiler will also know about the convention too, because the developer used different types, such as `int16_t` and `uint16_t`. However, the CPU does not know nor care: when adding two integers, the very same opcode and circuitry is used, regardless of the interpretation of the operands and the result. This is the main reason why two's complement is used[2].

Thus, back to subtraction: when using the subtraction algorithm on integers a and b represented over N limbs, and the numerical result $d=a-b$, then what we actually obtain is the two's complement representation of d. The output carry is the value of all the extra bits up to infinity: all zeros for a nonnegative result, all ones for a negative result.

Modular Addition And Subtraction

For cryptographic applications, we normally work with modular integers. After an addition or subtraction, the result may be out of range. To fix it, the modulus must optionally be subtracted or added. We now consider nonnegative integers a, b and m, all representable over N limbs; moreover, a and b are lower than modulus m. The algorithm for modular addition (computing $a+b \bmod m$) is then:

1. Compute the addition $a+b$, as described above; this yields $d=a+b \bmod W^N$, and a carry c.
2. If $c=1$, or if $d \geq m$, then subtract m from d (ignoring any resulting carry).

Note that the test "$d \geq m$" uses the unsigned interpretation of the value d we computed in the first step, ignoring the extra carry c. There is an alternate description, which merges the subtraction and the comparison (since subtraction and comparison are actually the same operation):

1. Compute the addition $a+b$, as described above; this yields $d=a+b \bmod W^N$, and a carry c.
2. Compute $d-m$, as described above; this yields $e=a+b-m \bmod W^N$, and a carry c'.
3. If $c=c'$, the result is e; otherwise, the result is d.

This alternate description is also more amenable to constant-time implementations, since the two internal operations (addition and subtraction) are always done. In the final step, there can be three cases:

- $c=c'=0$: the addition did not overflow the N limbs, but the result d was not lower than m, hence the subtraction was needed.
- $c=c'=1$: the addition overflowed the N limbs, meaning that the true mathematical value $a+b$ requires an extra bit for its representation (which is the carry). The final carry of the subtraction (c') consumes that extra bit.
- $c=0$ and $c'=1$: the addition does not overflow; subtracting m would yield a negative value, so the result to keep is

d, not e.

The fourth case ($c = 1$ and $c' = 0$) is not possible: it would require $a + b \geq W^N + m$, which cannot happen if the operands are in the proper range (i.e. $a, b < m < W^N$).

Note that nothing here assumes that N is minimal. There is no obligation for a, b and m to fill all N limbs up to the last bit. These algorithms also work when m is smaller. This means that there is nothing special to do when the modulus size (in bits) is not a multiple of the limb size.

The modular subtraction algorithm is similar:

1. Compute the subtraction $a - b$, yielding $d = a - b \bmod W^N$, and a carry c.
2. Compute $d + m$, yielding $e = d + m \bmod W^N$, and a carry c'.
3. If $c = c'$, the result is e; otherwise, the result is d.

There again, three cases are possible in the final step; you cannot obtain $c = 1$ and $c' = 0$ since that would mean that $a - b$ is negative and adding m back is not sufficient to make it nonnegative.

General Modular Reduction

In the previous section, we saw how to fix results of additions and subtractions to get a modular result in the proper range. However, we also need a more general modular reduction operation, when the input is potentially much larger than the modulus. This is used in particular with RSA, for private key operations: the value to exponentiate is an integer x modulo n, where $n = pq$ is the modulus. However, we prefer to compute the exponentiation modulo p and modulo q separately, because doing so is much faster (since p and q are typically twice smaller than n); the final result will be assembled through the CRT (https://en.wikipedia.org/wiki/Chinese_remainder_theorem).

A general modular reduction entails an Euclidean division: given and integer $x \geq 0$, and $m > 0$, compute the quotient q and remainder r such that:

$$\begin{aligned} 0 &\leq q \\ 0 &\leq r < m \\ x &= qm + r \end{aligned}$$

For reduction of x modulo m, we don't care about the quotient q; we just want the remainder r, which is the result.

The basic algorithm for that is the one taught in school: the quotient is computed digit by digit, starting with the most significant non-zero digit. School pupils often gripe with the "guessing" phase: when you have placed the divisor, properly shifted to the left, below the current dividend, you must find the next quotient digit, which you usually guess by looking at the first (leftmost) digits of the dividend and the divisor. Sometimes the guess is wrong, over or underestimated, and you must get out your eraser and try again. In binary, things are simpler: since a digit can be only 0 or 1, a simple comparison is sufficient. In fact, the comparison is a subtraction: if it "works" (no carry, i.e. result is nonnegative), then the next quotient digit is 1, and the new dividend is the subtraction result. Otherwise, the next quotient digit is 0, and the dividend is unchanged for this step.

Instead of shifting the divisor, we can shift the dividend. Remember that, in binary, shifting a value to the left is equivalent to multiplying by two (just like, in decimal, adding a zero on the right means multiplying by ten). This yields the following algorithm; we have an input value x expressed over k bits (from least significant x_0 to most significant x_{k-1}); we want to reduce that value modulo m, which is represented over N limbs in basis W.

1. Set: $d \leftarrow 0$.
2. For $i = k - 1$ down to 0:
    - Set: $d \leftarrow 2d + x_i$, with output carry in c.
    - Set: $e \leftarrow d - m$, with output carry in c'.
    - If $c = c'$, then set: $d \leftarrow e$

The "left-shifting", i.e. multiplication of d by 2, can be done with the addition algorithm seen previously; a dedicated shifting algorithm can also be implemented. In either case, we need a carry, to indicate when the dropped leftmost bit was non-zero. Note that adding x_i cannot trigger any carrying, since the left shift necessarily left a zero in the least significant bit; the x_i bit merely replaces that zero.

Some notes:

- At each iteration, on input, we have $d < m$. Thus, $d \leq m - 1$, and $2d + x_i \leq 2m - 1$. Subtracting m at most once is enough to bring the result back to the 0 to $m - 1$ range.
- As with modular addition, the case $c = 1$ and $c' = 0$ is not possible if the operands are correct (an incorrect operand would be $m = 0$).
- If the modulus m has binary size t bits, i.e. $2^{t-1} \leq m < 2^t$, then the first $t - 1$ iterations are simpler: none of the shifts overflows (hence $c = 0$), and all subtractions output a carry ($c' = 1$). We can then optimize these $t - 1$ initial steps by simply copying the most significant $t - 1$ bits of x into d, and start from there.

Now this algorithm is simple enough to implement, but its bit-by-bit processing is rather slow. It is possible to speed processing up by injecting a full limb of x at each iteration. If x is now represented over k *limbs* (x_0 to x_{k-1}), the algorithm looks like this:

```
1. Set: $d \leftarrow 0$.
2. For $i = k - 1$ down to $0$:
    ◦ Set $d_h$ to the top (most significant) limb of $d$.
    ◦ Set: $d \leftarrow Wd + x_i$; the previous top limb of $d$ is dropped (this is the value $d_h$).
    ◦ “Guess” the next quotient limb $v$, which is such that $0 \leq d_h W^N + d - vm < m$.
    ◦ Subtract $vm$ from $d_h W^N + d$; result is the new value of $d$.
```

This entails substantially more complicated operations. First, we have a multiplication of m by a small integer v. Note that v is always lower than W here: on input at each iteration, we have $d \leq m - 1$, and $x_i \leq W - 1$; therefore, $Wd + x_i \leq mW - 1$. Multiplication of m by v is an extension of the addition:

```
1. Set: $c \leftarrow 0$
2. For $i = 0$ to $N - 1$:
    ◦ Set: $z \leftarrow m_i v + c$
    ◦ Set: $m_i \leftarrow z \bmod W$
    ◦ Set: $c \leftarrow \lfloor z/W \rfloor$
```

This algorithm returns in c the extra top limb of vm (since m fits on N limbs and $v < W$, at most one extra limb is needed to contain the complete result).

The “guessing” step in the optimized reduction algorithm is more complex. First, we need to extract the “true” top word of m; indeed, m may use less than N limbs, or even if it uses N limbs, its most significant limb might be lower than $W/2$ (i.e. the top bit is zero). To simplify, we assume here that the top limb of m is not zero (i.e. the implementation will first reduce N if it is too large for the modulus; this is desirable for performance anyway). Then, we define the extra shift count s which is the unique integer such that:

$$W^N/2 \leq 2^s m < W^N$$

It is easily seen that $0 \leq s < \log_2 W$; i.e. if we use limbs of 31 bits, s will be at most 30.

Suppose that, at some step of the algorithm, we have $d_h W^N + d$, and we want to find v such that $0 \leq d_h W^N + d - vm < m$. We know that the result v will be less than W. To compute v, we first scale up values:

$$\begin{aligned} a &= 2^s(d_h W^N + d) \\ b &= 2^s m \end{aligned}$$

Scaling things up means, in binary, adding s zeros on the right; it does not change the quotient, i.e. we still want $v = \lfloor a/b \rfloor$. Now split a and b into limbs:

$$\begin{aligned} a &= (a_t W + a_h) W^{N-1} + a_l \\ b &= b_h W^{N-1} + b_l \end{aligned}$$

with a_t, a_h and b_h being limb-sized (i.e. less than W), and a_l and b_l being less than W^{N-1}. In other words, we isolate the top limb of b (the scaled divisor m) and the top two limbs of a (the scaled dividend $d_h W^N + d$). Moreover, since we scaled things up by s bits, we know that $W/2 \leq b_h < W$ (this was the point of scaling things up). In order to *estimate* the quotient v of a by b, we perform the Euclidean division of $a_t W + a_h$ by b_h:

$$a_t W + a_h = b_h q + r$$

with $0 \leq r < b_h$. Since $a_t W + a_h$ fits on two words, and b_h is a single word, then this operation can be handled without fiddling with big integers at all.

It can be shown that, thanks to the scaling up, the following holds:

$$q - 2 \leq v \leq q$$

i.e. the value $q - 1$ is a reasonably good approximation of v. We can then subtract $(q-1)m$ from $d_h W + d$, then "fix" the result by optionally subtracting m one more time, or adding m, so that we obtain a nonnegative result less than m.

Implementation of that mechanism has some subtleties:

- The first few steps of the modular reduction are trivial: if m really uses W words (i.e. $m \geq W^{N-1}$), then the first $W - 1$ iterations incur no reduction at all. In other words, the injection of the twop $W - 1$ words of the operand x are "free".
- We don't have to actually shift whole integers. We formally define a and b as values scaled up by s bits, but all we want are a_t, a_h and b_h. When v is obtained, we subtract vm from the non-shifted value.
- Scale count s, and the top word b_h, are fixed over the course of the whole modular reduction. We don't have to compute them repeatedly.
- The result q of the Euclidean division of $a_t W + a_h$ by b_h ranges from 0 to W, *inclusive*. If it is 0, then $q - 1$ is negative, which is a problem; in that case, the guessed quotient should be q, not $q - 1$. At the other end of the range, W might be "too big": if W is the natural word size of the computer, and the CPU division opcode is used, then a quotient equal to W is an overflow condition, which can have deleterious effects (on x86 CPU, this triggers a CPU exception)[3]. It can be shown that, with the way we defined values, this case is reached only when $a_t = b_h$, so we can use that test as a precondition to fix things.

Montgomery Reduction And Multiplication

Assuming we have a generic modular reduction mechanism, we can use it to implement modular multiplication by performing "normal" integer multiplication, then reducing the result. We can even intertwine the integer multiplication with limb-wise reductions (the individual steps of the modular reduction, as described above) so as not to require any extra buffer. However, the Euclidean divisions and the guess/fix pattern tend to be expensive. It is more efficient to use a trick known as *Montgomery multiplication*.

Conceptually, the inefficiency of generic modular reduction is in the "guess" phase: we estimate the limb-wise reduction quotient v by looking at the top limbs of the operands, but we may be off, thus requiring some extra fixing steps. The error in the estimate comes from carry propagation from lower limbs. The role of the quotient v is to cancel the extra limb (d_h) at the top of the integer. The principle of Montgomery multiplication is to instead cancel the *least significant* limb of the integer; this can be done more efficiently because there are no carries that propagate from lower limbs and make the estimate off; moreover, it avoids the Euclidean division on words, using instead a much simpler multiplication.

Montgomery multiplication is done as follows. Operands are integers a and b modulo m (i.e. they are nonnegative and less than m). Integers are encoded as sequences of N limbs in basis W. **Important:** we now assume that W is a power of two.

```
1. Set: $g \leftarrow -m_0^{-1} \bmod W$
2. Set: $d \leftarrow 0$ (big integer, $N$ limbs)
3. Set: $d_h \leftarrow 0$ (one limb-size variable)
4. For $i = 0$ to $N - 1$:
   ◦ Set: $f \leftarrow (d_0 + a_i b_0) g \bmod W$
   ◦ Set: $c \leftarrow 0$
   ◦ For $j = 0$ to $N - 1$:
      ▪ Set: $z \leftarrow d_j + a_i b_j + f m_j + c$
      ▪ If $j > 0$, set: $d_{j-1} \leftarrow z \bmod W$
      ▪ Set: $c \leftarrow \lfloor z/W \rfloor$
   ◦ Set: $z \leftarrow d_h + c$
   ◦ Set: $d_{N-1} \leftarrow z \bmod W$
   ◦ Set: $d_h \leftarrow \lfloor z/W \rfloor$
5. If $d_h \neq 0$ or $d \geq m$, set: $d \leftarrow d - m$
6. Return: $d$
```

Let’s analyse this algorithm:

- There is inside this algorithm a “schoolbook” multiplication. We compute all a_ib_j products and add them at the proper slots, with carry propagation. Every outer iteration corresponds to computing a_ib, and adding it at the right place in the result.
- However, for every outer iteration, we *also* add a corrective value fm. The point of this correction is to clear the low bits of the result. Indeed, f is computed exactly such that $d + a_ib + fm = 0 \bmod W$. Thus, at the first inner iteration (when $j = 0$), the intermediate value z will be equal to zero modulo W.
- Since the low limb of the intermediate d is all-zero, we can easily divide that value by W, by simply right-shifting the words. This is the meaning of writing into d_{j-1} instead of d_j. The division is exact, since the limb which is dropped is all-zero. Therefore, what each outer iteration computes really is $(d + a_ib + fm)/W$.
- It can be shown that the final d may be larger than m, but will be less than $2m$. This is why the final corrective step (5) is a simple conditional subtraction.
- In total, what this algorithm compute is thus a value $d = ab/(W^N) \bmod m$.

Note that some intermediate variables have a slightly larger range than usual. Value z may be larger than W^2 (but it will be less than $2W^2$). Similarly, d_h and c may be larger than W (but they will be less than $2W$). Thus, if you, for instance, use $W = 2^{32}$, then c and d_h will need 33 bits, and z will need 65 bits (this is, in a nutshell, why BearSSL’s “i31” implementation uses $W = 2^{31}$ instead).

A necessary condition for the algorithm is that m_0 be invertible modulo W. If W is a power of two, this means that m must be odd. Montgomery’s multiplication does not work with even integers. Fortunately, we don’t compute things modulo even integers in usual cryptographic algorithms[4]. Support of even moduli can be done by splitting an even modulus m into $m = 2^km'$ with m' odd, computing modulo 2^k and modulo m' separately, and recombining with the CRT. This is complex and hard to do in constant-time code (if the value k must remain hidden).

To compute $m_0^{-1} \bmod W$, when W is a power of two, there is a remarkable trick. Suppose that $y = x^{-1} \bmod 2^k$, i.e. y is an inverse of x for the low k bits. Then there is an integer r such that $xy = 1 + 2^kr$. Therefore:

$$\begin{aligned} xy(2 - xy) &= (1 + 2^kr)(1 - 2^kr) \\ &= 1 - 2^{2k}r^2 \end{aligned}$$

Thus, $y(2 - xy)$ is an inverse for x modulo 2^{2k}: we just doubled the number of bits for our inverse. This yields the following algorithm for inverting m_0 modulo W (when W is a power of two):

```
1. Set: $y \leftarrow m_0 \bmod W$
2. For $i = 2$ to $\log_2 W$:
    ◦ Set: $y \leftarrow y(2 - m_0y) \bmod W$
3. Return $y$
```

Note that for every odd integer m_0, m_0 is its own inverse modulo 4; thus, the initial step sets y to the correct value for the low two bits. Each extra iteration doubles the number of bits, so four iterations are sufficient for W up to 2^{32} (five iterations for up to 2^{64}).

Let’s see what this Montgomery multiplication really *means*. To make notations clearer, let’s define $R = W^N \bmod m$. Given a and b, the Montgomery multiplication computes the product ab, to which it adds some multiple of m, and then divides the result by W exactly N times. Finally, the obtained value is reduced modulo m. All the divisions are exact, so the Montgomery multiplication result really is:

$$\frac{ab + km}{W^N} \bmod m$$

for some integer k. Now, W is relatively prime to m (we assumed that W is a power of two, and m is odd). This means that W^N is invertible modulo m; indeed, its inverse is $R^{-1} \bmod m$. This means that what we computed is:

$$(ab + km)R^{-1} \bmod m$$

Since we work modulo m, this is equivalent to:

$$\frac{ab}{R} \bmod m$$

This is not exactly what we wanted (the product of a abd b modulo m) but it is close enough.

We define the **Montgomery representation** of an integer a modulo m: it is an integer equal to $aR \bmod m$. Then, notice that:

$$aR + bR = (a+b)R \bmod m$$
$$\frac{(aR)(bR)}{R} = abR \bmod m$$

Therefore, to compute additions and multiplications of modular integers, we can use modular additions and Montgomery products, respectively, of their Montgomery representations. An algorithm that involves many such operations (e.g. modular exponentiations, or elliptic curve computations) can simply convert inputs to Montgomery representation at the start, do all operations on the Montgomery representations, and convert the results to their usual representation at the end.

Converting from Montgomery representation to the normal representation is called **Montgomery reduction**. A simple way to do that is to use Montgomery multiplication with 1 (it will compute $(aR)1/R = a \bmod m$). A dedicated function can be written, that takes advantage of that special case to be faster than a generic Montgomery multiplication routine; this is a trade-off (more code size for faster computations).

The inverse operation (converting *to* Montgomery reduction) can also be done with a Montgomery multiplication, this time with R^2 (because $(aR^2)/R = aR \bmod m$). This requires the use of $R^2 \bmod m$. That value may be precomputed, if the same modulus is used repeatedly; this applies in particular to the case of elliptic curves, where standard curves are defined over specific, fixed fields, for whom the corresponding R^2 constants can be hardcoded.

Another method for converting integer a to Montgomery representation is to use repeated doubling; if $W = 2^k$, then we can use the following:

- For $i = 1$ to kN, do:
  - Set: $a \leftarrow 2a \bmod m$

Multiplication by 2 can be done with an explicit left-shift (we saw that operation already, in the implementation of generic modular reduction) or with a modular addition of a with itself. This method is simple and tends to reuse code which is already there. It is also relatively slow. This is usually not a problem when dealing with expensive operations such as modular exponentiation or curve point multiplication, because the relative cost of converting to Montgomery representation will be negligible.

Since conversion to and from Montgomery representation really are products with constants, it often happens that these operations can be smuggled into other computations for free. For instance, suppose that you have inputs x and y (modulo m) and want to compute $xy \bmod m$; the generic method would be to do the following:

- Convert x to Montgomery representation $xR \bmod m$.
- Convert y to Montgomery representation $yR \bmod m$.
- Perform the Montgomery product of xR and yR to obtain $xyR \bmod m$.
- Convert back the result into normal representation $xy \bmod m$.

Now, if what you want is indeed the product in normal representation, and are not interested in the Montgomery representation of xy, then you can simplify the above into:

- Convert x to Montgomery representation $xR \bmod m$.
- Perform the Montgomery product of xR and y to obtain $(xR)y/R = xy \bmod m$.

Thus, one conversion to Montgomery representation, and the conversion back, have been saved.

Modular Exponentiation

Modular exponentiation is about computing $x^e \bmod m$, given modulus m, integer x modulo m (i.e. in the 0 to $m-1$ range), and a nonnegative exponent e. The straightforward method of performing $e-1$ multiplications is not applicable when e is large (it would be too expensive, possibly taking longer than the age of the Universe). The basic general method of modular exponentiation is called *square-and-multiply*.

Suppose, for instance, that you want to compute $x^{197} \bmod m$. This can be done with a relatively small sequence of squarings and multiplications; in the table below, I detail on the left column the operations, and on the right column the contents of y after that operation:

Operation	Result
$y \leftarrow x$	x
$y \leftarrow y^2 \bmod m$	$x^2 \bmod m$
$y \leftarrow yx \bmod m$	$x^3 \bmod m$
$y \leftarrow y^2 \bmod m$	$x^6 \bmod m$
$y \leftarrow y^2 \bmod m$	$x^{12} \bmod m$
$y \leftarrow y^2 \bmod m$	$x^{24} \bmod m$
$y \leftarrow y^2 \bmod m$	$x^{48} \bmod m$
$y \leftarrow yx \bmod m$	$x^{49} \bmod m$
$y \leftarrow y^2 \bmod m$	$x^{98} \bmod m$
$y \leftarrow y^2 \bmod m$	$x^{196} \bmod m$
$y \leftarrow yx \bmod m$	$x^{197} \bmod m$

What happens here is best explained on the binary representation of the exponent. In binary, 197 is 11000101. We start with $y = x^1$, i.e. x raised to an exponent v which is, at that point, equal to 1. Squaring y means replacing x^v with x^{2v}, i.e. it *doubles* the exponent. In binary, the action of doubling is equivalent to a left-shift by 1 bit; i.e. if the exponent v was (for instance) 110 (in binary), then after the squaring it has become 1100. Multiplying by x, on the other hand, adds 1 to the current exponent v; if the exponent, at that time, ends with a zero (in binary), then adding 1 changes that zero into a one. For instance, if the exponent v is 1100 (in binary), then the effect of multiplying y by x is to make v become 1101.

The gist of the square-and-multiply algorithm is to start with $v = 1$ (i.e. $y = x$) and then use squarings and multiplications to turn v into exponent e, one bit at a time. We push bits of e into v, starting from the left (most significant bits); to append a zero on the right of v, we square y; to append a one, we square y *then* multiply y by x. In total, there won't be more multiplications by x than there are squarings, and there will be $n - 1$ squarings if the exponent e has length n bits (i.e. is less than 2^n). This scales up nicely to very large exponents e. The algorithm is expressed as follows:

```
1. Set: $y \leftarrow 1$
2. For $i = n - 1$ down to $0$ (where $e < 2^n$):
     ∘ Set: $y \leftarrow y^2 \bmod m$
     ∘ If $\lfloor e/2^i \rfloor \bmod 2 = 1$, then set: $y \leftarrow yx \bmod m$
3. Return: $y$
```

Here we made y start at 1 instead of x; this has the benefit of not making the first step special, i.e. it will work even if the actual length of e is *less* than n bits (this will come handy for constant-time exponentiation that tries to hide the true length of the exponent). On the other hand, it makes one extra multiplication by x, and it also mishandles the corner case of $x = 0$ (it will return 1 for 0^e instead of 0, which is mathematically wrong).

There is a dual version of the square-and-multiply algorithm which uses bits of e from the right to the left:

```
1. Set: $y \leftarrow 1$
2. For $i = 0$ to $n - 1$ (where $e < 2^n$):
     ∘ If $\lfloor e/2^i \rfloor \bmod 2 = 1$, then set: $y \leftarrow yx \bmod m$
     ∘ Set: $x \leftarrow x^2 \bmod m$
3. Return: $y$
```

In this version, we simply multiply together the values of $x^{2^i} \bmod m$ for the non-zero bits e_i of the exponent e. This algorithm basically uses the same number of squarings and multiplications as the other one; which one is best to use depends on operational conditions such as whether the exponentiation is done in-place or not.

In a constant-time implementation, where the exponent value is secret, the conditional multiplication must be avoided; instead, the multiplication should *always* be performed, and the result kept or discarded (with a constant-time conditional copy primitive). This makes the cost rise to $2n$ multiplications (half of them being squarings) for a n-bit exponent.

Note that a squaring can be computed with a generic multiplication routine. It is also possible to make a specialized squaring routine which will be a tad faster than a generic multiplication, taking into account that $x_i x_j = x_j x_i$, and with more caching of data in registers (since there is one operand instead of two, there is less data to load from RAM, so a bigger proportion of it will fit in registers, and there could conceptually be less traffic between memory and registers). Practical improvements of up to about 20% have been reported (i.e. a squaring may cost down to about 0.8 times a generic multiplication). That kind of saving is much harder to obtain for *generic* code, where the modulus size is not known in advance; also, specialized routines mean more code, which runs afoul of the goal of minimizing code footprint.

The window optimization is a common trick to save on the number of multiplications (but not on the squarings). Suppose that, in the square-and-multiply algorithm (first version, with exponent bits considered from left to right), the next three exponent bits to apply are 101. This is nominally done with three squarings, and two multiplications by x (in a constant-time implementation, three squarings and three multiplications by x, the middle one being ultimately discarded). However, if the value $x^5 \bmod m$ is available, then the processing of these three bits can be reduced to three squarings, and *one* multiplication by x^5 (notice how 5 is 101 when written in binary).

The window optimization consists in precomputing a table of values $x^j \bmod m$ for all j from 0 to $2^w - 1$, with w being the “window width”. Then, the square-and-multiply will do one squaring for each bit of the exponent, but multiplications will be done only once every w bits of the exponent.

Making the window larger has diminishing returns; if you raise w, then the initial precomputation is more expensive, both in CPU (these are 2^w values to compute) and in RAM usage (the 2^w values must be stored somewhere). On the other hand, the window optimization does nothing to reduce the number of squarings, whose cost tends to dominate. Consider for instance a constant-time square-and-multiply: $2n$ multiplications (half of which being squarings). With 2-bit windows, there will be $1.5n$ multiplications (including the n squarings), a cost diminution of 25%. Going to 3-bit windows lowers that number to $1.333n$, i.e. a gain of only 11% over the 2-bit windows, and that’s without counting the extra overhead for computing and storing the table contents (values x^0 to x^7). Increasing to a 4-bit window scrapes only an extra 6.25%, again not counting the table construction overhead.

There are variants of the window optimization that save on the table construction by making multiplications at exponent-dependent emplacements within the sequence of squarings (e.g. instead of doing four squarings *then* multiplying by x^8, one can make one squaring, multiply by x, then do three other squarings; this allows not precomputing x^j for even values of j). However, these extra optimizations make it much harder to obtain constant-time implementations.

Modular Inversion And Division

Modular inversion is about, given x modulo m, computing $x^{-1} \bmod m$, which is such that $xx^{-1} = 1 \bmod m$. This is defined only if x is invertible modulo m. The generalization is modular division: given x and y modulo m, compute $y/x \bmod m$, which is such that $x(y/x) = y \bmod m$. There again, this is ill-defined when x is not prime with m.

If the modulus is prime, then there is a straightforward method that leverages Fermat’s Little Theorem, which says that, when m is prime and $x \neq 0 \bmod m$, then $x^{m-1} = 1 \bmod m$. Thus, an inverse of x modulo m can be computed as:

$$x^{-1} = x^{m-2} \bmod m$$

This method is simple but not especially fast. Moreover, it only works for prime moduli. If m is not prime, Fermat’s Little Theorem can be extended, but not in pleasant ways from an implementation point of view[5]. A better method is the Extended Euclidean algorithm (https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm), and in particular its variant known as the **binary GCD**, which is described below.

There are several descriptions of the binary GCD. The one below, given x and y modulo m, will compute the GCD δ of x and m, as well as an integer u such that $\delta y = xu \bmod m$. In particular, if $\delta = 1$ (which means that y is invertible modulo m), then $u = y/x \bmod m$.

1. Set:

$$\begin{aligned} a &\leftarrow x \\ b &\leftarrow m \\ u &\leftarrow y \\ v &\leftarrow 0 \end{aligned}$$

2. While $a \neq b$:
   - If a is even, then:

     $$\begin{aligned} a &\leftarrow a/2 \\ u &\leftarrow u/2 \bmod m \end{aligned}$$

   - Otherwise, if b is even, then:

     $$\begin{aligned} b &\leftarrow b/2 \\ v &\leftarrow v/2 \bmod m \end{aligned}$$

   - Otherwise, if $a > b$ is even, then:

     $$\begin{aligned} a &\leftarrow (a-b)/2 \\ u &\leftarrow (u-v)/2 \bmod m \end{aligned}$$

   - Otherwise, we know that $b > a$, and we do:

     $$\begin{aligned} b &\leftarrow (b-a)/2 \\ v &\leftarrow (v-u)/2 \bmod m \end{aligned}$$

3. The common value of a and b is δ, the GCD of x and m. The values u and v are also equal to each other; if $\delta = 1$ then $u = y/x \bmod m$.

This algorithm works as long as m is odd (it is not necessary that m is prime). The divisions by 2 of a, b, $a-b$ and $b-a$ are exact (when they are applied, the value to divide is even); however, the divisions by 2 of u, v, $u-v$ and $v-u$ may require some special treatment. Since we assume that m is odd, then if u is odd, $u+m$ will be even. Thus, $u/2 \bmod m$ can be computed in the following way:

- If u is odd, then compute $u \leftarrow u+m$.
- Now that u is even, divide it by 2 with a right-shift.

An equivalent method is:

- Right-shift u by one bit.
- If the dropped bit (the least significant bit of u before the shift) is non-zero, then add $(m+1)/2$ to the result.

In the binary GCD as described above, the following invariants are kept throughout the algorithm:

$$\begin{aligned} ay &= xu \bmod m \\ by &= xv \bmod m \end{aligned}$$

This is obviously true of the starting point. Then, whenever we update a (respectively b), we also update u (respectively v) to maintain our invariants. Therefore, when the algorithm converges, the value u is the division result we want (provided that the GCD is 1).

As described above, the number of steps in the binary GCD algorithm is not fixed; this is a problem if we want constant-time code. However, we can bound the number of iterations: each iteration will necessarily reduce the size of a or of b by at least one bit. Therefore, if m has size k bits (i.e. $2^{k-1} \leq m < 2^k$), then there can be no more than $2k-1$ iterations. A constant-time implementation will run for exactly $2k-1$ iterations.

For integers of size k bits, the complexity of the binary GCD is $O(k^2)$, i.e. quadratic, which compares favourably to the modular exponentiation of Fermat's Little Theorem (which is $O(k^3)$ with Montgomery multiplication). However, the binary GCD is bit-by-bit, which means that the constant hidden in the "O" notation is quite big, making the binary GCD not much faster than Fermat's Little Theorem method. The binary GCD still has the added advantage of handling non-prime (odd) moduli; this is useful in some cases, especially RSA key pair generation.

It is possible to optimize the binary GCD in the following way: notice that, at each iteration, the choice of operations is based mostly on the least significant bit of a and b (even or odd), and on the few most significant bits of a and b (to know which one is the greater of the two). Thus, we can run the loop as:

- Get the two upper words of a as a_h, and the low word as a_l.

- Get the two upper words of b as b_h, and the low word as b_l.

- Perform $t = \log W$ rounds of binary GCD on $A = a_h W + a_l$ and $B = b_h W + b_l$. The operations are aggregated into reduction factors p_a, p_b, q_a and q_b which are such that $Ap_a + Bp_b$ and $Aq_a + Bq_b$ are both multiple of W, but of minimal size.
- Set:

$$\begin{aligned} a &\leftarrow (ap_a + bp_b)/W \\ b &\leftarrow (aq_a + bq_b)/W \\ u &\leftarrow (up_a + vp_b)/W \bmod m \\ v &\leftarrow (uq_a + vq_b)/W \bmod m \end{aligned}$$

  The operations on a and b are exact (the value is a multiple of W), but operations on u and v are modular. The "division by W" is performed as in the inner loop of the Montgomery multiplication.

The use of A and B is an approximation: these are not the real a and b, but three-limb values that give similar results in the binary GCD algorithm. Thus, it is not guaranteed that a and b are always reduced by t bits; however, it can be shown that they are reduced by at least $t - 1$ bits, which is good enough. With this technique, the "i31" implementation of modular division in BearSSL costs about 36 times that of a Montgomery multiplication (with a 521-bit odd modulus), while using Fermat's Little Theorem would raise that factor to about 600, while being at the same time restricted to prime moduli only. On the other hand, a dedicated optimized binary GCD implementation is extra code, hence (again) a greater code size for faster computations.

Karatsuba Multiplication

In this section, I present Karatsuba's multiplication algorithm for a more complete treatment; however, it is *not* used in BearSSL. The main reason for that is that while use of Karatsuba has definite performance benefits, especially for integers in the range of typical RSA primes (1024 bits), it also requires some extra storage buffers. BearSSL strives to support constrained systems, in particular embedded system for which stack space is scarce. Using Karatsuba for the largest RSA moduli sizes (e.g. 4096 bits) would require more than 3 kB of stack buffers, which is more than some systems can afford.

Karatsuba's multiplication algorithm is a well-known optimization technique for multiplication of big integers (it also works on polynomials). Suppose that you want to multiply integers a and b together, each consisting of N limbs in basis W. The "schoolbook" method is quadratic: you compute the N^2 products $a_i b_j$, then add them together in the right slots. With Karatsuba multiplication, you split a and b into two halves:

$$\begin{aligned} a &= a_h W^{N/2} + a_l \\ b &= b_h W^{N/2} + b_l \end{aligned}$$

with $0 \leq a_h, a_l, b_h, b_l < W^{N/2}$. Then, remark that:

$$\begin{aligned} ab &= (a_h W^{N/2} + a_l)(b_h W^{N/2} + b_l) \\ &= a_h b_h W^N + (a_h b_l + a_l b_h) W^{N/2} + a_l b_l \\ &= a_h b_h W^N + ((a_h + a_l)(b_h + b_l) - a_h b_h - a_l b_l) W^{N/2} + a_l b_l \end{aligned}$$

In other words, the product ab of two integers of size N limbs can be computed with only *three* products of integers of size (about) $N/2$ limbs:

$$\begin{gathered} a_h b_h \\ a_l b_l \\ (a_h + a_l)(b_h + b_l) \end{gathered}$$

Applied recursively, this method yields a complexity of $O(N^{\log 3}) \approx O(n^{1.585})$, which is better than $O(N^2)$.

Implementing Karatsuba multiplication on integers has some slightly vexing characteristics, namely that $a_h + a_l$ and $b_h + b_l$ may overflow. In other words, you start out with computing a product of two 1024-bit integers, you split each into 512-bit halves, and you now have to compute three products, two on 512-bit integers, and one on 513-bit integers. At the next recursion depth, you have nine products: four on 256-bit integers, four on 257-bit integers, and one on 258-bit integers. These extra bits are irksome, especially when trying to make a generic multiplication routine not fixed to a specific integer size.

Modular multiplication can be handled with Montgomery multiplication, albeit with a larger basis. Let m be the (odd) modulus of size N limbs; one can (pre)compute $-m^{-1} \bmod W^N$. Then, the Montgomery product of a and b can be computed as:

$$D = ab + ((ab \bmod W^N)(-m^{-1} \bmod W^N) \bmod W^N)m$$

It can be seen that:

- $D = 0 \bmod W^N$
- $D = ab \bmod m$
- D can be computed from three products over integers of N limbs:
  - Non-modular product of a and b;
  - Product of the low half of ab (just computed above) with $-m^{-1} \bmod W^N$, truncated to the low N limbs;
  - Product of the value computed above with m.

  One of these three products is furthermore truncated, meaning that its upper half needs not be computed.

Thus, D can be divided *exactly* by W^N (this is a simple right-shift), and D/W^N is equal to ab/R modulo m (the value D/W^N may require one extra subtraction to be really reduced modulo m).

BearSSL Implementations

As will be described below, there are four big integer implementations in BearSSL, called “i15”, “i31”, “i32” and “i62”. The i62 code is the same code as i31 with just a single extra function (for modular exponentiations), but it uses the same types and representations; we will thus no longer speak of it except in the section related to modular exponentiations.

All operations are implemented for i15, i31 and i32, except for modular division, which is not implemented for i32. In the text below, i31 is used as main example; in some cases, it will be compared to i32, to highlight the reasons why i31 is considered superior to i32, the latter being mostly deprecated as a consequence. The i15 code is very similar to the i31 code, with everything scaled down; thus, it won’t be explained any further.

Implementation Requirements

As the rest of BearSSL, implementations of big integers strive for correctness, security, and economy of resources. The following notes apply:

- Big integers are *not* part of the public API. They exist only to support the cryptographic algorithms implemented by BearSSL. When big integers are to be exposed on the API (e.g. RSA key elements), they are shown as sequences of bytes. This means that the internal representation can be changed at will, and big integer functions may have somewhat complicated usage restrictions (e.g. interdiction of operand overlap). Also, there is no need for any support of negative integers.

- Elliptic curves typically work over finite fields which are fixed by some standard. RSA, on the other hand, will use varying moduli of many possible sizes. For better code compacity, implementations must be generic, and handle values of all sizes (up to some internal limit) without limitations such as “modulus size must be a multiple of 16 bits”.

- Code footprint must be as low as is practical. Since different algorithms may use different functions, all individual functions must be segregated into different source files, so that static linking pulls in only the required functions.

- Functions cannot perform dynamic memory allocation. Stack buffers may be used but only with strict bounds, so that total stack usage by BearSSL stays below about 3 kB. This will impose restrictions such as a maximum RSA key size (which should be at least 4096 bits).

- Everything must be constant-time. Notably, support of RSA requires implementation of modular reductions and exponentiations such that not only the values themselves, but also the moduli, are secret. It is acceptable to leak the *sizes* of the moduli (everybody knows that in a 2048-bit RSA key, the prime factors are usually of length 1024 bits each).

- Code that targets the ARM Cortex M0+ and M3 must use only 32→32 multiplications, since the opcode that yields the upper 32 bits of the 64-bit product is not constant-time on the M3, and does not exist at all on the M0+. Most other 32-bit architectures are perfectly happy with a 32→64 multiplications, and substantial performance benefits are expected from doing so. *Some* 64-bit architectures offer a 64→128 multiplication opcode, and performance of modular exponentiation is greatly enhanced when that opcode is used; unfortunately, there is no truly portable way to use it (it depends on both the architecture and the compiler brand). Thus, we'll need at least three distinct implementations, at least for some functions.

Internal Representation

Since there is no dynamic memory allocation in BearSSL, all operands must be caller-allocated.

Modular division and exponentiation require some extra buffers, which will be taken on the stack, and dynamically split; in particular, window-based optimizations of modular exponentiations call for an internal format which can be "manually allocated" in an array without any alignment issue. This basically means that a big integer must be represented as an array of a basic type (`uint16_t`, `uint32_t` or `uint64_t`), and be self-contained.

The design is then the following:

- A big integer is represented in basis $W = 2^{15}$ (for the "i15" implementation), 2^{31} (for "i31"), or 2^{32} (for "i32"). Implementation "i15" is for ARM Cortex M0+ and M3, and similar platforms that are best used with 32→32 multiplications. Implementation "i31" is the generic code for everything else. Implementation "i32" is an historical attempt, which is slower than "i31". There is also an "i62" implementation that is identical to "i31", except for an optimized modular exponentiation routine that leverages 64→128 multiplication opcodes.

- The integer is represented in an array of `uint16_t` integers (for i15) or `uint32_t` integers (for i31 and i32). If `x` points to the integer representation, then the least significant limb of the integer is `x[1]`; other limbs are in `x[2]`, `x[3]`… This is *little-endian encoding*. Take care that this notion of "little-endian" relates *only* to the ordering of limbs in the array, not to the ordering of bits or bytes in the in-memory representation of a word[6]. In the i15 and i31 implementations, limbs don't use the complete storage word: they leave the upper bit unused. We define the representation to *always* leave that extra bit to 0 (i.e. functions MUST ensure they don't put a 1 in the high bit of a value word, but they can assume that all input value words have a 0 as high bit).

- For an integer pointed to by `x`, the first word `x[0]` is the *header word*. It encodes the bit length of the integer. This, in particular, indicates how many words are significant in the array, so the integer is self-contained and there is no need for an extra length parameter.

The header word is used for two purposes:

- To allow easy retrieval of the size, in limbs, of the integer. This is needed for all loops that iterate over the limbs, e.g. in additions and multiplications.

- To allow easy retrieval of the size, in bits, of the integer. This is needed to compute encoding sizes, e.g. signature size for RSA.

For the i32 implementation, this is easily done: the header word just contains the bit length. To compute the number of value words, it suffices to divide the bit length by 32, rounded up; this is done with an expression such as `(x[0] + 31) >> 5` (right-shifting by 5 bits is equivalent to a truncating division by 32; the prior addition of 31 guarantees an up-rounding).

For the i31 and i15 implementations, this is not as easy. If the bit length is used as is, then the length in limbs requires computing a division by 31 or 15, which is cumbersome and inefficient. Instead, the header word is set to the **encoded bit length**. For a bit length k, the encoded bit length (for the i31 implementation) is $k' = k + \lfloor k/31 \rfloor$. This formula has the following consequences:

- The non-encoded bit length can be recomputed as: `x[0] - (x[0] >> 5)`
- The number of value words is: `(x[0] + 31) >> 5`
- The low-order five bits of the encoded bit length are equal to the bit length modulo 31: $k' \bmod 32 = k \bmod 31$.

For i15, the same mechanism is used, but using $k' = k + \lfloor k/15 \rfloor$.

Each integer has a **true bit length** and an **announced bit length**. This is independent of the encoding described above. The true bit length of a positive integer $x > 0$ is the unique integer $k \geq 1$ such that $2^{k-1} \leq x < 2^k$ (by convention, the true bit length of zero is zero). This is a mathematical concept that depends on the integer value. The announced bit length is the one which is encoded in the header word; it relates to the number of bits used to represent the integer in memory. The announced bit length cannot be lower than the true bit length (otherwise, the integer would be missing some parts), but it can be greater. The general rules are the following:

- All functions use the announced bit length to control loops and other memory access patterns. Therefore, the announced bit length cannot be kept secret from attackers who can observe such patterns.
- Most integers in cryptographic algorithms are modular integers. In BearSSL, most functions require that:
    - The announced bit length of the modulus m matches its true bit length.
    - For any integer x modulo m, the announced bit length of x matches that of m (i.e. the true bit length of m).

For instance, the integer 257871904, for the i15 implementation, and with an announced bit length of 35 bits, will be encoded as four `uint16_t` values:

```
0025 5020 1EBD 0000
```

which is analyzed as follows:

- Announced bit length is $35 = 2 \times 15 + 5$. It is then encoded as $2 \times 16 + 5$, which is 37 (0x25 in hexadecimal).
- Since the announced bit length is 35 bits, there will be three value words (15 bits per value word).
- $257871904 = 7869 \times 2^{15} + 20512$. Thus, the low value word is 0x5020 (that's 20512 in hexadecimal) while the next one is 0x1EBD (for 7869). The third value word is zero. Unused bits (top bit in each value word, and extra bits beyond the announced bit length in the top word) are set to zero.

There is a dark hack that lurks inside the code, though. In some situations, the true bit length of the integer is allowed to slightly extend beyond the announced bit length. We'll see this with additions and subtractions.

API Rules

The following rules are used for the API:

- All function names are prefixed with the implementation name. For instance, all i31 functions have a name that starts with `br_i31_`.
- Integers are passed around as a single pointer to the header word. A `const` qualifier is added for inputs that are not supposed to be modified by the function. The main point of using a `const` is to allow the use of hardcoded values which are *defined* with a `const`, which means that they end up in ROM instead of RAM. Note that even if a parameter is tagged with `const`, the function cannot assume that the value does not change, in particular if it overlaps with one of the output parameters.
- "Out" parameters (where data is written) usually come first in the list of parameters, using the same convention as the standard function `memcpy()` (which, itself, mimics the assignment operator, that goes right-to-left).
- "In" parameters (`const`-qualified) may overlap arbitrarily. In general, such parameters MUST NOT overlap with any of the output parameters. Output parameters must not overlap with each other. Some functions allow overlaps with some restrictions; this must be documented explicitly.
- Some function use in/out parameters, i.e. receive an already initialized integer whose value is updated. This is a preferred design, because it tends to minimize both the number of required (stack) buffers, and also because it allows passing fewer parameters to the function, which means a smaller code footprint at the call site.
- Some functions have an optional "control" parameter: when that value is zero, the function goes through the motions of doing the computation, but does not *in fine* modify the values in output parameters (it still performs all memory read and write accesses, but arranges for the updated value to be equal to its previous contents). That control parameter is called `ctl`, comes last in the list of parameters, has type `uint32_t`, and must have value 0 or 1. We do not use a boolean type (`_Bool` in C99) because we do *not* want the compiler to assume that it is a boolean, and begin to use non-constant-time constructions such as conditional jumps.

Additions and Subtractions

The i31 addition function (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_add.c;

h=2ca47c6b81fb36d698718fc1400ab4748dcf70c7;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27) is reproduced below:

```
uint32_t
br_i31_add(uint32_t *a, const uint32_t *b, uint32_t ctl)
{
        uint32_t cc;
        size_t u, m;

        cc = 0;
        m = (a[0] + 63) >> 5;
        for (u = 1; u < m; u ++) {
                uint32_t aw, bw, naw;

                aw = a[u];
                bw = b[u];
                naw = aw + bw + cc;
                cc = naw >> 31;
                a[u] = MUX(ctl, naw & (uint32_t)0x7FFFFFFF, aw);
        }
        return cc;
}
```

Let’s see in details how it works:

- The function takes as parameters an in/out value (a), another input parameter (b), and a control word (`ctl`). It is meant to add value `b` to value `a`; returned value is the extra carry if the top word overflows. If `ctl` is zero, all memory accesses are still performed, and the carry computed and returned, but the value in a is not updated (or, rather, it is replaced with a copy of itself). Values a and b MUST have the same announced bit length, and the function simply assumes it. The API is documented in `inner.h` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/inner.h;h=8c7f04e1682b66d3832815fd58a6596798091f69;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l1297).

- The `m` value is the number of value words, plus one (the “plus one” is done because value words start at index 1, not 0, in the array; to get this “plus one” operation, 63 is used instead of 31 in the computation of `m`). Note that since a and `b` are assumed to have the same announced bit length, the computation of `m` works for both. The onus is on the caller to ensure that this property holds; this allows `br_i31_add()` to avoid any test, and thus compiles to very compact code.

- For every index, we take the two words from a and `b`, perform the addition (including the running carry `cc`), and then write back the result. The intermediate result (naw) is a 32-bit integer (since the carry is either 0 or 1, and the two value words are less than 2^{31} each, the intermediate sum is less than 2^{32} and therefore fits in an `uint32_t`). The top bit is the new carry, which we extract with a shift.

- The expression that writes back the new value word uses the `MUX()` function (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/inner.h;h=8c7f04e1682b66d3832815fd58a6596798091f69;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l742), which is a constant-time primitive defined in `inner.h`. If `ctl` is 1, then it returns its second operand (here the newly computed limb, naw, excluding the top bit, which is the carry into the next index). If `ctl` is 0, then `MUX()` will simply return its third operand, which is the old value of the limb of a at that index. That way, regardless of the value of `ctl`, the same sequence of memory reads and writes will be performed, except not with the same values.

It is instructive to compare that code with the one for i32 (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i32_add.c;h=620baffd7a3ddc83285bd94198df2831d0473fd3;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l28):

```
uint32_t
br_i32_add(uint32_t *a, const uint32_t *b, uint32_t ctl)
{
        uint32_t cc;
        size_t u, m;

        cc = 0;
        m = (a[0] + 63) >> 5;
        for (u = 1; u < m; u ++) {
                uint32_t aw, bw, naw;

                aw = a[u];
                bw = b[u];
                naw = aw + bw + cc;

                /*
                 * Carry is 1 if naw < aw. Carry is also 1 if naw == aw
                 * AND the carry was already 1.
                 */
                cc = (cc & EQ(naw, aw)) | LT(naw, aw);
                a[u] = MUX(ctl, naw, aw);
        }
        return cc;
}
```

This one is very similar, except for the computations of the carry and the next word. Since, in i32, limbs are exactly 32 bits, they are naturally truncated to 32 bits when being written in an `uint32_t` variable, so there is no need to mask the “carry bit”. On the other hand, the carry bit is *not* in the result word, and must be recovered in some other way. Here, a comparison is used: if the result of the addition is numerically lower than one of the operands, then this means that it “wrapped around”, i.e. that there was a carry. Unfortunately, this is complicated for two reasons:

- There is a special case, depending on the incoming carry. If the result (naw) is equal to the first operand (aw), then this could be either because bw+cc was 0, or because bw+cc was 2^{32}. Both are possible, but not at the same time; a 0 is possible only if `cc` is 0, while 2^{32} is possible only if `cc` is 1.

- We do not want to use the normal C operators for comparisons, because the C compiler knows that comparison operators return booleans (0 or 1), and may decide to “optimize” things with a conditional jump, which would break constant-time discipline. This depends a lot on the compiler brand, target architecture, and compilation flags. In order to get reasonably reliable constant-time behaviour on a wide variety of platforms, we must use comparison functions that get the result arithmetically (the `EQ()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/inner.h;h=8c7f04e1682b66d3832815fd58a6596798091f69;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l751) and `LT()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/inner.h;h=8c7f04e1682b66d3832815fd58a6596798091f69;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l775) functions).

In practice, this makes `br_i32_add()` larger and slower than `br_i31_add()`, even though the i31 code needs a bit more space for integers (a 1024-bit integer uses 32 limbs with i32, but 34 with i31).

The i31 subtraction function (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_sub.c;h=391089518d750978f18db1aa69ed69f541132694;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27) is very similar:

```
uint32_t
br_i31_sub(uint32_t *a, const uint32_t *b, uint32_t ctl)
{
        uint32_t cc;
        size_t u, m;

        cc = 0;
        m = (a[0] + 63) >> 5;
        for (u = 1; u < m; u ++) {
                uint32_t aw, bw, naw;

                aw = a[u];
                bw = b[u];
                naw = aw - bw - cc;
                cc = naw >> 31;
                a[u] = MUX(ctl, naw & 0x7FFFFFFF, aw);
        }
        return cc;
}
```

We recognize the same steps: computation of integer length (in limbs), limb-by-limb subtraction, extraction of the carry. In this function, the carry is still 0 or 1 (mathematically, you could also define things with a carry of value 0 or -1, but it is more convenient to use a nonnegative carry definition here).

The "dark hack" is in the details. In both `br_i31_add()` and `br_i31_sub()`, if the result does not fit in the announced bit length, then the top word may end up being out of range. For instance, suppose that we are adding two integers of announced bit length 9 bits. If the source values are, for instance, 400 and 300, then the sum is 700, and 700 does *not* fit over 9 bits. In that case, the top word will use 10 bits, which is in violation of our defined internal format. However, it so happens that if `br_i31_add()` and `br_i31_sub()` may produce results that use extra bits in the top word, they also tolerate quite well inputs that similarly violate the internal format. Thus, the usage pattern must be one of the two following:

- The result is known to fit in the announced bit length.
- The oversized result is immediately fixed with another operation.

In practice, you'll see code like this one (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/ec/ec_c25519_i31.c;h=aa88dd610db9a76d2d8d3cf51de0485361621442;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l136):

```
        ctl = br_i31_add(t, b, 1);
        ctl |= NOT(br_i31_sub(t, C255_P, 0));
        br_i31_sub(t, C255_P, ctl);
```

This is a modular addition of `b` to `t`; the modulus is a constant value held in array `C255_P` (this is part of an implementation of Curve25519). First the addition is performed; this may produce an extra carry (overflow, and the value does not even fit in the allocated limbs) or a value which may or may not exceed the modulus. On the second line, a subtraction is done with the modulus, but with a control bit set to 0: this means that it really is a comparison, which will return 1 if and only if `C255_P` happens to be greater than `t` at this point. Crucially, if `t` (after the addition) violates the format with an extra bit in the top word, this still works, because `br_i31_sub()` accepts such invalid formats on input. The third line performs the subtraction of the modulus `C255_P` from `t` conditionally on that bit `ctl`.

Note that the condition on doing the modulus subtraction really is: the intermediate result is greater than or equal to the modulus. This may manifest as either a zero result from the comparison, or an extra carry from the addition[7].

What justifies that kind of temporary violation of inner formats is that it is, indeed, an inner format: the big integer functions are not part of the public API, and thus can have relatively complex usage conditions, that the callers are assumed to know perfectly, since they are part of the same library. If the big integer code were to be made public, the `br_i31_add()` and `br_i31_sub()` functions would have to carefully truncate the top word of the result, and return an oversized result as an output non-zero carry, not as an extra bit in the top word.

Multiplications

The i31 Montgomery multiplication (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_montmul.c;h=80668086ff813e5a88ecf38aefab61c14bb95743;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27) has an inner loop that looks like this:

```
        for (u = 0; u < len; u ++) {
                uint32_t f, xu;
                uint64_t r, zh;

                xu = x[u + 1];
                f = MUL31_lo((d[1] + MUL31_lo(x[u + 1], y[1])), m0i);

                r = 0;
                for (v = 0; v < len4; v += 4) {
                        uint64_t z;

                        z = (uint64_t)d[v + 1] + MUL31(xu, y[v + 1])
                                + MUL31(f, m[v + 1]) + r;
                        r = z >> 31;
                        d[v + 0] = (uint32_t)z & 0x7FFFFFFF;
                        z = (uint64_t)d[v + 2] + MUL31(xu, y[v + 2])
                                + MUL31(f, m[v + 2]) + r;
                        r = z >> 31;
                        d[v + 1] = (uint32_t)z & 0x7FFFFFFF;
                        z = (uint64_t)d[v + 3] + MUL31(xu, y[v + 3])
                                + MUL31(f, m[v + 3]) + r;
                        r = z >> 31;
                        d[v + 2] = (uint32_t)z & 0x7FFFFFFF;
                        z = (uint64_t)d[v + 4] + MUL31(xu, y[v + 4])
                                + MUL31(f, m[v + 4]) + r;
                        r = z >> 31;
                        d[v + 3] = (uint32_t)z & 0x7FFFFFFF;
                }
                /* ... */
```

This code follows the Montgomery multiplication algorithm, but with some unrolling of the inner loop: four iterations are made successively. It is a trade-off: this slightly increases the code size, but also offers some performance improvement. Let’s concentrate on a single instance:

```
        z = (uint64_t)d[v + 1] + MUL31(xu, y[v + 1])
                + MUL31(f, m[v + 1]) + r;
        r = z >> 31;
        d[v + 0] = (uint32_t)z & 0x7FFFFFFF;
```

The `MUL31()` is a macro that computes a multiplication of two values less than 2^{31}, with a result of type `uint64_t`. BearSSL includes several versions (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/inner.h;h=8c7f04e1682b66d3832815fd58a6596798091f69;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l929) of this macro, in order to try to get constant-time multiplications even on uncooperative architectures (ctmul.html). Unless explicitly configured otherwise, `MUL31()` evaluates as a simple product opcode. The C compiler sees that the operands fit on 32 bits, and will use the 32→64 multiplication opcode of the CPU.

We see here that the sum of the previous result word (`d[v + 1]`), the two products, and the carry, still fits in the 64-bit temporary variable `z`. This is a consequence of using 31-bit limbs: since operand limbs are less than $W = 2^{31}$, products of such values are less than 2^{62}, and we can thus accumulate several such products in a single 64-bit slot. In the i32 implementation (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i32_montmul.c;h=7edb376cddd7bba9a093b8b8e6f1cb3f82c395e0;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27), things are less easy:

```
        z = (uint64_t)d[v + 1] + MUL(xu, y[v + 1]) + r1;
        r1 = z >> 32;
        t = (uint32_t)z;
        z = (uint64_t)t + MUL(f, m[v + 1]) + r2;
        r2 = z >> 32;
        if (v != 0) {
                d[v] = (uint32_t)z;
        }
```

With full-width limbs (32 bits), individual products may range up to $2^{64} - 2^{33} + 1$, which is very close to 2^{64}, and the intermediate sum would need 65 bits. This forces the code to split the expression into two parts, with *two* carry words (`r1` and `r2`). This is the main reason why i31 code is faster than i32.

Modular Reduction

BearSSL defines an internal function called `br_i31_muladd_small()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_muladd.c;h=eecd9e2c6c53230c1e0553844d6bb159c2149d3b;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27) which is used for several operations, in particular generic modular reduction. This function takes three parameters: a big integer `x`, a small integer `z`, and a modulus `m`. The `x` value MUST be smaller than `m` before the call. What this function does is that it multiplies `x` by the basis W (2^{31} for the i31 code), then adds `z`, and finally reduces the result modulo `m`.

Mathematically, multiplying by W and adding z, with $z < W$, is equivalent to shifting all limbs by one slot, and setting the newly added least-significant limb to z. If $x < m$, then $xW + z < mW$; thus, the modular reduction is only a matter of a single Euclidean division by m, that necessarily yields a quotient lower than W. This is done with the approximation method explained in a previous section. Since the approximation uses an elementrary Euclidean division, this requires a constant-time implementation of that step, which is provided as the `br_divrem()` function (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i32_div32.c;h=d8b8023d842d380ffd52b0c69d157013ca3ce26a;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27):

```
uint32_t
br_divrem(uint32_t hi, uint32_t lo, uint32_t d, uint32_t *r)
{
        /* TODO: optimize this */
        uint32_t q;
        uint32_t ch, cf;
        int k;

        q = 0;
        ch = EQ(hi, d);
        hi = MUX(ch, 0, hi);
        for (k = 31; k > 0; k --) {
                int j;
                uint32_t w, ctl, hi2, lo2;

                j = 32 - k;
                w = (hi << j) | (lo >> k);
                ctl = GE(w, d) | (hi >> k);
                hi2 = (w - d) >> j;
                lo2 = lo - (d << k);
                hi = MUX(ctl, hi2, hi);
                lo = MUX(ctl, lo2, lo);
                q |= ctl << k;
        }
        cf = GE(lo, d) | hi;
        q |= cf;
        *r = MUX(cf, lo - d, lo);
        return q;
}
```

This function uses the “schoolbook” method of comparing the dividend with the suitably scaled divisor, and performing a subtraction when possible; this works in binary, so every quotient digit is either 0 or 1. Dividend is a 64-bit word (two 32-bit limbs `hi` and `lo`), while divisor is `d`. In the inner loop:

- `k` is the current “scaling” for the quotient.
- `w` is set to the top dividend word, with the current scaling (there may be an extra top bit at that point; it will be found in `hi >> k`).
- `ctl` is set to 1 if the scaled divisor must be subtracted from the dividend; this happens when `w` is greater than or equal to `d`, or when the extra top bit is not zero. In other words, `ctl` is the next quotient bit.
- The subtraction of the scaled divisor from the dividend is computed; it conditionally replaces the current dividend (with

the `MUX()` calls) if the subtraction must indeed occur, i.e. when the quotient bit is 1.

- The last iteration (when scaling is zero) is specialized in the final code.

Division is known to be tricky: if the divisor is zero, or if the quotient cannot fit on 32 bits, then there is no well-defined result that is returned. The `br_divrem()` function will not throw an exception (no `SIGFPE`) but what it returns in such cases is mathematically unsound, and the callers should ensure that this case does not happen (or work around it).

With the `br_i31_muladd_small()` function, the following are implemented:

- Generic modular reduction: `br_i31_reduce()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_reduce.c;h=5c9523ed152f1023b49f3a851d029c99c165c9fe;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27). The value to reduce is injected limb by limb, starting with the most significant; each injection means shifting the already present limbs, and adding the new one. This is exactly what `br_i31_muladd_small()` computes.

- Generic decoding from bytes combined with modular reduction: `br_i31_decode_reduce()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_decred.c;h=43db6624c8e4d57749c32177aed899f196e21443;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27).

- Conversion to Montgomery representation: `br_i31_to_monty()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_tmont.c;h=4798ff65d7c961c2ffb1adf7a9511d2f6a215939;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27). Since converting to Montgomery representation means multiplying by W^N, it suffices to perform N multiplications by W; each is a call to `br_i31_muladd_small()` with an extra `z` parameter set to zero.

We may note that `br_i31_muladd_small()` is relatively expensive: while it has nominally linear cost ($O(N)$), it has a large overhead, due notably to the `br_divrem()` invocation. Another method for modular reduction uses only Montgomery multiplication, assuming that a given constant is precomputed:

- Let $S = W^{2N-1} \bmod m$. This is the Montgomery representation of W^{N-1}.
- Start with x set to the top $N-1$ input limbs (we assume here that $m \geq W^{N-1}$). Then, for each next chunk of $N-1$ limbs to inject, multiply x with W^{N-1} (i.e., perform a Montgomery multiplication with S), then add the next limbs; the addition of the next limbs is made modular with a conditional subtraction of the modulus.

Such a method is more efficient, but requires some extra buffers if the modulus is known only dynamically, which is why it is not implemented in BearSSL. Also, in practice (when implementing RSA), modular reduction cost is negligible with regards to the modular exponentiation that follows.

Modular Exponentiation

The `br_i31_modpow()` function (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_modpow.c;h=4ef3f5d5ac871a3086a0b76f9b25540de58765b1;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27) implements a simple square-and-multiply algorithm. It follows the variant where exponent bits are scanned from right (least significant) to left (most significant). Some salient points are the following:

- The exponent is expected as bytes with big-endian encoding. This maps to the practical situation of using RSA: the private exponents are provided encoded, and need not be converted to the i31 internal representation.

- If the source value is 0, the returned value will be 1, which is mathematically wrong, but not a problem in practical situations, where either 0 does not happen (e.g. the input when encrypting data with RSA, or signing, *cannot* be zero after padding), or where neither a 0 or 1 output is valid (e.g. when decrypting, or when verifying a signature, neither 0 or 1 will have a correct padding, so no harm is done is returning the wrong invalid value). There again, we exploit the fact that the big integer functions are *internal*, and not exposed as a public API.

- Some extra buffers are needed (to store two integers). Since the code is generic, it cannot make assumptions on the size of the operands; thus, the extra buffers must be provided by the caller. The need for extra buffers comes from the fact that the Montgomery multiplication function (`br_i31_montymul()`) does not support in-place modifications (the output operand MUST be disjoint from the input operands).

- The code is constant-time. For each exponent bit, two multiplications are performed (one is actually a squaring); a conditional copy (`CCOPY()` function) is used to keep or discard the result of the multiplication of `x` by `t1`. Moreover, the true exponent length is hidden: all provided bits will be used, even if the top bits are zeros.

- Mixed Montgomery multiplication is used. This uses the fact that the Montgomery product of xR (which is x in

Montgomery representation) with y yields xy (i.e. the product, but *not* in Montgomery representation). Thus, an initial conversion *to* Montgomery representation is needed (`br_i31_to_monty()`) but no conversion *from* Montgomery representation is required.

There is a corresponding `br_i32_modpow()` function (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i32_modpow.c;h=034aba06db1d06e077a1587784c1168cd7dfc19c;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27) which is identical, except that it uses the i32 primitives.

Window optimzations are implemented in an extra function called `br_i31_modpow_opt()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_modpow2.c;h=0b8f8cf7ea70d2b3b0770c555f39be134b442525;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l27). There is no i32 counterpart for this function (if you want more speed, you should first switch to i31 anyway). It is worth detailing the implementation:

```
uint32_t
br_i31_modpow_opt(uint32_t *x,
        const unsigned char *e, size_t elen,
        const uint32_t *m, uint32_t m0i, uint32_t *tmp, size_t twlen)
{
        size_t mlen, mwlen;
        uint32_t *t1, *t2, *base;
        size_t u, v;
        uint32_t acc;
        int acc_len, win_len;

        /*
         * Get modulus size.
         */
        mwlen = (m[0] + 63) >> 5;
        mlen = mwlen * sizeof m[0];
        mwlen += (mwlen & 1);
        t1 = tmp;
        t2 = tmp + mwlen;
```

The API raises x to the power e modulo `m`. As in `br_i31_modpow()`, the exponent is provided in encoded big-endian representation. The `m0i` parameter is the usual $-m^{-1} \bmod W$ value for Montgomery multiplication (arguably, this function could recompute it, the cost would be negligible in practice). Some extra space must be provided by the caller, as an array of words. Contrary to `br_i31_modpow()`, this space needs not be pre-split; instead, the function will ensure splitting.

Note that the function returns a value. It will report an error if not enough extra temporary space is provided, given the operand sizes. This test occurs immediately afterwards:

```
        /*
         * Compute possible window size, with a maximum of 5 bits.
         * When the window has size 1 bit, we use a specific code
         * that requires only two temporaries. Otherwise, for a
         * window of k bits, we need 2^k+1 temporaries.
         */
        if (twlen < (mwlen << 1)) {
                return 0;
        }
        for (win_len = 5; win_len > 1; win_len --) {
                if ((((uint32_t)1 << win_len) + 1) * mwlen <= twlen) {
                        break;
                }
        }
```

It is allowed to report an error “early”: this does not break constant-time discipline, because that test is on the announced lengths of the operands, which are non-secret.

This code snippet will compute the window width (`win_len`). The largest possible window width is used, given the size of the temporaries, with a maximum at 5 bits (this is heuristically an appropriate maximum; larger windows yield only negligible speed-ups, and table construction time tends to cancel these speed-ups for large windows). Window width depends only on announced operand lengths, and thus is not secret either.

```
	/*
	 * Everything is done in Montgomery representation.
	 */
	br_i31_to_monty(x, m);

	/*
	 * Compute window contents. If the window has size one bit only,
	 * then t2 is set to x; otherwise, t2[0] is left untouched, and
	 * t2[k] is set to x^k (for k >= 1).
	 */
	if (win_len == 1) {
		memcpy(t2, x, mlen);
	} else {
		memcpy(t2 + mwlen, x, mlen);
		base = t2 + mwlen;
		for (u = 2; u < ((unsigned)1 << win_len); u ++) {
			br_i31_montymul(base + mwlen, base, x, m, m0i);
			base += mwlen;
		}
	}
```

We compute the table, depending on the window size. When the window is down to 1 bit, there is not much to compute; that case is kept as a separate code path because it makes it use no more space than `br_i31_modpow()`; this makes `br_i31_modpow_opt()` a complete substitute.

The first slot of the table (for $x^0 = 1$) is not modified; it will be used as a temporary buffer later on.

```
	/*
	 * We need to set x to 1, in Montgomery representation. This can
	 * be done efficiently by setting the high word to 1, then doing
	 * one word-sized shift.
	 */
	br_i31_zero(x, m[0]);
	x[(m[0] + 31) >> 5] = 1;
	br_i31_muladd_small(x, 0, m);
```

Result variable is initialized to 1, but we need it in Montgomery representation, because everything is in Montgomery representation here. The `br_i31_to_monty()` function is somewhat expensive, but we can optimize things here: we first set the top limb to 1 (this sets the value to W^{N-1}, for N limbs), then multiply it by W modulo m with `br_i31_muladd_small()`.

```
	/*
	 * We process bits from most to least significant. At each
	 * loop iteration, we have acc_len bits in acc.
	 */
	acc = 0;
	acc_len = 0;
	while (acc_len > 0 || elen > 0) {
		int i, k;
		uint32_t bits;

		/*
		 * Get the next bits.
		 */
		k = win_len;
		if (acc_len < win_len) {
			if (elen > 0) {
				acc = (acc << 8) | *e ++;
				elen --;
				acc_len += 8;
			} else {
				k = acc_len;
			}
		}
		bits = (acc >> (acc_len - k)) & (((uint32_t)1 << k) - 1);
		acc_len -= k;
```

We use the square-and-multiply variant that processes exponent bits from left (most significant) to right (least significant); this is the variant that can be optimized with windows. The code above gets the next `win_len` bits. This may entail reading one extra exponent byte, but no more than one, since the window width was limited to at most 5 bits. The bits which were read from the exponent, but not used for this iteration, are stored in `acc` (with `acc_len` stored bits).

Also, when reaching the end of the exponent, we may have fewer than `win_len` bits. Indeed, the window width is not necessarily a strict divisor of the exponent length; for instance, if the window width is 3 bits, and the exponent length is 1024 bits (128 bytes), then the last iteration will gather only one bit, not three. The code above gathers the exponent bits in `bits`, and the number of said bits in `k`.

```
		/*
		 * We could get exactly k bits. Compute k squarings.
		 */
		for (i = 0; i < k; i ++) {
			br_i31_montymul(t1, x, x, m, m0i);
			memcpy(x, t1, mlen);
		}
```

These are the squarings. We use the generic integer multiplication function.

```
                /*
                 * Window lookup: we want to set t2 to the window
                 * lookup value, assuming the bits are non-zero. If
                 * the window length is 1 bit only, then t2 is
                 * already set; otherwise, we do a constant-time lookup.
                 */
                if (win_len > 1) {
                        br_i31_zero(t2, m[0]);
                        base = t2 + mwlen;
                        for (u = 1; u < ((uint32_t)1 << k); u ++) {
                                uint32_t mask;

                                mask = -EQ(u, bits);
                                for (v = 1; v < mwlen; v ++) {
                                        t2[v] |= mask & base[v];
                                }
                                base += mwlen;
                        }
                }
```

The lookup in the table is constant-time. To perform it, we first store an all-zero value in `t2` (it's a temporary buffer with the size of `m`), then combine it (with bitwise OR) with all the table values, with proper masks. The `mask` value will be an all-zero word for all outer loop iterations, except the one where `u` equals the value of the exponent bits (`bits`), in which case the `mask` will be an all-one word.

The net result is that `t2[0]` is set to the value `t2[bits]`. Note that if the exponent bits are all zeros, then `t2` contains 0, which is wrong (mathematically, it should contain 1); this will be fixed afterwards.

```
                /*
                 * Multiply with the looked-up value. We keep the
                 * product only if the exponent bits are not all-zero.
                 */
                br_i31_montymul(t1, x, t2, m, m0i);
                CCOPY(NEQ(bits, 0), x, t1, mlen);
        }
```

Here, we compute the multiplication with the value which was extracted from the table. `br_i31_montymul()` requires non-overlapping operands, so the result is not written in `x`, but in temporary buffer `t1`. A final copy is needed to put it back in `x`. This is the point where we fix the case of `bits` being equal to zero: in that case, we don't actually want to overwrite `x`. `CCOPY()` performs that conditional copy in a constant-time way.

(Alternatively, I could have fixed `t2` *before* the multiplication, with something like: `t2[1] |= EQ(bits, 0);`. Then, the `CCOPY()` could have been transformed into a `memcpy()`.)

```
        /*
         * Convert back from Montgomery representation, and exit.
         */
        br_i31_from_monty(x, m, m0i);
        return 1;
}
```

Finally, we need to convert back the result from Montgomery representation, since everything was computed in Montgomery representation.

Note that the *only* error condition (a returned value of zero) is about the size of the temporary area with regards to the size of the modulus; thus, the returned value is not secret.

On 64-bit architectures, there is a `br_i62_modpow_opt()` function (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob; f=src/int/i62_modpow2.c;h=2db537f0a80a3ece1088879b2174726bfbe4c4be; hb=8ef7680081c61b486622f2d983c0d3d21e83caad) which leverages 64→128 multiplication opcodes. The following notes apply:

- There is no standard way to get a 64→128 multiplication. We must use a language extension, which will depend on the compiler brand and version. On GCC and Clang, there is an `unsigned __int128` type. On 64-bit x86 with Visual Studio, the extension involves some “intrinsics”, specifically the `_umul128()` and `_addcarry_u64()` functions, that the compiler handles specially. All of this is activated only when some macro-based detection code (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/inner.h;h=8c7f04e1682b66d3832815fd58a6596798091f69;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l376) says that the feature is available. In case of failure[8], this can be deactivated explicitly by setting `BR_INT128` and `BR_UMUL128` explicitly to `0` in `config.h` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/config.h;h=8ea4d8af8d61beb5a02305dccc9c7f851fb5a00a;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l196).

- When there is no detected support for 64→128 multiplications, or when that support was explicitly disabled, `br_i62_modpow_opt()` is still defined, with an alternate implementation (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i62_modpow2.c;h=2db537f0a80a3ece1088879b2174726bfbe4c4be;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l464) which is a simple wrapper around `br_i31_modpow_opt()`.
- 64-bit architectures tend to be “large”, i.e. to not have strong constraints on RAM and ROM size. Thus, `br_i62_modpow_opt()` may use larger code. In practice, the `i62_modpow2.c` code contains some specializations of the `br_i31_montymul()` and `br_i31_from_monty()` functions, modified to leverage the 64→128 multiplications.

The implementation of `br_i62_modpow_opt()` is otherwise similar to that of `br_i31_modpow_opt()`. The main structural difference is that operands are provided in i31 format, and must at some point be converted to i62 format (limbs modulo $W = 2^{62}$, held in 64-bit slots). This means using a few extra buffers in the provided temporaries. Therefore, there are situations in which `br_i62_modpow_opt()` does not have enough space to work, but `br_i31_modpow_opt()` would be fine; in these case, `br_i31_modpow_opt()` will be automatically called. A consequence is that any code referencing `br_i62_modpow_opt()` will also pull in `br_i31_modpow_opt()` at link time; as explained above, this is considered not much of a problem, under the assumption that 64-bit architectures tolerate somewhat larger code footprints.

It shall be noted that switching between arrays of `uint32_t` and arrays of `uint64_t` can lead to alignment issues[9]. In strictly standard C, in general, pointers to integer types of different sizes must be not carelessly cast. However, the standard rules on `uint32_t` and `uint64_t` imply that when these two types exist, the size (in bytes) of `uint64_t` will be exactly twice the size of `uint32_t`, from which it follows that a pointer to an array of `uint64_t` *can* be converted to a pointer to an array of `uint32_t`. Take care that we are only talking about alignment here! In no way are we allowed to *write* a value in a `uint64_t` slot, and try to read it back as `uint32_t` words; we are just arguing about reusing the same memory area to store temporary values.

Thanks to this alignment feature, we can automatically invoke `br_i31_modpow_opt()`, that needs an array of `uint32_t` for temporary storage, with a (suitably cast) pointer to the received array of `uint64_t`. There is a dual case, incarnated by `br_i62_modpow_opt_as_i31()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i62_modpow2.c;h=2db537f0a80a3ece1088879b2174726bfbe4c4be;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l481): this is a simply wrapper function that invokes `br_i62_modpow_opt()` but has an API which is identical to that of `br_i31_modpow_opt()`. This implies a cast *from* `uint32_t*` *to* `uint64_t*`, which can run afoul of alignment rules. This wrapper is used in one case: the `br_rsa_i31_keygen_inner()` function (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/rsa/rsa_i31_keygen_inner.c;h=9ec881b5f945224a43b10be51106dbe73e40d358;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l456), which performs RSA key pair generation, expects as parameter a pointer to a modular exponentiation function that can work on i31 values. Using `br_i62_modpow_opt_as_i31()` allows leveraging the faster `br_i62_modpow_opt()` without duplicating the calling code. However, this also implies an extra requirement on the caller, namely that the array of temporaries, provided as an `uint32_t*` pointer, is also suitably aligned if cast to `uint64_t*`. To that effect, `br_rsa_i31_keygen_inner()` allocates these temporaries with a `union`:

```
        union {
                uint32_t t32[TEMPS];
                uint64_t t64[TEMPS >> 1];  /* for 64-bit alignment */
        } tmp;
```

This is arguably quite ugly. This kind of hack is tolerable because, there again, this is about using an *internal* API. For a public API, arcane rules on data alignment are too complicated and must be shunned, even if documented at length.

Modular Division

Modular division is implemented by `br_i31_moddiv()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_moddiv.c;h=995059119767441 9d97f545a224a9503eda4a955;hb=8ef7680081c61b486622f2d983c0d3d21e83caad). This is a complicated code that implements the binary GCD algorithm explained in a previous section. It furthermore includes the optimization of trying to group modifications to values using multiplications.

The code is heavily commented, to highlight the specific contorsions that must be done at some points. A notable implementation point is the following: nominally, in C, signed integer types MUST NOT be allowed to go “out of range”, because that would trigger “undefined behavior”, which can have all sorts of consequences, including crashing the computer or corrupting the memory contents. This contrasts with unsigned integer types, that are specified to implement arithmetics modulo a power of two, with clean, defined “wrap-around” semantics. Therefore, it is always valid to convert any integer value to an unsigned type (this will perform the modulo), but the reverse is dangerous. Moreover, the right-shift of a negative value has “implementation-defined results”; it won’t crash anything, but the resulting value is not specified by the standard, and each compiler/architecture is free to return whatever it wants (provided that it is documented somewhere).

In the modular division code, this happens in a couple of places, namely in the internal functions `co_reduce()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_moddiv.c;h=9950591197674419d97f545a224a9503eda4a955;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l111) and `co_reduce_mod()` (https://www.bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/int/i31_moddiv.c;h=9950591197674419d97f545a224a9503eda4a955;hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l187). Consider the following excerpt from `co_reduce()`:

```
                /*
                 * For the new values of cca and ccb, we need a signed
                 * right-shift; since, in C, right-shifting a signed
                 * negative value is implementation-defined, we use a
                 * custom portable sign extension expression.
                 */
#define M   ((uint64_t)1 << 32)
                tta = za >> 31;
                ttb = zb >> 31;
                tta = (tta ^ M) - M;
                ttb = (ttb ^ M) - M;
                cca = *(int64_t *)&tta;
                ccb = *(int64_t *)&ttb;
#undef M
```

At that point in the code, we have two values that are, nominally, signed integers (they can be negative), but held in *unsigned* variables `za` and `zb` (of type `uint64_t`). We want to perform an *arithmetic right-shift* of these values (i.e. the holes on the left shall be filled with copies of the “sign bit”).This is the code that I would have wanted to use:

```
                cca = (int64_t)za >> 31;
                ccb = (int64_t)zb >> 31;
```

This code would fail on two counts:

- The conversion from `uint64_t` to `int64_t` triggers undefined behaviour if the value of `za` is outside of the range allowed for an `int64_t` (which would happen whenever the mathematical value I wanted to store in `za` was negative).
- If the conversion result is negative, then right-shifting it is implementation-defined. Most compilers do the right thing, i.e. an arithmetic right shift, but, for portability, I prefer not to rely on it.

Thus, I must first perform an arithmetic right shift “by hand” on the unsigned types. These two lines do a logical right shift; the upper 31 bits of `tta` and `ttb` will be set to zero:

```
                tta = za >> 31;
                ttb = zb >> 31;
```

Then the following lines “extend” the former top bit. If that bit was a 0, then the XOR with the constant `M` flips that bit to a 1, and the subtraction flips it back to 0. On the other hand, if the bit was a 1, the XOR makes it a 0, and the subtraction then tries to turn it into -1, which implies carry propagation, and all upper bits are set to 1.

```
                tta = (tta ^ M) - M;
                ttb = (ttb ^ M) - M;
```

Finally, the conversion to `int64_t` is done in a safe way:

```
                cca = *(int64_t *)&tta;
                ccb = *(int64_t *)&ttb;
```

The reason why it is safe is convoluted. In general, in-memory representation of integer types is left to the compiler, and it can have some unnamed padding bits, and “trap representations” that trigger undefined behavior when accessed. However, for the “exact width” integer types like `uint32_t` or `int64_t` (and unlike the usual types such as `unsigned int` and `long long`), there are some constraints:

- Exact-width integer types do not have padding bits or trap representations.
- Signed exact-width integer types use two’s complement encoding for negative values.
- The signed and unsigned variants of an exact-width type store value bits in compatible formats, i.e. the bit of value 2^k in an `int64_t` is at the same position as the bit of value 2^k in an `uint64_t`.
- The signed and unsigned variants of an exact-width type have identical alignment requirements.

Taken together, this means that I can store an `uint64_t` in memory, then read it back (with a pointer cast) as an `int64_t`, and it MUST make the conversion that I expect without triggering any undefined behavior. This is what the code above does. Let me stress again that this relies on requirements imposed by the C standard on exact-width types which do not necessarily apply to other integer types; i.e. don’t try that to convert an `unsigned int` into an `int`.

Fortunately, optimizing C compilers are smart enough to transform the whole write-to-memory-then-read into nothingness, i.e. to just keep the values in their respective registers.

---

1. In the line of languages that includes Basic and is descendants such as Microsoft Visual Basic, the prefix is `&H`.↩
2. Historically, some computers and compilers have used other representations, such as one’s complement or sign+mantissa. In one’s complement, a value is negated by inverting all bits, but without adding 1 afterwards. In sign+mantissa, negation is obtained by inverting the top bit only. These alternate conventions made some operations somewhat simpler to implement in hardware at that time, but also increased complexity of others, and are generally thought not to be worth the effort for integers; they also induce some extra headaches, such as the presence of *two* zeros, a positive and a negative one. Note that while two’s complement is now universally used for computers in hardware less than three decades old, the sign+mantissa convention is in widespread usage for floating-point numbers.↩
3. In any case, division opcodes are usually not constant-time, so you’ll want to use a “manual” implementation that performs the division bit by bit in a constant-time way.↩
4. Except at some point in RSA key pair generation, where we must invert the public exponent e modulo both $p-1$ and $q-1$, which are even. For that operation, BearSSL must employ additional tricks (https://www.bearssl.org/gitweb /?p=BearSSL;a=blob;f=src/rsa/rsa_i31_keygen_inner.c;h=9ec881b5f945224a43b10be51106dbe73e40d358; hb=8ef7680081c61b486622f2d983c0d3d21e83caad#l391).↩
5. Fermat’s Little Theorem can be expressed algebraically as: when m is prime, the group of invertible elements modulo m has order $m-1$. When m is not prime, that group has order $\varphi(m)$, where φ is Euler’s totient function. Euler’s totient function is hard to compute unless the factorization of m into prime numbers is known.↩
6. In general, code should be endian-neutral with regards to the in-memory representation of integer types. If some code is not endian neutral, then this means that it writes data using one integer type, and reads it back with another different-sized integer type, which is tantamount of breaking aliasing rules, and this leads to much sorrow, wailing and grinding of teeth. Endian-neutral code is also code that has a better chance of not breaking when using high optimization levels in newer compilers; and it is more portable.↩
7. In this case, the extra carry bit from the addition is not actually feasible, since the modulus is a 255-bit integer, and will use 9 limbs, that would be good for up to 279 bits. This code could thus be simplified a tiny bit by not looking at the result of the `br_i31_add()`, and simply setting `ctl` to the opposite of the `br_i31_sub()` on the second line. But this would make things even less clear.↩
8. Experimentally, using language extensions and intrinsics is a great way to trigger “internal compiler errors”, especially in the first versions in which the feature is enabled.↩
9. Penalty for misaligned accesses depends on the architecture. From the point of view of the C standard, this is “undefined behavior” and anything goes. In practice, consequences will range from a benign negligible delay upon

access (e.g. on x86 or POWER8+) to much less negligible delays (a 1000-cycle penalty and a syslog message per unaligned access on the Alpha), to immediate process termination with a bus error (Sparc64, older 64-bit PowerPC), to arguably even worse cases, such as reading wrong values or performing slightly out-of-bounds accesses. Alignment issues are not to be trifled with.↩

(https://twitter.com/BearSSLnews)