Optimized Binary GCD for Modular Inversion

Thomas Pornin

NCC Group, `thomas.pornin@nccgroup.com`

6 August 2020

Abstract. In this short note, we describe a practical optimization of the well-known extended binary GCD algorithm, for the purpose of computing modular inverses. The method is conceptually simple and is applicable to all odd moduli (including non-prime moduli). When implemented for inversion in the field of integers modulo the prime $2^{255} - 19$, on a recent x86 CPU (Coffee Lake core), we compute the inverse in 7490 cycles, with a fully constant-time implementation.

1 Introduction

We study here the following problem: given an odd integer $m \geq 3$ and a value $y \in \mathbb{Z}_m$, compute its modular inverse x which is such that $xy = 1 \bmod m$. This is a notoriously expensive operation, compared with addition and multiplication in $\mathbb{Z}_m$. Its cost is the main reason why, for instance, operations on elliptic curves are often done with projective coordinates or other similar systems where coordinates are handled as fractions, so that all costly inversions can be mutualized into a final inversion, once the whole computation is finished.

There are some applications of modular inversion where the value to invert and/or the modulus is secret. In elliptic curve cryptography, the modulus m is normally prime and publicly known, but the value to invert is private. In RSA key pair generation, modular inversion is needed to compute the private exponent, and the CRT reconstruction factor; in these cases, the value to invert and the modulus are both secret, and when computing the private exponent from the public exponent, the modulus is not even prime. In all situations where secret values are handled, a *constant-time* implementation is needed (i.e. one with an execution time and a memory access pattern that do not vary depending on secret data).

When m is prime, a simple method is to use Fermat's little theorem:

$$1/y = y^{m-2} \mod m$$

This method requires $O(\log m)$ multiplications modulo m, hence it usually has a cost cubic in the size of m (assuming a quadratic cost for multiplication, which is the usual case for moduli used in cryptographic applications). When $m = 2^{255} - 19$, and implementing on a recent 64-bit x86 CPU, modular multiplications are quite fast, and this is favourable to Fermat's little theorem; our implementation of this inversion method, on an Intel Core i5-8259U at 2.30 GHz (Coffee Lake core), can perform a constant-time inversion in 9175 cycles. The goal is to find something faster.

Apart from Fermat's little theorem, the main method to compute modular inverses is the Extended Euclidean Algorithm for GCD. While Euclid's algorithm was presented by Euclid himself[7], the extension to compute values u and v such that $au + bv = \text{GCD}(a, b)$ was first described by Āryabhaṭa[1]. The binary GCD is a variant of Euclid's algorithm that performs only comparisons, subtractions and divisions by 2 (i.e. right shifts), and is therefore more amenable to fast and constant-time implementations than the general divisions in Euclid's algorithm; that algorithm was first presented, albeit a bit cryptically, in the Nine Chapters on the Mathematical Art[5]. A more accessible description was published by Stein in 1967[14]. A variant called the plus-minus algorithm was presented by Brent and Kung in 1983[4]. More recently, a new (and quite complex) algorithm was described by Bernstein and Yang[3]. See also [12] for extensive benchmarks for inversions modulo primes with the $2^n - r$ format (for a small r). Asymptotically fast but not constant-time GCD algorithms are presented in [11].

Bernstein and Yang claim to hold the current speed record for computing inverses modulo $2^{255} - 19$ on recent 64-bit x86 CPUs (Skylake cores and similar variants). The field of integers modulo the prime $p = 2^{255} - 19$ is used by some well-known elliptic curves, Curve25519 and Edwards25519, thus highlighting the relevance of this particular modulus. Bernstein and Yang obtain a performance of 8520 cycles on a Coffee Lake core[1]. With the method presented here, we do better: 7490 cycles.

Our algorithm was added to the BearSSL library[2] in 2018; it was also deployed as part of the implementation of key pair generation in the reference code for the Falcon post-quantum signature scheme[8]. This note formally describes the algorithm and proves its correctness.

2 Optimizing the Extended Binary GCD

Algorithm 1 describes the classic extended binary GCD.

Algorithm 1 Extended Binary GCD (classic algorithm)

```
Require: Odd modulus $m$ ($m \geq 3$, $m \bmod 2 = 1$) and value to invert $y$ ($0 \leq y < m$)
Ensure: $1/y \bmod m$ (if GCD($y, m$) = 1), or zero
 1: $a \leftarrow y, u \leftarrow 1, b \leftarrow m, v \leftarrow 0$
 2: while $a \neq 0$ do
 3:     if $a = 0 \bmod 2$ then
 4:         $a \leftarrow a/2$                          ▷ $a$ is even, so this division is exact.
 5:         $u \leftarrow u/2 \bmod m$
 6:     else
 7:         if $a < b$ then
 8:             $(a, u, b, v) \leftarrow (b, v, a, u)$     ▷ Conditional swap to ensure $a \geq b$.
 9:         $a \leftarrow (a - b)/2$                    ▷ $a$ and $b$ are odd, so this division is exact.
10:         $u \leftarrow (u - v)/2 \bmod m$
11: if $b \neq 1$ then
12:     return 0 (value $y$ is not invertible)       ▷ $b$ contains GCD($y, m$) at this point.
13: return $v$
```

[1]I am deeply grateful to Bo-Yin Yang, who sent me a copy of their code so that I may benchmark it on the same test system as the one used for the algorithm described in this note. This speed is almost identical to the 8543 cycles on a Kaby Lake core, as reported in the published article; this makes sense, since Coffee Lake and Kaby Lake use the same internal scheduler and execution units, and differ only in external pieces that should not matter much for a compact, L1-cache only computational piece of code.

In this algorithm, values a and b are nonnegative integers, while u and v are integers modulo m. It is not hard to see that:

– b is always an odd integer;
– $\mathrm{GCD}(a, b)$ is constant throughout the algorithm;
– when $a = 0$ (exit condition), b contains the GCD of y and m;
– divisions by 2 to compute the new value of a are exact;
– each iteration maintains the invariants $a = uy \bmod m$ and $b = vy \bmod m$;
– the sum of the lengths, in bits, of a and b is decremented by at least one at each iteration.

Therefore, if m is a n-bit integer, then at most $2n - 1$ iterations are needed to reach the exit condition. In order to have a constant-time implementation, we need to *always* perform $2n-1$ iterations; if $a = 0$, the extra iterations will not modify b or v, so they won't modify the result. A constant-time implementation will also need to perform each iteration in a constant-time way, replacing all conditions with bitwise combinations of values.

The extended binary GCD is quadratic in the length of the modulus: it has $O(n)$ iterations, and each iteration involves a fixed number of operations which can be computed in time $O(n)$ (comparisons, subtractions, halvings). If implemented as is, its performance is comparable to Fermat's little theorem for $m = 2^{255} - 19$, but somewhat worse for platforms with fast large multiplications; for larger moduli, it becomes better. It also properly handles non-prime moduli.

We will now present two complementary optimization methods that apply to this algorithm and substantially improve performance in practice.

Optimization 1: mutualize updates to u and v: In the description above, u and v record the modifications applied to a and b, but no decision is taken based on their values. We can thus delay and group the updates to u and v:

– Initialize the *update factors*: $f_0 \leftarrow 1, g_0 \leftarrow 0, f_1 \leftarrow 0, g_1 \leftarrow 1$
  The update factors are such that the new value of (u, v) should be:

$$(u, v) \leftarrow (f_0 u + g_0 v, f_1 u + g_1 v)$$

  We update f_0, f_1, g_0 and g_1 instead of u and v in the algorithm. Once every k iterations, while the update factors are still small (e.g. they fit in a single register each), we apply them to u and v using multiplication opcodes, which is much faster than making k linear updates. After each such update, the update factors are reinitialized.
– When swapping a with b (in case a and b are both odd, and $a < b$), exchange f_0 with f_1 and g_0 with g_1.
– When subtracting b from a, subtract f_1 from f_0 and g_1 from g_0.
– At each iteration, we should divide f_0 and g_0 by 2 (corresponding to the division of a by 2), but this would entail making them non-integral. Instead, we multiply f_1 and g_1 by 2.

The effect of the last item is that the update factors, after t iterations, are in fact themselves wrong by a factor of exactly 2^t. This is not much of a problem; we can perform a division of the final result v by 2^{2n-1}, after the $2n - 1$ iterations (if m is known in advance, this division is merely a multiplication by a precomputed constant; even in the general case, this division is the same thing as a pair of Montgomery reductions, and has a cost no higher than that of a couple of modular multiplications).

The update factors are signed integers; after k iterations, the maximum (absolute) value of any of them is 2^k. It can also be seen that -2^k is not possible, only $+2^k$, and only for f_1 or g_1. Thus, one can set k to be up to one less than the size of the registers on the implementation architecture.

Optimization 2: use approximations of *a* and *b*: While the update factors allow grouping the updates of u and v, and performing them efficiently with the help of integer multiplication opcodes, the values of a and b are still large. However, the update factors *record* the operations which have been performed on a and b, to be mimicked by u and v; thus, if we could compute the update factors *first*, we may also apply them to a and b, thereby grouping and optimizing the updates to a and b too.

The issue here is that all update decisions depend on the values of a and b; specifically, whether a is even or odd (thus using the low bit of a), and whether a is greater than b (which requires mostly looking at the high bits of a and b). It can easily be seen that for k iterations, the value of the low bit of a will only depend on the initial k low bits of a and b. The optimization idea is to make *approximations* of a and b that will fit in a single register each, and can tell us in a mostly correct way whether a is greater than b or not.

Suppose that the local architecture offers $2k$-bit registers. Let $n = \max(\text{len}(a), \text{len}(b))$, where $\text{len}(x)$ is the length of x in bits (the smallest integer i such that $x < 2^i$). Thus, n is the current length of the largest of a and b. If $n \leq 2k$, then the values a and b already fit into a single register each, and we can use the classic extended GCD algorithm to compute the exact update factors. Otherwise, extract the k low bits and the k top bits of a and b, and assemble them into $\bar{a}$ and $\bar{b}$:

$$\bar{a} = (a \bmod 2^k) + 2^k \lfloor a/2^{n-k} \rfloor$$
$$\bar{b} = (b \bmod 2^k) + 2^k \lfloor b/2^{n-k} \rfloor$$

Then, apply the extended binary GCD on $\bar{a}$ and $\bar{b}$ for k iterations, yielding update factors f_0, g_0, f_1 and g_1. Finally, apply these factors to update a, b, u and v:

$$(a, b) \leftarrow (f_0 a + g_0 b, f_1 a + g_1 b)$$
$$(u, v) \leftarrow (f_0 u + g_0 v, f_1 u + g_1 v)$$

Note that the computed update factors are then approximations; they may be slightly wrong, and thus fail to shrink a and b as much as expected. They may also lead to a negative value for a or b; in case a negative a is obtained, it suffices to negate it, and also to negate u (this time modulo m), and similarly for b. In practice, it is convenient to:

1. update a with the update factors f_0 and g_0;
2. if the new value is negative, negate it, and also negate f_0 and g_0;
3. apply the (possibly negated) update factors f_0 and g_0 to compute the new value of u.

The tricky point is computing how many iterations are needed in the worst case; for a constant-time implementation, we need an absolute bound. Appendix A is dedicated to proving that each sequence of k iterations ensures that $\text{len}(a) + \text{len}(b)$ is reduced by at least k;

therefore, $2\text{len}(m) - 1$ iterations are sufficient (just like the classic binary GCD algorithm). This bound is optimal: if $y = 2^{\text{len}(m)-1}$, then all $2\text{len}(m) - 1$ iterations are needed.

It may be convenient, for implementation purposes, to slightly raise the total number of iterations so that it is a multiple of k. Alternatively, the last iterations may be specialized away, which may allow for some extra optimizations, especially when the modulus m is prime; this will be detailed later on.

The optimized algorithm is described below (algorithm 2).

Algorithm 2 Extended Binary GCD (optimized algorithm)

```
Require: Odd modulus $m$ ($m \geq 3$, $m \bmod 2 = 1$), value $y$ ($0 \leq y < m$), and $k > 1$
Ensure: $1/y \bmod m$ (if $\text{GCD}(y, m) = 1$)
 1: $a \leftarrow y, u \leftarrow 1, b \leftarrow m, v \leftarrow 0$
 2: for $1 \leq i \leq \lceil (2\text{len}(m) - 1)/k \rceil$ do
 3:     $n \leftarrow \max(\text{len}(a), \text{len}(b), 2k)$
 4:     $\bar{a} \leftarrow (a \bmod 2^k) + 2^k \lfloor a/2^{n-k} \rfloor$        ▷ $\bar{a} < 2^{2k}$
 5:     $\bar{b} \leftarrow (b \bmod 2^k) + 2^k \lfloor b/2^{n-k} \rfloor$        ▷ $\bar{b} < 2^{2k}$
 6:     $f_0 \leftarrow 1, g_0 \leftarrow 0, f_1 \leftarrow 0, g_1 \leftarrow 1$
 7:     for $1 \leq j \leq k$ do
 8:         if $\bar{a} = 0 \bmod 2$ then
 9:             $\bar{a} \leftarrow \bar{a}/2$
10:         else
11:             if $\bar{a} < \bar{b}$ then
12:                 $(\bar{a}, \bar{b}) \leftarrow (\bar{b}, \bar{a})$
13:                 $(f_0, g_0, f_1, g_1) \leftarrow (f_1, g_1, f_0, g_0)$
14:             $\bar{a} \leftarrow (\bar{a} - \bar{b})/2$
15:             $(f_0, g_0) \leftarrow (f_0 - f_1, g_0 - g_1)$
16:         $(f_1, g_1) \leftarrow (2f_1, 2g_1)$
17:     $a \leftarrow af_0 + bg_0$
18:     if $a < 0$ then
19:         $(a, f_0, g_0) \leftarrow (-a, -f_0, -g_0)$
20:     $u \leftarrow uf_0 + vg_0 \bmod m$
21:     $b \leftarrow af_1 + bg_1$
22:     if $b < 0$ then
23:         $(b, f_1, g_1) \leftarrow (-b, -f_1, -g_1)$
24:     $v \leftarrow uf_1 + vg_1 \bmod m$
25: $v \leftarrow v/2^{k\lceil (2\text{len}(m)-1)/k \rceil} \bmod m$
26: if $b \neq 1$ then
27:     return 0 (value $y$ is not invertible)
28: return $v$
```

When y is known in advance to be either 0 or an invertible number modulo m (this is always the case when m is prime), then the last two outer iterations can be specialized and simplified, because, at that point:

– If $y = 0$, then the update factors after k inner iterations will be $f_0 = 1, g_0 = 0, f_1 = 0$ and $g_1 = 2^k$, regardless of the value of $\bar{b}$.
– Otherwise, it is known that $\text{len}(a) + \text{len}(b) < 2k$, which means that $\bar{a} = a$ and $\bar{b} = b$; the computed update factors will then be exact, and will remain exact even if more than k iterations are performed. Depending on the implementation of the routines that apply the update factors, it may be possible to merge the last two outer iterations.
– At the end of the final iteration, it is not necessary to compute the new values of a, b and u, only v (since the update factors are exact at that point, they cannot lead to a negative b).

In the general case, when modulus m is not necessarily prime, and a non-invertible but distinct from zero input y is possible, these optimizations on the final steps can still be applied, but an extra verification step is required. It suffices to verify that the computed value, when multiplied with the original operand, yields the value 1. The cost of this extra check is small (typically less than 2% of the overall cost, for a modulus of about 256 bits).

3 Implementation and Benchmarks

We implemented our method in the case of the prime modulus $p = 2^{255} - 19$. Our code is available at:

`https://github.com/pornin/bingcd`

Implementation is in C, with some inline assembly, as well as calls to intrinsic functions. It should compile and run on any Linux or similar system, using GCC or Clang, and an x86 CPU with BMI2 extensions (in the Intel line, this means using a Haswell core or newer).

Extraction of the approximated words $\bar{a}$ and $\bar{b}$ is done with constant-time selection primitives to obtain the highest non-zero limbs in a OR b (bitwise Boolean OR), then the `lzcnt` opcode to count the number of leading zeros. This opcode is available on all x86 CPUs that also feature the BMI2 opcodes. This extraction is constant-time under the assumption that bitwise shifts execute in time independent of the shift count, an assumption that is valid on all relevant x86 CPUs to date[2].

Updates to a, b, u and v use $64 \times 64 \rightarrow 128$ multiplication opcodes, accessed through the non-standard `unsigned __int128` type[3]. Large integers are represented over 256 bits as four 64-bit limbs; carry propagation uses the `adc` and `sbb` opcodes, accessed through the `_addcarry_u64()` and `_subborrow_u64()` intrinsic functions.

The whole point of the optimization described in this note is to make the core of the inner loop as simple and fast as possible. In our implementation, it boils down to the following routine, implemented using inline assembly:

[2]The last Intel CPU on which shifts were not constant-time with regard to the shift count was the Pentium IV, in the early 2000s, but that CPU does not offer `lzcnt`.

[3]Note that the Microsoft Visual C compilers do not support this 128-bit type; they do not support inline assembly in 64-bit mode either. Thus, our example implementation is mostly restricted to GCC and Clang, and perhaps some other compilers that strive at GCC compatibility, including extensions to the language syntax. An MSVC implementation would use the `_umul128()` intrinsic function, and an external assembly module.

```
# Copy f0, g0, f1, g1, xa and xb into r10..r15
movq   %rax, %r10
movq   %rbx, %r11
movq   %rcx, %r12
movq   %rdx, %r13
movq   %rsi, %r14
movq   %rdi, %r15

# Conditional swap if xa < xb
cmpq   %rdi, %rsi
cmovb  %r15, %rsi
cmovb  %r14, %rdi
cmovb  %r12, %rax
cmovb  %r10, %rcx
cmovb  %r13, %rbx
cmovb  %r11, %rdx

# Subtract xb from xa
subq   %rdi, %rsi
subq   %rcx, %rax
subq   %rdx, %rbx

# If xa was even, override the operations above
testl  $1, %r14d
cmovz  %r10, %rax
cmovz  %r11, %rbx
cmovz  %r12, %rcx
cmovz  %r13, %rdx
cmovz  %r14, %rsi
cmovz  %r15, %rdi

# Now xa is even; apply shift.
shrq   $1, %rsi
addq   %rcx, %rcx
addq   %rdx, %rdx
```

Only simple instructions that can be executed in several distinct CPU ports, thus amenable to parallelism, are used. Conditional swaps and selection are performed with `cmov`; this instruction is constant-time, has a 1-cycle latency, and two of them can be executed in a single cycle on a Coffee Lake core, thus each sequence of six `cmov` conceptually needs only 3 cycles to execute. The non-conditional `mov` are even more amenable to parallelism (four ports are eligible, leading to up to four `mov` executing in the same cycle) and some may be in fact eliminated by the register renaming unit. The `sub` and `add` opcodes can similarly execute on four ports. Note that multiplications by 2 of f_1 and g_1 (`rcx` and `rdx` in the code above) are performed with `add` instead of `shl`, because the latter may execute on only two ports and was measured to slightly decrease performance.

Counting each `cmov` at 0.5 cycle, `sub`, `add`, `cmp` and `test` at 0.25 cycle each, and `shr` at 0.5 cycle (using the reported reciprocal throughputs from Agner Fog's comprehensive tables[10]), the sequence above requires a minimum of 8.25 cycles, not counting the `mov` opcodes. Simple measurements show that this sequence, when used in a loop, requires 9 cycles per iteration (on our test system), which means that some of the `mov` opcodes are eliminated and replaced with internal free data routing. In total, the 508 iterations represent 4572 cycles by themselves; all the other operations (extraction of the properly shifted words, updates to a, b, u and v, final multiplication by 2^{-508}) use slightly less than 3000 cycles. This highlights the importance of optimizing that inner loop as much as possible.

Measures are made on an Intel Core i5-8295U CPU, clocked at 2.3 GHz. Host system is Linux (Ubuntu 20.04, kernel 5.4.0-42). C compiler is Clang-10.0.0 and compilation uses optimization flags `-O3 -march=native`. TurboBoost is disabled (in the BIOS), so there is no frequency scaling above 2.3 GHz. SMT (HyperThreading) is disabled. Clock cycles are obtained with the `rdtsc` opcode before and after a sequence of 100 dependent inversions (i.e. each inversion works over the output of the previous one); a serializing `lfence` opcode is also used to ensure that the CPU scheduler will run `rdtsc` at the right place in the sequence of instructions. The bench program first runs some extensive unit tests, to verify correction of the computations, and also act as a "warm-up" to make sure the CPU has gone out of energy-saving mode and has reached its nominal frequency. 2000 measures are then performed; the first 1000 are discarded (they are used to make sure caches are populated and branch prediction is properly trained), then the median of the remaining 1000 is reported[4].

On this system, our inversion routine runs in 7490 cycles. This is better than the previously published record[3], which runs in 8520 cycles on the same system. For comparison purposes, we also implemented the usual inversion method with Fermat's little theorem (FLT); for this, we use handwritten assembly routines for multiplications, squarings, and sequences of squarings modulo $2^{255} - 19$. These routines are inspired from, and very similar to, the ones described in [13] and [12], though our FLT inversion routine ends up running in 9175 cycles, which is very slightly faster than the previous record (9301 cycles, in [12]): apparently, our multiple squaring routine is faster by about half a cycle per iteration. We don't really know why. In any case, the extended binary GCD is still faster.

A noteworthy feature of our code is that it can work with other moduli, with the same performance; supported moduli have the format $2^{255} - r$ for any odd r between 1 and $2^{31} - 1$ (this limit is imposed by the way we perform modular reduction after a multiplication). If the modulus is not prime, the FLT inversion method no longer works, but the extended binary GCD still returns the correct result (in that case, our code includes the extra verification step, to also cover non-invertible inputs; that extra verification step costs about 100 cycles).

[4]The deactivation of TurboBoost and the warm-up steps ensure that the CPU runs at exactly the frequency used by the TSC counter. We also tried to use the performance counters, namely the CPU_CLK.UNHALTED counter which is supposed to count "true" cycles, and can be read with `rdpmc` at index `0x40000001`; this does not yield different results. Our test program uses `rdtsc` because it is readable by non-root userland processes by default. Regardless, some measurement variance remains, of unclear provenance. It does not seem to be correlated with the value being inverted (since we observe that variance even when using the same values again and again), and thus should not contradict our claim of constant-time behaviour. It is known that modern complex CPU are full of accumulated heuristics, many of them undocumented.

4 Conclusion

We described a simple and practical optimization to the classic extended binary GCD algorithm. In the specific case of inversion modulo $2^{255} - r$ (for a small r), this optimization happens to lead to a routine which is faster than the usual method based on Fermat's little theorem, and also slightly faster than the previously best reported inversion routine for that class of moduli, the recent (and more complicated) algorithm from Bernstein and Yang. This does not mean that our optimization makes binary GCD the best algorithm in all cases; it depends on many parameters, such as modulus size and format and target implementation architecture. We also did not try to apply it to polynomial GCDs, though a similar optimization might also provide practical benefits there. One can view our method as an improvement in *data locality*, thereby reducing data routing cost (in our primary example, by keeping values in single registers, thereby avoiding RAM traffic in the innermost loop).

The same optimization technique should, conceptually, apply to the computation of the Jacobi symbol[6]. The Jacobi symbol is useful to test the quadratic residue status of a finite field element, which in turn is part of the operations used for constant-time hashing into elliptic curves[9].

References

1. Āryabhaṭa, *Āryabhaṭīya*, 499.
2. *BearSSL, a smaller SSL/TLS library*, `https://www.bearssl.org/`
3. D. Bernstein and B.-Y. Yang, *Fast constant-time gcd computation and modular inversion*, `https://gcd.cr.yp.to/papers.html#safegcd`
4. R. Brent and H. Kung, *Systolic VLSI arrays for linear-time GCD computation*, VLSI 1983, pp. 145-154, 1983.
5. 九章算术, chapter 1, section 6, 1st century AD.
6. H. Cohen, *A Course in Computational Algebraic Number Theory*, Springer, pp. 29-31, 1993.
7. Euclid, *Elements*, book 7, propositions 1 and 2, c. 300 BC.
8. *Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU*, `https://falcon-sign.info/`
9. A. Faz-Hernandez, S. Scott, N. Sullivan, R. Wahby and C. Wood, *Hashing to Elliptic Curves*, `https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-09`
10. A. Fog, *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, `https://www.agner.org/optimize/instruction_tables.pdf`
11. N. Möller, *On Schönhage's algorithm and subquadratic integer GCD computation*, Mathematics of Computation, vol. 77, pp. 589-607, 2008.
12. K. Nath and P. Sarkar, *Efficient Arithmetic In (Pseudo-)Mersenne Prime Order Fields*, `https://eprint.iacr.org/2018/985`
13. , T. Oliveira, J. López, H. Hisil, A. Faz-Hernández and F. Rodríguez-Henríquez, *How to (pre-)compute a ladder – improving the performance of X25519 and X448*, SAC 2017, Lecture Notes in Computer Science, vol. 10719, pp. 172-191.
14. J. Stein, *Computational problems associated with Racah algebra*, Journal of Computational Physics, vol. 1, issue 3, pp. 397–405, 1967.

A Bounding the Number of Iterations

In this section, we prove that the inner loop of algorithm 2 ensures that the k iterations always reduce $\mathrm{len}(a) + \mathrm{len}(b)$ by at least k bits.

Each excution of the inner loop consists of k iterations. We number iterations from 0 to $k - 1$; this means that iteration t happens just after t iterations have completed. We will note x_t the value of the quantity x (for any of the quantities we manipulate, namely a, b, $\bar{a}$ and $\bar{b}$) when *exiting* iteration $t - 1$ and *entering* iteration t.

A.1 Maximum Error Bound

We first need to show that $\bar{a}$ and $\bar{b}$ remain "good approximations" of a and b over the k iterations.

First, notice that after t iterations of the inner loop ($0 \le t \le k$), the following inequalities hold:

$$
\begin{aligned}
|f_0| + |g_0| &\le 2^t \\
|f_1| + |g_1| &\le 2^t
\end{aligned}
$$

They are obviously correct at the start ($t = 0$) since, at that point, $f_0 = g_1 = 1$ and $f_1 = g_0 = 0$. Then, at each successive iteration:

- If $\bar{a}$ is even, then:
$$
\begin{aligned}
|f_0| + |g_0| &\le 2^t < 2^{t+1} \\
|2f_1| + |2g_1| &= 2(|f_1| + |g_1|) \le 2^{t+1}
\end{aligned}
$$
- If $\bar{a}$ is odd, then the conditional swap preserves the inequalities, and the subtraction implies that:
$$
\begin{aligned}
|f_0 - f_1| + |g_0 - g_1| &\le |f_0| + |f_1| + |g_0| + |g_1| \\
&\le |f_0| + |g_0| + |f_1| + |g_1| \\
&\le 2^{t+1}
\end{aligned}
$$

Suppose that the inner loop started with a and b, approximated into $\bar{a}$ and $\bar{b}$. We have $n = \max(\mathrm{len}(a), \mathrm{len}(b))$. If $n \le 2k$, then $\bar{a} = a$ and $\bar{b} = b$, and there is no approximation; the algorithm then proceeds exactly as the classic algorithm (algorithm 1) and ensures a gain of at least 1 bit per iteration. We thus consider thereafter that $n > 2k$.

Considering the situation after t iterations of the inner loop ($0 \le t \le k$), we define:

$$
\begin{aligned}
\bar{a}_t &= (\bar{a}f_0 + \bar{b}g_0)/2^t \\
\bar{b}_t &= (\bar{a}f_1 + \bar{b}g_1)/2^t \\
a_t &= (af_0 + bg_0)/2^t \\
b_t &= (af_1 + bg_1)/2^t
\end{aligned}
$$

These are, respectively, the values of $\bar{a}$ and $\bar{b}$ after t iterations, and the corresponding values of a and b, should the update factors be applied at that point. Note that, by construction, the divisions by 2^t are exact.

Note that $\bar{a}$ and $\bar{b}$ are approximations of a and b in the sense that bits k to $2k-1$ of $\bar{a}$ (respectively $\bar{b}$) are exactly the same as bits $n-k$ to $n-1$ of a (respectively b). This can be expressed with the following inequalities:

$$|a - 2^{n-2k}\bar{a}| < 2^{n-k}$$
$$|b - 2^{n-2k}\bar{b}| < 2^{n-k}$$

Consider now the following:

$$\begin{aligned} a_t - 2^{n-2k}\bar{a}_t &= \frac{(af_0 + bg_0) - 2^{n-2k}(\bar{a}f_0 + \bar{b}g_0)}{2^t} \\ &= \frac{(a - 2^{n-2k}\bar{a})f_0 + (b - 2^{n-2k}\bar{b})g_0}{2^t} \end{aligned}$$

Therefore, applying the triangular inequality:

$$\begin{aligned} |a_t - 2^{n-2k}\bar{a}_t| &\le 2^{-t}(|2^{n-k}||f_0| + |2^{n-k}||g_0|) \\ &\le 2^{n-k-t}(|f_0| + |g_0|) \end{aligned}$$

Since $|f_0| + |g_0| \le 2^t$, we then get that:

$$|a_t - 2^{n-2k}\bar{a}_t| \le 2^{n-k}$$

and similary for b:

$$|b_t - 2^{n-2k}\bar{b}_t| \le 2^{n-k}$$

This means that, throughout the k iterations, the upper bits (k to $2k-1$) of $\bar{a}$ and $\bar{b}$ remain a close approximation of the upper bits ($n-k$ to $n-1$) of the values a and b that they represent, i.e. the values we would get by applying the update factors right away.

A.2 Length Reduction at Divergence Point

As in the previous section, consider the values a and b at the start of an inner loop. The k iterations of the inner loop will perform halvings, swaps and subtractions based on the bits in $\bar{a}$ and $\bar{b}$, used as representations of a and b. The halvings and subtractions rely on the k low bits of $\bar{a}$ and $\bar{b}$, which are always exact; only the conditional swaps may differ between the classic and optimized algorithms, since, in the latter case, the relative ordering of $\bar{a}$ and $\bar{b}$ might not match that of a and b.

If, throughout the k iterations, the inner loop makes the exact same decisions that it would take with the actual updated values of a and b, then it follows the same steps as the classic algorithm (algorithm 1) on the same inputs, and thus obtains the same reduction in length of a and b, i.e. at least k bits in total. Otherwise, we call the *divergence point* the iteration d ($0 \le d < k$) at which the inner loop of algorithm 2 takes a different decision from that would have been taken by the classic algorithm. Let us observe the situation at that point, using the notations from

– The algorithm has $\bar{a}_d$ and $\bar{b}_d$, which are both odd. Without loss of generality, we can assume that $\bar{a}_d \ge \bar{b}_d$. The inner loop of algorithm 2 decides to subtract $\bar{b}_d$ from $\bar{a}_d$.

– However, the actual values a_d and b_d (updated values of a and b) would be such that $a_d < b_d$. The subtraction will then yield a negative value.

As we saw in section A.1, $\bar{a}_d$ and $\bar{b}_d$ are good approximations of a_d and b_d, respectively. Write that:

$$2^{n-2k}(\bar{a}_d - \bar{b}_d) - (a_d - b_d) = (2^{n-2k}\bar{a}_d - a_d) + (b_d - 2^{n-2k}\bar{b}_d)$$

and thus:

$$|2^{n-2k}(\bar{a}_d - \bar{b}_d) - (a_d - b_d)| \leq |2^{n-2k}\bar{a}_d - a_d| + |b_d - 2^{n-2k}\bar{b}_d| \leq 2^{n-k+1}$$

Since $2^{n-2k}(\bar{a}_d - \bar{b}_d) \geq 0$ and $a_d - b_d < 0$, and their difference is no greater in absolute value than 2^{n-k+1}, it follows that:

$$0 \leq \bar{a}_d - \bar{b}_d < 2^{k+1}$$

Therefore, at the divergence point d, the inner loop will compute a new value $\bar{a}_{d+1} = (\bar{a}_d - \bar{b}_d)/2$ which will be strictly lower than 2^k, hence such that all its upper bits will be zero. Similarly:

$$0 > a_d - b_d \geq -2^{n-k+1}$$

which implies that $|a_{d+1}| \leq 2^{n-k}$. Note that $|a_{d+1}|$ may be equal to 2^{n-k} only if $\bar{a}_d = \bar{b}_d$, in which case $\bar{a}_{d+1} = 0$ and all subsequent iterations of the inner loop will decide to divide $\bar{a}_t$ (and thus a_t) by 2, without any further subtraction.

A.3 Minimal Length Reduction

We again consider a situation at the start of an inner loop, such that:

- One of a and b (or both) has length at least $2k + 1$ bits.
- The inner loop execution includes a divergence point at iteration d.

In all other inner loop cases, the execution follows the steps of the classic binary GCD algorithm, thereby ensuring a total length reduction of (a, b) by at least k bits.

After the divergence point, the following properties apply:

- Dividing a_t by 2 does not change the sign of a_t.
- If $a_t > 0$ and $b_t < 0$, then $a_t - b_t > 0$.
- If $a_t < 0$ and $b_t > 0$, then $a_t - b_t < 0$.

Therefore, in all subsequent steps, one of the values will be negative, and the other one will be positive. It cannot happen that two strictly negative values, or two nonnegative values are obtained, unless a reaches zero (in which case the algorithm has converged).

Moreover, if $|a_t| \geq 2^{n-k}$ and $|b_t| < 2^{n-k}$, then:

$$\left|\frac{a_t - b_t}{2}\right| \leq \frac{|a_t| + |b_t|}{2} < |a_t|$$

and if $|a_t| < 2^{n-k}$ and $|b_t| < 2^{n-k}$, then:

$$\left|\frac{a_t - b_t}{2}\right| \leq \frac{|a_t| + |b_t|}{2} < 2^{n-k}$$

As remarked in section A.1, if the divergence point yields $a_{d+1} = -2^{n-k}$, then all subsequent iterations will use $\bar{a}_t = 0$ and none of them will try to perform a subtraction (or a swap); otherwise, $|a_{d+1}| < 2^{n-k}$. It follows that, after the point of divergence, even if an iteration fails to reduce the sum of the lengths of a_t and b_t, it will not allow any value a or b above 2^{n-k} to grow, or any value below 2^{n-k} to regrow beyond $n - k$ bits.

Using these properties, we analyze the possible cases, depending on the initial lengths of the values.

Case 1: $\text{len}(a) = n$ and $\text{len}(b) \leq n - k$

This situation means that, when starting the inner loop, b was shorter than a by at least k bits.

In that case, $\bar{b} < 2^k$ and $\bar{a} \geq 2^{2k-1}$; thus, after t successive steps, the minimal value of $\bar{a}_t$ is $\bar{a}_t - (2^t - 1)\bar{b}_t$ (if b_t was subtracted from a_t at each iteration). For $t \leq k - 1$, we have:

$$\begin{aligned}
\bar{a}_t - (2^t - 1)\bar{b}_t &> 2^{2k-1} - (2^t - 1)(2^k - 1) \\
&> 2^{2k-1} - (2^{k-1} - 1)(2^k - 1) \\
&> 2^{2k-1} - (2^{2k-1} - 2^{k-1} - 2^k + 1) \\
&> 2^{k-1} + 2^k - 1 \\
&> \bar{b}_t
\end{aligned}$$

Thus, up to and including the last iteration, $a_t > b_t$ and the inner loop will not perform a swap. It follows that, after k iterations, the reduction factors are such that $f_0 = 1$, $0 \geq g_0 \geq -(2^k - 1)$, $f_1 = 0$, and $g_1 = 2^k$. Since $a > 2^{k-1}b$, similar inequalities imply that if there is a divergence point, then it may only happen at the last iteration; that last iteration will then compute a negative a_k, but such that $|a_k| < 2^{n-k}$ (since it will be the result of subtracting b_{k-1}, still equal to b at that point, from a nonnegative a_{k-1}). The conditional negation will make the next a positive, and of length at most $n - k$ bits. In total, a has been reduced by at least k bits, while b was unchanged.

Case 2: $\text{len}(a) \leq n - k$ and $\text{len}(b) = n$

This case is similar to the previous case. It may start with some divisions of a by 2; if s such steps happen before reaching an odd $\bar{a}$, then we can apply the analysis of the previous case, replacing k with $k-s$: a swap occurs at iteration s, and after the swap, the new a (previously b) has length n bits while the new b (previously a) has length at most $n-k-s$ bits. The subsequent $k - s$ iterations will each reduce a by at least one bit each, but may reach a divergence point only on the last iteration, and only if $s = 0$; the total reduction will still be at least k bits.

Case 3: $\text{len}(a) = n$ and $\text{len}(b) \geq n - k + 1$, or $\text{len}(a) \geq n - k + 1$ and $\text{len}(a) = n$

Suppose that when the divergence point (d) is reached, $b_d < 2^{n-k}$. Since the divergence point is such that $a_d < b_d$, this implies that $a_d < 2^{n-k}$ as well. Therefore, the two values have already been reduced by at least $k + 1$ bits in total at this point. Since operations beyond the divergence point won't allow values to regrow beyond 2^{n-k} in absolute value, it follows that the total reduction will be at least $k + 1$ bits.

We now assume that when the divergence point is reached, $\text{len}(b_d) = n - k + j$ for some $j \geq 0$. Since $a_d < b_d$ (this is a divergence point), it implies that $\text{len}(a_d) \leq n - k + j$ as well. Since one of the starting values had length n bits, this means that at least $k - j$ bits of reduction have been achieved at that point. Then, the new value of a is a_{d+1} which is such that $0 > a_{d+1} \geq -2^{n-k}$. If $a_{d+1} \neq -2^{n-k}$, then this represents an extra reduction of at least j bits in length, hence $k - j + j = k$ bits at least in total. As remarked above, subsequent steps won't allow any value of more than k bits to grow, so this reduction level cannot be compromised; after the k iterations, total reduction will still be at least k bits.

The remaining case is when $a_{d+1} = -2^{n-k}$, exactly. At that point, we have a reduction of at least $k-1$ bits. Moreover, since $\bar{a}_{d+1} = 0$, subsequent steps, if any, will divide a by 2. Thus, if $d < k - 1$, then there will be at least one such extra step, and, again, at least k bits of reduction will be achieved. We thus now assume that the divergence step happens at the last iteration of the inner loop ($d = k - 1$). Since $a_{d+1} = -2^{n-k} = (a_d - b_d)/2$, then $a_d = b_d - 2^{n-k+1}$; however, $\bar{a}_d = \bar{b}_d$, and we know (see section A.1) that:

$$|a_d - 2^{n-2k}\bar{a}_d| \leq 2^{n-k}$$
$$|b_d - 2^{n-2k}\bar{b}_d| \leq 2^{n-k}$$

This implies that, in that case, a_d and b_d are apart from each other by exactly 2^{n-k+1}, but cannot be apart from the common value $2^{n-2k}\bar{a}_d$ by more than 2^{n-k}. It then follows that $a_d = 2^{n-2k}\bar{a}_d - 2^{n-k}$ and $b_d = 2^{n-2k}\bar{a}_d + 2^{n-k}$. This is not possible, since, by construction, a_d and b_d are both odd at this point.

Conclusion: In all cases, we saw that a reduction of at least k bits is achieved, which completes the proof.

Commentary: It may be surprising that even using approximations, we do not need more iterations than the classic binary GCD, for which the worst case bound $2\text{len}(m) - 1$ is already optimal. Intuitively, the reason is that we have some extra "hidden" iterations: the conditional negation of a and b, after their respective updates every k iterations, is the point where the approximation is resolved. In that sense, the optimized algorithm really needs $k + 1$ iterations in the worst case to achieve a k-bit reduction; the last of these $k + 1$ iterations masquerades as a pair of conditional subtractions.