A study of Fermat's difference of squares and an attempt at solving the factorization problem using LLL.

Author: Essbee Vanhoutte
Email: big_polar_bear1@proton.me
Twitter: big_polar_bear2
Bluesky: bigpolarbear1
Website: sandboxescaper.com
Github: big-polar-bear

Introduction and acknowledgments

As I have no formal education due to dropping out of high-school, I am not very literate in formal math notation, hence I will demonstrate my findings mainly by example to avoid confusion.
Please review my work objectively and fairly, that is all ask.

I am not making any claims of producing novel work, or solving the factorization problem, these are merely my own findings to whom it may be of interest.
More than likely this work is just a weaker version of the work by the likes of Coppersmith.
Either way, perhaps it will provide some insight in how one might attempt to solve a complex problem with limited technical know-how.
In theory I suspect this approach scales logarithmically, but the initial scaling is quite steep, and factoring even small numbers is very slow.
But as it uses LLL, with enough CPU cores, it may perhaps outperform other algorithms (running on similar computing power).
However, I am not math educated, and complexity analysis on a lattice reduction algorithm is not my strength.
Additionally, many improvements can be made in the lattice reduction portion of the algorithm as most of that work is done by trial and error in only the last month. The main bulk of my work is the number theoretical portion in chapter 3.
If anything useful comes out of this research project, it will be there.
Odds are however none of it is very interesting or novel.
I gave it my best effort, and this is what I have to share. That is all. Nothing more, nothing less.

This paper is dedicated to my former manager at Microsoft and friend Roger P, who was unfairly treated by the company he worked at for nearly 3 decades and subsequently fired after trying to prevent Microsoft from firing me after I had a mental breakdown in July of 2023.
It is thanks to Roger and my other former managers that I discovered my passion for cryptography and found many software security issues in components such as IKE, OpenSsl, Secure Channel, Kerberos and CryptoApi.
I was able to soar to heights I would have never been able to dream of as a high-school dropout.
Additionally I would also like to thank all my other friends, family and those who believed in me and supported me.

I have learned in life, that nothing really matters except for your friends and those who believe in you, they are the only thing standing between you and the absurdity of the world.

Chapters

-I. From integer to quadratic
-II. From quadratic to quadratic congruence
-III. From quadratic congruence to multiple-choice subset-sum problem
    -a. Rebasing to 0
    -b. Raising the modulus
    -c. Subtracting partial results and reducing the modulus
-IV. From multiple choice subset-sum problem to LLL
    -a. The index and split column
    -b. The mask column
    -c. The aid column
-V. Proof of Concept read-me and performance
-VI. Closing thoughts
-VII. References

I. From integer to quadratic

A semi-prime n is a product of two factors, p and q (pq=n).

Example: p = 41 and q = 107 => n = 4387

(note: In this paper I will use p=41 and q=107 as the main example)

We can represent factorization as a quadratic of the following form:

$x^2+yx+n = 0$

x is the lower factor negated and y is the sum of p and q.

=> $x = -41$ and $y = 107+41$

Throughout this paper it will hopefully become clear to the reader why this choice was made, as on the surface it may seem rather arbitrary.

Plugging x and y into the quadratic above:

$-41^2+(148*-41)+4387 = 0$
=> $1681-6068+4387 = 0$

Knowing only y we can also quickly solve for the factors by using the quadratic formula:

(note: ^0.5 refers to taking the square root)

$a = (y^2-4n)^{0.5}$
lower factor = $|(a-y)/2|$
upper factor = $(a+y)/2$

If we plug in y=148:

$a = (148^2-4*4387)^{0.5}$
=> $a = 4356^{0.5} = 66$
lower factor = $|(66-148)/2| = 41$
upper factor = $(66+148)/2 = 107$

This reduces the problem to finding y (p+q).

Another observation happens when we divide p+q by 2:

$148/2 = 74$

This is the value where the distance between both factors is equal. 74-41 and 107-74 = 33.
This notably is also the value Fermat's factorization algorithm will try to find (difference of squares).

$74^2 = 5476 - 4387 = 1089^{0.5} = 33$

This relation results in a bunch of interesting properties which we will attempt to exploit further down. However instead of 74 and 33 we will instead look at their doubles: 148 and 66, but nothing stops you from modifying the proof of concept and divide everything by two (multiply by the inverse of 2 for congruences) and work with 74 and 33 instead.

II. From quadratic to quadratic congruence

Solving the above quadratic boils down to finding an integer solution. However this appears to be a hard problem.
As we're usually trying to factor very large numbers, it would make sense to use modular reduction and see if we can find a pattern which we can leverage to find the integer solution.

Our representation now becomes the same quadratic but with modular reduction:

$x^2+yx+n = 0 \mod m$

Instead of finding the factors, we find the correct residues mod m now.

Example:

$n = 4387 = 6 \mod 13$
$p = 41 = 2 \mod 13$
$q = 107 = 3 \mod 13$

The residue of n mod 13 is the residue of p*q mod 13 (2*3 = 6 mod 13).
The residue of y mod 13 is p+q mod 13 (2+3 = 5 mod 13)

If y is 5 mod 13 and we know our residue of n mod 13 is 6 then only 2+3 and 3+2 can be our two residues for p and q mod 13. Thus x is either -3 or -2.

Plugging in for y = 5 and x = -3 or -2 in mod 13:

$-2^2+5*-2+4387 = 0 \mod 13$
$\Rightarrow 4-10+6 = 0 \mod 13$

$-3^2+5*-3+4387 = 0 \mod 13$
$\Rightarrow 9-15+6 = 0 \mod 13$

In real life we don't know y is 5 (since we do not know p+q). We do know n mod 13 is 6. Thus all possible y solutions can be enumerated by summing up each of the two residues mod 13 that multiply to 6.

Hence a y solution can also be told to exist, if for a given y value an x solution exists. This can be trivially determined using the Legendre symbol (see function: squareRootExists in factor.sage)

All x and y solutions mod 13 that solve the quadratic congruence:

y: 1 x: -5 *=> -5^2+1*-5+4387 = 0 mod 13*
y: 1 x: -9 *=> -9^2+1*-9+4387 = 0 mod 13*
y: 5 x: -2 *=> -2^2+5*-2+4387 = 0 mod 13*
y: 5 x: -3 *=> -3^2+5*-3+4387 = 0 mod 13*
y: 6 x: -7 *=> -7^2+6*-7+4387 = 0 mod 13*
y: 6 x: -12 *=> -12^2+6*-12+4387 = 0 mod 13*
y: 7 x: -1 *=> -1^2+7*-1+4387 = 0 mod 13*
y: 7 x: -6 *=> -6^2+7*-6+4387 = 0 mod 13*
y: 8 x: -10 *=> -10^2+8*-10+4387 = 0 mod 13*
y: 8 x: -11 *=> -11^2+8*-11+4387 = 0 mod 13*
y: 12 x: -4 *=> -4^2+12*-4+4387 = 0 mod 13*
y: 12 x: -8 *=> -8^2+12*-8+4387 = 0 mod 13*

III. From quadratic congruence to multiple-choice subset-sum problem

For the majority of this project, I was stuck at representing the problem as quadratic congruences. Until I found a way to transform the problem into a (modular) multiple-choice subset-sum problem.

Say we have the following y solutions for mod 3, 5, 7, 11 and 13 (we ignore the x solutions in the quadratic and look at the y solutions in isolation for most of the algorithm):

mod 3 = 1, 2
mod 5 = 2, 3
mod 7 = 0, 1, 6
mod 11 = 1, 2, 5, 6, 9, 10
mod 13 = 1, 5, 6, 7, 8, 12

One strategy is to find every combination of the above solutions in mod 3*5*7*11*13 (one way is using CRT).

This gives us:

mod 15015 = 83, 97, 98, 112, 148, 188, 203, 287, 307, 343, 358, 398, 428, 463, 482, 512, 538, 617, 643, 658, 727, 742, 812, 827, 853, 937, 967, 1007, 1022, 1028, 1058, 1072, 1112, 1253, 1267, 1282, 1288, 1358, 1373, 1442, 1457, 1462, 1483, 1513, 1553, 1567, 1568, 1618, 1637, 1652, 1813, 1828, 1847, 1897, 1912, 1982, 2003, 2008, 2023, 2092, 2107, 2113, 2177, 2183, 2198, 2267, 2282, 2393, 2437, 2443, 2458, 2477, 2528, 2542, 2612, 2638, 2653, 2722, 2723, 2737, 2738, 2807, 2822, 2828, 2848, 2932, 2983, 3002, 3023, 3037, 3067, 3158, 3178, 3262, 3268, 3277, 3283, 3353, 3368, 3437, 3452, 3478, 3548, 3563, 3613, 3632, 3647, 3697, 3752, 3808, 3823, 3892, 3893, 3907, 3947, 3977, 3983, 3998, 4003, 4087, 4102, 4178, 4192, 4193, 4207, 4388, 4402, 4432, 4438, 4453, 4493, 4523, 4577, 4607, 4633, 4648, 4718, 4753, 4802, 4817, 4907, 4922, 4948, 4978, 5032, 5062, 5102, 5117, 5153, 5257, 5312, 5348, 5362, 5363, 5377, 5468, 5543, 5557, 5572, 5578, 5608, 5648, 5663, 5732, 5747, 5803, 5818, 5858, 5908, 5923, 5942, 5972, 6007, 6077, 6103, 6118, 6187, 6202, 6272, 6287, 6293, 6313, 6397, 6467, 6488, 6518, 6532, 6572, 6623, 6727, 6733, 6742, 6748, 6818, 6832, 6833, 6902, 6917, 6943, 6973, 7013, 7027, 7028, 7078, 7097, 7112, 7118, 7162, 7273, 7288, 7357, 7372, 7442, 7448, 7463, 7468, 7547, 7552, 7567, 7573, 7643, 7658, 7727, 7742, 7853, 7897, 7903, 7918, 7937, 7987, 7988, 8002, 8042, 8072, 8098, 8113, 8182, 8183, 8197, 8267, 8273, 8282, 8288, 8392, 8443, 8483, 8497, 8527, 8548, 8618, 8702, 8722, 8728, 8743, 8813, 8828, 8897, 8912, 8938, 9008, 9043, 9073, 9092, 9107, 9157, 9197, 9212, 9268, 9283, 9352, 9367, 9407, 9437, 9443, 9458, 9472, 9547, 9638, 9652, 9653, 9667, 9703, 9758, 9862, 9898, 9913, 9953, 9983, 10037, 10067, 10093, 10108, 10198, 10213, 10262, 10297, 10367, 10382, 10408, 10438, 10492, 10522, 10562, 10577, 10583, 10613, 10627, 10808, 10822, 10823, 10837, 10913, 10928, 11012, 11017, 11032, 11038, 11068, 11108, 11122, 11123, 11192, 11207, 11263, 11318, 11368, 11383, 11402, 11452, 11467, 11537, 11563, 11578, 11647, 11662, 11732, 11738, 11747, 11753, 11837, 11857, 11948, 11978, 11992, 12013, 12032, 12083, 12167, 12187, 12193, 12208, 12277, 12278, 12292, 12293, 12362, 12377, 12403, 12473, 12487, 12538, 12557, 12572, 12578, 12622, 12733, 12748, 12817, 12832, 12838, 12902, 12908, 12923, 12992, 13007, 13012, 13033, 13103, 13118, 13168, 13187, 13202, 13363, 13378, 13397, 13447, 13448, 13462, 13502, 13532, 13553, 13558, 13573, 13642, 13657, 13727, 13733, 13748, 13762, 13903, 13943, 13957, 13987, 13993, 14008, 14048, 14078, 14162, 14188, 14203, 14273, 14288, 14357, 14372, 14398, 14477, 14503, 14533, 14552, 14587, 14617, 14657, 14672, 14708, 14728, 14812, 14827, 14867, 14903, 14917, 14918, 14932

Each of these solutions mod 15015 will map to a unique combination of solutions in mod 3,5,7,11 and 13.

We can spot the solution that we are looking for "148" in the list.

If we kept growing the modulus and solutions by means of CRT an important insight is that eventually 148 will be the first element in the ordered list of solutions and no matter how much further we grow the modulus, it will remain 148 while everything else will grow to something bigger then n.
Hence with a big enough modulus the problem can be reduced to finding the "smallest" solution.

For a long time I investigated the possible existence of an algorithm to only keep track of the smallest solution, knowing that if I grew the modulus big enough, this would be my real solution (p+q), but I was unable too find anything useful that doesn't scale unfavorably in complexity.

Y or p+q=148 encodes all the y solutions for any quadratic congruence modulo m where m is less then n (148 is the non-modular reduced solution).
Representing the quadratic this way is not an arbitrary choice. It lets us define a concept of "distance" to the factors. 148 is double 74, and 74 is the point between both of the factors where the distance to the upper and lower factor is equal. x=-41 represents the distance to the lower factor.
Hence with each increase in the modulus we decrease the y solution by 2 and increase the x value by 1, we do this because x and y represent distance, not simply p and p+q.

Representing distance this way is important, because it led to the insight of applying a trick called "rebasing to 0" which is described further down, and it is due this representation of distance that this trick works.
While you can get the x and y value simply by calculating 148 mod m or -41 mod m, in reality the non-modular reduced x and y value increases/decreases with each increase/decrease of the modulus. Because with each increase in the modulus we are getting closer to the lower factor or the point where the distance between the factors is equal (times 2).

(note: If you want to see more examples check out printall in toolbox.sage)

6 = 4387 mod 13 y: 5 x: -2 total y: 122 total x: -28
7 = 4387 mod 12 y: 4 x: -5 total y: 124 total x: -29
9 = 4387 mod 11 y: 5 x: -8 total y: 126 total x: -30
7 = 4387 mod 10 y: 8 x: -1 total y: 128 total x: -31
4 = 4387 mod 9 y: 4 x: -5 total y: 130 total x: -32
3 = 4387 mod 8 y: 4 x: -1 total y: 132 total x: -33
5 = 4387 mod 7 y: 1 x: -6 total y: 134 total x: -34
1 = 4387 mod 6 y: 4 x: -5 total y: 136 total x: -35
2 = 4387 mod 5 y: 3 x: -1 total y: 138 total x: -36
3 = 4387 mod 4 y: 0 x: -1 total y: 140 total x: -37
1 = 4387 mod 3 y: 1 x: -2 total y: 142 total x: -38
1 = 4387 mod 2 y: 0 x: -1 total y: 144 total x: -39
0 = 4387 mod 1 y: 0 x: 0 total y: 146 total x: -40
0 = 4387 mod 0 y: 0 x: 0 total y: 148 total x: -41

Plugging in the total y and x solutions into the quadratic we get:

-41^2+148-41+4387 = 0*
-40^2+146-40+4387 = 147 (1*147)*
-39^2+144-39+4387 = 292 (2*146)*
-38^2+142-38+4387 = 435 (3*145)*
-37^2+140-37+4387 = 576 (4*144)*
-36^2+138-36+4387 = 715 (5*143)*

At the start of my research project in the summer of 2023, I did pattern analysis of modulo reduction applied to n. A pattern that I noticed is this:

31 = 4387 mod 66 total remainder: 31
32 = 4387 mod 65 total remainder: 32
35 = 4387 mod 64 total remainder: 35
40 = 4387 mod 63 total remainder: 40
47 = 4387 mod 62 total remainder: 47
56 = 4387 mod 61 total remainder: 56
7 = 4387 mod 60 total remainder: 67
21 = 4387 mod 59 total remainder: 80
37 = 4387 mod 58 total remainder: 95
55 = 4387 mod 57 total remainder: 112
19 = 4387 mod 56 total remainder: 131
42 = 4387 mod 55 total remainder: 152
13 = 4387 mod 54 total remainder: 175
41 = 4387 mod 53 total remainder: 200
19 = 4387 mod 52 total remainder: 227
1 = 4387 mod 51 total remainder: 256

We can see that at 4387 mod 4387^0.5 (66), the remainder starts at 31.

at n mod 65 the remainder is: 31 + 1^2 = 32
at n mod 64 the remainder is: 31 + 2^2 = 35
at n mod 63 the remainder is: 31 + 3^2 = 40
at n mod 62 the remainder is: 31 + 4^2 = 47
at n mod 61 the remainder is: 31 + 5^2 = 56

Knowing that it is just adding squares to 31 starting at the square root of n, we can calculate the non-modular reduced total remainder for any modulus.

(note: A long time ago, my initial approach was to find a pattern in these remainders that would allow me to determine the distance to the factors.)

Circling back to mod 1, 2, 3, 4, 5 for which we can now calculate the total remainder and our previous result:

0 = 4387 mod 1 total remainder: 4256
1 = 4387 mod 2 total remainder: 4127
1 = 4387 mod 3 total remainder: 4000
3 = 4387 mod 4 total remainder: 3875
2 = 4387 mod 5 total remainder: 3752

Now plugging in the absolute remainders instead:

-41^2+148-41+4387 = 0*
-40^2+146-40+4256 = 16 (1*16)*
-39^2+144-39+4127 = 32 (2*16)*
-38^2+142-38+4000 = 48 (3*16)*

-37^2+140-37+3875 = 64 (4*16)*
-36^2+138-36+3752 = 80 (5*16)*

From the square root of 4387 (66), at which the y value is **16** we need to add 8 in the modulus or in other words subtract **16** from the y value to get to mod 74 (148/2), at n mod 74 the y solution is 0 and x solution is 33 (This gives us: 33^2+0*33+4387 = 0 mod 74). 74 is the point where the distance to both factors is equal, hence resulting in a y solution of 0.

Because of this relation between the remainders and the quadratic and due to the fact that representing things this way makes everything work, I am fairly confident that this is the correct approach. I do not have the mathematical background to prove this and most of this is intuition based on the pattern analysis I did.

(note: for more examples check out toolbox.sage)

Anyway, back to the algorithm..

a. Rebasing to 0

Because with each increase/decrease in the modulus we increase/decrease the Y value by 2 to solve the congruence (we are increasing/decreasing the distance from p and p+q/2), we can use a trick.
For lack of a better term, I will call this trick "rebasing to 0".
Rebasing to 0 in a way lets us partially escape the modular world to the world of integers again.
This makes solving a system of quadratic congruences much easier.

(note: I call this rebasing to 0 as it is basically the same as changing the modulus to 0 in our representation. However mod 0 implies division by 0, but this is how I have abstracted it in my head and the name stuck, even though if you want to nitpick it is not entirely the correct use of words.)

<u>Pay attention now, this trick is important, even if it may not seem like it!</u>

Say we have a y solution to the quadratic congruence mod 3, 5,7 and 11. By rebasing first to 0 we can quickly combine these 4 solutions to mod 1155.

Example of y solutions:

(note: this basically represents a system of quadratic congruences)

y = 1 mod 3
y = 3 mod 5
y = 1 mod 7
y = 5 mod 11

Knowing the "total" y value increase by two each time we decrease the modulus, we add twice the modulus to Y to rebase to 0.

Example of "rebased to 0" y solutions (we do not apply modular reduction to the result):

(note: See rebase in factor.sage)

3: y = 1+6 = 7
5: y = 3+10 = 13
7: y = 1+14 = 15
11: y = 5+22 = 27

b. Raising the modulus

Now that we have rebased the y solutions to 0, we can quickly raise them to mod 1155 (3*5*7*11) by iteratively multiplying the rebased y by every other prime and their inverse in the accumulated modulus.

I will refer to raising to a common modulus as raising to a "shared modulus" in this chapter.

(note: See create_partial_results in factor.sage)

mod 3 raised to 1155

*inverse of 5 in mod 3 = 2 => 7*2*5 = 70 mod 1155*
*inverse of 7 in mod 15 = 13 => 70*13*7 = 595 mod 1155*
*inverse of 11 in mod 105 = 86 => 595*86*11 = 385 mod 1155*

thus 7 raised to mod 1155 = 385

mod 5 raised to 1155

*inverse of 3 in mod 5 = 2 => 13*2*3 = 78 mod 1155*
*inverse of 7 in mod 15 = 13 => 78*13*7 = 168 mod 1155*
*inverse of 11 in mod 105 = 86 => 168*86*11 = 693 mod 1155*

thus 13 raised to mod 1155 = 693

mod 7 raised to 1155

*inverse of 3 in mod 7 = 5 => 15*5*3 = 225 mod 1155*
*inverse of 5 in mod 21 = 17 => 225*17*5 = 645 mod 1155*
*inverse of 11 in mod 105 = 86 => 645*86*11 = 330 mod 1155*

thus 15 raised to mod 1155 = 330

mod 11 raised to 1155

*inverse of 3 in mod 11 = 4 => 27*4*3 = 324 mod 1155*
*inverse of 5 in mod 33 = 20 => 324*20*5 = 60 mod 1155*
*inverse of 7 in mod 165 = 118 => 60*118*7 = 385 mod 1155*

thus 27 raised to mod 1155 = 1050

Now adding up each result: = 385+693+330+1050 = 148 mod 1155

There we have it. These solutions in mod 3,5,7 and 11 rebased to 0 and raised to mod 1155 result in 148 mod 1155.

This is useful, because now we can raise y solutions in different moduli to a shared modulus, at almost no computational cost. We don't even need to combine them. Just generating the partial results (as in the above example: 385, 693, 330, 1050) is ideal to utilize in an algorithm like LLL. The "partial result" means it only encodes the solution for the modulus it was raised from, and basically adds a "0" solution (a.k.a is a multiple of) for the other moduli.

(note: My use of the term "partial result" is another one of my weird abstractions that will probably make no sense to real mathematicians, but I like my abstractions as it helps me to reason about complex topics I lack proper vocabulary for.)

Example 1: 385 = 1 mod 3, 385 = 0 mod 5, 385 = 0 mod 7 and 385 = 0 mod 11
Example 2: 693 = 0 mod 3, 693 = 3 mod 5, 693 = 0 mod 7 and 693 = 0 mod 11
Example 3: 330 = 0 mod 3, 330 = 0 mod 5, 330 = 1 mod 7 and 330 = 0 mod 11
Example 4: 1050 = 0 mod 3, 1050 = 0 mod 5, 1050 = 0 mod 7 and 1050 = 5 mod 11

<u>Quick reminder for the reader:</u>
Despite referring to these "y solutions" in isolation, all of this is still in the context of our quadratic congruence: x^2+yx+n = 0 mod m. Or system of quadratic congruences when talking about different moduli. All the tricks described here are only possible because of how that system of quadratic congruences behaves. Please keep this in mind, otherwise none of this will make sense.

Using p=41 and q=107 as an example, let us generate the partial results for moduli whom multiplied together are larger then n (4387), to illustrate something interesting.
Again, the partial results are generated by using y solutions for which there exists an x solution in our quadratic, then rebasing that y solution to 0 and raising to a shared modulus, in the example below the shared modulus is 3*7*13*19 (5187)

mod 3 = 1729, 3458 (rebased to 0 and raised from 1, 2 in mod 3)
mod 7 = 0, 4446, 741 (rebased to 0 and raised from 0, 1, 6 in mod 7)
mod 13 = 1197, 798, 1995, 3192, 4389, 3990 (rebased to 0 and raised from 1, 5, 6, 7, 8, 12 in mod 13)
mod 19 = 3003, 3822, 1638, 2457, 273, 4914, 2730, 3549, 1365, 2184 (rebased to 0 and raised from 1, 3, 4, 6, 7, 12, 13, 15, 16, 18 in mod 19)

From each modulus we "choose one partial result" and add them together to generate a possible y solution for the shared modulus (mod 5187).

Example 1: 3458 + 741 + 3990 + 2184 = 5186 mod 5187
Example 2: 1729 + 4446 +798 + 3549 = 148 mod 5187
Example 3: 1729 + 0 + 1197 + 3003 = 742 mod 5187

We know 148 is the correct solution (since we know p+q is 148), however, let us assume we do not know the factors. How can we find the correct combination here?

There are two strategies that come to mind.
If we keep growing the modulus, only 148 will remain 148. All others will grow to a value larger then n. Hence, the problem can be reduced to finding a combination that yields the smallest possible value, and if the modulus is large enough, this is almost guaranteed to be the correct solution.

This is called a variant on the sub-set sum problem. Or more specific, modular multiple-choice subset-sum problem where we try to find the minimum value.

I am however not aware of a way to solve a subset-sum type problem where the target value is not strictly defined with LLL but is instead defined as finding the smallest value.

Thus let us look at the second strategy:

Let us generate two different shared moduli from two different sets of moduli.

First shared modulus (mod 5187)

mod 3 = 1729, 3458 (from 1, 2 mod 3)
mod 7 = 0, 4446, 741 (from 0, 1, 6 mod 7)
mod 13 = 1197, 798, 1995, 3192, 4389, 3990 (from 1, 5, 6, 7, 8, 12 mod 13)
mod 19 = 3003, 3822, 1638, 2457, 273, 4914, 2730, 3549, 1365, 2184 (from 1, 3, 4, 6, 7, 12, 13, 15, 16, 18 mod 19)

Second shared modulus (mod 21505)

mod 5 = 8602, 12903 (from 2, 3 mod 5)
mod 11 = 13685, 5865, 3910, 17595, 15640, 7820 (from 1, 2, 5, 6, 9, 10 mod 11)
mod 17 = 0, 12650, 10120, 16445, 7590, 13915, 5060, 11385, 8855 (from 0, 2, 5, 6, 8, 9, 11, 12, 15 mod 17)
mod 23 = 0, 18700, 7480, 1870, 17765, 14960, 6545, 3740, 19635, 14025, 2805 (from 0, 1, 5, 7, 9, 10, 13, 14, 16, 18, 22 mod 23)

From mod 5187 we select: 1729+4446+798+3549 = 148 mod 5187
and from mod 21505 we select: 12903+3910+11385+14960 = 148 mod 21505

In both shared moduli we have a combination that results in 148.
I would like to hypothesize that the sum of our two factors is always going to be below the ceiling:

*(n^0.5)*2+(n^0.5)/2*

And if so, I would also like to hypothesize that if we construct multiple sufficiently large shared moduli, the chance of them sharing a common combination below the ceiling we just defined, is very unlikely.
Thus any combination we find in all sets of partial results (example: 148 is found in both mod 5187 and mod 21505) has a very high probability of being the correct one.
This reduces the problem to finding a combination of partial results which can be found in all shared moduli as well.

c. Subtracting partial results and reducing the modulus

First I will show how to subtract partial results from two different shared moduli. After this I will show you how to reduce two different shared moduli to the same modulus and then subtract the partial results from each other.

Let us say we have two sets of partial results (in mod 5187 and mod 21505):

1729+4446+798+3549 = 148 mod 5187
12903+3910+11385+14960 = 148 mod 21505

If we negate the results in mod 21505 instead to indicate subtraction:

1729+4446+798+3549 = 148 mod 5187
-12903-3910-11385-14960 = -148 mod 21505

=> 148-148 = 0.

If we do not have the same result (148 and -148) in both shared moduli, then subtracting these from each other will likely not result in 0.

This can work in LLL! We set the "target value" to 0. One shared modulus has positive partial results and another one negated partial results.
We must find a combination in both of them, such that subtracted from each other, we get 0.

Furthermore, if we want to find a result below a certain 'ceiling', we can reduce the shared moduli.
Let us reduce both mod 5187 and mod 21505 to mod 4387 if for demonstration's sake we take n as our ceiling.

1729+4446+798+3549 = 148 mod 5187
-12903-3910-11385-14960 = -148 mod 21505

Reduced to mod 4387 becomes:

1729+59+798+3549 = 1748 mod 4387
-4129-3910-2611-1799 = -3675 mod 4387

And we need to do one final adjustment:

5187 mod 4387 = 800 mod 4387
21505 mod 4387 = 3957 mod 4387

*=> 1748-i*800 = 148 mod 4387 (i=2)*
*=> -3675+i*3957 = -148 mod 4387 (i=2)*

And we arrive at the same result again, but in a different (smaller) modulus.

Let us circle back to Fermat's difference of squares.

y^2-4n = a^2

plugging in 148 for y and 4387 for n:

148^2-17548 = 66^2

What I noticed is that by squaring, we can reduce the number of y solutions by half.
This is a massive advantage as it lowers the density of the subset-sum problem.
Let us look at what happens when we square y solutions, and then compute the partial results.

Starting with the y solutions:

mod 5187:

mod 3 = 1, 2
mod 7 = 0, 1, 6
mod 13 = 1, 5, 6, 7, 8, 12
mod 19 = 1, 3, 4, 6, 7, 12, 13, 15, 16, 18

mod 21505:

mod 5 = 2, 3
mod 11 = 1, 2, 5, 6, 9, 10
mod 17 = 0, 2, 5, 6, 8, 9, 11, 12, 15
mod 23 = 0, 1, 5, 7, 9, 10, 13, 14, 16, 18, 22

Squaring each solution:

mod 5187:

mod 3 = 1,1
mod 7 = 0, 1, 1
mod 13 = 1, 12, 10, 10, 12, 1

mod 19 = 1, 9, 16, 17, 11, 11, 17, 16, 9, 1

mod 21505:

mod 5 = 4, 4
mod 11 = 1, 4, 3, 3, 4, 1
mod 17 = 0, 4, 8, 2, 13, 13, 2, 8, 4
mod 23 = 0, 1, 2, 3, 12, 8, 8, 12, 3, 2, 1

We can see the the amount of unique solutions is halved. Thus we need only create partial results for half of them.
Let us create the partial results for each unique squared Y solution (by rebasing to 0 and raising to a shared modulus):

mod 5187

mod 3 = 1729 (from 1 mod 5)
mod 7 = 0, 4446 (from 0,1 mod 7)
mod 13 = 1197, 3990, 1596 (from 1,12,10 mod 13)
mod 19 = 3003, 1092, 1365, 4368, 1911 (from 1,9,16,17,11 mod 19)

mod 21505

mod 5 = 17204 (from 4 mod 5)
mod 11 = 13685, 11730, 19550 (from 1,4,3 mod 11)
mod 17 = 0, 3795, 7590, 12650, 17710 (from 0,4,8,2,13 mod 17)
mod 23 = 0, 18700, 15895, 13090, 9350, 20570 (from 0,1,2,3,12,8 mod 23)

Before we were trying to find a combination resulting in 148, but since we squared everything, we are now looking for a combination resulting in 148**2 (mod 5187 and mod 21505).

In mod 5187: 1729+4446+3990+1365 = 1156 mod 5187 and 148^2 = 1156 mod 5187
In mod 21505: 17204+19550+7590+20570 = 399 mod 21505 and 148^2 = 399 mod 21505

However, since 148^2 is fairly large, there is a trick we can do. Remember how 148^2 - 4n = 66^2
So instead of looking for 148^2 we can look for 66^2 instead.
All we need to do is subtract 4n.

Let us simply subtract 4n from the partial results in mod 19 and the partial results in mod 23, which gives us:

mod 5187

mod 3 = 1729
mod 7 = 0, 4446
mod 13 = 1197, 3990, 1596
mod 19 = 3003-17548, 1092-17548, 1365-17548, 4368-17548, 1911-17548
=> mod 19 = 1016, 4292, 4565, 2381, 5111

mod 21505

mod 5 = 17204
mod 11 = 13685, 11730, 19550
mod 17 = 0, 3795, 7590, 12650, 17710
mod 23 = 0-17548, 18700-17548, 15895-17548, 13090-17548, 9350-17548, 20570-17548
=> mod 23 = 3957, 1152, 19852, 17047, 13307, 3022

If we calculate the (correct) combinations again.. we should have 66^2 mod 5187 and mod 21505 this time around:

In mod 5187: 1729+4446+3990+4565 = 4356 mod 5187 and 148^2 = 4356 mod 5187
In mod 21505: 17204+19550+7590+3022 = 4356 mod 21505 and 148^2 = 4356 mod 21505

Much better!

There is one more improvement I will leave for the reader. Instead of 66 and 148, we can divide both by two and look for the squares of 33 and 74 instead (in the modular world dividing by 2 is done by multiplying with the inverse of 2). This should increase performance of our algorithm slightly, since we're dealing with even smaller values, thus can work with smaller constraints/ceiling.

IV. From multiple choice subset-sum problem to LLL

a. The index and split column

Using all the transformations from above, lets get to the real heart of the algorithm. LLL!

Let us say we want to subtract partial results in mod 21505 from partial results in mod 5187, we would list both partial results together in a column in our LLL matrix.

We have to select 4 from the positives and 4 from the negatives (since both 5187 and 21505 are created from 4 different prime moduli). But how to make sure it selects *at-least* one from each subset? (note: subset = solutions within a single prime modulus)

Solution: We can add an index column like this (the correct combination of rows is in bold)

3	***1729***
7	*0*
7	***4446***
13	*1197*
13	***3990***
13	*1596*
19	*1016*
19	*4292*
19	***4565***
19	*2381*
19	*5111*
5	***-17204***
11	*-13685*
11	*-11730*
11	***-19550***
17	*-0*
17	*-3795*
17	***-7590***
17	*-12650*
17	*-17710*
23	*-3957*
23	*-1152*
23	*-19852*
23	*-17047*
23	*-13307*
23	***-3022***

0	***+25105*** *(to simulate mod 25105)*
0	***-5187*** *(to simulate mod 5187)*

----

target=98	*0*

The target of the index column is simply: 3+7+13+19+5+11+17+23 = 98
Since the index column has to sum up to 98, we will have a high probably that it will select at-least one from each subset.

b. The mask column

Another issue is the following scenario: What if it selects 2 from mod 23 and subtracts one from mod 23? In the index column this would result in a single 23. I came up with the following solution:

2	1729
30	0
300	4446
55000	1197
505000	3990
550000	1596
7777000000	1016
70777000000	4292
77077000000	4565
77707000000	2381
77770000000	5111
1100000000000	-17204
13130000000000000	-13685
1300130000000000000	-11730
1313000000000000000	-19550
171717170000000000000000000	0
17001717170000000000000000000	-3795
17170017170000000000000000000	-7590
17171700170000000000000000000	-12650
17171717000000000000000000000	-17710
191919191900000000000000000000000000000	-3957
19001919191900000000000000000000000000000	-1152
19190019191900000000000000000000000000000	-19852
19191900191900000000000000000000000000000	-17047
19191919001900000000000000000000000000000	-13307
19191919190000000000000000000000000000000	-3022
-2	0
-30	0
-300	0
-5000	0
-50000	0
-500000	0
-7000000	0
-70000000	0
-700000000	0
-7000000000	0
-70000000000	0
-1100000000000	0
-130000000000000	0
-13000000000000000	0
-1300000000000000000	0
-170000000000000000000	0
-17000000000000000000000	0
-1700000000000000000000000	0
-170000000000000000000000000	0
-17000000000000000000000000000	0
-1900000000000000000000000000000	0
-190000000000000000000000000000000	0
-19000000000000000000000000000000000	0
-1900000000000000000000000000000000000	0
-190000000000000000000000000000000000000	0
-19000000000000000000000000000000000000000	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	21505
0	-5187
0	0

Adding a pattern like this will discourage the selection of multiple rows from the same subset.

Let us see why this works:

```
                23      0222
                23      2022
                23      2202
                23      2220
                0       -2
                0       -20
                0       -200
                0       -2000
-------------------------------
target=         23      0
```

If we select two rows and subtract one row, no matter in what order we do it, we always end up with a "4".

Example:

```
 0222
+2022
----------
=2244
-2220
----------
=0024
```

The only way to get rid of 0024 is by -20 once and selecting -2 twice.
However, if we set the constraint so that it will select -2 (or any of the others) at most once, we can hopefully prevent multiple rows from the same subset being selected. From testing, this does seem to help. The only downside being that it doubles the amount of rows, thus making everything much slower.

I will refer to this column as the "mask" column.
We can also discourage a row from being subtracted by zeroing out the mask column for that row, as that will result in even more "4s" should multiple rows end up getting selected in the same subset. Or discourage selection entirely by adding some bogus number like 1337, which can not be reduced to 0 using only -2,-20,-200,-200 (etc).

So far we have covered three column types, the mask, the index and the column where we subtract the two partial results, this last one I will refer to as the "split column".

There is one more column we can add to improve performance, the "aid" column

c. The aid column

Update: I found out a day before release that completely removing these aid columns from my matrix actually improves performance massively, but I will leave it in the paper and it can be enabled via setting if you must. We're still pre-computing combinations though, as it will help to reduce the density regardless.

Since we have multiple CPU cores available, it would not be a bad idea to pre-calculate possible combinations in advance, to further reduces the items that can be selected, reduce the density and create multiple instances running in parallel. We don't need to combine this with aid columns, without may also work if we simply reduce the density by using pre-computed combinations. But it is another tool in the box the reader can mess around with.

Let us say we want to pre-calculate all possible combinations in mod 3, mod 5 and mod 7.
We can use the same tools from chapter III.

Calculating every y solution that solves our quadratic congruence for 0 in mod 3,5 and 7 we get:

mod 3 = 1, 2
mod 5 = 2, 3
mod 7 = 0, 1, 6

Next we square them and remove duplicates:

mod 3 = 1
mod 5 = 4
mod 7 = 0, 1

Next rebase each solution to 0:

mod 3 = 7
mod 5 = 14
mod 7 = 14, 15

Next create partial results by raising to the shared modulus (3*5*7 = 105):

mod 3 = 70
mod 5 = 84
mod 7 = 0, 15

Since we are now working with 148^2 but we want to work 66^2 instead, subtract 4n from mod 7

mod 3 = 70
mod 5 = 84
mod 7 = 0-17548, 15-17548 mod 105
=> mod 7 = 92, 2

Next we simply find each combination, selecting one from each subset, in this case we only have two combinations:

70+84+92 = 36 mod 105
70+84+2 = 51 mod 105 (note 66^2 = 51 mod 105)

Having 36 and 51 pre-calculated, we can create two instances of our algorithm. One will match the solutions in mod 3,5 and 7 to 36 and another will match the solutions in mod 3,5,7 to 51. This means we can delete the solutions that don't match, and reduce the density. Thus the more CPU cores we have, the more we can reduce the density of our subset sum problem.

Let us look at the instance running against the pre-calculated combination 51. This means we need to find a combination which reduced to mod 105 results in 51. This is how we would add two aid columns (one for mod 5187 and mod 21505):

1729	0
4446	0
1197	0
3990	0
1596	0
1016	0
4292	0
4565	0
2381	0
5111	0
0	17204
0	13685
0	11730
0	19550
0	0
0	3795
0	7590
0	12650
0	17710
0	3957
0	1152
0	19852
0	17047
0	13307
0	3022
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
-5187	0
0	-21505
0	0
0	0
-105	-105
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0
51	51

We can also merge these two columns into one by raising all partial results to mod 5187*21505.

For example partial result 1729 in mod 5187 raised to 5187*21505 is calculated like this:

Step 1. Take the inverse of 21505 in 5187 = 3844
*Step 2. Multiply 1729 by the inverse = 1729*3844 = 6646276 mod 111546435*
*Step 3. Multiply 6646276 by 21505 = 6646276*21505 = 37182145 mod 111546435*

We can do this for every partial result and instead of two separate columns, we now have a single column:

```
 37182145
 95611230
  8580495
102965940
 85804950
104944400
 81460940
  5139695
 57977480
 75590075
 44618574
 30421755
 10140585
 91265265
        0
 65615550
 19684665
 32807775
 45930885
 59422272
 30323202
  1224132
 83671497
 44872737
 49722582
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
-111546435
        0
        0
        0
        0
     -105
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
       51
```

I have not been able to determined what performs best. Simply adding the merged column, or adding two un-merged columns, or adding the merged column and two un-merged columns. I will leave this up to the reader for experimentation.

To lower the search range / ceiling we can also reduce the modulus as described in the chapter 3. I will not go into the calculation again, but it will look like this:

```
  3035297
 10244110
  8580495
   525396
   437830
  2503856
 13167244
  5139695
  6757208
  7296379
 10471726
 13348331
 10140585
  5898145
        0
 14395278
  2611241
 15734351
 11784037
  8202000
 13249778
  1224132
 15377801
 10725889
 15575734
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
-17073424
 -9105891
     -105
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
       51
```

In the picture above, we are showing a merged aid column, reduced to 4386*4386 (19236996) whereas the un-reduced merged aid column before was 5187*21505 (111546435). All these calculations are also performed in my proof of concept should you need more clarification.

As a side note, when pre-computing combinations and creating different instances over multiple CPUs, any of those instances can end up finding the correct solution, not just the correct pre-computed combination. I suspect this happens when we take the square root of the result from the LLL matrix and round it. It may also happen if the pre-computed combination matches at-least some of the solutions, which if true, would make the algorithm a little more flexible and useful, and the cores not running on the incorrect combinations wouldn't be a complete waste.

Finally another important component is setting the constraints (the values in the diagonal of the matrix):

```
      0          0          0          0      0
1/16384          0          0          0      0
      0    1/16384          0          0      0
      0          0    1/16384          0      0
      0          0          0    1/16384      0
      0          0          0          0   1/26
      0          0          0          0      0
      0          0          0          0      0
      0          0          0          0      0
      0          0          0          0      0
      0          0          0          0      0
      0          0          0          0      0
      0          0          0          0      0
    1/2        1/2        1/2        1/2    1/2
```

I have tried to optimize this in the code, but this is one area to improve performance of the algorithm. I do not have a whole lot of experience in lattice reduction, so finding the optimal constraint values has been more trail and error than science.

At a larger level, the algorithm will basically start with small constraints and then increase the constraints until it find the correct solution. It will additional also add more columns with reduced modulus, as from testing I have found this helps significantly. Furthermore, if it does end up subtracting a row, we can use the mask column to try and discourage subtraction of that particular row.

(Note: Instead of using the strategy I am using, something like bkz reduction may make more sense.)

A full lattice matrix without aid columns with two split columns where one has a reduced modulus would look like this (some of the left side is missing, but the diagonal just continues until it reaches the top row):

```
  0     0     0     0     0     0     0     0     0     0  4096                  2   2948   2948]
  0     0     0     0     0     0     0     0     0     0     3                 30  -5005   -745]
  0     0     0     0     0     0     0     0     0     0     5                500  -9009   -489]
  0     0     0     0     0     0     0     0     0     0     7               7000      0      0]
  0     0     0     0     0     0     0     0     0     0     7              70000 -10725  -2205]
  0     0     0     0     0     0     0     0     0     0    11          111100000  -1365  -1365]
  0     0     0     0     0     0     0     0     0     0    11        11001100000  -5460  -1200]
  0     0     0     0     0     0     0     0     0     0    11        11110000000  -4095  -4095]
  0     0     0     0     0     0     0     0     0     0    13    131300000000000  -4397   -137]
  0     0     0     0     0     0     0     0     0     0    13  13001300000000000  -5552  -1292]
  0     0     0     0     0     0     0     0     0     0    13  13130000000000000  -6707  -2447]
  0     0     0     0     0     0     0     0     0     0     0                 -2      0      0]
  0     0     0     0     0     0     0     0     0     0     0                -30      0      0]
  0     0     0     0     0     0     0     0     0     0     0               -500      0      0]
  0     0     0     0     0     0     0     0     0     0     0              -7000      0      0]
  0     0     0     0     0     0     0     0     0     0     0             -70000      0      0]
  0     0     0     0     0     0     0     0     0     0     0           -1100000      0      0]
  0     0     0     0     0     0     0     0     0     0     0         -110000000      0      0]
  0     0     0     0     0     0     0     0     0     0     0       -11000000000      0      0]
  0     0     0     0     0     0     0     0     0     0     0     -1300000000000      0      0]
  0     0     0     0     0     0     0     0     0     0     0   -130000000000000      0      0]
  0     0     0     0     0     0     0     0     0     0     0 -13000000000000000      0      0]
  1     0     0     0     0     0     0     0     0     0     0                  0      0      0]
  0 1/832     0     0     0     0     0     0     0     0     0                  0  15015      0]
  0     0 1/832     0     0     0     0     0     0     0     0                  0  -4096      0]
  0     0     0     1     0     0     0     0     0     0     0                  0      0      0]
  0     0     0     0     1     0     0     0     0     0     0                  0      0      0]
  0     0     0     0     0 1/832     0     0     0     0     0                  0      0   4260]
  0     0     0     0     0     0 1/832     0     0     0     0                  0      0  -4260]
  0     0     0     0     0     0     0 1/832     0     0     0                  0      0   2235]
  0     0     0     0     0     0     0     0 1/832     0     0                  0      0  -4096]
1/2   1/2   1/2   1/2   1/2   1/2   1/2   1/2   1/2   1/2  4135                  0      0      0]
```

There is still a lot of room for optimization, but in the end I came to the conclusion, if I really want to get the most out of this, I need to spent a significant time studying various math topics first and approach it more like a scientist. The things I was able to get away with as a software vulnerability researcher, I cannot get away with here. I believe I have reached a point that pure determination cannot overcome, and only time investment and lots of studying will. But as I have been unemployed for a year, time has ran out sadly.

V. Proof of Concept read-me and performance

(note: The key generation may not calculate correct public and private exponents, I have not checked, but as it is not relevant to factorization, I am not going to waste time on it)

In general, the most important parameters are cores, *aidlen* and the *lift* related settings. These will determine how many combinations are pre computed and over how many cores they are spread and aside from messing with the constraints and the settings related to *deleting partial results*, you probably shouldn't change much. On a general consumer machine this will perform poorly, but with enough cpu cores, it should eventually overtake many of the other factorization algorithms, or at-least in theory. Ideally you want about 1 or 2 pre-computed combinations per core, especially because at larger bit-lengths, the LLL instances will take quite some time to finish. Either way, a proper BKZ reduction algorithm will likely perform much better then what I am doing, and this is just a proof of concept, nothing more.

Overview of the settings

(note: some of these can be set with command line (type: -?) others can be changed in factor.sage)

General Settings

method: Algorithm to use.
"LLL" Factors using LLL. Slower then "Simple" for small keys but has better theoretical scaling.
"Simple" Simply iterates over every y solution less then mod (n^0.5)*2+(n^0.5)/2
Default: "LLL" (str)
strat: Strategy to use.
0 for unsquared y solutions.
1 for squared.
Default: 1 (int)
key: Define a custom key to factor instead of generation one. Default: 0 (int)
keysize: Define a key size for a key to be generated. This refers to the bitlength in the modulus not exponent Default: 20 (int)
aidlen: Define an amount of primes from which we should generate precomputed combinations. Primes are chosen starting from 2. The amount of combinations automatically gets spread of the amount of cores available (defined by cores). Increases the likelyhood of finding the corret solution but also requires more cores to be effective Default: 7 (int)
cores: Amount of CPU to use. Default: 16 (int)
mode: Ceiling for the reduced modulus.
2 (n^0.5)*2+(n^0.5)/2 (fastest but if using strat 1, the constraint in g_cmult needs to start high enough, as the moduli will need to be subtracted/added more)
1 (n^0.5)*(n^0.5)/2 (normal)
0 n (slow).
-1 Use a custom ceiling (use in combination with -custom_mode)
Default: 1 (int)

note: You can added your own limits to the code, the smaller they are, the faster the LLL instances will run, but the looser the constraints have to be. It may be interesting to see what you can get away with and still reliably find the factors.

custom_mode: Define your own ceiling Default: 10000 (int)

Algorithm parameters

Constraint specific:

g_cmult: Constraint used for the moduli in the matrix. Default: 16 (int)
g_cmult_max: Stop algorithm once we get to this value. Default: 200 (int)
g_cmult_inc: Increase g_cmult by this amount each iteration. Default: 8 (int)
g_climit: Amount of columns using reduced moduli. Default: 5 (int)
g_climit_max: Increase g_cmult and reset once we get to this value. Default: 30 (int)
g_climit_inc: Increase g_climit by this amount each iteration. Default: 4 (int)
g_constr_min: Constraint values for the aid column. Default: 0 (int)
g_constr_max: Increase g_climit and rest once we get to this value. Default: 4 (int)

Partial result deletion specific:

g_maxdepth: Amount of iterations after deleting subtracted partial results defined by g_deletion amount. Default: 10 (int)
g_deletion_amount: Amount of subtracted partial results to delete each iteration. Default: 20 (int)
g_deletion_mode: Deletion strategy.
- **0** Delete only subtracted partial results when both shared moduli have the same combination
- **1** Favor the above but when not available delete anyway
- Default: 1 (int)

Lifting settings:

g_liftlimit: Attempt to lift any solutions for primes below this value. Default:3 (int)
g_lift_for_2: Attempt to lift 16 by this amount. Default: 8 (int)
g_lift_for_3: Attempt to lift 3 by this amount. Default: 2 (int)
g_lift_for_others: Attempt to lift primes, excluding 2 and 3 by this amount. Default: 2 (int)
g_lift_threshold: Defines the density threshold for when lifting should occure. Default: 3 (int)

Other:

g_include_unmerged_col:
- **1** To include unmerged aid columns.
- **0** Only include merged aid columns.
- Default: 0 (int)

g_mod_red_amount: Defines a multiplier for how strongly we increase or decrease the modulus with each round of g_climit. Default: 6 (int)
g_merged_aid_list: Strategies for the aid columns
- **0** Only use the prime moduli
- **1** Only use the shared modulus
- **2** Use both
- Default: 2 (int)

g_use_aid_cols: Use aid columns
- **0** Don't use aid columns (recommended, they don't help)
- **1** Use aid columns
- Default: 0 (int)

Debug settings

show: Whether or not to print the LLL matrix
- **0** Do not print
- **1** Show the complete input matrix
- **2** Show the complete input and output matrix
- **-1** Show a truncated input matrix (define printcols)
- **-2** Show a truncated input and output matrix (define printcols)
- Default: 0 (int)

printcols: Amount of columns to print when -show is in truncated mode. Default: 60 (int)
debug: Show debug output
- **0** Do not show
- **1** Show
- Default: 0 (int)

g_enable_custom_factors: Enable the use of custom factors
- **1** Enable
- **0** Disable

Default: 0 (int)
g_p: Define custom factor p (int)
g_q: Define custom factor q (int)

Do not change

upperweight: Don't touch, it will break the PoC without changing values in init_matrix and runLLL aswell. Default: 1 (int)
scalar: Scalar for all the matrix weights. Reduce if using the show setting to print the matrix. Default: 10000000000 (int)

Performance

Benchmarks using the default settings (note: they can be optimized for specific bit lengths for faster factorization):

Reproduce by inputting the modulus in -key argument or defining custom factors (example: sage factor.sage -key 689083)

Modulus size: 20
sage factor.sage -key 689083
Prime p: 701
Prime q: 983
Modulus (p*q): 689083
Factorization took: 5.118209427993861 (seconds)

Modulus size: 22
sage factor.sage -key 2660443
Prime p: 1831
Prime q: 1453
Modulus (p*q): 2660443
Factorization took: 1.1144341789913597 (seconds)

Modulus size: 24
sage factor.sage -key 10828877
Prime p: 2647
Prime q: 4091
Modulus (p*q): 10828877
Factorization took: 2.121110129999579 (seconds)

Modulus size: 26
sage factor.sage -key 39501379
Prime p: 6871
Prime q: 5749
Modulus (p*q): 39501379
Factorization took: 3.12278330999834 (seconds)

Modulus size: 27
sage factor.sage -key 94065857
Prime p: 8389
Prime q: 11213
Modulus (p*q): 94065857
Factorization took: 3.1465786379994825 (seconds)

Modulus size: 30
sage factor.sage -key 684159461
Prime p: 21031
Prime q: 32531
Modulus (p*q): 684159461
Factorization took: 4.123098841999308 (seconds)

Modulus size: 32
sage factor.sage -key 3554547547
Prime p: 57641
Prime q: 61667
Modulus (p*q): 3554547547

Factorization took: 7.137495260001742 (seconds)

Modulus size: 34
sage factor.sage -key 10312953307
Prime p: 125711
Prime q: 82037
Modulus (p*q): 10312953307
Factorization took: 32.1541292139882 (seconds)

Modulus size: 36
sage factor.sage -key 43724277749
Prime p: 213887
Prime q: 204427
Modulus (p*q): 43724277749
Factorization took: 93.26531726100075 (seconds)

Modulus size: 38
sage factor.sage -key 172889774297
Prime p: 402527
Prime q: 429511
Modulus (p*q): 172889774297
Factorization took: 75.21932730299886 (seconds)

Beyond this point you need to increase the *aidlen* and *lifting* settings to generate more combinations and run the PoC on more cores for reliable results. Rewriting the PoC in c++ will also help massively too.

While the factorization time on small bit lengths is not very impressive, the idea is that the more cores you have, the more combinations you can pre-calculate to reduce the density of the sub-set sum problem and spread over multiple cores.
From testing, it seems to scale strongly logarithmically (but I lack the experience in lattice reduction to make a proper complexity analysis).
The amount of pre-calculated combinations to be guaranteed the correct solution without too many iterations does not scale exponentially but sub-exponentially. If all is optimized, it may even be possible to achieve factorization of very high bit-lengths, given that you have plenty of CPU cores available to run the algorithm. The whole idea is to use LLL because it has far superior scaling then anything else. But the initial scaling is very steep.

As a final note on this, if we set our ceiling to be n, then of-course, as the bit-length increase, we need to multiply more primes together, which means more partial results to add to the LLL matrix, and the slowdown that comes with that. But that should still scale vastly better then any of the near exponential scaling in traditional algorithms. The trade-off is that those traditional algorithms are also way faster for smaller bit-lengths. This algorithm needs a good amount of CPU cores to perform well, at least that is my hypothesis as an uneducated amateur.

VI. Closing thoughts

I think this was a good first math project and introduction to math in general. When I started, I didn't know basic algebra. I do admit that I spent many months at a time trying to figure out basic number theoretical concepts that I could have just read about in a book, but I prefer not to see it as lost time.
I wish I would have had more time to work this angle, but I have been unemployed and basically broke for a year now. I do feel somewhat disappointed, as it feels I have not achieved anything novel or of significance. And in retrospect, my stubbornness to figure things out myself instead of properly learning math from books and doing exercises, resulted in just months being wasted on trivial stuff.

I feel like I'm trying to reach for something, that's just behind the corner, and I feel an insane compulsion to keep trying to reach for it.. a vital insight just out of sight, that will solve everything, but alas, time has ran out.
I regret dropping out of high-school, perhaps in another life things would have turned out differently. But it is as it is. By the time you read this, I will most likely be on my way to the Arctic, or already be there. One more great adventure.

VII. References

(note: Please e-mail me if you feel like I have forgotten to include a reference. It is important to me to give credit where credit is due as I know very well what it feels like when people don't. Due to my limitations in math education I was not able to make much sense of most papers on factorization, however it is very likely collisions may have occured so feel free to point these out.)

-An Illustrated Theory of Numbers - Martin H. Weissman
-Prime Numbers A Computational Perspective - Richard Crandall , Carl Pomerance
-An Introduction to the theory of numbers - Ivan Niven, Herber S. Zuckerman, Hugh L. Montgomery
-Cryptography Lecture Series - Christof Paar:
https://www.youtube.com/playlist?list=PL2jrku-ebl3H50FiEPr4erSJiJHURM9BX
-Subset Sum from lattice reduction - Alex Xiong:
https://hackmd.io/@alxiong/ssp-from-lll
-Solving Modular SubSet Sum with LLL:
https://github.com/dxt99/LLL-Modular-Subset-Sum
-Tonelli-Shanks in python:
https://github.com/anonylouis/404CTF-2023---Write-ups/blob/cebae3f05f7ac3542601a63868fa7be8f77c2758/Cryptanalyse/La_ou_les_nombres_n_existent_pas/solve_quadratic_congruence.py
-Creating RSA keys:
http://www.alljchome.com/index/jc/lid/1178/id/15990