Lower Bounds for Straight Line Factoring

Daniel R. L. Brown[*]

May 31, 2010

Abstract

Straight line factoring algorithms include a variant Lenstra's elliptic curve method. This note proves lower bounds on the length of straight line factoring algorithms.

Key Words: Factoring, Straight Line Programs.

1 Introduction

Straight line factoring algorithms include a variant Lenstra's elliptic curve method. This note proves lower bounds on the length of straight line factoring algorithms.

1.1 Summary of this Work

A straight line program P is essentially a fixed sequence of ring operations $+$, $-$, and $\times$. A straight line factoring algorithm is specified by a straight line program P. On input n to be factored, it does the following two steps

1. It evaluates P in the ring $\mathbb{Z}/(n)$ of integers modulo n to get a result value r.

2. It computes $\gcd(r, n)$ (using the extended Euclidean algorithm, for example).

If $\gcd(r, n) \neq 1, n$, then the straight line factoring algorithm is deemed to have successfully factored n.

Note that P does not depend n. So, straight line factoring algorithms cannot adjust P to depend on n. Typically, a single straight line factoring algorithm will target a finite set of n, such that the set of all n consist of a product of two primes of some bit length s.

Although this class of algorithms may appear severely limited, and thereby insignificant, we claim that it includes variants of Lenstra's elliptic curve method (ECM), with similar performance to Lenstra's ECM. For certain ranges of n, Lenstra's ECM is the fastest known algorithm. Furthermore, some implementations of the fastest known algorithm for n beyond a certain size, the General Number Field Sieve (GNFS), make use of Lenstra's ECM as an intermediate step. For these two reasons, we argue that the class of straight line factoring algorithms is a significant class.

This note proves two lower bounds on the number of steps in a straight line program is successful to factor randomly chosen RSA keys. The first lower bound is abstract in the sense it depends on

[*]Certicom Research

an infinite set of very large integers and the concept of a length of integer, which can be difficult to determine even for much smaller integers than those in the set. The second lower bound, derived from the first, is much more concrete, but too weak to provide sufficient assurance for cryptography. The second lower bound says that the straight line program has at least a number of steps that is about equal to the number of bits in the prime factors of the RSA modulus to be factor.

Future research may improve these results. Stronger lower bounds on the lengths of integers, especially compared to their factorization may help. Better understanding the aforementioned sets of integers may help. Generalizations to a broader class of algorithms, with division, randomization, and even forms of branching, may be also be worthwhile, but may be somewhat moot unless weak concrete lower bounds for the simpler case of straight line factoring can be improved substantially.

1.2 Previous Work

Riesel [Rie94] describes the Pollard rho factoring method in his *algebraic model*. This algebraic model is essentially the class of straight line factoring algorithms. Riesel also noticed that such algorithms may be described as simply computing a huge integer modulo the number to be factored. On page 223 and 224, Riesel states

> Summarizing this line of thought: Pollard's rho method may be regarded as a technique for generating huge integers, which, after i steps, contain all the small primes up to some limit Ci^2 as factors, and the amount of work needed to find a specific factor of p of N is therefore $O(\sqrt{p})$ steps. An "ideal" prime generator could, after i steps, contain the primes up to about Cr^i and would thus require only $O(\ln p)$ steps, with each step performing in $O(\log N)^{k+1}$ seconds, in order to identify p as a factor of N. This corresponds to polynomial time performance.
>
> Even if such an ideal prime generator cannot be constructed, the enormous gap between the orders of magnitude $\sqrt{p}$ and $(\log p)(\log N)^{k+1}$ means that there is certainly much room for improvement of existing algorithms. Ultimately, the ideal could be approximated so closely that in practice, a factorization algorithm performing in nearly polynomial would be achieved.

Two interpretations of the Riesel's paragraphs are as follows:

1. Riesel is conjecturing that there exists a straight line factoring algorithm with a linear number of steps. This is supported by both his quoting the word *ideal* and conditioning the construction of such an ideal thing. Also, the word *only* suggests that an upper bound on the number is being described.

2. Riesel is stating a lower bound on the number of steps in a straight line factoring algorithm, thereby anticipating the result of this paper. Evidence against interpretation is the word *only*, which should have been *at least* if a lower bound were being described.

Even under the interpretation that Riesel is stating a lower bound, no formal proof of such a lower bound is evident.

Note that Riesel does not identify Lenstra's ECM as subject to his algebraic model. Perhaps this is why his algebraic model did not receive as much attention as it deserved.

For a brief survey of other work related to lower bounds, straight line programs and factorization—none mentioning any lower bounds on factoring—see Appendix A.

2 Definitions

Definition 1. *A straight line program P of length L is a sequence $(\sigma_1, \ldots, \sigma_L)$ of steps of the form*

$$\sigma_k = (i_k, \circ_k, j_k) \tag{1}$$

with operation $\circ_k \in \{+, -, \times\}$ and integer indices $0 \leqslant i_k, j_k < k$.

Definition 2. *A straight line program P computes an integer $P()$ as follows. Let $x_0 = 1$. Let*

$$x_k = x_{i_k} \circ_k x_{j_k}. \tag{2}$$

Let $P() = x_L$.

Lemma 1. *If P is a straight line program of length L, then $P() \bmod n$ can be computed using L arithmetic operations modulo n.*

Proof. Evaluate $x_k \bmod n$ using (2) and a single arithmetic operation modulo n. □

Note that the naive method of computing $P()$ as integer first, using L integer arithmetic operations, then reducing it modulo n, may be greatly exceed the cost of the method provided by Lemma 1, because as the integers x_k grow in size, so does the cost of each integer arithmetic operations upon them.

Definition 3. *A straight line factoring algorithm F is specified by a straight line program P. On input of integer n, algorithm A does the following:*

1. *It evaluates $r = P() \bmod n$, using Lemma 1.*
2. *It evaluates $f = \gcd(r, n)$, using the extended Euclidean algorithm.*

If $f \neq 1, n$, then F has succeeded.

Note that $f = \gcd(P(), n)$.

Definition 4. *Let n be a random variable, taking integer values. Let F be a straight line factoring algorithm. The success rate μ of F for factoring F is the probability that F succeeds when its given input is from the random variable n.*

3 Straight Line Factoring Algorithms

3.1 Trial division

This section describes a straight line variant of the trial division factoring algorithm.

Definition 5. *Let $m \geqslant 3$ be integer. Let $L = 2m - 3$. Let P be the straight line program of length L with steps*

$$\begin{aligned}
\sigma_1 &= (0, +, 0), \\
\sigma_2 &= (1, +, 0), \\
\sigma_{2k-4} &= (2k-6, +, 0), \qquad \forall 4 \leqslant k \leqslant m, \\
\sigma_{2k-3} &= (2k-4, \times, 2k-5), \qquad \forall 3 \geqslant k \leqslant m.
\end{aligned}$$

Lemma 2. $P() = m!$.

Proof. By definition, $x_0 = 1$. Since $\circ_1 = +$, we see $x_1 = 1 + 1 = 2$. Since $\sigma_2 = (1, +, 0)$, we see $x_2 = x_1 + x_0 = 3$.

We claim that $x_{2k-4} = k$ for all $k \geqslant 3$, by induction. The base of induction is $k = 3$, was established above: $x_{2(3)-4} = x_2 = 3$. For $k \geqslant 4$, we have $x_{2k-4} = x_{2k-6} + x_0$ because $\sigma_{2k-4} = (2k - 6, +, 0)$. Now $x_{2k-6} = x_{2(k-1)-4}$, which, by inductive hypothesis, equals $k - 1$. Since $x_0 = 1$, we see $x_{2k-4} = (k - 1) + 1 = k$.

We next claim that $x_{2k-3} = k!$ for all $k \geqslant 2$. The base case of induction is $k = 2$, which follows from $x_{2(2)-3} = x_1 = 2 = 2!$, as shown above. For $k \geqslant 3$, we have $x_{2k-3} = x_{2k-4}x_{2k-5}$ because $\sigma_{2k-3} = (2k - 4, \times, 2k - 5)$. We already established above that $x_{2k-4} = k$. Now $x_{2k-5} = x_{2(k-1)-3}$, so, by induction, we have $x_{2k-5} = (k - 1)!$. Therefore, $x_{2k-4} = k \times (k - 1)! = k!$.

In particular, $P() = x_L = x_{2m-3} = m!$. □

Lemma 3 (Heuristic)**.** *If m is sufficiently large, and n is an integer random variable uniformly distributed between 1 and m^2, then the success rate μ of the straight line program specified by P is approximately* $\log 2$.

Sketch. A heuristic result of Dickman states that the number of positive integers below x with no prime factors larger $x^{1/a}$ is approximately $x\rho(a)$ where $\rho(a)$ is the Dickman function.

Taking $x = m^2$ and $a = 2$, one can see that $\rho(2) = 1 - \log 2$. Therefore, the probability that n has a prime factor larger than m is approximately $\log 2$.

For such integers n, we have $\gcd(m!, n) < n$, because n has prime factor $p > m$, so that $p \nmid m!$. If n is composite, then we will also have $\gcd(m!, n) > 1$, since any composite n must have a prime factor between 2 and $\sqrt{n} < m$.

The probability that n is prime is, by the prime number theorem, at most approximately $1/(2\log(m))$, which for sufficiently large m is negligible compared to $\log 2$. □

Other straight line variants of the trial division algorithm are possible. For example, instead of computing $m!$, the program P can compute the product of all primes $\leqslant m$. This can be obtained from the program above by insertion of some extra addition steps, and removal of multiplication. Actual construction of such a program is itself a slightly difficult problem. In this paper, however, we will ignore the difficulty associated with construction of straight line programs, and only consider the overall length of the straight line program.

3.2 Pollard's $p - 1$ factoring algorithm

Pollard's $p - 1$ factoring algorithm is essentially straight line factoring algorithm. Its core consists of computing $f = \gcd(a^M - 1, n)$ for some small integer a, such as $a = 2$, and some large integer M which is also product of small factors. If $n = pq$ for primes p and q and $p - 1|M$ but $a^M \not\equiv 1 \bmod q$, then $f = p$.

Of course, Pollard's $p - 1$ algorithm does not compute $a^M - 1$ fully as an integer, but rather only modulo n, as $a^M - 1$. This may be done by a straight line program.

For example putting $M = m!$ and $a = 2$, then the following straight line program P compute $2^M - 1$ in the following $L = 2 + \binom{m}{2}$ steps:

$$\begin{aligned} \sigma_1 &= (0, +, 0) \\ \sigma_k &= (k-1, \times, t(k)+1), \qquad 2 \leqslant k < L \\ \sigma_L &= (L-1, -, 0) \end{aligned}$$

where $t(k)$ is the largest triangular integer less than $k - 1$.

Note that a shorter program to compute the same integer can be obtained as follows. The program P above has stages wherein the j^{th} stage it compute b^j from b (where b is the result of the previous stage) by $j - 1$ repeated multiplications by b. Each stage can be shortened by using a square-and-multiply algorithm to compute b^j from b, which takes at most $2 \log_2 j$ steps. The total length is then at most about $2m \log m$.

The success rate of Pollard's $p - 1$ factoring algorithm is only significantly better than trial division if n has exactly one prime factor p such that $p - 1$ is a product of small primes. So, it might seem that the algorithm is not cryptographically relevant, since for n chosen with random prime factor p, the algorithm has a low success rate, unless the length is infeasibly large.

Nevertheless, this algorithm is important because Lenstra generalized it to elliptic curves. Before getting into this generalization, we can review what we have done so far.

We have described two straight line programs that compute large integers with many small factors. The first program simply multiplied small integers together to compute $m!$. The second program instead computed $2^{m!} - 1$. It so happens that the sequence $t_n = 2^n - 1$ has the property that $a|b$ implies $t_a|t_b$. Therefore, if n has many small factors, such as d, then t_n has many small factors, such as t_d. This property of a sequence is called *divisibility*. We could try to generalize this by finding other divisibility sequences.

Of course, being a divisibility sequence is no guarantee of having the small factors necessary to make a good straight line factoring algorithm. Also needed is the that earliest occurrence of a prime dividing a member of the sequence should be small.

Another view is that we work over the group $\mathbb{F}_p^*$, and Lenstra's generalized this group to to elliptic curve groups. Elliptic curve groups, with operations also defined in $\mathbb{F}_p$, have orders other than $p - 1$, but still close to p, and it is these other orders that will provide an improvement to the success rate of factorization.

3.3 Lenstra's elliptic curve method

This section first loosely outlines a straight line variant of Lenstra's elliptic curve method (ECM) factoring algorithm [Len87], which we conjecture to have approximately the same complexity as Lenstra's original algorithm. Second, a more formal description is describe, including some example code.

To make Lenstra's ECM algorithm into a straight line factoring algorithm,

1. fix various parameters, such as those that depend on target class of numbers to factor, and fix the random choices, such as the choice of random elliptic curves and points,

2. perform the elliptic curve arithmetic without using division, such as by using projective coordinates, and

3. multiply together each of certain values computed during the course of Lenstra's ECM algorithm to obtain r. Such a value could be the projective z-coordinate of a smooth large multiple of an elliptic curve point.

It suffices to compute the z-coordinates, and not to bother with the x and y coordinates. One can do this division polynomials, or more simply with elliptic divisibility sequences. This results in a brief, albeit not optimal, implementation. We give an example.

Table 1, written in J, implements a straight line variant of Lenstra's ECM. The number of curves to try is fixed to 40, and a point on each curve is effectively multiplied by 30!. The number to

```
NB. J 6.0.2
r =: 2 1 0 1
s =: 3 0 1
t =: s,:r
C =: 0,.(t&,@(0&,.))^:3 t
F =: |:|.|:|.C
D =: (|.s),}:F
E =: |:|.|:|.D
mod =: x: */ p: ? 2 # 10 ^ 5
pow =: |:@(*/)@(0|:#~)
evn =: mod | pow & C - pow & D
odd =: mod | pow & E - pow & F
ini =: }.@:|:@:((x:@(-@|.,0&,)@(1 1&,))"1)
raw =: [`((evn@$:<.@-:)`(odd@$:<.@-:)@.(2|]))@.(0<])
mid =: 3&{
cur =: 3+i. 40 2
set =: ~. mod +. mid (ini cur) raw  ! 30x
```

Table 1: J code for Straight Line Variant of Lenstra's ECM

factor in this code is `mod`, a product of two random primes, uniformly chosen from the first 100000 primes. The factors found, if any, are stored in the result `set`. Built-in J verb `+.` computes final gcd. Actually, the implementation above is not a strict straight line factoring algorithm because it does not multiply all the partial results, but rather computes the gcd of each partial result with the number to factor. (It could be regarded as a multi-output straight line program.)

There is no branching except in `raw`, where branching is made according to the bits of the multiplier, in this case 30!. Since we will fix the multiplier, we therefore have implemented a straight line program. Stange's adaptation of Shipsey's recursion, which is similar to Montgomery's ladder, for elliptic divisibility sequence is used, as follows.

For each bit in the multiplier, in this case 30!, eight consecutive entries in the elliptic curve divisibility sequence are computed. These eight entries are viewed as a vector, and the recursion involves raising these vectors to the power of matrices with nonnegative entries (for example, $(2,3)^{\left(\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right)} = (6,2)$, which is the job of the defined verb `pow`). The matrices needed are in the variables `C`, `D`, `E`, and `F`. The initial vector is determined by applying `ini` to the 40 fixed curve parameters in `cur`.

The length of the straight line program corresponding to the implementation above is about 242000. It can find prime factors up to size about 2^{20}. The straight line version of trial division described in §3.1, for a similar success rate, might have length about 2^20, or about 8 times longer. Of course, as one factors larger and larger numbers, the ECM based programs become shorter than the comparably successful trial division based programs. A question to ask to what is the shortest straight line program that has a comparable success rate for the same class of numbers to factor. Later, we will show some theoretical and weak concrete lower bounds on the lengths of such programs.

4 Lengths of Integers

The *length* $\lambda(n)$ of a given integer n is the shortest length L of a straight line program P such that $P() = n$.

Note that Shub and Smale [SS96] and Moreira [Mor97] write $\tau(n)$ for $\lambda(n)$, Moreira calls it the *cost* of the integer, while Cheng [Che04] calls it *complexity*.

Some weak bound for the length of a positive integer n are

$$1 + \log_2(\log_2(n)) \leqslant \lambda(n) \leqslant 2\log_2(n) \tag{3}$$

The lower bound is strict except at $n = 2^{2^m}$. The lower bound follows by comparison to the straight line whose first step a double, and all other steps squarings of the previous value. The upper bound is obtained from double-and-add algorithm.

Moreira proves that, for almost all n, the lower bound

$$\lambda(n) \geqslant \frac{\log n}{\log\log n} \tag{4}$$

and a slightly upper bound that holds for all sufficiently large n. Note that here log is to the natural base.

5 Lower Bounds for Straight Line Factoring

A straight line factoring algorithm A computes the homomorphic image of some integer I, using a straight line program P, in the ring $\mathbb{Z}/(n)$ where n is the integer to factored. It follows that the length L of this factoring algorithm is at least $L \geqslant \lambda(I)$.

5.1 An Abstract Lower Bound

Assuming that A has success rate μ for random variable n, the probability that $\gcd(n, I) \notin \{1, n\}$ is at least μ. Let $I(n, \mu)$ be the set of integers I with this property. Therefore, we have a lower bound on the length L of a straight line factoring algorithm:

$$L \geqslant \Lambda(n, \mu) = \min_{I \in I(n,\mu)} \lambda(I) \tag{5}$$

Of course, with no concrete knowledge about $\Lambda(n, \mu)$, this remains merely an abstract lower bound.

5.2 A Weak Concrete Lower Bound

In this section, for special cases of n and μ of potential interest, we find a concrete lower bound below the abstract lower bound $\Lambda(n, \mu)$. The concrete lower bound, though, does not provide much assurance.

Suppose that $\mu = 1/2$. Suppose that the random variable n is generated as $n = pq$ where p and q are identically distributed independent random variables uniformly distributed in the set S of primes integers in the interval $(2^s, 2^{s+1})$. For the sake of simplicity, suppose that the number of primes in S is even. (If not, just remove one prime.) For specificity, we denote this random variable by n_s. This n_s is somewhat similar to the distribution of RSA public key moduli.

The set $I(n_s, 1/2)$ is the set of integers which are divisible by exactly half of the primes in S. Using the lower bound $\lambda(n) \geqslant 1 + \log_2 \log_2(n)$, we get a lower bound

$$\Lambda(n_s, 1/4) \geqslant 1 + \log_2 \log_2 \frac{(2^s + |S|/2)!}{2^s!}, \tag{6}$$

by noting that for $I \in I(n_s, 1/2)$, we must have I at least the product of the first $|S|/2$ numbers larger than 2^s.

The right-hand side of (6) is not entirely concrete, because $|S|$ may be unknown. Rigorous concrete lower bounds can be provided, but in this paper, mere approximations will be given, but the rigorous lower bounds should be quite similar. From the prime number theorem, we derive an approximation $|S|/2 \approx \frac{2^{s+1}}{4\log(2^{s+1})} = \frac{2^{s-1}}{(s+1)\log(2)}$. Let $2^s + |S|/2 = 2^s(1+\epsilon)$ where $\epsilon \approx 1/(2(s+1)\log(2))$. Taking Stirling's approximation, that $\log(n!) = n\log(n) - n + \frac{1}{2}\log(n) + \frac{1}{2}\log(\pi)$, we have an approximation:

$$\log \frac{(2^s(1+\epsilon))!}{2^s!} \approx 2^s \epsilon \log 2^s - \epsilon 2^s + 2^s(1+\epsilon)\log(1+\epsilon)$$

Using the fact that ϵ is small, we get an approximation $\log(1+\epsilon) \approx \epsilon$. Throwing away dominated terms, gives a crude approximation:

$$\log \frac{(2^s(1+\epsilon))!}{2^s!} \approx 2^{s-1}$$

Therefore, we get a lower bound for L of approximately s.

5.3 Failure of Applying Moreira's Lower Bound

A heuristic one could try to apply is to assume that Moreira's bound holds for all $I \in I(n, \mu)$, since it holds for almost all integers. This heuristic would lead to the following bound:

$$\Lambda(n_s, 1/2) \geqslant \frac{\log}{\log\log}\left(\frac{(2^s + |S|/2)!}{2^s!}\right), \tag{7}$$

which, under the approximations above, amounts to something like $\Lambda(n_s, 1/2) \geqslant 2^{s-1}/s$. This lower bound seems too high, because straight line versions of Lenstra's elliptic curve method of factoring seem to have lower lengths than this heuristic lower bound. In fact, applying this heuristic yields a lower bound approximately matching the cost of straight line version of the trial division method. So, assuming the Moreira lower bound over all of $I(n, \mu)$ seems like an incorrect heuristic. In other words, Lenstra's ECM computes an integer I for which $\lambda(I)$ is significantly less than Moreira's lower bound.

6 Conclusion

A linear lower bound on the length of integral-generic factoring algorithms was given. A linear lower bound can be given for the number of bit operations for the addition and multiplication operations of the ring corresponding to the number to be factored. This results in a quadratic lower bound for the number of bit operations for an integral-generic factoring algorithm. This lower bound is no better than the cost of RSA decryption operation. So, there is room for improvement. It is natural to hope that the lower bounds can be improved, and such would be worthy exercise even if one only ever obtains a polynomial lower bound.

Acknowledgments

Dan Bernstein pointed out Riesel's work. Anonymous CRYPTO 2010 reviewers revealed faults in at least the exposition of this material, whose resolution is ongoing.

References

[AM09] D. Aggarwal and U. Maurer. *Breaking RSA generically is equivalent to factoring.* In A. Joux (ed.), *Advances in Cryptology — EUROCRYPT 2009*, no. 5479 in LNCS, pp. 36–53. IACR, Springer, Apr. 2009.

[BV98] D. Boneh and R. Venkatesan. *Breaking RSA may be easier than factoring.* In K. Nyberg (ed.), *Advances in Cryptology — EUROCRYPT '98*, no. 1403 in LNCS, pp. 59–71. IACR, Springer, May 1998. http://crypto.stanford.edu/~dabo/abstracts/no_rsa_red.html.

[Che04] Q. Cheng. *On the ultimate complexity of factorials.* Theoretical Computer Science, **326**(1–3):419–429, Oct. 2004.

[DK09] I. Damgård and M. Koprowski. *Generic lower bounds for root extraction and signature schemes in general groups.* In L. Knudsen (ed.), *Advances in Cryptology — EUROCRYPT 2002*, no. 2332 in LNCS, pp. 256–271. IACR, Springer, Apr. 2009.

[Len87] H. W. Lenstra. *Factoring integers with elliptic curves.* Annals of Mathematics, **126**:649–673, 1987.

[Lip94] R. J. Lipton. *Straight line complexity and integer factorization.* In L. M. Adleman and M.-D. Huang (eds.), *Algorithmic Number Theory*, no. 877 in LNCS, pp. 71–79. Springer, may 1994.

[LR06] G. Leander and A. Rupp. *On the equivalence of RSA and factoring regarding generic ring algorithms.* In X. Lai and K. Chen (eds.), *Advances in Cryptology — ASIACRYPT 2006*, no. 4284 in LNCS, pp. 241–251. IACR, Springer, Dec. 2006.

[Mor97] C. G. T. D. A. Moreira. *On asymptotic estimates for arithmetic cost functions.* Proceedings of the American Mathematical Society, **125**(2):347–353, Feb. 1997.

[Rie94] H. Riesel. *Prime Numbers and Computer Methods for Factorization*, Progress in Mathematics, vol. 126. Birkhäuser, 2nd edn., 1994.

[Sha79] A. Shamir. *Factoring numbers in $O(\log n)$ arithmetic steps.* Information Processing Letters, **8**(1):28–31, 1979.

[Sho97] V. Shoup. *Lower bounds for discrete logarithms and related problems.* In W. Fumy (ed.), *Advances in Cryptology — EUROCRYPT '97*, no. 1233 in LNCS, pp. 256–266. IACR, Springer, May 1997.

[SS96] M. Shub and S. Smale. *On the intractability of Hilbert's nullstellensatz and an algebraic version of "$NP \neq P$".* Duke Mathematical Journal, **81**(1):47–54, 1996.

A Related Work

Shoup [Sho97] introduced the generic group model and proved lower bound the difficulty of solving the discrete logarithm problem in this model. Damgård and Koprowski [DK09] extended Shoup's model to groups with unknown order, and imply a lower bound of the difficulty of group order in this model. Their lower bound is larger than the cost of existing factoring algorithms, but such algorithms are not restricted to group operations only. Leander and Rupp [LR06] introduced the generic ring model, and Aggarwal and Maurer [AM09] proved that the RSA problem is as difficult as the factoring problem, in this model.

Boneh and Venkatesan [BV98] essentially proved that a generic factoring algorithm with an RSA oracle could be converted into one without an RSA oracle.

Upper bounds on the difficulty of generic factoring are provided by examples of generic factoring algorithms. The trial division factoring algorithm has a version which is integral-generic. Lenstra's [Len87] elliptic curve method has version that is an integral-generic factoring algorithm, and which may have similar running time.

Shamir [Sha79] provides an asymptotic upper bound of $O(\log n)$ arithmetic steps to factor n. This very low upper bound proves that factoring can only have very low lower bound in terms of number of arithmetic steps. However, Shamir's arithmetic steps include integer division, which is the floor of rational division, and these steps are excluded from the generic ring model: Shamir's algorithm is not a generic factoring algorithm and a lower bound on the number of steps in a generic factoring algorithm could exceed Shamir's upper bound on arithmetic steps.[1] Furthermore, the steps in Shamir's algorithm involve arbitrarily large integers, whereas in the generic factoring, the steps measured are operations on inputs of fixed bit size. Upon conversion to bit operations, Shamir's bound becomes quite large.

Lipton [Lip94] provides effectively yet another kind of upper bound: showing the factoring is no harder than computing, with a straight line program, polynomials with many distinct rational roots.

Cheng [Che04] recasts Lenstra's elliptic curve method factoring as an upper bound on the straight line complexity of certain integers (which we shall call the length of integers). Our lower bounds may be expressed in terms of the length of integers, we study lower bounds for these. Cheng

[1]The extra operations in Shamir's algorithm, integer division, can be used to implement the Euclidean algorithm for finding modular inverses, which seems to be hard to do with mere ring operations, unless factoring is easy.

[Che04] later speculates about lower bounds on the lengths of integers and the possibility of certain factoring algorithms, but does not give an explicit lower bound on factoring.

Moreira [Mor97] provides lower bounds on the length of almost all integers. Lower bounds of the lengths of integers are exactly what is needed in this paper, but because Moreira's bound only applies to almost all integers, it is not directly applicable, unless some heuristic assumptions are made. An adversary is not bound to choose a compute integer, but may seek one that can be computed more efficiently than average.

Shub and Smale [SS96] show that the short length of certain (sets of) integers is equivalent to an algebraic version of $NP \neq P$. Assuming the hardness of the latter also provides a conditional lower bound on the lengths of certain integers. Perhaps this conditional lower bound can be extended to become a lower bound on integral-generic factoring algorithms.

B Schools of Thought

One rather strict school of thought in cryptology is that a cryptographic technique should only be considered secure to the extent that it can be proved secure. This is mainly applied to advance a preference between two cryptographic techniques, where the one with better proofs of security deemed as secure(r). Such preference may in fact be contrary to a well-established history of remaining unbroken under heavy use and intense scrutiny. A rationale may be that an unforeseen attack may just be waiting around the corner.

Nevertheless, many such proofs of security actually rely on unproven security assumptions, such as factoring being hard, whose security is assured primarily on well-established history of reaming unbroken under heavy and intense scrutiny. Strict adherence to the above school of thought demands seeking either proofs of security for such assumptions or other assumptions that have security proofs (or both). So, strictly speaking, this school of thought should prefer discrete logarithm cryptography (DLC) over integer factoring cryptography (IFC), because the former has better security proofs, provided by Shoup's proof, whereas integer factoring cryptography has no useful proofs. The fact that Shoup's proof for DLC has limitations is a given, but IFC has no proofs at all, so the case for DLC is better than nothing.

C Lambda Examples

This section gives some examples of λ computations.

First, $\lambda(83) = 5$. The only length 5 integral-generic program computing 83 is:

$$((+,0,0),(+,0,1),(\times,2,2),(\times,3,3),(+,1,4)).$$

Second, $\lambda(720) = 6$. Seven integral-generic programs compute 720, with intermediate values given by one of the five rows of the following array:

1	2	3	9	27	729	720
1	2	3	9	81	80	720
1	2	3	9	81	729	720
1	2	4	6	24	30	720
1	2	4	16	20	36	720

The last two rows account for two programs each because 4 can be computed as either $2 + 2$ or 2×2.

The $n \leqslant 10000$ for which $\lambda(n) = \lceil 1 + \log_2 \log_2 n \rceil$ are:

$$2, 3, 4, 5, 6, 8, 9, 16, 17, 18, 20, 24, 25, 27, 32, 36, 64, 81, 256, 257, 258, 260, 272, 288,\\ 289, 320, 324, 400, 512, 576, 625, 729, 1024, 1296, 4096, 6561.$$