# FAST SQUARE-FREE DECOMPOSITION OF INTEGERS USING CLASS GROUPS

ERIK MULDER

Abstract. Let $n = a^2 b$, where $b$ is square-free. In this paper we present an algorithm based on class groups of binary quadratic forms that finds the square-free decomposition of $n$, i.e. $a$ and $b$, in heuristic expected time:

$$\widetilde{\mathcal{O}}(L_b[1/2, 1] \ln(n) + L_b[1/2, 1/2] \ln(n)^2).$$

If $a, b$ are both primes of roughly the same cryptographic size, then our method is currently the fastest known method to factor $n$. This has applications in cryptography, since some cryptosystems rely on the hardness of factoring integers of this form.

## 1. Introduction

One of the classic questions in computational number theory is whether the square-free decomposition of a given mathematical object can be computed 'fast'. The question for polynomials is easy to solve: given a polynomial $f = f_1^2 f_2$, where $f_2$ is square-free, we can compute $g = \gcd(f, f')$. Then $g$ is a multiple of $f_1$, and with some more care, the exact square factor $f_1$ can be found in polynomial time [35]. For integers however, the question is still open. Given $n = a^2 b$, with $b$ square-free, we present a novel technique that uses class groups to find the square-free decomposition of $n$, i.e. $a$ and $b$. The algorithm runs in heuristic expected time

$$\widetilde{\mathcal{O}}(L_b[1/2, 1] \ln(n) + L_b[1/2, 1/2] \ln(n)^2)$$

where $L_b$ is the $L$-notation:

$$L_b[\alpha, c] = e^{(c+o(1)) \ln(b)^\alpha (\ln \ln(b))^{1-\alpha}}.$$

In the special case that $a = p$ and $b = q$ are distinct primes and $p \approx q$, our algorithm is currently the fastest known method for computing the square-free decomposition if $q$ is roughly in the range $[10^{20}, 10^{5000}]$. The upper bound should be taken with a grain of salt, see Section 6.1. Numbers $n$ of this form might seem like a very specific case. It is true that the probability that a random large integer $n$ is of this form is very low. However, these numbers appear quite frequently in cryptographic systems [25, 24, 26, 29, 33]. In these cryptosystems, the assumption is usually made that factoring an integer $n = p^2 q$ with $p \approx q$ is as hard as factoring an arbitrary integer with 3 large prime factors. Our method shows that this is not true when $q$ is of cryptographic size. Therefore, larger moduli should be used in these cryptosystems than advised in those articles.

Another application of our algorithm is to determine the ring of integers of number fields, since this is polynomial time equivalent to finding the square-free decomposition of the discriminant of the number field [11]. In [7], this equivalence

*Date*: January 12, 2023.



**Submitted to Algor. Num. Th. Symp.**

is studied in great detail. Unfortunately, our algorithm does not run in polynomial time, but it can be useful when the square-free part of the discriminant is not too big. Another possible application is determining the endomorphism ring of an elliptic curve over a finite field [2], since that algorithm requires the square part of the discriminant of the characteristic equation of the Frobenius endomorphism.

Factoring integers that have repeated prime factors has been an area of active research for quite some time. Lattice algorithms are quite popular [3, 13, 21, 17], but also elliptic curves [27] and even class groups [9, 10] have been used before. In [4] an algorithm is presented that also uses class groups, which can be used to detect square-free numbers. Finally, it is good to mention that general purpose factorization algorithms such as the number field sieve [18] or the elliptic curve method [19] can of course also factor numbers of this form.

In 1984, Schnorr and Lenstra [30] presented the following algorithm to factor an integer $n$, which is very similar to other factorization algorithms such as the elliptic curve method and Pollard's $p-1$ method. Take a random form $f$ in the class group $C(-4n)$ and compute $g = f^k$, where $k$ is a large highly composite integer. If $h(-4n)$ is smooth, then $g \sim e_{-4n}$, where $e_{-4n}$ is the identity element of $C(-4n)$. Schnorr and Lenstra showed how to factor $n$ in this case. Based on heuristic assumptions, they claimed that the expected runtime of their algorithm is $\mathcal{O}(L_n[1/2, 1])$. Unfortunately, it was later found that if $n$ has a large square prime divisor, then this runtime is out of reach [20].

We adapt the algorithm of Schnorr and Lenstra such that it actually becomes *faster* when $n = a^2b$ has a large square divisor. We will use the fact that every form in $g \in C(-4n)$ can be derived from a unique form $\pi(g) \in C(-4b)$. We don't require that $g \sim e_{-4n} \in C(-4n)$, but instead that $\pi(g) \sim e_{-4b} \in C(-4b)$. Given such $g$, we show how to retrieve $a$, which gives the square-free decomposition of $n$. Our method is successful if the class number $h(-4b)$ is smooth. If $b$ is not too big, then the probability that this happens is much larger than the probability that $h(-4n)$ is smooth.

1.1. **Outline of this paper.** In Chapter 2 we will recall some basic properties of binary quadratic forms and how forms in $C(-4a^2b)$ can be derived from forms in $C(-4b)$. In Chapter 3, we look at the factorization algorithm of Schnorr and Lenstra [30] in more detail. In Chapters 4 and 5, we present our new square-free decomposition algorithm. In Chapter 6 we compare our algorithm to other factorization algorithms and we test its speed in practice.

**Acknowledgements.** The author would like to thank Wieb Bosma for the helpful discussions and for proofreading this paper. The author is also grateful for the comments by the anonymous referees, which greatly improved this article.

## 2. Class group of binary quadratic forms

2.1. **Preliminaries.** In this section we quickly recall important definitions from the theory of binary quadratic forms. For more details see Cox [14], Chapters 1 to 3.

A *binary quadratic form* $f$ is a polynomial of the form $f(x, y) = ax^2+bxy+cy^2 = (a, b, c)$, where $a, b, c \in \mathbb{Z}$. We say that $f$ *represents* $m$ if there exist $x, y \in \mathbb{Z}$ such that $f(x, y) = m$. The *discriminant* of $f$ is $D = b^2 - 4ac = 0, 1 \bmod 4$. If $D < 0$ and $a > 0$, then $f$ is *positive definite*. A form is *primitive* if $\gcd(a, b, c) = 1$. We

will always assume that our forms are primitive and positive definite, unless stated otherwise. Let $\Gamma$ be the classical modular group of $2 \times 2$ matrices $A$ with integer coefficients and $\det(A) = 1$. Two binary quadratic forms $f, g$ are *equivalent* if there exists a matrix $A = \left(\begin{smallmatrix} p & q \\ r & s \end{smallmatrix}\right) \in \Gamma$ such that $f(x, y) = g(A \cdot (x, y)^T) = g(px+qy, rx+sy)$. If this is the case, then we write $f \sim g$. This implies that equivalent forms have the same discriminant. A form $(a, b, c)$ of discriminant $D < 0$ is *reduced* if $|b| \leq a \leq c$, and $b \geq 0$ if either $|b| = a$ or $a = c$. Every form of negative discriminant is equivalent to a unique reduced form.

The *class group* $C(D)$ consists of the equivalence classes of primitive binary quadratic forms of discriminant $D$, the group operation is composition of forms. Given two forms $f = (a_1, b_1, c_2)$, $g = (a_2, b_2, c_2)$ in $C(D)$ with $\gcd(a_1, a_2, (b_1 + b_2)/2) = 1$, their (Dirichlet) composition is $f \cdot g = (a_1 a_2, B, \frac{B^2 - D}{4 a_1 a_2})$, for a suitable integer $B$. For more details about this operation, see Chapter 3 of [14]. This composition can be computed in $\widetilde{\mathcal{O}}(\ln(D))$ using a variant of fast GCD [31]. If $D = 0 \bmod 4$, then the identity element of this group is $e_D = (1, 0, \frac{-D}{4})$. The *class number* $h(D)$ is the order of $C(D)$. For $D < 0$, on average $h(D)$ is roughly $\sqrt{|D|}/\pi$ [8], page 84. A discriminant $D$ is *fundamental* if either $D = 1 \bmod 4$ and $D$ square-free, or $D = 4m$ for some $m$ with $m = 2, 3 \bmod 4$ and $m$ square-free. $D$ is called *non-fundamental* (non-fun) otherwise. In most number theory papers it is assumed that $D$ is fundamental, but for our purposes we will mainly focus on non-fundamental discriminants instead.

2.2. **Non-fundamental discriminants.** We will mainly follow Chapter 7 of Buell [8] for this subsection. Let $D$ be a discriminant and $r$ a positive integer. We will discuss how the class groups $C(D)$ and $C(Dr^2)$ are related. For this we use *transformation matrices* $A$, which are $2 \times 2$ integer matrices with $\det(A) = r$.

**Proposition 2.1.**

a) *Given any primitive form $f$ of discriminant $Dr^2$, there exists a form $g$ of discriminant $D$ and a transformation matrix $A$ with $\det(A) = r$ such that $f(x, y) = g(A \cdot (x, y)^T)$.*
b) *If $f_1$ and $f_2$ are primitive equivalent forms of discriminant $Dr^2$, then there exists a primitive form $g$ of discriminant $D$, together with transformation matrices $A_1$, $A_2$ such that $A_1 A_2^{-1} \in \Gamma$ and:*

$$\det(A_1) = \det(A_2) = r, \quad g(A_1 \cdot (x, y)^T) = f_1, \quad g(A_2 \cdot (x, y)^T) = f_2.$$

*Proof.* See Proposition 7.1 in [8]. □

Proposition 2.1 basically says that for every primitive form $f$ in the group of larger discriminant $Dr^2$, there is a unique (up to equivalence) primitive form $g$ in the group of smaller discriminant $D$ and a transformation matrix $A$ of determinant $r$ such that $f(x, y) = g(A \cdot (x, y)^T)$. In this case we say that $f$ is *derived* from $g$.

Buell also makes those transformations explicit; for this he uses the following notion. Define two transformation matrices $A_1$ and $A_2$ of determinant $r$ to be *right-equivalent* if there exists a matrix $B \in \Gamma$ such that $A_1 B = A_2$. It is easy to see that this is an equivalence relation. Furthermore, if $g$ is a form of discriminant $D$ and $f_1, f_2$ are forms derived from $g$ using right-equivalent transformations, then $f_1$ and $f_2$ are equivalent.

We first restrict ourselves to the case that $r$ is a prime $p$. The general case will be partially discussed in Lemma 4.4.

**Proposition 2.2.** *The right-equivalent transformations of determinant $p$ have as equivalence class representatives the $p+1$ transformations:*

$$\begin{pmatrix} p & h \\ 0 & 1 \end{pmatrix} \quad \text{for } 0 \le h \le p-1 \quad \text{and} \quad \begin{pmatrix} 1 & 0 \\ 0 & p \end{pmatrix}.$$

*Proof.* See Proposition 7.2 in [8]. □

This means that given a form $g = (a, b, c) \in C(D)$ with $\gcd(a, p) = 1$, we can create the following $p + 1$ forms of discriminant $Dp^2$:

$$\begin{gathered} (ap^2, p(b + 2ah), ah^2 + bh + c) \quad \text{for } 0 \le h \le p-1 \\ \text{and } (a, bp, cp^2) \end{gathered} \tag{2.1}$$

and no others, up to equivalence. In the last case, we say that $h = \infty$ was used in the transformation. Note that these forms are not necessarily reduced, even when $g$ is reduced.

Using the formulas from (2.1), we can also define a map that goes the other way. Given a form $f \in C(Dp^2)$, find a form $f_2$ of the form $(ap^2, bp, c)$ that is equivalent to $f$. Then define $\pi(f) = (a, b, c) \in C(D)$. This map is well-defined (up to equivalence) because of Proposition 2.1.

The next question is whether the forms from (2.1) are primitive and if they can be equivalent to each other. We will use the *Kronecker symbol* ([14] page 93), which we denote by $\left(\frac{D}{p}\right)$, for given integers $D, p$. Using this symbol, we can state the relation between $h(D)$ and $h(Dp^2)$. From this points onwards, we will focus on negative discriminants, because the important proposition below is not true for positive discriminants.

**Proposition 2.3.** *Given a form $(a, b, c)$ of discriminant $D < -4$ and an odd prime $p$ with $\gcd(a, p) = 1$, the $p + 1$ representative transformations of determinant $p$ produce exactly $p - \left(\frac{D}{p}\right)$ primitive forms of discriminant $Dp^2$ which are all pairwise inequivalent. It follows that:*

$$h(Dp^2) = h(D) \cdot (p - \left(\frac{D}{p}\right)).$$

*Proof.* See Propositions 7.3 and 7.4 in [8]. □

By applying this formula repeatedly, we get the following corollary:

**Corollary 2.4.** *Given a discriminant $D < -4$ and an odd integer $r = \prod_{i=1}^{k} p_k^{e_k}$, define*

$$\varphi_D(r) = \prod_{i=1}^{k} p_k^{e_k - 1}(p_k - \left(\frac{D}{p_k}\right)).$$

*Then*

$$h(Dr^2) = h(D) \cdot \varphi_D(r).$$ □

An important property of the embedding of $C(D)$ in $C(Dr^2)$ is that the derivedness property behaves well under composition of forms:

**Proposition 2.5.** *If $f_1, f_2 \in C(Dr^2)$ are derived from $g_1, g_2 \in C(D)$ respectively, then $f_1 \cdot f_2$ is derived from $g_1 \cdot g_2$.*

*Proof.* See Proposition 7.9 in [8]. Buell's map $1_\Delta$ is the map $\pi$ in our context. □

**Corollary 2.6.** *Suppose $f \in C(Dr^2)$ is derived from $g \in C(D)$, where $r$ is odd. Let $a$ be the order of $g$. Then the order of $f$ divides $a \cdot \varphi_D(r)$.*

*Proof.* The forms in $C(Dr^2)$ derived from $e_D \in C(D)$ form a subgroup in $C(Dr^2)$. We know from Corollary 2.4 that the order of this subgroup is $\varphi_D(r)$. Proposition 2.5 implies that $f^a$ is derived from $e_D$. Therefore, $f^{a \cdot \varphi_D(r)} = e_D$. □

## 3. Schnorr and Lenstra class group factorization

In 1984, Schnorr and Lenstra published a general purpose integer factorization algorithm which they claimed could heuristically factor an integer $n$ in $\mathcal{O}(L_n[1/2, 1])$ [30]. We will briefly discuss how the algorithm works before showing how we can use it in our square-free decomposition algorithm.

Let $n$ be the number you want to factor, possibly square-free. Consider the class group $C(-4n)$ and take a random form $f \in C(-4n)$. Construct a number $k$ that consists of powers of all primes up to some bound $B$. Now, if $f^k = e_{-4n}$ and the order of $f$ is even, then $g = f^{k/2^m}$ has order 2 for some integer $m$. Forms of order 2 are called *ambiguous*. They are of the form $(m, 0, \frac{n}{m})$, with $\gcd(m, \frac{n}{m}) = 1$, which makes it easy to retrieve a factor of $n$ from them. If this was successful and if $n$ is not yet completely factored, then we go again with a different $f$ and hopefully find another factor of $n$, until the complete factorization is found.

This method works as long as the order of $C(-4n)$ is smooth. If it is not smooth, then we can try again by considering the class group $C(-4ns)$ for some small positive integer $s$. An ambiguous form in that group will still lead to the factorization of $n$. We can repeat this process for different values of $s$ until the factorization of $n$ is found.

Schnorr and Lenstra made the following heuristic assumptions. Assumptions $a, b$ are (2.1), (2.2) in [30] respectively. Assumption $c$ is mentioned later in their article in equation (4.3). This final assumption turned out to be wrong, which we will see later in this chapter. Let $\Psi(x, y)$ be the number of positive integers $a \leq x$ such that $a$ is $y$-smooth.

**Assumptions 3.1.**

*a) The order of a class group of discriminant $D$ is at least as likely to be smooth as a random integer of size $\sqrt{D}$. More precisely, for all $n, t$:*

$$\#\{m \leq n : h(-m) \mid \prod_{i=1}^{t} p_i^{e_i}\}/(0.5n) \geq \#\{m \leq \sqrt{n} : m \mid \prod_{i=1}^{t} p_i^{e_i}\}/\sqrt{n},$$

*where the $e_i$ are defined in Algorithm 1.*

*b) A significant portion of integers are smooth. More explicitly, for all $n$ and $c \leq \sqrt{\ln(n)/\ln\ln(n))}$ we have that $\Psi(n, n^{1/c})/n \geq c^{-c}$.*

*c) Given an integer $n$, the smoothness bounds of $h(-4ns)$ are independent for all square-free integers $s$.*

It is good to mention that the Cohen-Lenstra heuristics [12] suggest that the odds of finding a class group with a smooth order is actually a bit better than that of a random integer of the same size.

**Algorithm 1** Schnorr and Lenstra stage 1

```
1: function GeneralClassGroupFactorization(n)
2:     c = sqrt(ln(n)/ln ln(n))
3:     B = n^{1/(2c)}                                   ▷ prime bound
4:     compute the first t primes p_1, ..., p_t up to B
5:     k = prod_{i=2}^{t} p_i^{e_i}, where e_i = max{v : p_i^v ≤ p_t^2}
6:     s = 1
7:     while no factorization found do
8:         pick random f ∈ C(−4ns)
9:         g = f^k
10:        if g = e_{−4ns} then
11:            go back and pick a new f
12:        end if
13:        for 1 ≤ i ≤ log_2(sqrt(n)) do
14:            g = g^2
15:            if g = e_{−4ns} then
16:                construct ambiguous forms
17:                return complete factorization of n
18:            end if
19:        end for
20:        update s to be the next square-free number after s
21:    end while
22: end function
```

We can now state the main result from Schnorr and Lenstra.

**Theorem 3.2.** *Assume Assumptions 3.1. Then for all composite integers $n$, Algorithm 1 will completely factor $n$ in expected $\mathcal{O}(L_n[1/2, 1])$ time.*

*Proof.* See Theorem 5 and the run time analysis section of Chapter 4 from [30]. □

Unfortunately, in 1992 it was found by Lenstra and Pomerance [20] (Chapter 11), that the above stated run time was incorrect for a large set of numbers. Can you guess which? Numbers that have a large prime square divisor!

Suppose that $n$ has a divisor $p^2$, where $p$ is prime. Suppose furthermore that $p-1$ and $p+1$ are both not smooth. Then by Proposition 2.3, we see that $C(-4ns)$ is divisible by either $p-1$ or $p+1$ for all integers $s$. Therefore, there will be no $s$ such that $C(-4ns)$ is smooth.

Interestingly enough, we will show that by adapting Lenstra's algorithm for integers of this form, we actually get an algorithm that is *faster* than the originally claimed time bound in this special case.

We now know that assumption $c$ is incorrect. Therefore, we will use the following assumptions instead, where we only tweak assumption $c$. These new assumptions are still just conjectures, but now at least the problem with large square divisors is taken care of. In Section 6.2, we will give experimental results that might convince one of the correctness of these assumptions.

**Assumptions 3.3.**

a) *The order of a class group of discriminant $D$ is at least as likely to be smooth as a random integer of size $\sqrt{D}$. More precisely, for all $n, t$:*

$$\#\{m \leq n : h(-m) \mid \prod_{i=1}^{t} p_i^{e_i}\}/(0.5n) \geq \#\{m \leq \sqrt{n} : m \mid \prod_{i=1}^{t} p_i^{e_i}\}/\sqrt{n},$$

*where the $e_i$ are defined in Algorithm 1.*

b) *A significant portion of integers are smooth. More precisely, for all $n$ and $c \leq \sqrt{\ln(n)/\ln\ln(n)}$ we have that $\Psi(n, n^{1/c})/n \geq c^{-c}$.*

c) *Given a square-free integer $n$, the smoothness bounds of $h(-4ns)$ are independent for all square-free integers $s$.*

We will rephrase Theorem 3.2 so that it uses the new assumptions and so that it can be used for the analysis of our algorithm.

**Theorem 3.4.** *Assume Assumptions 3.3. Let $n$ be square-free and composite. Let*

$$B = n^{1/(2c)} \in \mathcal{O}(L_n[1/2, 1/2])$$

*as stated in Algorithm 1. Then per multiplier $s$, the algorithm performs $\mathcal{O}(B)$ compositions. It takes expected $\mathcal{O}(B)$ tries to find a suitable $s$. Afterwards, another $\mathcal{O}(\ln(s)(B+s))$ compositions are needed to construct the ambiguous forms. In total, it takes expected $\mathcal{O}(B^2) = \mathcal{O}(L_n[1/2, 1])$ compositions to factor $n$.*

*Proof.* We again refer to the time analysis section of Chapter 4 of [30]. □

## 4. A new square-free decomposition algorithm

4.1. **The case of prime square factor.** For now, let $n = p^2 b$ with $p$ prime and $b$ positive, square-free and possibly composite. Later on we will allow $p$ to be composite as well. All our discriminants will contain a factor 4, since $-n$ is not a discriminant if $n = 1, 2 \bmod 4$, but $-4n$ always is.

In this section we will introduce an algorithm that computes the square-free decomposition of $n$ that uses class groups with non-fundamental discriminants. One of the main ingredients is the following proposition. This proposition contains the very restricting condition that $b > p^2$. But, we will see in Lemma 4.2 that we can drop this assumption.

**Proposition 4.1.** *Suppose we have a reduced form $f \in C(-4n)$ that is derived from $e_{-4b} = (1, 0, b) \in C(-4b)$ and $f \not\sim e_{-4n} = (1, 0, n)$. Furthermore, suppose that $b > p^2$. Then*

$$f = (p^2, 2pk, k^2 + b)$$

*for some $-p/2 \leq k \leq p/2$.*

*Proof.* Using the formulas from (2.1), we see that $f$ is equivalent to

$$g = (p^2, 2ph, h^2 + b)$$

for some $0 \leq h \leq p - 1$. Let's check if $g$ is reduced. We have $h^2 + b \geq b > p^2$, so that is a good start.
If $h \leq p/2$ then $|2ph| \leq p^2$, so in this case $g$ is reduced. Reduced forms are unique. Therefore $f = (p^2, 2ph, h^2 + b)$ (not just equivalent, really equal).
If $h \geq p/2$ then we do one reduction step with $A = \left(\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right)$:

$$g \sim g_2 = (p^2, 2ph - 2p^2, p^2 - 2ph + h^2 + b) = (p^2, 2p(h - p), (h - p)^2 + b).$$

Then since $|2p(h-p)| \leq |2p(p/2)| = p^2$, we see that $g_2$ is reduced. Therefore $f = (p^2, 2p(h-p), (h-p)^2 + b)$. □

The main takeaway from this proposition is the following factorization strategy. If we are able to find a non-trivial form in the larger group $C(-4bp^2)$, that is derived from the trivial form in the smaller group $C(-4b)$, then we can find $p^2$ by reading off the first coefficient of the reduced form of $f$. In Proposition 4.5, we will state a version of Proposition 4.1 where the square part can be composite.

It is good to mention that this factorization strategy in a general sense is very similar to other integer factorization algorithms. For example, in the elliptic curve method (ECM) by Lenstra [19], a non-trivial point $P$ on an elliptic curve $E(\mathbb{Z}/n\mathbb{Z})$ is constructed such that $P$ reduced mod $p$ is the identity element in the group $E(\mathbb{Z}/p\mathbb{Z})$ of smaller size, where $p \mid n$. By computing $\gcd(P_x, n)$, the factor $p$ of $n$ will be found. This will not be the last time that we compare our algorithm to the ECM, since there are many similarities.

The assumption that $b > p^2$ is quite limiting because then $h(-4b)$ will be fairly large, making the algorithm not very efficient. Fortunately, there is a great way to circumvent this problem, by introducing an integer $r$ with known factorization. In the next lemma we show that we can enlarge $b$ with a large factor $r^2$. This will help a lot, since then the form $f$ in Proposition 4.1 will be reduced, even if the original $b$ is not large compared to $p$.

**Lemma 4.2.** *Suppose $g \in C(-4n)$ is derived from $e_{-4b}$. Let $r$ be a positive integer with $\gcd(r,p) = 1$. Lift $g$ to some $h \in C(-4nr^2)$ using the formulas in (2.1). Then $l = h^{\varphi_{-4n}(r)}$ is not only derived from $e_{-4b}$, but also from $e_{-4br^2}$.*

*Proof.* First note that $h \in C(-4nr^2)$ is derived from $e_{-4b} \in C(-4b)$. Suppose $h$ is derived from $h_2 \in C(-4br^2)$. Then $h_2$ is also derived from $e_{-4b} \in C(-4b)$, because Proposition 2.1 implies that $h$ is derived from a unique form in $C(-4b)$. Corollary 2.6 now implies that the order of $h_2$ divides $\varphi_{-4b}(r)$.
Write $r = \prod_{i=1}^{k} p_k^{e_k}$, then since $\gcd(r,p) = 1$,

$$\varphi_{-4b}(r) = \prod_{i=1}^{k} p_k^{e_k-1}(p_k - \left(\frac{b}{p_k}\right)) = \prod_{i=1}^{k} p_k^{e_k-1}(p_k - \left(\frac{n}{p_k}\right)) = \varphi_{-4n}(r).$$

Using Proposition 2.5, we see that $l$ is derived from

$$h_2^{\varphi_{-4n}(r)} = h_2^{\varphi_{-4b}(r)} = e_{-4br^2}.$$ □

We can now state the first version of our algorithm. The main idea is to use Algorithm 1 to find a form $f \in C(-4n)$ that is derived from $e_{-4b} \in C(-4b)$. Using Lemma 4.2, we are then able to find a form derived from $e_{-4br^2} \in C(-4br^2)$ for a large enough $r$. Finally, we use Proposition 4.1 to find the factor $p^2$ of $n$. From this the square-free decomposition can be easily computed.

In the algorithm below, $n = p^2 b$ is an integer not divisible by 3 and $b_2$ is an upper bound for $b$. If no good upper bound is known, then a small value can be used initially, and it can be increased incrementally if no factorization is found. The first part of the algorithm is completely the same as Algorithm 1. The difference is that if Lenstra's algorithm is unable to find the factorization, then we do some additional steps which might lead to the square-free decomposition of $n$.

**Algorithm 2** Square-free decomposition stage 1

1: **function** SQUAREFREEDECOMPOSITION($n, b_2$)
2: $c = \sqrt{\ln(b_2)/\ln\ln(b_2))}$
3: $B = b_2^{1/(2c)}$ ▷ prime bound
4: compute the first $t$ primes $p_1, \ldots, p_t$ up to $B$
5: $k = \prod_{i=2}^{t} p_i^{e_i}$, where $e_i = \max\{v : p_i^v \leq p_t^2\}$
6: $s = 1$
7: $r = 3^{\lceil \log_3(\sqrt{n}))\rceil}$
8: **while** no factorization found **do**
9: pick random $f \in C(-4ns)$
10: $g = f^k$
11: **if** $g = e_{-4ns}$ **then**
12: go back and pick a new $f$
13: **end if**
14: **for** $1 \leq i \leq \log_2(\sqrt{n})$ **do**
15: $g = g^2$
16: **if** $g = e_{-4ns}$ **then**
17: construct ambiguous forms
18: **return** complete factorization of $n$
19: **end if**
20: **end for**
21: lift $g$ to a form $h \in C(-4nsr^2)$ using the formulas in (2.1)
22: $l = h^{\varphi_{-4ns}(r)}$
23: try to find $p^2$ using the form $l$ and Proposition 4.1
24: **if** $p^2$ is found **then**
25: $p = \sqrt{p^2}$ and $b = n/p^2$
26: **return** $p, b$
27: **end if**
28: update $s$ to be the next square-free number after $s$
29: **end while**
30: **end function**

**Theorem 4.3.** *Assume Assumptions 3.3. Then for all integers $n = p^2 b$, Algorithm 2 will find the square-free decomposition (and possibly the full factorization) of $n$ in expected time:*

$$\widetilde{\mathcal{O}}(L_b[1/2, 1]\ln(n) + L_b[1/2, 1/2]\ln(n)^2).$$

*Proof.* Let's first look at the correctness of the algorithm. We know from Theorem 3.4 that the code up to line 20 has a chance to completely factor the integer $n$. This is the case if $g$ is the trivial form in the group $C(-4ns)$. If that is not the case, then by using Proposition 2.5, we see that there is still a possibility that $g$ is derived from the trivial form in the underlying group $C(-4bs)$.
In that case, we know from Lemma 4.2 that the form $l$ in the larger group $C(-4nsr^2)$ is not just derived from $e_{-4bs} \in C(-4bs)$, but also from $e_{-4bsr^2} \in C(-4bsr^2)$. Thus, we can now apply Proposition 4.1 with the form $l$ to find $p^2$, because

$$bsr^2 \geq r^2 \geq n \geq p^2.$$

Now let's look at the running time of the algorithm. Until line 20, the code is the same as Algorithm 1, except that our prime bound depends on $b_2$ instead of $n$. If

$b_2$ is a good approximation of $b$, then Proposition 2.5 and Theorem 3.4 imply that we can expect to get a form derived from $e_{-4bsr^2}$ in $\mathcal{O}(L_b[1/2,1])$ group operations, i.e. $\widetilde{\mathcal{O}}(L_b[1/2,1]\ln(n))$ steps if we use a fast composition algorithm [31].
The additional lines 21 to 27 can be done in $\widetilde{\mathcal{O}}(\ln(n)^2)$ per value of $s$ that we try, since the discriminant of $h$ is $\mathcal{O}(-n^2 s)$ and the exponent in line 22 is $\mathcal{O}(\sqrt{n})$. From Theorem 3.4 we also know that the expected number of multipliers $s$ that we have to try is $\mathcal{O}(L_b[1/2,1/2])$. Hence, the total expected running time of the algorithm in this case is

$$\widetilde{\mathcal{O}}(\psi_b(n)) := \widetilde{\mathcal{O}}(L_b[1/2,1]\ln(n) + L_b[1/2,1/2]\ln(n)^2).$$

If a good approximation of $b$ is not known, then start with a small value for $b_2$ and try to find the factor $p^2$ in $\widetilde{\mathcal{O}}(\psi_{b_2}(n))$ time. If the algorithm is not successful in that time frame, then try again with $b_2 = 2b_2$, etc. A $b_2$ of size $b$ is found within $\log_2(b)$ steps. Thanks to the definition of $L_b$, the total complexity is still $\widetilde{\mathcal{O}}(\psi_b(n))$. □

4.2. **Extending to composite square factors.** Now that we know how to find the square part of integers of the form $n = p^2 b$, we will show in this subsection how this can be extended to composite square factors as well. To do this, we need a version of Proposition 4.1 for integers $n = a^2 b$ with $a, b$ both possibly composite.

The main difference is that the form $f$ can now also be derived from a trivial form that lies in a larger group than $C(-4b)$. There can be many intermediate groups between $C(-4b)$ and $C(-4a^2b)$ if $a$ has many prime factors. In the next lemma we compose transformations from (2.1). We restrict ourselves to transformations with $h \neq \infty$, which will make sense in the proof of Proposition 4.5.

**Lemma 4.4.** *Suppose $f \in C(-4a^2b)$ is derived from $g \in C(-4b)$ using the following transformations (see (2.1)), starting from left to right:*

$$\begin{pmatrix} p_1 & h_1 \\ 0 & 1 \end{pmatrix}, \ldots, \begin{pmatrix} p_r & h_r \\ 0 & 1 \end{pmatrix}$$

*where the $p_i$ are primes (not necessarily distinct) and $p_1 \cdots p_r = a$ and $0 \leq h_i \leq p_i - 1$ for all $i$. Then the transformation matrix of determinant $a$ that maps $g$ to $f$ can be written as*

$$\begin{pmatrix} a & h \\ 0 & 1 \end{pmatrix} \quad \text{where} \quad h = \sum_{i=1}^{r} h_i \prod_{j=1}^{i-1} p_j.$$

*Furthermore, $0 \leq h \leq a - 1$.*

*Proof.* This can be proven by doing induction on $r$, the number of prime factors of $a$. □

We can now state and prove the composite analogue of Proposition 4.1.

**Proposition 4.5.** *Suppose we have a reduced form $f \in C(-4a^2b)$ that is derived from $e_{-4b} = (1,0,b) \in C(-4b)$ and $f \not\sim e_{-4a^2b} = (1,0,a^2b)$. Then there exists a maximal $b_2 = (\frac{a}{a_2})^2 b$ for some $a_2 \mid a$, such that $f$ is derived from $e_{-4b_2}$, and if $b_2 > a_2^2$, then*

$$f = (a_2^2, 2a_2k, k^2 + b_2)$$

*for some $-a_2/2 \leq k \leq a_2/2$.*

*Proof.* Let $b_2 = (\frac{a}{a_2})^2 b$ be maximal such that $f$ is derived from $e_{-4b_2}$. We know that $f$ can be derived from $e_{-4b_2}$ by repeatedly applying the formulas from (2.1) for each prime factor of $a_2$. Claim: only the transformations with $h \neq \infty$ in (2.1) are used to produce $f$ from $e_{-4b_2}$.
In the chain of transformations from $e_{-4b_2}$ to $f$, suppose we have an intermediate form

$$f_2 \in C(-4a_3^2 b_2) = C(-4(\frac{a \cdot a_3}{a_2})^2 b)$$

which is produced from $e_{-4b_2}$ using the prime factors of $a_3 = p_1 \cdots p_r \mid a_2$, and suppose only transformations of the form

$$\begin{pmatrix} p_1 & h_1 \\ 0 & 1 \end{pmatrix}, \ldots, \begin{pmatrix} p_r & h_r \\ 0 & 1 \end{pmatrix}$$

were used up to this point. Suppose furthermore that the next prime $p \mid a_2$ in the chain does use $h = \infty$ to produce $f_3$. We know from Lemma 4.4 that the transformation from $e_{-4b_2}$ to $f_2$ is of the form

$$\begin{pmatrix} a_3 & h \\ 0 & 1 \end{pmatrix} \quad \text{with} \quad h = \sum_{i=1}^{r} h_i \prod_{j=1}^{i-1} p_j.$$

Therefore, the transformation from $e_{-4b_2}$ to $f_3$ can be written as

$$\begin{pmatrix} a_3 & h \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & p \end{pmatrix} = \begin{pmatrix} a_3 & hp \\ 0 & p \end{pmatrix}.$$

However, there is another way to produce $f_3$ from $e_{-4b_2}$. Namely, if we instead start with the prime $p$ and use $h = \infty$ again, then we produce

$$f_2' = (1, 0, p^2 b_2) = e_{-4p^2 b_2}$$

from $e$. Next, we produce $f_3'$ by using the following transformations for the prime factors of $a_3$:

$$\begin{pmatrix} p_1 & p \cdot h_1 \\ 0 & 1 \end{pmatrix}, \ldots, \begin{pmatrix} p_r & p \cdot h_r \\ 0 & 1 \end{pmatrix}$$

where $p \cdot h_i$ can be computed modulo $p_i$ for all $i$. Then using Lemma 4.4 again, we see that the transformation from $e_{-4b_2}$ to $f_3'$ is:

$$\begin{pmatrix} 1 & 0 \\ 0 & p \end{pmatrix} \begin{pmatrix} a_3 & hp \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} a_3 & hp \\ 0 & p \end{pmatrix}.$$

Hence $f_3 = f_3'$. We now see that $f_3$ is also derived from $e_{-4p^2 b_2}$. Therefore, $f$ is also derived from $e_{-4p^2 b_2}$. This contradicts the assumption that $b_2$ was maximal, hence the claim is proven.
To finish the proof of the proposition, we use Lemma 4.4 one final time, together with the claim, to see that the transformation from $e_{-4b_2}$ to $f$ is of the form

$$\begin{pmatrix} a_2 & l \\ 0 & 1 \end{pmatrix} \quad \text{for some } \ 0 \leq l \leq a_2 - 1.$$

We are now in the same situation as Proposition 4.1. By repeating the steps of that proof we see that $f$ is either equal to

$$(a_2^2, 2a_2 l, l^2 + b_2) \quad \text{or to} \quad (a_2^2, 2a_2(l - a_2), (l - a_2)^2 + b_2). \qquad \square$$

We can now state our algorithm in full generality. Proposition 4.5 is harder to use directly than Proposition 4.1, as there are cases where the complete square factor can't be read off immediately from the first coordinate. In the next proposition, we show that it is still possible to find the complete square factor in those cases.

**Proposition 4.6.** *Algorithm 2 can be extended to work for integers $n = a^2b$ having composite square factors, without increasing the asymptotic complexity.*

*Proof.* Until line 23 of Algorithm 2, we don't have to make any adjustments. In line 23, the $x^2$ coefficient of $l$ could be $a^2$. But, we know from Proposition 4.5, that this coefficient can also be a divisor of $a^2$, say $a_2^2$. This happens when our highly composite integer $k$ not only covers the prime factors of $h(-4bs)$, but also those of $h(-4(\frac{a}{a_2})^2bs)$.
To get the full factor $a$ of $n$, we can use Lenstra's original Algorithm 1, together with the same $s$ and $k$, to completely factor $b_2 = (\frac{a}{a_2})^2b$. By reading off the square part of the factorization of $b_2$, we get $\frac{a}{a_2}$. Combining this with the factor $a_2$ we found earlier, we now have computed the square-free decomposition of $n = (a_2\frac{a}{a_2})^2b$. Note that in this case, the only part of $n$ that we possibly don't have the full factorization of is $a_2$.
We only added a single iteration of Algorithm 1 to factor $b_2$. By looking carefully at Theorem 3.4, we see that this only takes

$$\widetilde{\mathcal{O}}(\ln(s)\ln(b_2)(L_b[1/2,1/2]+s)) \subseteq \widetilde{\mathcal{O}}(\psi_b(n))$$

since we use the same prime bound $B$ and multiplier $s$ that we also applied in Algorithm 2 with input $n = a^2b$. Therefore, the asymptotic complexity of this algorithm is the same as Algorithm 2. □

In some applications, we know a rough estimate of the size of $a$ compared to $b$. We can use this to formulate a neater version of the running time.

**Corollary 4.7.** *Fix $0 \le \alpha \le 1$ and assume Assumptions 3.3. Then for all integers $n = a^2b$ with $b = n^\alpha$, Algorithm 2 will find the square-free decomposition (and possibly the full factorization) of $n$ in expected time:*

$$\mathcal{O}(L_b[1/2,1]) = \mathcal{O}(e^{(1+o(1))\sqrt{\ln(b)\ln\ln(b)}}) = \mathcal{O}(L_n[1/2,\sqrt{\alpha}]).$$

*Proof.* First note that for any constant $c$

$$L_b[1/2,c] = L_n[1/2,c\sqrt{\alpha}].$$

Therefore, we can hide the factors $\ln(n)$ of $\psi_b(n)$ in the $o(1)$ terms to get

$$\widetilde{\mathcal{O}}(\psi_b(n)) = \widetilde{\mathcal{O}}(L_b[1/2,1]+L_b[1/2,1/2]) = \widetilde{\mathcal{O}}(L_b[1/2,1]).$$

Finally, we can drop the $\sim$ in $\widetilde{\mathcal{O}}(L_b[1/2,1])$ for the same reason. □

**Remark 4.8.** Let's make some remarks about our new factorization algorithm.

- In cryptographic applications, integers $n = p^2q$ are sometimes used, where $p, q$ are primes of roughly the same size. Thus a good approximation of $b \approx n^{1/3}$ is then known. In cases like these we can also choose $r$ smaller: $r = cn^{1/6}$, for some small $c$ that depends on how much $p$ and $q$ differ. Using this $r$, we still meet the requirement of Proposition 4.1.
  More generally, numbers of the $n = p^kq$ are sometimes used in cryptographic systems. Our algorithm is especially fast when $k$ is even, because then the square-free part of $n$ is only $q$, compared to $pq$ when $k$ is odd.

- The order in which we try the different multipliers $s$ might not be optimal in the current presentation. The optimization of this seems to be somewhat of an art. We analyze this (experimentally) in Appendix B of the longer version of this article [23].
- Instead of taking $r = 3^m$, we could also take $r = 2^m$ for a suitable $m$ (we did not state Corollary 2.4 for this case), or $r$ could be $r = \text{NextPrime}(\sqrt{n})$. The good thing about taking $r$ prime is that $r$ will be only a little bit bigger than $\sqrt{n}$. With $r = 3^m$ for example, you usually overshoot it by quite a bit. Smaller $r$ speed up the arithmetic in the group $C(-4nsr^2)$, which is not very important for this version of the algorithm, but it will make a practical difference in Chapter 5. However, the complexity analysis of the NextPrime function is somewhat messy, so we left it out of this presentation.
- As mentioned before, Lenstra's original algorithm struggled with numbers $n$ that have large prime square divisors. We now see that in that case we actually have a faster way to find at least the square-free decomposition of $n$. We can also restore Lenstra's algorithm to its former glory by amending it as follows. Given $n$, possibly square-free, run Algorithm 2. If an ambiguous form is found, then compute the full factorization of $n$ and stop. If factors $a, b$ are found such that $n = a^2 b$, then completely factor $b$ using Lenstra's original algorithm, which is possible since $b$ is square-free. It is possible that $a$ contains a square factor. Therefore, we repeat this algorithm recursively with the input $a$. There are at most $\log_2 \log_2(n)$ recursion steps. This produces the full factorization of $n$ in expected

  $$\mathcal{O}(L_n[1/2, 1] \ln \ln(n)) = \mathcal{O}(L_n[1/2, 1])$$

  which is the original running time that Schnorr and Lenstra claimed!
- After we discovered the trick of introducing the factor $r$, we found out that Castagnos and Laguillaumie [10] had already discovered it in 2009. In Theorem 3 of that article, they show how to find the square part of the discriminant given a form that is derived from the trivial form in the underlying group. Another article from 2009 [9] does something similar. Using Coppersmith's method, they show that when given a form that is derived from a form of small norm, you can find the square part of the discriminant. An example of such a form is of course the trivial form.
  A natural question for both articles is therefore: how can we find a form that is derived from the trivial form? As we now have seen, Lenstra's algorithm answers this question quite well, which raises the question why this combination has not been spotted before. One possible explanation is that Schnorr and Lenstra's algorithm is from 1984, whereas these articles are much more recent.

Now that we have a solid foundation for our factoring algorithm, we will look at similar algorithms and try to learn from them to optimize ours.

## 5. Stage 2

5.1. **Introduction.** In Algorithm 2 we constructed a large $k$ and computed $g = f^k$ for some $f \in C(-4ns)$. We then hoped that this form is derived from $e_{-4bs}$. If not, then we try the next $s$. Our algorithm is an example of *algebraic-group factorization*,

which is a more general factorization strategy that works in some other groups as well, such as finite fields and elliptic curves.

These algorithms can always be extended with a so-called 'stage 2'. If in stage 1 we used primes up to a bound $B$, then in stage 2 we extend our prime bound to $B_2 > B$. But, instead of doing exactly the same steps with a larger $k$, we now continue with the $g = f^k$ that we computed, and we check for every prime $p \in [B, B_2]$ separately if $g^p$ is derived from $e_{-4bs}$. The idea behind this is that if stage 1 was unsuccessful, then there is a good chance that we are just missing one prime factor $q$ of the order of $C(-4bs)$. Since we only are examining one prime $p$ at the time, we might be able to take $B_2$ quite a bit larger than $B$, without having to increase the runtime by a lot. If the factor $q$ lies in the interval $[B, B_2]$, then we find the square-free decomposition of $n$.

Note that if there are two (or more) primes $q_1, q_2 \in [B, B_2]$ that we are missing from the order of $C(-4bs)$, then we are not going to factor $n$ using stage 2, since we only look at one prime at the time. In stage 1 this problem does not exist. Nevertheless, we will see that on average, having a stage 2 improves the performance of the algorithm.

5.2. **Generic stage 2.** The first known 'stage 2' for algebraic-group factorization algorithms was already discovered in 1974 in Pollard's classic paper [28], Chapter 4. There, it was applied in the group $(\mathbb{Z}/n\mathbb{Z})^*$, which uses the Chinese remainder theorem to see that this group has a subgroup $\mathbb{F}_p^*$, where $p$ is prime and $p \mid n$. If an element in $b \in (\mathbb{Z}/n\mathbb{Z})^*$ is found that is derived from $1 \in \mathbb{F}_p^*$, then the factor $p$ can be found. Derived in this context means that $b = 1 \bmod p$, hence $\gcd(b-1, n)$ will provide $p$. Such $b$ can be found by computing $b = a^k \bmod n$ for a large highly composite $k$ and some random $a$, precisely as what we saw in Algorithms 1, 2. Because of the order of the group $\mathbb{F}_p^*$, methods like Pollard's are called $p-1$ algorithms.

After computing $b = a^k \bmod n$, Pollard continues by first precomputing $b^2, b^4, \ldots, b^{2m} \bmod n$, where $2m$ is the largest gap between two consecutive primes in the interval $[B, B_2]$. The best known unconditional bound on $m$ is $\mathcal{O}(B_2^{0.525})$ [1]. However, most conjectured bounds are much smaller, for example Cramér's bound is $\mathcal{O}(\ln(B_2)^2)$ [15].

Let $p_{t+1} \leq \cdots \leq p_u$ be the primes in the interval $[B, B_2]$. We can now compute $b^r \bmod n$ for each prime $r \in [B, B_2]$ by first computing $c = b^{p_{t+1}} \bmod n$ in $\mathcal{O}(\ln(B))$ multiplications mod $n$. Afterwards, we only have to do one multiplication mod $n$ to compute $b^{p_{t+2}} = c \cdot b^{p_{t+2}-p_{t+1}} \bmod n$, since we precomputed $b^{p_{t+2}-p_{t+1}} \bmod n$. Continuing like this covers all primes in the interval $[B, B_2]$ in $\mathcal{O}(\pi(B_2) - \pi(B))$ multiplications mod $n$, where $\pi$ is the prime counting function.

The prime number theorem states that $\pi(x) \approx x/\ln(x)$ for all large positive integers $x$. Therefore, this stage 2 takes $\mathcal{O}(B_2/\ln(B_2))$ multiplications mod $n$. If we take $B_2 = B\ln(B)$ then the number of multiplications becomes $\mathcal{O}(B)$. This is the same number of multiplications as in stage 1. Brent [6] (Section 9.2) mentions that using this stage 2 improves the runtime of the $p-1$ algorithm by a factor roughly $\ln\ln(p)$.

We call this the 'generic' stage 2 because it can be used in all groups that follow the algebraic-group factorization strategy. So, let's apply it to our algorithm. We won't state Algorithm 2 here again, Algorithm 3 will take place between lines 27 and 28 of Algorithm 2 (except for lines 2, 3 and 4, it is better if they are placed

before the while loop of Algorithm 2). We will state it for integers of the form $n = a^2 b$.

Unfortunately, the compositions of stage 2 will take place in the larger group $C(-4nsr^2)$, because otherwise we would have to lift the form $g^p$ to this group for every prime $p$ in the interval $[B, B_2]$, which is slower. This is where it is useful to take $r$ as small as possible.

**Algorithm 3** Generic stage 2 continuation

**function** STAGE2($n, b_2$)
  $B_2 = B \ln(B)$
  compute the primes $p_{t+1}, \ldots, p_u$ in the interval $[B, B_2]$
  $m = \max_{t+1 \le i \le u-1} (p_{i+1} - p_i)$
  precompute $l^2, l^4, \ldots, l^m \in C(-4nsr^2)$
  $p_{-1} = 0$ ▷ the previous prime
  **for** $p \in [p_{t+1}, \ldots, p_u]$ **do**
    $step = p - p_{-1}$
    $l_2 = l_2 \cdot l^{step}$ ▷ if $p_{-1} \neq 0$ then $l^{step}$ is precomputed
    try to find $a^2$ using the form $l_2$ and Proposition 4.1 or Proposition 4.6
    **if** $a^2$ is found **then**
      compute $a = \sqrt{a^2}$ and $b = n/a^2$
      **return** $a, b$
    **end if**
    $p_{-1} = p$ ▷ update the previous prime
  **end for**
**end function**

In the next proposition we summarize what we know about Algorithm 3.

**Proposition 5.1.** *Assume that the largest prime gap in $[B, B_2]$ is polynomial in $\ln(B_2)$. Then the generic stage 2 continuation of Algorithm 2 does not change the complexity per $s$ that we try. The number of class groups that have to be tried to factor $n = a^2 b$ this way is reduced by a factor roughly $\ln \ln(b)$.*

The generic stage 2 continuation uses $B_2 = B \ln(B)$. This is not bad, but there are methods used in other algebraic-group factorization algorithms that use a whopping $B_2 = B^2$, or something close to it, and still only take $\mathcal{O}(B)$ group operations. This provides a significant speedup, namely a factor of roughly $\ln(n)/\ln \ln(n)$, compared to only using stage 1 [30], page 300. Unfortunately, we were not able to find such a good stage 2 for our algorithm. However, some interesting possible approaches are explored in Appendix C of [23].

## 6. Timings and comparison to other algorithms

6.1. **Complexity comparisons.** In this subsection we will compare our algorithm to three other algorithms: the elliptic curve method (ECM), the number field sieve (NFS) and the lattice methods mentioned in Chapter 1. Let's start with the ECM.

As we saw in Proposition 4.6, given an integer $n = a^2 b$, we can compute the square-free decomposition of $n$ in expected time

$$\widetilde{\mathcal{O}}(L_b[1/2, 1] \ln(n) + L_b[1/2, 1/2] \ln(n)^2).$$

The ECM works a bit differently, if $p$ is the smallest prime factor of $n$, then the ECM can find $p$ in expected $\widetilde{\mathcal{O}}(L_p[1/2, \sqrt{2}]\ln(n))$ time [19] if fast arithmetic is used. Because of the $o(1)$ in the exponent of the $L_p$ function, the logarithmic improvement provided by the better stage 2 of the ECM is not shown in the complexity.

Both algorithms hope to find a group of smooth order. In our algorithm, we work with the class groups $C(-4bs)$, which have size roughly $\sqrt{b}$. In the ECM, you work with elliptic curves $E(\mathbb{F}_p)$, which have size roughly $p$. This is why the $\sqrt{2}$ term is *not* present in the $L$ function of the complexity of our algorithm.

It is good to mention that just like the ECM, there is a natural way to parallelize our algorithm. If you have $m$ processors, give each of them a different $s$ and try to factor $n$ using the class groups $C(-4ns)$. This will provide a linear speed-up in $m$.

If the factors $a, b$ of $n$ are not prime, then the ECM will most likely find *some* factor of $n$ before our algorithm computes the square-free decomposition. However, the ECM might take longer to find the complete square-free decomposition, depending on the structure of the remaining prime factors of $n$. If $n$ is of the form $n = p^2q$, where $p, q$ are primes, then purely looking at the asymptotic complexities, our method will be faster than the ECM when $p^2 > q$. In practice, we might need $p^2$ to be even larger compared to $q$, since the ECM has a better stage 2 and many more optimizations.

From now on, we will assume that $n$ is of the form $n = p^2q$, where $p, q$ are primes of roughly the same size. We do this, because this is the hardest case and it is probably the most common setting where our algorithm can be used. As mentioned in Chapter 1, there are quite a few cryptographic systems that use numbers of this form, where they rely on the assumption that these numbers are hard to factor. In this case $p^2$ is much larger than $q$, so our algorithm will be faster than the ECM when $n$ is large enough.

The number field sieve completely factors its input $n$ in expected time

$$\mathcal{O}(L_n[1/3, (64/9)^{1/3}]) = \mathcal{O}(e^{(64/9+o(1))^{1/3}\ln(n)^{1/3}\ln\ln(n)^{2/3}})$$

[16], page 288. From Corollary 4.7, we know that our method takes expected time

$$\mathcal{O}(L_n[1/2, \sqrt{1/3}]) = \mathcal{O}(e^{(\sqrt{1/3}+o(1))\sqrt{\ln(n)\ln\ln(n)}}).$$

The exponent of $\ln(n)$ in the complexity of the NFS is smaller than in our algorithm. Therefore, when $n$ is large enough, the NFS will factor $n$ faster than our algorithm, even when $n$ is in this special form. However, since the constant $(64/9)^{1/3} \approx 1.92$ is much bigger than $\sqrt{1/3} \approx 0.58$, our method will likely be faster up to some point. If we solve for $n$ in the equation

$$L_n[1/3, (64/9)^{1/3}] = L_n[1/2, \sqrt{1/3}]$$

then we find the massive $n \approx 10^{5613}$. This should be taken with a grain of salt, since this does not take into account any optimizations nor the $o(1)$ terms. But, it seems reasonable to assume that for integers $n = p^2q$ of cryptographic size, our method will be faster than the NFS. Table 2 also supports this claim.

Finally, let's have a look at the lattice methods that can factor integers of the form $n = p^rq$. If $p$ and $q$ are primes of roughly the same size, then a lattice attack can factor $n$ in deterministic $\widetilde{\mathcal{O}}(n^{1/(r+1)^2})$ time [17], Section 1.3. This means that for $r = 2$, this method takes $\widetilde{\mathcal{O}}(n^{1/9})$ time, which is not sub-exponential in $\ln(n)$ and therefore slower than our method if $n$ is large enough. But, if $r$ is large enough, then

the lattice methods will be faster. Especially if $r$ is odd, since then the square-free part of $n$ is $pq$ instead of $p$, significantly slowing down our algorithm.

6.2. **Speed in practice.** We implemented our algorithm in Magma [5] and ran it on a fairly standard desktop (3.35 GHz). Our basic implementation can be found in [22]. For ease of implementation, we used only one core when running each algorithm. But, each of the algorithms below would enjoy an almost linear speed-up in the number of cores. In this subsection we will compare the following four algorithms:

(1) Our combined algorithm, i.e. the combination of Algorithm 2 with the generic stage 2 continuation presented in Algorithm 3.
(2) Only stage 1 of our algorithm, i.e. Algorithm 2.
(3) The elliptic curve method (ECM). We used the optimized built-in version of Magma.
(4) The number field sieve (NFS). We used the optimized CADO-NFS implementation [34].

In Table 1 we compare the first three of these. The numbers that are were factored to create this table were of the form $n = p^2q$, where $p, q$ are primes of roughly the same size.

| | $q \approx 10^{15}$ | $q \approx 10^{20}$ | $q \approx 10^{25}$ | $q \approx 10^{30}$ | $q \approx 10^{35}$ |
|---|---|---|---|---|---|
| Mean time with stage 2 | 0.11 | 0.80 | 5.16 | 30.30 | 135.91 |
| Median time with stage 2 | 0.07 | 0.50 | 3.54 | 23.04 | 80.52 |
| Mean time only stage 1 | 0.16 | 1.63 | 11.65 | 66.71 | 357.74 |
| Median time only stage 1 | 0.10 | 0.98 | 7.80 | 43.91 | 267.61 |
| Mean ECM time | 0.07 | 1.01 | 14.15 | 141.78 | 1549.16 |
| Median ECM time | 0.05 | 0.65 | 11.05 | 102.76 | 832.52 |
| Number of $n$'s factored | 100 | 100 | 100 | 100 | 50 |

Table 1. Comparison between factorization algorithms, in seconds

We see that for smaller inputs the ECM is faster than our method, but for $q \geq 10^{20}$ our method becomes significantly faster. The fact that the ECM is faster for smaller inputs can be explained by the many years of optimizations that were done to improve the method.

If the reader has tried to avoid statistics as much as the author, then they might be wondering why the mean times seem to be larger than the median times. This is explained by the fact that in an exponential distribution, the mean is about $1/\ln(2)$

times the median [32]. For example, for $q = 10^{30}$ we have $23.04/\ln(2) = 33.24 \approx 30.30$.

Our method that picks the multipliers $s$ is quite straightforward, start at $s = 1$ and after each round go to the next square-free value. In Appendix B of [23] some heuristic arguments and numerical results are presented that suggest that there are other strategies to choose your multipliers $s$ that on average improve the runtime of the algorithm. This improvement probably won't change the complexity of the algorithm, but it can make a difference in practice.

In the next table, we show more details about our combined algorithm and we compare it to the NFS for large inputs. In Table 2, the numbers $n = p^2q$ that were factored were of the same form as in Table 1. We see that if $n$ has 150 digits, then on average we can factor it in about 3 hours using the combined algorithm, much faster than with the NFS.

| | $q \approx 10^{20}$ | $q \approx 10^{30}$ | $q \approx 10^{40}$ | $q \approx 10^{50}$ |
|---|---|---|---|---|
| Mean time NFS | 61.17 s | 22.82 m | 572.51 m | $\sim 8$ d [1] |
| Mean time combined alg. | 0.72 s | 33.63 s | 9.74 m | 174.47 m |
| Median time combined alg. | 0.54 s | 20.55 s | 7.20 m | 87.69 m |
| Successful in stage 1 | 27% | 26% | 20% | 24% |
| Successful in stage 2 | 73% | 74% | 80% | 76% |
| Mean number of groups | 6.19 | 26.80 | 63.54 | 206.36 |
| Median number of groups | 5 | 17 | 47 | 103 |
| Number of $n$'s factored | 100 | 100 | 50 | 25 |

TABLE 2. Timing of Algorithm 2 extended with Algorithm 3 compared to the NFS

Some notable examples: when we used

$$q = 372942026756888437229663910319202968572202753882 39,$$

the combined algorithm finished within 43 seconds, this is because the very first group that we tried was successful (in stage 2). On the contrary,

$$q = 533289614734184758945208832227278064453950167777 23$$

[1] 8 days is a rough estimate by extrapolation

took more than 13 hours, when finally the 965th group was successful ($s = 1581$). Upon closer inspection, we see that

$$h = h(-4 \cdot 1581 \cdot q) = 184735851610543000235261184 =$$
$$2^8 \cdot 3 \cdot 19 \cdot 113 \cdot 349 \cdot 6359 \cdot 25031 \cdot 39461 \cdot 51109.$$

We used $B = 229158$, which is bigger than all of the prime factors of $h$, therefore stage 1 was successful. However, if we had a method where we could have taken $B_2 = B^2$, then we would have been done after 51 groups ($s = 82$), since

$$h(-4 \cdot 82 \cdot q) = 12174482046333947562864 4536 =$$
$$2^4 \cdot 5 \cdot 7 \cdot 32633 \cdot 35993 \cdot 153521 \cdot 1161878987.$$

We see that $1161878987 > B$ and also $1161878987 > 2828306 \approx B \ln(B)$. But, $1161878987 < 52513388964 = B^2$. If we had access to such a method, then this factorization would have only taken about 45 minutes instead of 13 hours.

## References


[1] Roger C. Baker, Glyn Harman, and János Pintz. The difference between consecutive primes, II. *Proceedings of the London Mathematical Society*, 83(3):532–562, 2001.

[2] Gaetan Bisson and Andrew V. Sutherland. Computing the endomorphism ring of an ordinary elliptic curve over a finite field. *Journal of Number Theory*, 131(5):815–831, 2011.

[3] Dan Boneh, Glenn Durfee, and Nick Howgrave-Graham. Factoring $N = p^r q$ for large $r$. In *Advances in Cryptology-CRYPTO'99*, pages 326–337, Berlin, Heidelberg, 1999. Springer.

[4] Andrew R. Booker, Ghaith A. Hiary, and Jon P. Keating. Detecting squarefree numbers. *Duke Mathematical Journal*, 2015.

[5] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993). URL: `http://dx.doi.org/10.1006/jsco.1996.0125`.

[6] Richard P. Brent. Some integer factorization algorithms using elliptic curves, 1998. Accessed on: 2023-4-18. URL: `https://arxiv.org/abs/1004.3366`.

[7] Johannes A. Buchmann and Hendrik W. Lenstra Jr. Approximating rings of integers in number fields. *Journal de théorie des nombres de Bordeaux*, 6(2):221–260, 1994.

[8] Duncan A. Buell. *Binary quadratic forms: classical theory and modern computations*. Springer Science & Business Media, 1989.

[9] Guilhem Castagnos, Antoine Joux, Fabien Laguillaumie, and Phong Q. Nguyen. Factoring $pq^2$ with quadratic forms: Nice cryptanalyses. In *Asia-crypt*, volume 9, pages 469–486. Springer, 2009.

[10] Guilhem Castagnos and Fabien Laguillaumie. On the security of cryptosystems with quadratic decryption: the nicest cryptanalysis. In *Advances in Cryptology-EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings 28*, pages 260–277. Springer, 2009.

[11] Alexander Leonidovich Chistov. The complexity of the construction of the ring of integers of a global field. In *Doklady Akademii Nauk*, volume 306:5, pages 1063–1067. Russian Academy of Sciences, 1989.

[12] Henri Cohen and Hendrik W. Lenstra Jr. Heuristics on class groups of number fields. In *Number Theory Noordwijkerhout 1983: Proceedings of the Journées Arithmétiques held at Noordwijkerhout, The Netherlands July 11–15, 1983*, pages 33–62. Springer, 2006.

[13] Jean-Sébastien Coron, Jean-Charles Faugère, Guénaël Renault, and Rina Zeitoun. Factoring $N = p^r q^s$ for large $r$ and $s$. In *Topics in Cryptology-CT-RSA 2016: The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29-March 4, 2016, Proceedings*, pages 448–464. Springer, 2016.

[14] David A. Cox. *Primes of the Form $x^2 + ny^2$: Fermat, Class Field Theory, and Complex Multiplication*. John Wiley & Sons, 2nd edition, 2013.

[15] Harald Cramér. On the order of magnitude of the difference between consecutive prime numbers. *Acta arithmetica*, 2:23–46, 1936.
[16] Richard E. Crandall and Carl Pomerance. *Prime numbers: a computational perspective*. Springer, 2nd edition, 2005.
[17] David Harvey and Markus Hittmeir. A deterministic algorithm for finding $r$-power divisors. *Research in Number Theory*, 8(4):94, 2022.
[18] Arjen K. Lenstra and Hendrik W. Lenstra Jr. *The development of the number field sieve*, volume 1554. Springer Science & Business Media, 1993.
[19] Hendrik W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.
[20] Hendrik W. Lenstra Jr and Carl Pomerance. A rigorous time bound for factoring integers. *Journal of the American Mathematical Society*, 5(3):483–516, 1992.
[21] Alexander May. Using LLL-reduction for solving RSA and factorization problems. In *The LLL Algorithm: Survey and Applications*, pages 315–348. Springer, 2009.
[22] Erik Mulder. Computing the square-free decomposition of integers, 2023. URL: `https://github.com/erik-math/squarefree_decomposition`.
[23] Erik Mulder. Fast square-free decomposition of integers using class groups, 2023. Accessed on: 2023-12-23. URL: `https://arxiv.org/abs/2308.06130`.
[24] Tatsuaki Okamoto. A fast signature scheme based on congruential polynomial operations. *IEEE Transactions on Information Theory*, 36(1):47–53, 1990.
[25] Tatsuaki Okamoto and Shigenori Uchiyama. A new public-key cryptosystem as secure as factoring. In *Advances in Cryptology—EUROCRYPT'98: International Conference on the Theory and Application of Cryptographic Techniques Espoo, Finland, May 31–June 4, 1998 Proceedings 17*, pages 308–318. Springer, 1998.
[26] Sachar Paulus and Tsuyoshi Takagi. A new public-key cryptosystem over a quadratic order with quadratic decryption time. *Journal of Cryptology*, 13:263–272, 2000.
[27] René Peralta and Eiji Okamoto. Faster factoring of integers of a special form. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 79(4):489–493, 1996.
[28] John M. Pollard. Theorems on factorization and primality testing. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 76(3), pages 521–528. Cambridge University Press, 1974.
[29] Katja Schmidt-Samoa. A new Rabin-type trapdoor permutation equivalent to factoring. *Electronic Notes in Theoretical Computer Science*, 157(3):79–94, 2006.
[30] C.P. Schnorr and Hendrik W. Lenstra Jr. A Monte Carlo factoring algorithm with linear storage. *Mathematics of Computation*, 43(167):289–311, 1984.
[31] Arnold Schönhage. Fast reduction and composition of binary quadratic forms. In *Proceedings of the 1991 international symposium on Symbolic and algebraic computation*, pages 128–133, 1991.
[32] Joram Soch. Proof: Median of the exponential distribution. Accessed on: 2023-4-18. URL: `https://statproofbook.github.io/P/exp-med`.
[33] Tsuyoshi Takagi. Fast RSA-type cryptosystem modulo $p^k q$. In *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*, pages 318–326. Springer, 1998.
[34] The CADO-NFS Development Team. CADO-NFS, an implementation of the number field sieve algorithm. Development version 3.0.0. URL: `http://cado-nfs.inria.fr/`.
[35] David Y. Y. Yun. On square-free decomposition algorithms. In *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 26–35, 1976.

*Email address*: `erik-baltasar@live.nl`