Cryptanalysis of the RSA Subgroup Assumption from TCC 2005*

Jean-Sébastien Coron[1], Antoine Joux[2,3], Avradip Mandal[1], David Naccache[4], and Mehdi Tibouchi[1,4]

[1] Université du Luxembourg
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg, Luxembourg
{jean-sebastien.coron,avradip.mandal}@uni.lu
[2] Direction générale de l'armement (DGA)
[3] Université de Versailles–Saint-Quentin, Laboratoire PRISM
45, avenue des États-Unis, F-78035 Versailles CEDEX, France
antoine.joux@m4x.org
[4] École normale supérieure
Département d'informatique, Groupe de cryptographie
45, rue d'Ulm, F-75230 Paris CEDEX 05, France
{david.naccache, mehdi.tibouchi}@ens.fr

Abstract. At TCC 2005, Groth underlined the usefulness of working in small RSA subgroups of hidden order. In assessing the security of the relevant hard problems, however, the best attack considered for a subgroup of size $2^{2\ell}$ had a complexity of $\mathcal{O}(2^{\ell})$. Accordingly, $\ell = 100$ bits was suggested as a concrete parameter.
This paper exhibits an attack with a complexity of roughly $2^{\ell/2}$ operations, suggesting that Groth's original choice of parameters was overly aggressive. It also discusses the practicality of this new attack and various implementation issues.

Key-words: RSA moduli, hidden order, subgroup, cryptanalysis.

1 Introduction

In 2005, Jens Groth [6] proposed a collection of cryptographic primitives based on small RSA subgroups of $\mathbb{Z}_N^*$ of hidden orders. The motivation behind these constructions is improved efficiency and tighter security reductions.

The RSA moduli N used in [6] are of the form:

$$N = p \cdot q = (2p'r + 1) \cdot (2q's + 1)$$

* An extended abstract of this paper will appear at PKC 2011. This is the full version.

where p, p', q, q' are prime integers and r, s are random integers. Then there exists a unique subgroup $\mathbb{G}$ of $\mathbb{Z}_N^*$ of order $p'q'$. Letting g be a random generator of $\mathbb{G}$, the pair (N, g) is made public whereas everything else including the group order $p'q'$ is kept secret.

The best attack considered in [6] has complexity $\mathcal{O}(p')$. Therefore, when proposing concrete parameters, the author suggests to take the bit-lengths $\ell_{p'}$ and $\ell_{q'}$ of the primes p' and q' as $\ell_{p'} = \ell_{q'} = 100$.

This paper does not consider any specific scheme from [6]. Instead, it describes an attack that recovers the secret factors of N from the public data (N, g) in $\tilde{\mathcal{O}}(\sqrt{p'})$. This results in a 2^{50} attack making the choice $\ell_{p'} = \ell_{q'} = 100$ potentially insecure. We analyze the practicality of our attack with an implementation, for which we provide the source code in the Appendix.

Remark 1. In [6], Groth also considers RSA subgroups where r and s are smooth integers (*i.e.* all prime factors of r and s are smaller than some bound B). For this specific variant an attack in complexity $\mathcal{O}(\sqrt{p'})$ is given in [6], and consequently larger parameters ($\ell_{p'} = \ell_{q'} = 160$) are suggested. In this paper we do not consider this variant but directly focus on the general case.

Remark 2. Other works have proposed schemes based on small subgroups of $\mathbb{Z}_n^*$. The attack introduced in this paper applies to some, but not all of them. In particular, the scheme proposed by Damgård *et al.* in [5] uses a subgroup of *prime* order v of $\mathbb{Z}_n^*$, where v is a factor of both $p-1$ and $q-1$ (of around 160 bits). Since the group has the same order modulo p and q, the attack presented herein does not apply to this scheme. On the other hand, it does, in principle, apply to the subgroup variant of the Paillier cryptosystem [10]. The parameter choice from the original paper was more conservative than that of Groth, however (320-bit subgroup), making it out of reach of our new attack.

2 The New Attack

Using the notations above, we factor N in time $\tilde{\mathcal{O}}(\sqrt{p'})$ and memory $\mathcal{O}(\sqrt{p'})$ as follows. Recall that the RSA modulus $N = pq$ is such that:

$$N = p \cdot q = (2p'r + 1) \cdot (2q's + 1)$$

where p' and q' are prime; besides, g is a generator of the subgroup $\mathbb{G}$ of order $p'q'$. From $g^{p'q'} = 1 \mod N$ we get:

$$g^{p'} = 1 \mod p \tag{1}$$

Let ℓ denote the bit-length of p', which we assume is even without loss of generality, and write $\Delta = 2^{\ell/2}$. We then have

$$p' = a + \Delta \cdot b$$

with $0 \leq a, b < 2^{\ell/2}$. From (1), we get:

$$g^a = (g^{\Delta})^{-b} \mod p$$

If the prime factor p was known, one could carry out a baby-step giant-step attack by generating the following two lists:

$$L_p = \{g^i \bmod p : 0 < i < 2^{\ell/2}\}$$
$$L'_p = \{(g^{\Delta})^{-j} \bmod p : 0 \leq j < 2^{\ell/2}\}$$

and finding a collision between L_p and L'_p, which would reveal a, b and thus p' in total time and space $\mathcal{O}(2^{\ell/2})$.

Obviously p is unknown, so instead of computing L_p and L'_p, we generate the two following lists modulo N:

$$L = \{x_i = g^i \bmod N : 0 < i < 2^{\ell/2}\}$$
$$L' = \{y_j = (g^{\Delta})^{-j} \bmod N : 0 \leq j < 2^{\ell/2}\}$$

One could then compute $\gcd(x_i - y_j, N)$ for all $x_i \in L$ and all $y_j \in L'$. Since we have

$$x_a - y_b = 0 \mod p$$

this would reveal the factors of N for $i = a$ and $j = b$. However, the complexity of this naive approach is quadratic in Δ, and will thus require 2^{ℓ} computations, not $2^{\ell/2}$. Hence we proceed as follows instead:

1. Generate the polynomial:

$$f(x) = \prod_{x_i \in L} (x - x_i) \mod N$$

2. For all $y_j \in L'$, evaluate f at y_j and compute $\gcd(f(y_j), N)$.

Since we have

$$f(y_b) = \prod_{x_i \in L} (y_b - x_i) = (y_b - x_a) \cdot R = 0 \mod p$$

computing $\gcd(f(y_j), N)$ reveals the factors of N for $j = b$.

The attack is summarized in Algorithm 1. In the next section we show that it can be carried out in time quasi-linear in the cardinalities of L and L'.

Algorithm 1 Attack overview.

1: Let $\Delta \leftarrow 2^{\ell/2}$.
2: **for** $i = 0$ to $\Delta - 1$ **do**
3: $\quad x_i \leftarrow g^i \mod N$
4: $\quad y_i \leftarrow (g^{\Delta})^{-i} \mod N$
5: **end for**
6: Generate the polynomial

$$f(x) \leftarrow \prod_{i=1}^{\Delta-1}(x - x_i) \mod N$$

7: **for** $i = 0$ to $\Delta - 1$ **do**
8: $\quad$ Evaluate $f(y_i) \in \mathbb{Z}_N$
9: $\quad$ Attempt to factor N by computing $\gcd(f(y_i), N)$.
10: **end for**

3 Attack Complexity

This attack involves the computation and evaluation of a polynomial of the form:

$$f(x) = \prod_{i=1}^{d-1}(x - x_i) \mod N$$

with $d = 2^{\ell/2}$. It is a classical fact [1] that the coefficients of such a polynomial can be computed using a product tree, with a total number of operations in $\mathbb{Z}_N$ which is quasilinear in d (namely $\mathcal{O}(M(d)\log d)$, where $M(d)$ is the complexity of the multiplication of two polynomials of degree d). Similarly, with a remainder tree, this polynomial can be evaluated at all points y_j, $0 \leq j < d$ in $\mathcal{O}(M(d)\log d)$ operations.

Since this approach is easy to follow, we provide the source code of an implementation using this technique in Appendix C in SAGE [12] for the reader's convenience.

In our case, however, both (x_i) and (y_j) are geometric progressions, hence even more efficient algorithms exist: the Newton basis conversion algorithms of Bostan and Schost [4] make it possible to compute f using $\mathcal{O}(d)$ precomputations and a single *middle product* of polynomials of degree d, and to evaluate $f(y_j)$ for all j using $\mathcal{O}(d)$ precomputations, a product of polynomials of degree d and a middle product of polynomials of degree d. See the next section for details. This results in an overall complexity of $3M(d) + \mathcal{O}(d)$ for the complete attack, with a small constant in the $\mathcal{O}$. Space requirements are also $\mathcal{O}(d)$, to store a few polynomials of degree d.

Thus, for typical parameter sizes, the attack is *essentially linear in* $\sqrt{p'}$ *both in time and space.*

4 Algorithmic Details

As discussed above, we can break down the attack in two steps: first compute the coefficients of the polynomial $f(x) = \prod_{i=1}^{d-1}(x - x_i) \mod N$, and then evaluate $f \mod N$ at each of the points y_j. Since both (x_i) and (y_j) are geometric progressions, both of these steps reduce to a variant of the discrete Fourier transform, called the "chirp transform" (or its inverse) [11, 2]. In our implementation, we carry out these computations using the particularly efficient algorithms of Bostan and Schost [4], as described in [3, §5.5]. More precisely, Bostan gives pseudocode, reproduced as Algorithms 3 and 4 in Appendix A, to compute polynomial interpolation and polynomial evaluation at a geometric progression.

In our case, a number of futher simplifications are possible in the interpolation stage. Indeed, $f(x_i) = 0$ for $1 \leq i \leq d-1$ and $f(1) = \prod_{i=1}^{d-1}(1 - x_i)$, so with the notations of Algorithm 3, $v_0 = (-1)^{n-1}s_{n-1}$ and $v_i = 0$ for $i > 0$. This means in particular that the polynomial multiplication of Algorithm 3, Step 9 reduces to a simple scalar multiplication, and that the computations of Steps 10–12 can be carried out in the main loop. We can also have a slightly more conservative memory management, with only 4 polynomials of degree $n - 1$ kept in memory for both the interpolation and the evaluation step. Finally, the multiplications by s_i in Algorithm 4, Step 14 can be skipped altogether as we search for a factor of N with GCD computations, since the s_i's are prime to N by construction. We obtain the detailed procedure described in Algorithm 2.

The attack can again be broken down in three stages: interpolation in Steps 2–18, evaluation in Steps 19–36 and factor search in Steps 37–40. Complexity is dominated by the three quasi-linear multiplication steps: the middle products of Steps 15 and 32, and the polynomial multiplication of Step 36.

5 Implementation

We provide the source code of our attack in Appendix B. The implementation of polynomial interpolation and evaluation using Newton basis conversions largely follows the pseudocode from [3] (Figure 5.1 and 5.2), implemented in C using the FLINT library [8].

Algorithm 2 Detailed attack.

function ATTACK(g, n, N)
  $p \leftarrow 1$; $q \leftarrow 1$; $s \leftarrow 1$; $u \leftarrow 1$; $z \leftarrow 1$; $w \leftarrow 1$
  Initialize U, Z, S, W as zero polynomials of degree $n-1$
  $U_0 \leftarrow u$; $Z_0 \leftarrow z$; $W_0 \leftarrow w$
  for $i = 1$ to $n-1$ **do**
    $p \leftarrow p \cdot g \bmod N$
    $q \leftarrow q \cdot p \bmod N$
    $s \leftarrow s \cdot (p-1) \bmod N$
    $u \leftarrow u \cdot p/(1-p) \bmod N$
    $z \leftarrow (-1)^i u/q \bmod N$
    $w \leftarrow q/(s \cdot u) \bmod N$
    $U_i \leftarrow u$; $Z_i \leftarrow z$; $W_i \leftarrow w$
  end for
  $Z \leftarrow (-1)^{n-1} s_{n-1} \cdot Z \bmod N$
  $W \leftarrow \mathrm{mul}^t(n-1, U, W)$
  for $i = 0$ to $n-1$ **do**
    $W_i \leftarrow W_i \cdot Z_i \bmod N$
  end for
  $g \leftarrow 1/(p \cdot g) \bmod N$ ▷ $g \leftarrow g^{-\Delta}$
  $p \leftarrow 1$; $q \leftarrow 1$; $s \leftarrow 1$; $u \leftarrow 1$; $z \leftarrow 1$; $w \leftarrow 1$
  $U \leftarrow 0$; $Z \leftarrow 0$
  $U_0 \leftarrow u$; $Z_0 \leftarrow z$; $S_0 \leftarrow s$
  for $i = 1$ to $n-1$ **do**
    $p \leftarrow p \cdot g \bmod N$
    $q \leftarrow q \cdot p \bmod N$
    $s \leftarrow s/(p-1) \bmod N$
    $u \leftarrow u \cdot p/(1-p) \bmod N$
    $z \leftarrow (-1)^i u/q \bmod N$
    $S_i \leftarrow s$; $U_i \leftarrow u$; $Z_i \leftarrow z$
    $W_i \leftarrow W_i/z \bmod N$
  end for
  $W \leftarrow \mathrm{mul}^t(n-1, Z, W)$
  for $i = 0$ to $n-1$ **do**
    $W_i \leftarrow (-1)^i W_i \cdot U_i \bmod N$
  end for
  $W \leftarrow W \cdot S$
  for $i = 0$ to $n-1$ **do**
    if $\gcd(W_i, N) \neq 1$ **then return** $\gcd(W_i, N)$ ▷ Factor found!
    end if
  end for
end function

In Table 1, we provide the observed running time of our attack on an Intel Core2 Duo E8500 3.12 GHz, for 1024-bit RSA moduli. The program was linked to the following libraries: FLINT 1.6 (prerelease), MPIR 2.1.3 and MPFR 3.0, and ran on a single CPU core.

$\ell = \lceil \log_2 p' \rceil$	**running time**
26 bits	1.9 seconds
28 bits	4.0 seconds
30 bits	8.1 seconds
32 bits	16.5 seconds
34 bits	33.5 seconds
36 bits	68.9 seconds

Table 1. Experimental attack running times for 1024-bit moduli.

From Table 1 we see that, as expected, running times are essentially linear in $\sqrt{p'}$. Direct extrapolation yields the following estimates:

$\ell = \lceil \log_2 p' \rceil$	**running time**	**estimated number of clock cycles**
60 bits	3 days	2^{50}
80 bits	9 years	2^{60}
100 bits	9000 years	2^{70}

Table 2. Estimated attack running times for 1024-bit moduli.

Thus, even the parameter $\ell = 100$ suggested in Groth's paper [6] would require a large but not unachievable amount of computation, even by academic standards. As a comparison, the recent factorization of RSA 768 [9] required about 2000 CPU-years.

However, it is not obvious how the algorithm can be efficiently parallelized to distribute the computation. A naive parallelization strategy is to reduce the number of x_i's and increase the number of y_i's by some factor 2^k, but this only reduces time and memory by a factor of about 2^k while requiring 2^{2k} parallel nodes. It would be desirable to be able to distribute the full size computation—both the FFT steps (multiplication and middle product) and the precomputations—but this appears to be nontrivial.

Most importantly, it is difficult to deal with larger parameters because the attack is heavily memory-bound: the $\mathcal{O}(\sqrt{p'})$ memory requirement is a serious hurdle. In experiments, we encountered memory problems as early as $\ell \approx 38$ for a 1024-bit modulus, and even with much more careful memory management and the use of mass storage rather than RAM, it seems unlikely that parameters larger than $\ell \approx 60$ can be attacked unless storage can be efficiently distributed as well.

6 Conclusion

We have described an attack against the RSA subgroup of hidden order described in [6] that works in time $\tilde{\mathcal{O}}(\sqrt{p'})$ while the best attack considered in [6] had complexity $\mathcal{O}(p')$. We have implemented our attack and assessed its practicality. As expected, our attack exhibits a time complexity quasi-linear in $\sqrt{p'}$. In terms of CPU time alone, the parameters suggested in [6] appear to be within reach for a resourceful attacker. However, due to heavy memory requirements and parallelization problems, these parameters may remain unchallenged.

An interesting open question is to decrease the memory requirement: an algorithm similar to Pollard rho or Pollard lambda with constant memory would be the most convenient type of attack on this problem if it exists. If not, a method for distributing the computation and storage efficiently would be the simplest way to make the attack practical for larger parameters.

Acknowledgments. We are grateful to Luca De Feo, Marc Mezzarobba and anonymous referees for useful comments. This work was partly supported by the French ANR-07-TCOM-013-04 PACE Project and by the European Commission through the IST Program under Contract ICT-2007-216646 ECRYPT II.

References

1. D.J. Bernstein, *Fast multiplication and its applications*, Algorithmic number theory: lattices, number fields, curves and cryptography, MSRI Publications, vol. 44, Cambridge University Press, 2008, pp. 325–384.
2. L.I. Bluestein, *A linear filtering approach to the computation of the discrete Fourier transform*, IEEE Trans. Electroacoustics, vol. 18, 1970, pp. 451–455.
3. A. Bostan, *Algorithmique efficace pour des opérations de base en calcul formel* (in English), Ph.D. thesis, École polytechnique, 2003.
4. A. Bostan and E. Schost, *Polynomial evaluation and interpolation on special sets of points*, Journal of Complexity, vol. 21(4), Elsevier, 2005, pp. 420–446.

5. I. Damgård, M. Geisler and M. Krøigaard, *Efficient and secure comparison for on-line auctions*, Proceedings of ACISP 2007, LNCS, vol. 4586, Springer-Verlag, 2007, pp. 416–430.
6. J. Groth, *Cryptography in subgroups of* $\mathbb{Z}_n^*$, Proceedings of TCC 2005, LNCS, vol. 3378, Springer-Verlag, 2005, pp. 50–65.
7. G. Hanrot, M. Quercia and P. Zimmermann, *The middle product algorithm, I.*, Applicable Algebra in Engineering, Communication and Computing, vol. 14, Springer-Verlag, 2004, pp. 415–438.
8. W.B. Hart, D. Harvey *et al.*, *Fast library for number theory*, `www.flintlib.org`.
9. T. Kleinjung, K. Aoki, J. Franke, A.K. Lenstra, E. Thomé, J.W. Bos, P. Gaudry, A. Kruppa, P.L. Montgomery, D.A. Osvik, J.J. te Riele, A. Timofeev and P. Zimmermann, *Factorization of a 768-bit RSA modulus*, Proceedings of CRYPTO 2010, LNCS, vol. 6223, Springer-Verlag, 2010, pp. 333–350.
10. P. Paillier and D. Pointcheval, *Efficient public-key cryptosystems provably secure against active adversaries*, Proceedings of ASIACRYPT 1999, LNCS, vol. 1716, Springer-Verlag, 2004, pp. 165–179.
11. L.R. Rabiner, R.W. Schafer and C.M. Rader, *The chirp z-transform algorithm and its applications*, Bell System Tech. J., vol. 48, 1969, pp. 1249–1292.
12. Sage Library, available at `http://www.sagemath.org/`

A Bostan's algorithms

Algorithm 3 Polynomial interpolation: compute the polynomial F of degree $< n$ such that $F(p_i) = v_i$, where $p_i = q^i$, $0 \leq i \leq n-1$.

function InterpGeom($p_0, \ldots, p_{n-1}; v_0, \ldots, v_{n-1}$)
  $q_0 \leftarrow 1$; $s_0 \leftarrow 1$; $u_0 \leftarrow 1$; $z_0 \leftarrow 1$; $w_0 \leftarrow v_0$
  for $i = 1$ to $n-1$ **do**
    $q_i \leftarrow q_{i-1} \cdot p_i$
    $s_i \leftarrow s_{i-1} \cdot (p_i - 1)$
    $u_i \leftarrow u_{i-1} \cdot p_i/(1-p_i)$
    $z_i \leftarrow (-1)^i u_i/q_i$
  end for
  $H \leftarrow \left(\sum_{i=0}^{n-1} v_i/s_i x^i\right) \cdot \left(\sum_{i=0}^{n-1} (-x)^i q_i/s_i\right)$
  for $i = 1$ to $n-1$ **do**
    $w_i \leftarrow (-1)^i \, \texttt{Coeff}(H, i)/u_i$
  end for
  $G \leftarrow \mathrm{mul}^t(n-1, \sum_{i=0}^{n-1} u_i x^i, \sum_{i=0}^{n-1} w_i x^i)$
  return $\sum_{i=0}^{n-1} z_i \, \texttt{Coeff}(G, i) x^i$
end function

B Source code of the attack

```
#include <stdio.h>
```

Algorithm 4 Polynomial evaluation: evaluate the polynomial F at all points $p_i = q^i$, $0 \le i \le n-1$.

function EvalGeom($p_0, \ldots, p_{n-1}; F$)
  $q_0 \leftarrow 1$; $s_0 \leftarrow 1$; $u_0 \leftarrow 1$; $z_0 \leftarrow 1$; $g_0 \leftarrow 1$
  for $i = 1$ to $n-1$ **do**
    $q_i \leftarrow q_{i-1} \cdot p_i$
    $s_i \leftarrow s_{i-1} \cdot (p_i - 1)$
    $u_i \leftarrow u_{i-1} \cdot p_i/(1-p_i)$
    $z_i \leftarrow (-1)^i u_i / q_i$
  end for
  $G \leftarrow \mathrm{mul}^t(n-1, \sum_{i=0}^{n-1} z_i x^i, \sum_{i=0}^{n-1} \mathtt{Coeff}(F,i)/z_i x^i)$
  for $i = 1$ to $n-1$ **do**
    $g_i \leftarrow (-1)^i u_i \, \mathtt{Coeff}(G,i)$
  end for
  $W \leftarrow \left(\sum_{i=0}^{n-1} g_i x^i\right) \cdot \left(\sum_{i=0}^{n-1} s_i^{-1} x^i\right)$
  return $s_0 \, \mathtt{Coeff}(W,0), \ldots, s_{n-1} \, \mathtt{Coeff}(W, n-1)$
end function

```
#include <time.h>
#include <gmp.h>
#include "F_mpz_poly.h"
#include "F_mpz.h"

void F_mpz_poly_set_coeff_F_mpz(F_mpz_poly_t, ulong, const F_mpz_t);


void attack(F_mpz_t q, F_mpz_t m, unsigned long n)
{
    F_mpz_poly_t polW, polU, polZ, polS;
    F_mpz_t pi, qi, si, ui, zi, wi, x;
    unsigned long i;

    printf("Attack started.\n");

    F_mpz_poly_init2(polW, n);
    F_mpz_poly_init2(polU, n);
    F_mpz_poly_init2(polZ, n);
    F_mpz_poly_init2(polS, n);

    /* Step 1: interpolation */
    F_mpz_init(pi); F_mpz_set_ui(pi, 1);
    F_mpz_init(qi); F_mpz_set_ui(qi, 1);
```

```
F_mpz_init(si); F_mpz_set_ui(si, 1);
F_mpz_init(ui); F_mpz_set_ui(ui, 1);
F_mpz_init(zi); F_mpz_set_ui(zi, 1);
F_mpz_init(wi); F_mpz_set_ui(wi, 1);
F_mpz_init(x);

F_mpz_poly_set_coeff_F_mpz(polU, n-1, ui);
F_mpz_poly_set_coeff_F_mpz(polZ, 0, zi);
F_mpz_poly_set_coeff_F_mpz(polW, 0, wi);

for(i=1; i<n; i++) {
    F_mpz_mulmod2(qi, qi, pi, m);
    F_mpz_mulmod2(pi, pi, q, m);

    /* s_i = s_{i-1} * (p_i - 1) */
    F_mpz_sub_ui(x, pi, 1);
    F_mpz_mulmod2(si, si, x, m);

    /* u_i = u_{i-1} * p_i/(1 - p_i) */
    F_mpz_invert(x, x, m);
    F_mpz_mul2(ui, ui, pi);
    F_mpz_mul2(ui, ui, x);
    F_mpz_neg(ui, ui);
    F_mpz_mod(ui, ui, m);

    F_mpz_poly_set_coeff_F_mpz(polU, n-1-i, ui);

    /* z_i = (-1)^i u_i / q_i */
    F_mpz_invert(x, qi, m);
    F_mpz_mulmod2(x, x, ui, m);
    if(i & 1)
        F_mpz_neg(zi, x);
    else
        F_mpz_set(zi, x);

    F_mpz_poly_set_coeff_F_mpz(polZ, i, zi);

    /* w_i = q_i / (s_i * u_i) */
    F_mpz_mul2(x, x, si);
    F_mpz_invert(wi, x, m);
```

```
    F_mpz_poly_set_coeff_F_mpz(polW, i, wi);
}

/* W *= (-1)^{n-1} s_{n-1} */
if(!(n & 1))
    F_mpz_neg(si, si);
F_mpz_poly_scalar_mul(polW, polW, si);
F_mpz_poly_scalar_smod(polW, polW, m);

F_mpz_poly_mul_trunc_left(polW, polU, polW, n-1);
F_mpz_poly_right_shift(polW, polW, n-1);

for(i=0; i<n; i++) {
    F_mpz_mulmod2(polW->coeffs + i, polW->coeffs + i, polZ->coeffs + i, m);
}

printf("Polynomial interpolation complete.\n");

/* Step 2: evaluation */
F_mpz_mul2(q, q, pi);
F_mpz_invert(q, q, m);

F_mpz_set_ui(pi, 1);
F_mpz_set_ui(qi, 1);
F_mpz_set_ui(si, 1);
F_mpz_set_ui(ui, 1);
F_mpz_set_ui(zi, 1);
F_mpz_set_ui(wi, 1);

F_mpz_poly_zero(polU);
F_mpz_poly_zero(polZ);

F_mpz_poly_set_coeff_F_mpz(polU, 0, ui);
F_mpz_poly_set_coeff_F_mpz(polZ, n-1, zi);
F_mpz_poly_set_coeff_F_mpz(polS, 0, si);

for(i=1; i<n; i++) {
    F_mpz_mulmod2(qi, qi, pi, m);
    F_mpz_mulmod2(pi, pi, q, m);
```

```
    /* s_i = s_{i-1} / (p_i - 1) */
    F_mpz_sub_ui(x, pi, 1);
    F_mpz_invert(x, x, m);
    F_mpz_mulmod2(si, si, x, m);

    F_mpz_poly_set_coeff_F_mpz(polS, i, si);

    /* u_i = u_{i-1} * p_i/(1 - p_i) */
    F_mpz_mul2(ui, ui, pi);
    F_mpz_mul2(ui, ui, x);
    F_mpz_neg(ui, ui);
    F_mpz_mod(ui, ui, m);

    F_mpz_poly_set_coeff_F_mpz(polU, i, ui);

    /* z_i = (-1)^i u_i / q_i */
    F_mpz_invert(x, qi, m);
    F_mpz_mulmod2(x, x, ui, m);
    if(i & 1)
        F_mpz_neg(zi, x);
    else
        F_mpz_set(zi, x);

    F_mpz_poly_set_coeff_F_mpz(polZ, n-1-i, zi);

    /* w_i /= z_i */
    F_mpz_invert(x, zi, m);
    F_mpz_mulmod2(polW->coeffs + i, polW->coeffs + i, x, m);
}

F_mpz_poly_mul_trunc_left(polW, polZ, polW, n-1);
F_mpz_poly_right_shift(polW, polW, n-1);

F_mpz_poly_clear(polZ);

for(i=0; i<n; i++) {
    if(i & 1)
        F_mpz_neg(polU->coeffs + i, polU->coeffs + i);
    F_mpz_mulmod2(polW->coeffs + i, polW->coeffs + i, polU->coeffs + i, m);
```

```
    }

    F_mpz_poly_clear(polU);
    F_mpz_poly_mul(polW, polW, polS);
    F_mpz_poly_clear(polS);

    printf("Evaluation complete. Searching for a factor.\n");
    for(i=0; i<n; i++) {
        F_mpz_gcd(x, polW->coeffs + i, m);
        if(!F_mpz_is_one(x)) {
            printf("Factor found!\n");
            F_mpz_print(x);
            printf("\n");
            break;
        }
    }

    F_mpz_poly_clear(polW);

    F_mpz_clear(pi);
    F_mpz_clear(qi);
    F_mpz_clear(si);
    F_mpz_clear(ui);
    F_mpz_clear(zi);
    F_mpz_clear(wi);
    F_mpz_clear(x);
}


int main()
{
    F_mpz_t m, g;
    unsigned long d;
    clock_t c0, c1;

    printf("Enter parameters N, g, d.\n");
    F_mpz_init(m);
    F_mpz_init(g);

    F_mpz_read(m);
```

```
    F_mpz_read(g);
    scanf("%lu", &d);

    printf("\nParameters:\nN = ");
    F_mpz_print(m);
    printf("\ng = ");
    F_mpz_print(g);
    printf("\nd = %lu\n\n", d);

    c0 = clock();
    attack(g, m, 1L<<d);
    c1 = clock();

    printf("Elapsed time: %.3f seconds.\n",
            ((float)(c1-c0))/CLOCKS_PER_SEC);
}
```

C Simpler, less efficient tree-based version in SAGE

```
import sys,time

def GenSubgroup(n=30,l=100):
  "Generates a prime p=2*pp*r+1 of size l bits, with pp of size n bits"
  while True:
    pp=random_prime(2^n,lbound=2^(n-1))
    r=ZZ.random_element(2^(l-n))
    if 2*pp*r+1 in Primes(): break
  return 2*pp*r+1,pp,r

def ProdPoly(X,xi):
  "Computes \prod (X-x_i)"
  l=len(xi)
  if l==1: return X-xi[0]
  return ProdPoly(X,xi[:int(l/2)])*ProdPoly(X,xi[int(l/2):])

def ProdPolyTree(X,yi):
  "Returns the multiplication tree for polymomial \prod (X-y_i)"
  l=len(yi)
  if l==1: return (X-yi[0],False,False)
  t1=ProdPolyTree(X,yi[:int(l/2)])
```

```
  t2=ProdPolyTree(X,yi[int(l/2):])
  return (t1[0]*t2[0],t1,t2)

def EvaluationTree(f,tr):
  "Evaluates polynomial f using the multiplication tree for \prod (X-y_i)"
  (p,t1,t2)=tr
  if t1==False: return [f.quo_rem(p)[1]]
  f1=f.quo_rem(t1[0])[1]
  f2=f.quo_rem(t2[0])[1]
  return EvaluationTree(f1,t1)+EvaluationTree(f2,t2)

def Evaluation(X,f,yi):
  return EvaluationTree(f,ProdPolyTree(X,yi))

def Attack(n=24,l=512):
  print "n=",n
  print "Key generation"
  p,pp,r=GenSubgroup(n,l); q,qq,s=GenSubgroup(n,l)
  NN=p*q
  #print "NN=",NN,"\np=",p,"\nq=",q,"\npp=",pp,"\nqq=",qq
  g=ZZ.random_element(NN)
  g=power_mod(g,2*r*s,NN)
  k=2^(n//2)
  t0=time.time()
  print "Generation of the x_i's and y_i's"
  t=time.time()
  xi=[1]
  for i in range(1,k-1):
    xi.append(mod(xi[-1]*g,NN))
  yi=[power_mod(g,2^n,NN)]
  h=power_mod(g,-k,NN)
  for i in range(1,k):
    yi.append(mod(yi[-1]*h,NN))
  print "  Done in %.2f s" % (time.time()-t)

  R=PolynomialRing(IntegerModRing(NN),'X')
  X=R.gen()

  print "Construction of polynomial f(x)"
  t=time.time()
```

```
  f=ProdPoly(X,xi)
  print "  Done in %.2f s" % (time.time()-t)

  print "Evaluation of f(x)"
  t=time.time()
  fi=Evaluation(X,f,yi)
  print "  Done in %.2f s" % (time.time()-t)

  for f in fi:
    u=gcd(f,NN)
    if u!=1 and u!=NN:
      print "Total running time: %.2f s\n" % (time.time()-t0)
      return u==p or u==q

def test(l=512):
  for n in range(24,36,2):
    Attack(n,l)
```