Cryptanalysis of the RSA subgroup assumption from TCC 2005

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

[1]Université du Luxembourg

[3]Université de Versailles–Saint-Quentin/DGA

[2]École normale supérieure

PKC 2011

Outline

Context

- Cryptography in subgroups of $\mathbb{Z}_N^*$
- Original security analysis

Our contribution

- The new attack
- Implementation

Outline

Groth's paper from TCC 2005

- In a paper presented at TCC 2005, Groth showed how to construct a number of cryptographic primitives using small subgroups $\mathbb{G}$ of hidden order in $\mathbb{Z}_N^*$.
- N is an RSA modulus of a special form:

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

  (p, q prime; p', q' prime divisors of $p-1, q-1$; r, s random integers). Then $\mathbb{G}$ is the unique subgroup of $\mathbb{Z}_N^*$ of order $p'q'$.
- Based on a computational assumption similar to Strong RSA but restricted to $\mathbb{G}$, Groth proposes standard model constructions for:
  - EUF-ACMA-secure signatures;
  - statistically hiding, computationally binding commitments;
  - IND-CPA-secure encryption (for a slightly different N).
- Due to the relatively small size of $\mathbb{G}$, these schemes tend to be more efficient than Strong RSA-based constructions.

The strong RSA subgroup assumption

In the schemes proposed by Groth, the public key contains an RSA subgroup pair (N, g), consisting of N as above:

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

and a generator g of the subgroup $\mathbb{G} \subset \mathbb{Z}_N^*$ of order $p'q'$.

Then, the security is based on the following assumption on the RSA subgroup pairs (N, g) produced by the key generation algorithm.

Definition (Strong RSA subgroup assumption)

It is infeasible to find $u, v \in \mathbb{Z}_N^*$ and $d, e > 1$ such that:

$$g = uw^e \bmod N \quad \text{and} \quad u^d = 1 \bmod N$$

In particular, it should be hard to find e-th roots of g for any e.

Outline

Factorization attacks

Consider an RSA subgroup pair (N, g), with $N = p \cdot q = (2p'r + 1) \cdot (2q's + 1)$. Groth made the following observations.

- If an attacker can find the hidden subgroup order $p'q'$ or factor N, she can compute e-th roots of g and thus break the strong RSA subgroup assumption.
- We have $g^{p'} = 1 \bmod p$, so if an attacker can find p', she can recover $p = \gcd(N, g^{p'} - 1)$, factor N, and break the assumption again.

Concrete parameters

As seen previously, the bit lengths $\ell_N, \ell_{p'}, \ell_{q'}$ of N, p', q' should be chosen large enough that:

1. factoring N is infeasible;
2. recovering p', q' or the hidden order $p'q'$ is infeasible.

Since no better attacks on the problem are known, Groth suggested concrete parameters based on these two criteria:

1. $\ell_N = 1024$ (for roughly 80 bits of security against GNFS);
2. $\ell_{p'} = \ell_{q'} = 100$, as Pollard's lambda method gives a method to recover the hidden group order in $O(\sqrt{p'q'})$ time and constant space (the choice gives 100 bits of security against this attack).

Concrete parameters

As seen previously, the bit lengths $\ell_N, \ell_{p'}, \ell_{q'}$ of N, p', q' should be chosen large enough that:

1. factoring N is infeasible;
2. recovering p', q' or the hidden order $p'q'$ is infeasible.

Since no better attacks on the problem are known, Groth suggested concrete parameters based on these two criteria:

1. $\ell_N = 1024$ (for roughly 80 bits of security against GNFS);
2. $\ell_{p'} = \ell_{q'} = 100$, as Pollard's lambda method gives a method to recover the hidden group order in $O(\sqrt{p'q'})$ time and constant space (the choice gives 100 bits of security against this attack).

This talk: evidence that this choice of $\ell_{p'}, \ell_{q'}$ is overly optimistic.

Outline

Main result

Consider an RSA subgroup pair (N, g), with $N = p \cdot q = (2p'r + 1) \cdot (2q's + 1)$. An attacker wants to break the strong RSA subgroup assumption for (N, g) by factoring N.

While the best attack considered originally ran in $O(\sqrt{p'q'})$, we introduce a new attack in time and space $\tilde{O}(\sqrt{p'})$, based on a variant of the baby-step giant-step algorithm.

Thus, in principle, choosing $\ell_{p'} = 100$, as originally suggested, only provides about 50 bits of security against this attack.

We will now describe this new attack and discuss its practicality.

Baby-step giant-step

Recall how the baby-step, giant-step algorithm can reveal the hidden order n of a cyclic group $\mathbb{G}$ with generator g in time and space roughly linear in $\sqrt{n}$.

If n is of bit length ℓ, we can write:

$$n = a + \Delta \cdot b \quad \text{with } \Delta = 2^{\lceil \ell/2 \rceil} \text{ and } 0 \leq a, b < \Delta$$

Now, in time and space $O(\sqrt{n})$, we can compute:

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

A collision $x_i = y_j$ between those two lists (obtained by sorting, search trees, etc., in time quasi-linear in $\sqrt{n}$) gives a nontrivial pair (i, j) such that $g^{i+\Delta \cdot j} = 1$. We have $(i, j) = (a, b)$ and $n = a + \Delta \cdot b$ is recovered.

Applying BSGS to our setting (I)

To do something similar in our setting, we can write the RSA subgroup $\mathbb{G}$ as $\mathbb{G}_p \times \mathbb{G}_q$, where $\mathbb{G}_p$ is the mod-p group, of order p', and $\mathbb{G}_q$ is the mod-q group, of order q'.

In particular, g mod p is a generator of $\mathbb{G}_p$ and has multiplicative order p'. Now let $\ell = \ell_{p'}$ be the bit length of p', and write

$$p' = a + \Delta \cdot b \quad \text{with } \Delta = 2^{\lceil \ell/2 \rceil} \text{ and } 0 \leq a, b < \Delta$$

as before. We would like to recover p' in time and space linear in $\sqrt{p'}$ by applying the baby-step giant-step algorithm in $\mathbb{G}_p$. However, we cannot compute the two lists:

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

because p is unknown!

Applying BSGS to our setting (II)

Consider the following two lists instead:

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

Then we can test if x_i and y_j "collide mod p" by computing $\gcd(N, x_i - y_j)$.

If we compute all gcd values $\gcd(N, x_i - y_j)$, we will in particular evaluate $\gcd(N, x_a - y_b) = p$ and thus factor N.

But this is still not what we want: this requires computing $\Delta^2 = O(p')$ gcd values, which cannot be done in $\tilde{O}(\sqrt{p'})$ time.

Attaining $\tilde{O}(\sqrt{p'})$ time

We can make the gcd trick work as follows.

1. Instead of just computing the list of all values $x_i = g^i$, $0 < i < \Delta$, form the following polynomial:

$$f(x) = \prod_{0<i<\Delta} (x - x_i) \mod N$$

2. For $0 \leq j < \Delta$, evaluate the polynomial f at $y_j = (g^\Delta)^{-j}$, and compute $\gcd(N, f(y_j))$. For $j = b$, this reveals p as before.

This variant now runs in time quasi-linear in Δ (or equivalently $\sqrt{p'}$). Indeed, we can compute the coefficients of f with a product tree and evaluate it at all the y_j's with a remainder tree, both in time $O(M(\Delta)\log\Delta)$.

Improving complexity further

Since the x_i's and the y_j's are both in geometric progression, computing the polynomial:

$$f(x) = \prod_{0<i<\Delta} (x - x_i) \mod N$$

and evalutating it at all y_j's can be done faster than with generic product and remainder tree techniques, using the Newton basis interpolation and evaluation algorithms by Bostan and Schost (which simplify further in our setting).

Overall complexity:

Time: $3M(\Delta) + O(\Delta)$ arithmetic operations in $\mathbb{Z}_N$.

Space: $4\Delta + O(1)$ elements of $\mathbb{Z}_N$.

Outline

Implementation details

- Newton basis conversions following Bostan's thesis.
- Arbitrary precision arithmetic using MPIR/MPFR.
- Fast polynomial arithmetic over $\mathbb{Z}_N$ using the FLINT library. (Lack of a built-in middle-product leads to some efficency loss in time and space).
- Single-threaded implementation in C.
- Tested on a single core of an Intel Core2 Duo E8500 3.12GHz CPU, for a 1024-bit modulus N, and various sizes for p', q'.

Experimental results

$\ell = \lceil \log_2 p' \rceil$	**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

For the tested sizes, we get a very regular increase in running time, by a factor of about 2 for every two bits of ℓ: essentially linear in $\sqrt{p'}$ as expected.

Extrapolation to larger parameters

Direct extrapolation of the experimental running times gives the following estimates for running times with larger ℓ.

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

We would expect an attack on Groth's proposed parameters to take 9000 CPU years: a lot, but not absurdly so. The RSA-768 factoring effort took around 2000 CPU years.

However, there are serious hurdles to overcome before such large parameters can be attacked in practice.

Limitations

This attacks works well for small parameters but is difficult to carry out in practice for larger ones, due to two main limitations.

- Difficult to parallelize: due to the sequential nature of Bostan's algorithm, distributing the full size computation on several CPUs appears to be nontrivial.
- Large memory requirements: the $O(\sqrt{p'})$ space requirement is a major difficulty. Even $\ell = 60$ would require storage for 4 polynomials of degree 2^{30} over $\mathbb{Z}_N$, or about 500 GB of fast access memory.

  In our implementation, due to suboptimal memory management, we actually ran into memory problems as early as $\ell \approx 38$.

Conclusion

- Proposed a new attack on hidden order subgroups of $\mathbb{Z}_N^*$ in time and space $\tilde{O}(\sqrt{p'})$.
- Extrapolated running time suggests that the parameter choice of the original TCC 2005 paper is insecure.
- However, distributing computations and storage remains a stumbling block before this parameter choice can be attacked in practice.

- Open questions:
  - Better memory management?
  - Parallel implementation?
  - Ideally: constant-space variant à la Pollard lambda?

Thank you!