

Faster Algorithms for Approximate Common Divisors: Breaking Fully-Homomorphic-Encryption Challenges over the Integers

Yuanmi Chen* and Phong Q. Nguyen†

July 20, 2012

Abstract

At EUROCRYPT ’10, van Dijk, Gentry, Halevi and Vaikuntanathan presented simple fully-homomorphic encryption (FHE) schemes based on the hardness of approximate integer common divisors problems, which were introduced in 2001 by Howgrave-Graham. There are two versions for these problems: the partial version (PACD) and the general version (GACD). The seemingly easier problem PACD was recently used by Coron, Mandal, Naccache and Tibouchi at CRYPTO ’11 to build a more efficient variant of the FHE scheme by van Dijk *et al.*. We present a new PACD algorithm whose running time is essentially the “square root” of that of exhaustive search, which was the best attack in practice. This allows us to experimentally break the FHE challenges proposed by Coron *et al.* Our PACD algorithm directly gives rise to a new GACD algorithm, which is exponentially faster than exhaustive search: namely, the running time is essentially the $3/4$ -th root of that of exhaustive search. Interestingly, our main technique can also be applied to other settings, such as noisy factoring, fault attacks on CRT-RSA signatures, and attacking low-exponent RSA encryption.

1 Introduction

Following Gentry’s breakthrough work [10], there is currently great interest on fully-homomorphic encryption (FHE), which allows to compute arbitrary functions on encrypted data. Among the few FHE schemes known [10, 29, 8, 3, 12], the simplest one is arguably the one of van Dijk, Gentry, Halevi and Vaikuntanathan [29] (vDGHV), published at EUROCRYPT ’10. The security of the vDGHV scheme is based on the hardness of *approximate integer common divisors problems* introduced in 2001 by Howgrave-Graham [16]. In the general version of this problem (GACD), the goal is to recover a secret number p (typically a large prime number), given polynomially many near-multiples x_0, \dots, x_m of p , that is, each integer x_i is of the hidden form $x_i = pq_i + r_i$ where each q_i is a very large integer and each r_i is a very small integer. In the partial version of this problem (PACD), the setting is exactly the same, except that x_0 is chosen as an exact multiple of p , namely $x_0 = pq_0$ where q_0 is a very large integer chosen such that no non-trivial factor of x_0 can be found efficiently: for instance, [8] selects q_0 as a rough number, *i.e.* without any small prime factor.

By definition, PACD cannot be harder than GACD, and intuitively, it seems that it should be easier than GACD. However, van Dijk *et al.* [29] mention that there is currently no PACD algorithm that does not work for GACD. And the usefulness of PACD is demonstrated by the recent construction [8], where Coron, Mandal, Naccache and Tibouchi built a much more efficient variant of the FHE scheme by van Dijk *et al.* [29], whose security relies on PACD rather than GACD. Thus, it is very important to know if PACD is actually easier than GACD.

The hardness of PACD and GACD depends on how the q_i ’s and the r_i ’s are exactly generated. For the generation of [29] and [8], the noise r_i is extremely small, and the best attack known is simply gcd exhaustive search: for GACD, this means trying every noise (r_0, r_1) and check whether $\gcd(x_0 - r_0, x_1 - r_1)$ is sufficiently large and allows to recover the secret key; for PACD, this means trying every noise r_1 and check whether $\gcd(x_0, x_1 - r_1)$ is sufficiently large and allows to recover the secret key. In other words, if ρ is the bit-size of the noise r_i , then breaking GACD (resp. PACD) requires $2^{2\rho}$ (resp. 2^ρ) polynomial-time operations, for the parameters of [29, 8].

*ENS, Dépt. Informatique, 45 rue d’Ulm, 75005 Paris, France.

†INRIA, France & Tsinghua University, Institute for Advanced Study, 100084 Beijing, China. <http://www.di.ens.fr/~pnguyen/>.

OUR RESULTS. We present new algorithms to solve PACD and GACD, which are exponentially faster in theory and practice than the best algorithms considered in [29, 8]. More precisely, the running time of our new PACD algorithm is $2^{\rho/2}$ polynomial-time operations, which is essentially the “square root” of that of gcd exhaustive search. This directly leads to a new GACD algorithm running in $2^{3\rho/2}$ polynomial-time operations, which is essentially the 3/4-th root of that of gcd exhaustive search. Our PACD algorithm relies on classical algorithms to evaluate univariate polynomials at many points, whose space requirements are not negligible. We therefore present additional tricks, some of which reduce the space requirements, while still providing substantial speedups. This allows us to experimentally break the FHE challenges proposed by Coron *et al.* in [8], which were assumed to have comparable security to the FHE challenges proposed by Gentry and Halevi in [11]: the latter GH-FHE-challenges are based on hard problems with ideal lattices; according to Chen and Nguyen [4], their security level are respectively 52-bit (Toy), 61-bit (Small), 72-bit (Medium) and 100-bit (Large). Table 1 gives benchmarks for our attack on the FHE challenges, and deduces speedups compared to gcd exhaustive search. We can conclude that the FHE challenges of [8] have a much lower security level than those of Gentry and Halevi [13].

Table 1: Time required to break the FHE challenges by Coron *et al.* [8]. Size in bits, running time in seconds for a single 2.27GHz-core with 72Gb of RAM. Timings are extrapolated for RAM > 72 Gb.

Name	Toy	Small	Medium		Large
Size(public key)	0.95Mb	9.6Mb	89Mb		802Mb
Size(modulus)	1.6×10^5	0.86×10^6	4.2×10^6		19×10^6
Size(noise)	17	25	33		40
Expected security level	≥ 42	≥ 52	≥ 62		≥ 72
Running time of gcd-search	2420 40 mins ≈ 42	8.3×10^6 96 days ≈ 54	1.96×10^{10} 623 years ≈ 65		1.8×10^{13} 569193 years ≈ 75
Concrete security level					
Running time of the new attack implemented	99 1.6 min $d = 2^8$	25665 7.1 hours $d = 2^{12}$	1.635×10^7 190 days $d = 2^{13}$	6.6×10^6 76 days $d = 2^{15}$	6.79×10^{10} 2153 years $d = 2^{10}$
Parameters			≤ 72 Gb	≈ 240 Gb	≤ 72 Gb
Memory	≤ 130 Mb	≤ 15 Gb			2.9×10^8 9 years $d = 2^{19}$ ≈ 25 Tb
Speedup	24	324	1202	2997	264
New security level	≤ 37.7	≤ 45.7	≤ 55	≤ 54	62543 ≤ 59

Interestingly, we can also apply our technique to different settings, such as noisy factoring and attacking low-exponent RSA encryption. A typical example of noisy factoring is the following: assume that p is a divisor of a public modulus N , and that one is given a noisy version p' of p , which differs from p by at most k bits at unknown positions, can one recover p from (p', N) faster than exhaustive search? This may have applications in side-channel attacks. Like in the PACD setting, we obtain a square-root attack: for a 1024-bit modulus, the speedup can be as high as 1200 in practice. Similarly, we speed up several exhaustive search attacks on low-exponent RSA encryption.

RELATED WORK. Multipoint evaluation of univariate polynomials has been used in public-key cryptanalysis before. For instance, it is used in factoring (such as in the Pollard-Strassen factorization algorithm [23, 28] or in ECM speedup [19]), in the folklore square-root attack on RSA with small CRT exponents (mentioned by Boneh and Durfee [1], and described in [24, 20]), as well as in the recent square-root attack [7] by Coron, Joux, Mandal, Naccache and Tibouchi on Groth’s RSA Subgroup Assumption [14]. But this does not imply that our attack is trivial, especially since the authors of [8] form a subset of the authors of [7]. In fact, in most cryptanalytic applications (including [7]) of multipoint evaluation, one is actually interested in the following problem: given two lists $\{a_i\}_i$ and $\{b_j\}_j$ of numbers modulo N , find a pair (a_i, b_j) such that $\gcd(a_i - b_j, N)$ is non-trivial. Instead, we use multipoint evaluation differently, as a way to compute certain products of m elements modulo N in $\tilde{O}(\sqrt{m})$ polynomial-time operations, where $\tilde{O}()$ is the usual notation hiding poly-logarithmic terms. More precisely, it applies to products $\prod_{i=1}^m x_i \bmod N$ which can be rewritten under the form $\prod_{j=1}^{m_1} \prod_{k=1}^{m_2} (y_j + z_k) \bmod N$ where both m_1 and m_2 are $O(\sqrt{m})$. The Pollard-Strassen factorization algorithm [23, 28] can be viewed as a special case of this technique: it computes $m! \bmod N$ to factor N .

Very recently, Cohn and Heninger [5] announced a new attack on PACD and GACD, based on Coppersmith’s small root technique. This attack is interesting from a theoretical point of view, but from a practical point of view, we show

in App. A that for the FHE challenges of [8], it is expected to be slower than gcd exhaustive search, and therefore much slower than our attack.

ROADMAP. In Sect. 2, we describe our square-root algorithm for PACD, and apply it to GACD. In Sect. 3, we discuss implementation issues, present several tricks to speed up the PACD algorithm in practice, and we discuss the impact of our algorithm on the fully-homomorphic challenges of Coron *et al.* [8]. Finally, we apply our main technique to different settings: noisy factoring (Sect. 4) and attacking low-exponent RSA (Sect. 5).

ACKNOWLEDGEMENTS. We thank Jean-Sébastien Coron, Steven Galbraith, Éric Schost and Mehdi Tibouchi for helpful discussions. Part of this work is supported by the Commission of the European Communities through the ICT program under contract ICT-2007-216676 ECRYPT II.

2 A Square-Root Algorithm for Partial Approximate Common Divisors

In this section, we describe our new square-root algorithm for the PACD problem, which is based on evaluating univariate polynomials at many points. In the last subsection, we apply it to GACD.

2.1 Overview

Consider an instance of PACD: $x_0 = pq_0$ and $x_i = pq_i + r_i$ where $0 \leq r_i < 2^\rho$, $1 \leq i \leq m$. We start with the following basic observation due to Nguyen (as reported in [8, Sect 6.1]):

$$p = \gcd\left(x_0, \prod_{i=0}^{2^\rho-1} (x_1 - i) \pmod{x_0}\right) \quad (1)$$

At first sight, this observation only allows to replace 2^ρ gcd computations (with numbers of size $\approx \gamma$ bits) with essentially 2^ρ modular multiplications (where the modulus has $\approx \gamma$ bits): the benchmarks of [8] report a speedup of ≈ 5 for the FHE challenges, which is insufficient to impact security estimates.

However, we observe that (1) can be exploited in a much more powerful way as follows. We define the polynomial $f_j(x)$ of degree j , with coefficients modulo x_0 :

$$f_j(x) = \prod_{i=0}^{j-1} (x_1 - (x + i)) \pmod{x_0} \quad (2)$$

Letting $\rho' = \lfloor \rho/2 \rfloor$, we notice that:

$$\prod_{i=0}^{2^\rho-1} (x_1 - i) \equiv \prod_{k=0}^{2^{\rho'}+(\rho \bmod 2)-1} f_{2\rho'}(2^{\rho'} k) \pmod{x_0}.$$

We can thus rewrite (1) as:

$$p = \gcd\left(x_0, \prod_{k=0}^{2^{\rho'}+(\rho \bmod 2)-1} f_{2\rho'}(2^{\rho'} k) \pmod{x_0}\right) \quad (3)$$

Clearly, (3) allows to solve PACD using one gcd, $2^{\rho'}+(\rho \bmod 2) - 1$ modular multiplications, and the multi-evaluation of a polynomial (with coefficients modulo x_0) of degree $2^{\rho'}$ at $2^{\rho'}+(\rho \bmod 2)$ points, where $\rho' + (\rho \bmod 2) = \rho - \rho'$. We claim that this costs at most $\tilde{O}(2^{\rho'}) = \tilde{O}(\sqrt{2^\rho})$ operations modulo x_0 , which is essentially the square root of gcd exhaustive search. This is obvious for the single gcd and the modular multiplications. For the multi-evaluation part, it suffices to use classical algorithms (see [30, 17]) which evaluate a polynomial of degree d at d points, using at most $\tilde{O}(d)$ operations in the coefficient ring. Here, we also need to compute the polynomial $f_{2\rho'}(x)$ explicitly, which can fortunately also be done using $\tilde{O}(\sqrt{2^\rho})$ operations modulo x_0 . We give a detailed description of the algorithms in the next subsection.

2.2 Description

We first recall our algorithm to solve PACD, given as Alg. 1, and which was implicitly presented in the overview.

Algorithm 1 Solving PACD by multipoint evaluation of univariate polynomials

Input: An instance (x_0, x_1) of the PACD problem with noise size ρ .

Output: The secret number p such that $x_0 = pq_0$ and $x_1 = pq_1 + r_1$ with appropriate sizes.

1: Set $\rho' \leftarrow \lfloor \rho/2 \rfloor$.

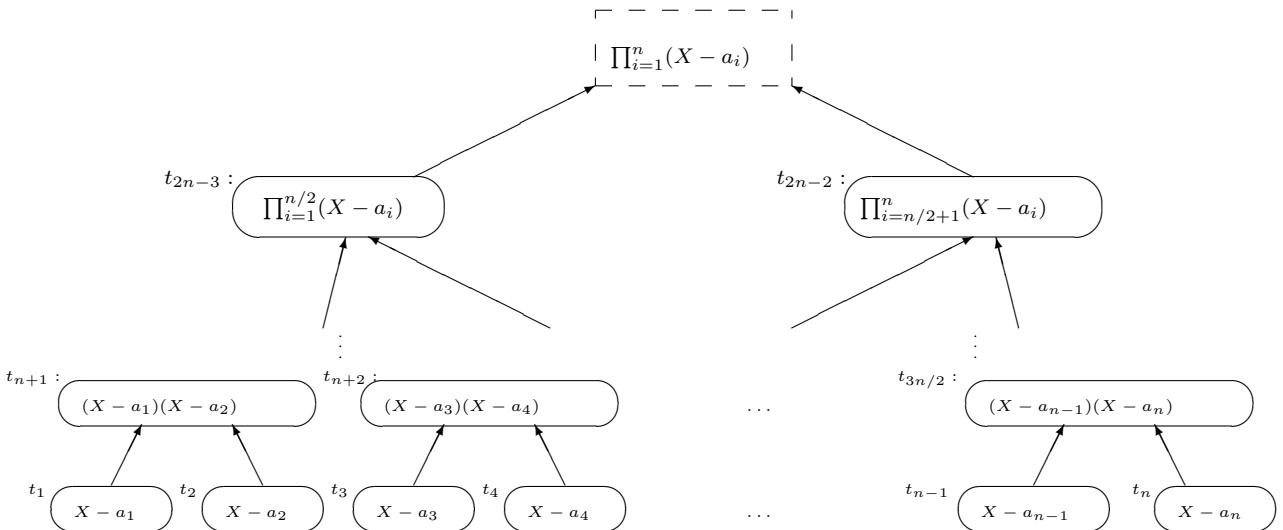
2: Compute the polynomial $f_{2\rho'}(x)$ defined by (2), using Alg. 2.

3: Compute the evaluation of $f_{2\rho'}(x)$ at the $2^{\rho'} + (\rho \bmod 2)$ points $0, 2^{\rho'}, \dots, 2^{\rho'}(2^{\rho'} + (\rho \bmod 2) - 1)$, using $2^{\rho \bmod 2}$ times Alg. 3 with $2^{\rho'}$ points. Each application of Alg. 3 requires the computation of a product tree, using Alg. 2.

Alg. 1 relies on two classical subroutines (see [30, 17]):

- a subroutine to (efficiently) compute a polynomial given as a product of n terms, where n is a power of two: Alg. 2 does this in $\tilde{O}(n)$ ring operations, provided that quasi-linear multiplication of polynomials is available, which can be achieved in our case using Fast Fourier techniques. This subroutine is used in Step 2. The efficiency of Alg. 2 comes from the fact that when the algorithm requires a multiplication, it only multiplies polynomials of similar degree.
- a subroutine to (efficiently) evaluate a univariate degree- n polynomial at n points, where n is a power of two: Alg. 3 does this in $\tilde{O}(n)$ ring operations, provided that quasi-linear polynomial remainder is available, which can be achieved in our case using Fast Fourier techniques. This subroutine is used in Step 3, and requires the computation of a tree product, which is achieved by Alg. 2. Alg. 3 is based on the well-known fact that the evaluation of a univariate polynomial at a point α is the same as its remainder modulo $X - \alpha$, which allows to factor computations using a tree.

Figure 1: Polynomial product tree $T = \{t_1, \dots, t_{2n}\}$ for $\{a_1, \dots, a_n\}$.



Algorithm 2 $[T, D] \leftarrow \text{TreeProduct}(A)$

Input: A set of $n = 2^l$ numbers $\{a_1, \dots, a_n\}$.

Output: The polynomial product tree $T = \{t_1, \dots, t_{2n-1}\}$, corresponding to the evaluation of points $A = \{a_1, \dots, a_n\}$ as shown in Figure 1.

$D = [d_1, \dots, d_{2n-1}]$ descendant indices for non-leaf nodes or 0 for leaf node.

```

1: for  $i = 1 \dots n$  do
2:    $t_i \leftarrow X - a_i$  {Initializing leaf nodes}
3:    $d_j \leftarrow 0$ 
4: end for
5:  $i \leftarrow 1$  {Index of lower level}
6:  $j \leftarrow n + 1$  {Index of upper level}
7: while  $j \leq 2n - 1$  do
8:    $t_j \leftarrow t_i \cdot t_{i+1}$ 
9:    $d_j \leftarrow i$ 
10:   $i \leftarrow i + 2$ 
11:   $j \leftarrow j + 1$ 
12: end while
```

Algorithm 3 $V \leftarrow \text{RecursiveEvaluation}(f, t_i, D)$

Input: A polynomial f of degree n .

A polynomial product tree rooted at t_i , and whose leaves are $\{X - a_k, \dots, X - a_m\}$

An array $D = [d_1, \dots, d_{2n-1}]$ descendant indices for non-leaf nodes or 0 for leaf node.

Output: $V = \{f(a_k), \dots, f(a_m)\}$

```

1: if  $d_i = 0$  then
2:   return  $\{f(a_i)\}$  {When  $t_i$  is a leaf, we apply an evaluation directly.}
3: else
4:    $g_1 \leftarrow f \bmod t_{d_i}$  {left subtree}
5:    $V_1 \leftarrow \text{RecursiveEvaluation}(g_1, t_{d_i}, D)$ 
6:    $g_2 \leftarrow f \bmod t_{d_i+1}$  {right subtree}
7:    $V_2 \leftarrow \text{RecursiveEvaluation}(g_2, t_{d_i+1}, D)$ 
8:   return  $V_1 \cup V_2$ 
9: end if
```

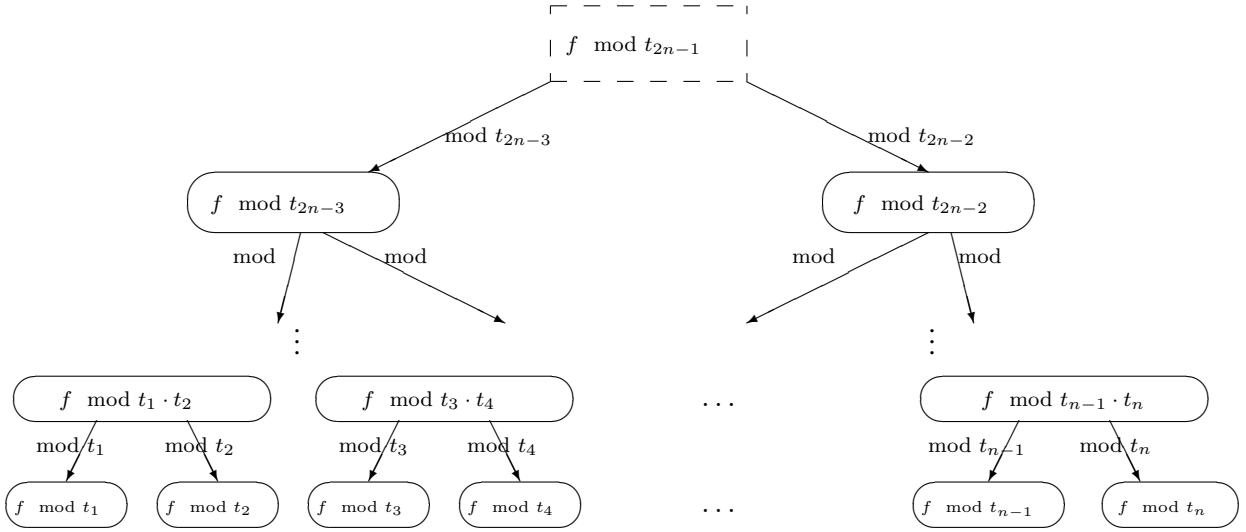
It follows that the running time of Alg. 1 is $\tilde{O}(2^{\rho'}) = \tilde{O}(\sqrt{2^\rho})$ operations modulo x_0 , which is essentially the ‘‘square root’’ of gcd exhaustive search. But the space requirement is $\tilde{O}(2^{\rho'}) = \tilde{O}(\sqrt{2^\rho})$ polynomially many bits: thus, Alg. 1 can be viewed as a time/memory trade-off, compared to gcd exhaustive search.

2.3 Logarithmic speedup

In the previous analysis, the time complexity $\tilde{O}(n)$ actually stands for $O(n \log^2(n))$ ring multiplications. Interestingly, Bostan, Gaudry and Schost showed in [2] that when the structure of the factors are very regular, there is an algorithm which speeds up the theoretical complexity by a logarithmic term $\log(n)$. This BGS algorithm is tailored for the case where we want to estimate a function f on a set of points with what we call a hypercubic structure. An important subprocedure is `ShiftPoly` which, given as input a polynomial f of degree at most 2^d , and the evaluations of f on a set of 2^d points with hypercubic structure, outputs the evaluation of f on a shifted set of 2^d points, using $O(2^d)$ ring operations. More precisely:

Theorem 2.1 (see Th. 5 of [2]) Let α, β be in ring \mathbb{P} and d be in \mathbb{N} such that $\mathbf{d}(\alpha, \beta, d)$ is invertible, with $\mathbf{d}(\alpha, \beta, d) = \beta \cdot 2 \dots d \cdot (\alpha - d\beta) \dots (\alpha + d\beta)$. And suppose also that the inverse of $\mathbf{d}(\alpha, \beta, d)$ is known. Let $F(\cdot) \in \mathbb{P}[X]$ of degree at

Figure 2: Evaluation on the polynomial tree $T = \{t_1, \dots, t_{2n-1}\}$ for $\{a_1, \dots, a_n\}$.



most d and $x_0 \in \mathbb{P}$. There exists an algorithm ShiftPoly which, given as input $F(x_0), F(x_0 + \beta), \dots, F(x_0 + d\beta)$, outputs $F(x_0 + \alpha), F(x_0 + \alpha + \beta), \dots, F(x_0 + \alpha + d\beta)$ in time $2M(d) + O(d)$ time and space $O(d)$. Here, $M(d)$ is the time of multiplying two polynomial of degree at most d .

We note $E(k_1, \dots, k_j)$ for $\left\{ \sum_{i=1}^j p_{k_i} 2^{k_i} \right\}$ with each p_{k_i} ranging over $\{0, 1\}$. This is the set enumerating all possibilities of bits $\{k_1, \dots, k_j\}$. Given a set A and an element and p , $A + p$ is defined as $\{a + p \mid \forall a \in A\}$. Then we have

$$E(k_1, \dots, k_{j+1}) = E(k_1, \dots, k_j) \cup \left(E(k_1, \dots, k_j) + 2^{k_{j+1}} \right).$$

This is what we call a set with hypercubic structure.

Given a linear polynomial $f(x)$ and a set with hypercubic structure of 2^ρ points, the proposed algorithm iteratively calls Alg.4 which uses ShiftPoly , and calculates the evaluation of $F_i(X) = \prod_{Y \in E(k_1, \dots, k_i)} f(X + Y)$ on $E(b_{k-i}, \dots, k_\rho)$ until $i = \lfloor n/2 \rfloor$. The i -th iteration costs $O(2^i)$ ring operations, thus the total complexity amounts to $O(2^{\rho/2})$ ring operations.

Algorithm 4 i -th iteration of the evaluation of $F_i(X)$

Input: For $i = 1, \dots, \lfloor \rho/2 \rfloor$, the evaluation of $F_i(X)$ on points $X \in E(k_{\rho-i+1}, \dots, k_\rho)$

Output: the evaluation of $F_{i+1}(X)$ on points $X \in E(k_{\rho-i}, \dots, k_\rho)$

- 1: $F_i(X)$ for $X \in E(k_{\rho-i+1}, \dots, k_\rho) + 2^{k_{\rho-i}} \leftarrow \text{ShiftPoly}(F_i(X), X \in E(k_{\rho-i+1}, \dots, k_\rho))$
 - 2: $F_i(X)$ for $X \in E(k_{\rho-i}, \dots, k_\rho) + 2^{k_{i+1}} \leftarrow \text{ShiftPoly}(F_i(X), X \in E(k_{\rho-i}, \dots, k_\rho))$
 - 3: $F_{i+1}(X) = F_i(X) \cdot F_i(X + 2^{k_{i+1}})$, for all $X \in E(k_{\rho-i}, \dots, k_\rho)$
-

2.4 Application to GACD

Any PACD algorithm can be used to solve GACD, using the trivial reduction from GACD to PACD based on exhaustive search over the noise r_0 . More precisely, for an arbitrary instance of GACD:

$$x_i = pq_i + r_i \text{ where } 0 \leq r_i < 2^\rho, 0 \leq i \leq m$$

we apply our PACD algorithm for all pairs $(x_0 - r_0, x_1)$ where r_0 ranges over $\{0, \dots, 2^\rho - 1\}$.

It follows that GACD can be solved in $\tilde{O}(2^{3\rho/2})$ operations modulo x_0 , using $\tilde{O}(2^{\rho/2})$ polynomially many bits. This is exponentially faster than the best attack of [29], namely gcd exhaustive search, which required $2^{2\rho}$ gcd operations. Note that in [29], another hybrid attack was described, where one performs exhaustive search over r_0 and factor the resulting number using ECM, but because of the large size of the prime factors (namely, a bit-length $\geq \rho^2$), this attack is not faster: it also requires at least $2^{2\rho}$ operations.

Following our work, it is noted with [9] that one can heuristically beat the GACD bound $\tilde{O}(2^{3\rho/2})$ using more samples of x_i , by removing the "smooth part" of $\gcd(y_1, \dots, y_s)$ where $y_i = \prod_{j=0}^{2^\rho-1} (x_i - j)$ and s is large enough. The choice of s actually gives different time/memory trade-offs. For instances, if $s = \Theta(\rho)$, the running time is heuristically $\tilde{O}(2^\rho)$ poly-time operations and similar memory. From a practical point of view however, our attack is arguably more useful, due to memory requirements and better $\tilde{O}()$ constants.

3 Implementation of the Square-Root PACD Algorithm

We implemented both Alg. 1 and the logarithmic speedup using the NTL library [26]. In this section, we describe various tricks that we used to implement efficiently Alg 1. The implementation was not straightforward due to the size of the FHE challenges.

3.1 Obstructions

The main obstruction when implementing Alg. 1 is memory. Consider the Large FHE-challenge from [8]: there, $\rho = 40$, so the optimal parameter is $\rho' = 20$, which implies that $f_{2^{\rho'}}$ is a polynomial of degree 2^{20} with coefficients of size 19×10^6 bits. In other words, simply storing $f_{2^{\rho'}}$ already requires $2^{20} \times 19 \times 10^6$ bits, which is more than 2Tb, while we also need to perform various computations. This means that in practice, we will have to settle for suboptimal parameters.

More precisely, assume that we select an additional parameter d , which is a power of two less than $2^{\rho'}$. We rewrite (3) as:

$$p = \gcd\left(x_0, \prod_{k=0}^{2^\rho/d-1} f_d(dk) \pmod{x_0}\right) \quad (4)$$

This gives rise to a constrained version of Alg. 1, called Alg. 5.

Algorithm 5 Solving PACD by multipoint evaluation of univariate polynomials, using fixed memory

Input: An instance (x_0, x_1) of the PACD problem with noise size ρ , and a polynomial degree d (which must be a power of two).

Output: The secret number p such that $x_0 = pq_0$ and $x_1 = pq_1 + r_1$ with appropriate sizes.

- 1: Compute the polynomial $f_d(x)$ defined by (2), using Alg. 2.
 - 2: Compute the evaluation of $f_d(x)$ at the $2^\rho/d$ points $0, d, 2d, \dots, d(2^\rho/d - 1)$, using $2^\rho/d^2$ times Alg. 3 with d points.
Each application of Alg. 3 requires the computation of a product tree, using Alg. 2.
-

The running time of Alg. 5 is $\frac{2^\rho \tilde{O}(d)}{d^2}$ elementary operations modulo x_0 , and the space requirement is $\tilde{O}(d)$ polynomially many bits. Note that each of the $2^\rho/d^2$ times applications of Alg. 3 can be done in parallel.

3.2 Tricks

The use of Alg. 5 allows several tricks, which we now present.

3.2.1 Minimizing the Product Tree

Each application of Alg. 3 requires the computation of a product tree, using Alg. 2. But this product tree requires to store $2n - 1$ polynomials. Fortunately, these polynomials have coefficients which are in some sense much smaller than the

modulus x_0 : this is because we evaluate the polynomial $f_d(x)$ at points in $\{0, \dots, 2^\rho - 1\}$, which is very small compared to the modulus x_0 . However, a naive implementation would not exploit this. For instance, consider the polynomial $(X - a_1)(X - a_2) = X^2 - (a_1 + a_2)X + a_1a_2$, which belongs to the product tree. In a typical library for polynomial computations, the polynomial coefficients would be represented as positive residues modulo x_0 . But if $a_1 + a_2$ is small, then $-(a_1 + a_2) + x_0$ is actually big. This means that many coefficients of the product tree polynomials will actually be as big as x_0 , if they are represented as positive residues modulo x_0 , which drastically reduces the choice of the degree d .

To avoid this problem, we instead slightly modify the polynomial $f_d(X)$, in order to evaluate at small negative numbers inside $\{0, \dots, 1 - 2^\rho\}$, so that each polynomial of the product tree has “small” positive coefficients. This drastically reduces the storage of the product tree. More precisely, we rewrite (4) as:

$$p = \gcd \left(x_0, \prod_{k=0}^{2^\rho/d^2-1} \prod_{\ell=0}^{d-1} f'_{d,k}(-\ell d) \pmod{x_0} \right) \quad (5)$$

where

$$f'_{d,k}(x) = \prod_{i=0}^{d-1} (x_1 - 2^\rho - x + dk - i) \pmod{x_0} \quad (6)$$

Each product $\prod_{\ell=0}^{d-1} f'_{d,k}(-\ell) \pmod{x_0}$ is computed by applying Alg. 3 once, using the d points $0, -d, -2d, \dots, -d(d-1)$.

3.2.2 Powers of Two

We need to compute the polynomial $f'_{d,k}(x)$ defined by (6) before each application of Alg. 3, using a simplified version of Alg. 2, which only computes the root rather than the whole product tree. However, notice that the degree of each polynomial of the product tree is exactly a power of two, which is the worst case for the polynomial multiplication implemented in the NTL library [26]. For instance, in NTL, multiplying two 512-degree polynomials with Medium-FHE coefficients takes 50% more time than multiplying two 511-degree polynomials with Medium-FHE coefficients.

To circumvent threshold phenomena, we notice that each polynomial of the product tree is a monic polynomial, except the leaves (for which the leading coefficient is -1). But the product of two monic polynomials whose degree is a power of two can be derived efficiently from the product of two polynomials with degree strictly less than the power of two, using:

$$(X^n + P(X)) \times (X^n + Q(X)) = X^{2n} + X^n(P(X) + Q(X)) + P(X)Q(X).$$

We apply this trick to speed up the computation of the polynomial $f'_{d,k}(x)$.

3.2.3 Precomputations

Now that we use (5), we change several times the polynomial $f'_{d,k}(x)$, but we keep the same evaluation points $0, -d, -2d, \dots, -d(d-1)$, and therefore the same product tree. This allows to perform precomputations to speed up Alg. 3. Indeed, the main operation of Alg. 3 is computing the remainder of a polynomial with one of the product tree polynomials, and it is well-known that this can be sped up using precomputations depending on the modulus polynomial. One classical way to do this is to use Newton’s method for remainder (Alg. 6). This algorithm requires the following notation: for any polynomial f of degree n and for any integer $m \geq n$, we define the m -degree polynomial $\text{rev}(f, m)$ as $\text{rev}(f, m) = f(1/X) \cdot X^m$. In Alg. 6, Line 1 is independent of f . Therefore, whenever one needs to compute many remainders with respect to the same modulus g , it is more efficient to precompute and store h , so that Line 1 does not need to be reexecuted. Hence, in an offline phase, we precompute and store (on a hard disk) the polynomial \bar{g} of Line 1 for each product tree polynomial. And for each remainder required by Alg. 3, we execute the last two lines of Alg. 6.

It follows that each remainder operation of Alg. 3 is reduced to two polynomial multiplications.

The NTL library also contains routines for doing remainders with precomputations, but Alg. 6 turns out to be more efficient for our setting. This is because many factors impact the performance of polynomial arithmetic, such as the size of the modulus and the degree.

Algorithm 6 Remainder using Newton’s method (see [17, Sect 7.2])

Input: Polynomials $f \in \mathbb{R}[X]$ of degree $2n - 1$, $g \in \mathbb{R}[X]$ of degree n .
Output: The polynomial $h = f \bmod g$

- 1: $\bar{g} \leftarrow \text{Inverse}(\text{rev}(g, n)) \bmod X^n$
- 2: $s \leftarrow \text{rev}(f, 2n - 1) \cdot \bar{g} \bmod X^n$
- 3: $h \leftarrow f - \text{rev}(s, n - 1) \cdot g$

3.3 Logarithmic Speedup and Further Tricks

We also implemented the BGS algorithm described in Sect. 2.3, which offers an asymptotical logarithmic speedup, but our implementation was not optimized due to lack of time: a good implementation would require the so-called middle product [2], which we instantiated by a normal product. On the FHE challenges, our implementation turned out to be twice as slow as Alg. 1 for Medium and Large, and marginally slower (resp. faster) for Toy (resp. Small).

Since memory is the main obstruction for choosing d , it is very important to minimize RAM requirements. Since Alg. 3 can be reduced to multiplications using precomputations, one may consider the use of special multiplication algorithms which require less memory than standard algorithms, such as in-place algorithms. We note that there has been recent work [25, 15] in this direction, but we did not implement these algorithms. This suggests that our implementation is unlikely to be optimal, and that there is room for improvement.

3.4 New Security Estimates for the FHE Challenges

Table 1 reports benchmarks for our implementation on the fully-homomorphic-encryption challenges of Coron *et al.* [8], which come in four flavours: Toy, Small, Medium and Large. The security level ℓ is defined in [8] as follows: the best attack should require at least 2^ℓ clock cycles on a standard single core. The row “Expected security level” is extracted from [8].

Our timings refer to a single 2.27GHz-core with 72Gb of RAM. First, we assessed the cost of gcd exhaustive search, by measuring the running time of the (quasi-linear) gcd routine of the widespread gmp library, which is used in NTL [26]: timings were measured for each modulus size of the four FHE-challenges. This gives the “concrete security level” row, which is slightly higher than the expected security level of [8].

We also report timings for our implementation of our square-root PACD algorithm: these timings are below the expected security level, which breaks all four FHE-challenges of [8]. For the Toy and Small challenges, the parameter d was optimal, and we did not require much memory: the speedup is respectively 24 and 324, compared to gcd exhaustive search. For the Medium and Large challenges, we had to use a suboptimal parameter d , due to RAM constraints: we used $d = 2^{13}$ (resp. $d = 2^{10}$) for Medium (resp. Large), instead of the optimal $d = 2^{16}$ (resp. $d = 2^{20}$). But the speedups are already significant: 1202 for Medium, and 264 for Large. The timings are obtained by suitably multiplying the running time of a single execution of Alg. 3 and Alg. 2: for instance, in the Large case, this online phase took between 64727s to 65139.4s, for 5 executions, and the precomputation storage was 21Gb.

Table 1 also provides extrapolated figures if the RAM was ≥ 72 Gb, which allows larger values of d : today, one can already buy servers with 4-Tb RAM. For the Large challenge, the potential speedup is over 60,000. Using a more optimized implementation, we believe it is possible to obtain larger speedups, so the New security level row should only be interpreted as an upper bound. But our implementation is already sufficient to show that the FHE-challenges of [8] fall short of the expected security level.

Hence, one needs to increase the parameters of the FHE scheme of [8], which makes it less competitive with the FHE implementation of [13]. It can be noted that the new security levels of the challenges of [8] are much lower than those given by [4] on the challenges of Gentry and Halevi [13], namely 52-bit (Toy), 61-bit (Small), 72-bit (Medium) and 100-bit (Large).

4 Applications to Noisy Factoring

Consider a typical “balanced” RSA modulus $N = pq$ where $p, q \leq 2\sqrt{N}$. A celebrated lattice-based cryptanalysis result of Coppersmith [6] states that if one is given half of the bits of p , either in the most significant positions, or the least significant positions, then one can recover p and q in polynomial time. Although this attack has been extended in several works (see [18] for a survey), all these lattice-based results require that the unknown bits are consecutive, or spread across extremely few blocks. This decreases its potential applications to side-channel attacks where errors are likely to be spread unevenly.

This suggests the following setting, which we call noisy factoring. Assume that one is given a noisy version p' of the prime factor p , which differs from p by at most k bits, not necessarily consecutive, under either of the following two cases:

- If the k positions of the noisy bits are known, we can recover p (and therefore q) by exhaustive search using at most 2^k polynomial-time operations: we stress that in this case, we assume that we do not know if each of the k bits has been flipped, otherwise no search would be necessary.
- If instead, none of the positions is known, but we know that exactly k bits have been modified, we can recover p by exhaustive search using at most $\binom{n}{k}$ polynomial-time operations, where n is the bit-length of p . If we only know an upper bound on the number of modified bits, we can simply repeat the attack with decreasing values of k .

These running times do not require that p and q are balanced.

In this section, we show that our previous technique for PACD can be adapted to noisy factoring, yielding new attacks whose running time is essentially the “square root” of exhaustive search, that is, $\tilde{O}(2^{k/2})$ or $\tilde{O}(\sqrt{\binom{n}{k}})$ polynomial-time operations, depending on the case. Finally, we also extend our method to fault attacks of RSA signature scheme.

4.1 Known positions

We assume that the prime number p has n bits, so that: $p = \sum_{i=0}^{n-1} p_i 2^i$, where $p_i \in \{0, 1\}$ for $0 \leq i \leq n - 1$.

In this subsection, we assume that all the bits p_i are known, except possibly at k positions b_1, \dots, b_k , which we sort, so that: $0 \leq b_1 \leq \dots \leq b_k < n$. Denote by $p^{(1)}, \dots, p^{(2^k)}$ the 2^k possibilities for p , when (pb_1, \dots, pb_k) ranges over $\{0, 1\}^k$. With high probability, all the $p^{(i)}$ ’s are coprime with N , except one, which would imply that:

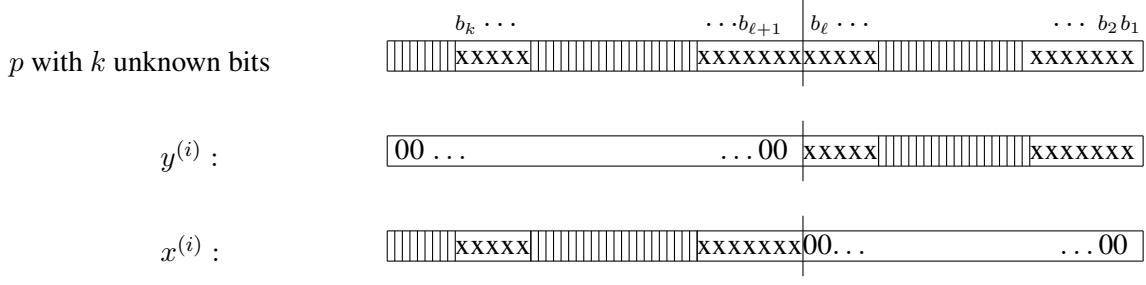
$$p = \gcd \left(N, \prod_{i=1}^{2^k} p^{(i)} \pmod{N} \right) \quad (7)$$

A naive evaluation of (7) costs 2^k modular multiplications, and one single gcd. We now show that this evaluation can be performed more efficiently using $\tilde{O}(2^{k/2})$ arithmetic operations with numbers with the same size as N .

The unknown bits pb_1, \dots, pb_k can be regrouped into two sets $\{pb_1, \dots, pb_\ell\}$, and $\{pb_{\ell+1}, \dots, pb_k\}$ of roughly the same size $\ell = \lfloor k/2 \rfloor$, as illustrated in Figure 3:

- For $1 \leq i \leq 2^\ell$, let $y^{(i)} = \sum_{j=0}^{n-1} y_j^{(i)} 2^j$, where $y_j^{(i)} = \begin{cases} 0 & \text{if } j > b_\ell \\ t\text{-th bit of } i & \text{if } \exists t \leq \ell, j = b_t \\ p_j & \text{otherwise} \end{cases}$
- For $1 \leq i \leq 2^{k-\ell}$, let $x^{(i)} = \sum_{j=0}^{n-1} x_j^{(i)} 2^j$, where $x_j^{(i)} = \begin{cases} 0 & \text{if } j \leq b_\ell \\ t\text{-th bit of } i & \text{if } \exists t > \ell, j = b_t \\ p_j & \text{otherwise} \end{cases}$

Figure 3: Splitting the Unknown Bits in Two



Hence, by definition of $x^{(i)}$ and $y^{(i)}$, we have:

$$\prod_{i=1}^{2^k} p^{(i)} \equiv \prod_{i=1}^{2^\ell} \prod_{j=1}^{2^{k-\ell}} (x^{(j)} + y^{(i)}) \pmod{N} \quad (8)$$

which gives rise to a square-root algorithm (Alg. 7) to solve the noisy factorization problem with known positions.

Algorithm 7 Noisy Factorization With Known Positions

Input: An RSA modulus $N = pq$ and the bits p_0, \dots, p_{n-1} of p , except the k bits p_{b_1}, \dots, p_{b_k} , where the bit positions $b_1 \leq b_2 \leq \dots \leq b_k$ are known.

Output: The secret factor $p = \sum_{i=0}^{n-1} p_i 2^i$ of N .

- 1: Compute the polynomial $f(X) = \prod_{i=1}^{2^\ell} (X + y^{(i)}) \pmod{N}$ of degree 2^ℓ , with coefficients modulo N , using Alg. 2.
 - 2: Compute the evaluation of $f(X)$ at the points $\{x^{(1)}, \dots, x^{(2^{k-\ell})}\}$, using $1 + (k \bmod 2)$ times Alg. 3 with 2^ℓ points.
 - 3: return $p \leftarrow \gcd(N, \prod_{i=1}^{2^{k-\ell}} (f(x^{(i)})) \pmod{N})$
-

Similary to Section 2, the cost of Alg. 7 is $\tilde{O}(2^{k/2})$ polynomial-time operations. This is an exponential improvement over naive exhaustive search, but Alg. 7 requires exponential space. In practice, the improvement is substantial. Using our previous implementation, Alg. 7 gives a speedup of about 1200 over exhaustive division to factor a 1024-bit modulus, given a 512-bit noisy factor with 46 unknown bits at known positions.

Furthermore, in this setting, the points to be enumerated happen to satisfy the hypercubic property, thus we may apply the logarithmic speedup described in Sect. 2.3.

Remember that the factor p can be calculated with formula (8). Now we can restate it as

$$\prod_{i=1}^{2^k} p^{(i)} \equiv \prod_{y \in E(b_{\ell+1}, \dots, b_k)} \prod_{x \in E(b_1, \dots, b_\ell)} (x + y + M_p) \pmod{N}, \quad (9)$$

here $M_p = \sum_{i \notin \{b_1, \dots, b_k\}} p_i 2^i$ is the known bits of p . We define $F_i(X) = \prod_{y \in E(b_1, \dots, b_i)} (X + y + M_p)$.

As discussed in Section 2, the cost of Alg. 8 is faster than Alg. 7 by a factor of $O(k)$ on a theoretical basis.

4.2 Unknown positions

In this subsection, we assume that p' differs from p by exactly k bits at unknown positions, and that p' has bit-length n . Our attack is somewhat reminiscent of Coppersmith's baby-step/giant-step attack on low-Hamming-weight discrete logarithm [27], but that attack uses sorting, not multipoint evaluation. To simplify the description, we assume that both k and n are even, but the attack can easily be adapted to the general case.

Algorithm 8 Improved Noisy Factorization With Known Positions

Input: An RSA modulus $N = pq$ and a number p' differing from p by exactly k bits of unknown position.

Output: The secret factor p .

- 1: $F_0(0) \leftarrow M_p$
 - 2: **for** $i = 1, \dots, \lfloor k/2 \rfloor$ **do**
 - 3: Call Alg. 4 to calculate the evaluation of $F_i(X)$ on $E(b_{k-i}, b_k)$ given the evaluation of $F_{i-1}(X)$ on $E(b_{k-i+1}, b_k)$
 - 4: **end for**
 - 5: **if** k is odd **then**
 - 6: The evaluation $F_{\lfloor k/2 \rfloor}(X)$ for $X \in E(b_{\lfloor k/2 \rfloor+2}, \dots, b_k) + 2^{b_{\lfloor k/2 \rfloor+1}}$
 $\leftarrow \text{ShiftPoly}(F_{\lfloor k/2 \rfloor}(X), X \in E(b_{\lfloor k/2 \rfloor+2}, \dots, b_k), 2^{b_{\lfloor k/2 \rfloor+1}})$
 - 7: **end if**
 - 8: $p'' = \gcd(N, \prod_{X \in E(b_{\lfloor k/2 \rfloor+1}, \dots, b_k)} (F_{\lfloor k/2 \rfloor}(X)))$
 - 9: **return** p''
-

Pick a random subset S of $\{0, \dots, n-1\}$ containing exactly $n/2$ elements. The probability that S contains the indices of exactly $k/2$ flipped bits is: $\binom{n/2}{k/2}^2 / \binom{n}{k} \approx \frac{1}{\sqrt{k}}$. We now assume that this event holds, and let $\ell = \binom{n/2}{k/2}$. Similarly to the previous subsection, we define:

- Let $x^{(i)}$ for $1 \leq i \leq \ell$ be the numbers obtained by copying the bits of p' at all the positions inside S , and flipping exactly $k/2$ bits: all the other bits are set to zero.
- Let $y^{(i)}$ for $1 \leq i \leq \ell$ be the numbers obtained by copying the bits of p' at all the positions outside S , and flipping exactly $k/2$ bits: all the other bits are set to zero.

Now, with high probability over the choice of (p, q) , we may write:

$$p = \gcd(N, \prod_{i=1}^{\ell} \prod_{j=1}^{\ell} (x^{(j)} + y^{(i)}) \pmod{N}) \quad (10)$$

which gives rise to a square-root algorithm (Alg. 9) to solve the noisy factorization problem with unknown positions.

Algorithm 9 Noisy Factorization With Unknown Positions

Input: An RSA modulus $N = pq$ and a number p' differing from p by exactly k bits of unknown position.

Output: The secret factor p .

- 1: **repeat**
 - 2: Pick a random subset S of $\{0, \dots, n-1\}$ containing exactly $n/2$ elements.
 - 3: Compute the integers $x^{(i)}$ and $y^{(i)}$ for $1 \leq i \leq \ell = \binom{n/2}{k/2}$.
 - 4: Compute the polynomial $f(X) = \prod_{j=1}^{\ell} (X + y^{(j)}) \pmod{N}$.
 - 5: Compute the evaluation of $f(X)$ at the ℓ points $\{x^{(1)}, \dots, x^{(\ell)}\}$.
 - 6: $p'' \leftarrow \gcd(N, \prod_{i=1}^{\ell} (f(x^{(i)})) \pmod{N})$
 - 7: **until** $p'' > 1$
 - 8: **return** p''
-

Similary to Section 2, the expected cost of Alg. 9 is $\tilde{O}(\ell \sqrt{k})$ polynomial-time operations, where $\ell = \binom{n/2}{k/2}$ is roughly $\sqrt{\binom{n}{k}}$. This is an exponential improvement over naive exhaustive search, but Alg. 9 requires exponential space.

Alg. 9 is randomized, but like Coppersmith’s baby-step/giant-step attack on low-Hamming-weight discrete logarithm [27], it can easily be derandomized using splitting systems. Deterministic versions are slightly less efficient, by a small polynomial factor: see [27].

4.3 Application to fault attacks on CRT-RSA signatures

Consider an RSA signature $s = m^d \bmod N$, where $N = pq$. In practice, this calculation is often accelerated using the Chinese remainder theorem. More precisely, s is derived from s_p and s_q , where $s_p = m^d \bmod p$ and $s_q = m^d \bmod q$. It is well-known that if a fault occurs during the computation of s_p (but not s_q), then the output s' will satisfy $s' \not\equiv m^d \bmod p$ and $s' \equiv m^d \bmod q$. Then the factorization of N is disclosed by

$$p = \gcd(s'^e - m \bmod N, N)$$

However, the message m may have already gone through some padding, so that only part of its bits are known. The positions of the unknown bits might be known, or not, which corresponds to the two situations presented in the previous discussion. The same technique to speed up the enumeration can be applied, by substituting (7) with

$$p = \gcd \left(N, \prod_{i=1}^{2^k} (s'^e - m^{(i)}) \pmod{N} \right). \quad (11)$$

5 Applications to Low-Exponent RSA

In this section, we show that our previous algorithms for noisy factoring can be adapted to attacks on both low-exponent RSA encryption. Consider an RSA ciphertext $c = m^e \bmod N$, where the public exponent e is very small. Assume that one knows a noisy version m' of the plaintext m , which differs from m by at most k bits, not necessarily consecutive, under either of the following two cases:

- If the k positions of the noisy bits are known, we can recover m by exhaustive search using at most 2^k polynomial-time operations: we stress that in this case, we assume that we do not know if each of the k bits has been flipped, otherwise no search would be necessary.
- If instead, none of the positions is known, but we know that exactly k bits have been modified, we can recover m by exhaustive search using at most $\binom{n}{k}$ polynomial-time operations, where n is the bit-length of m . If we only know an upper bound on the number of modified bits, we can simply repeat the attack with decreasing values of k .

This setting is usually called stereotyped RSA encryption [6]: there are well-known lattice attacks [6, 18] against stereotyped RSA, but they require that the unknown bits are consecutive, or split across extremely few blocks.

5.1 Known Positions

Assume that m is a plaintext of n bits, among which only k bits are unknown, whose (arbitrary) positions are b_1, \dots, b_k . Let $c = m^e \bmod N$ be the raw RSA ciphertext of m . If e is small (say, constant), we can “square root” the time of exhaustive search, using multipoint polynomial evaluation.

Let $\ell = \lceil (k - \log_2 e)/2 \rceil$, and assume that $k > 0$.

$$m : \quad \begin{array}{cccc|c|ccccc} b_k \dots & & & \dots b_{\ell+1} & | & b_\ell \dots & & & \dots b_2 b_1 \\ \hline \text{xxxxx} & \text{xxxxx} & \text{xxxxx} & \text{xxxxx} & | & \text{xxxxx} & \text{xxxxx} & \text{xxxxx} & | & \text{xxxxx} \end{array}$$

Let m_0 be derived from m by keeping all the known $n - k$ bits, and setting all the k unknown bits to 0.

$$m_0 : \quad \boxed{00000} \quad \boxed{0000000000000000} \quad \boxed{0000000}$$

For $1 \leq i \leq 2^{k-\ell}$, let the x_i 's enumerate all the integers when $(b_{\ell+1}, \dots, b_k)$ ranges over $\{0, 1\}^{k-\ell}$.

$$x_i : \quad \boxed{0000xxxx} \quad \boxed{000000000000xxxxxx} \quad \boxed{000000000000000000000000}$$

Similarly, for $1 \leq j \leq 2^\ell$, let the y_j 's enumerate all the integers when (b_1, \dots, b_ℓ) ranges over $\{0, 1\}^\ell$.

$$y_i : \quad \boxed{000000000000000000000000} \quad \boxed{xxxxx} \quad \boxed{000000000000xxxxxx}$$

Thus, by construction, there is a unique pair (i, j) such that:

$$c = (m_0 + x_i + y_j)^e \bmod N.$$

Now, we define the polynomial $f(X) = \prod_{i=1}^{2^\ell} ((m_0 + y_i + X)^e - c) \bmod N$, which is of degree $e2^\ell$. If x_t corresponds to the correct guess for the bits $b_{\ell+1}, \dots, b_k$, then $f(x_t) = 0$. Hence, if we evaluate $f(X)$ at $x_1, \dots, x_{2^{k-\ell}}$, we would be able to derive the $k - \ell$ higher bits $b_{\ell+1}, \dots, b_k$, which gives rise to Alg. 10.

Algorithm 10 Decrypting Low-Exponent RSA With Known Positions

Input: An RSA modulus $N = pq$ and a ciphertext $c = m^e \bmod N$, where all the bits of m are known, except at k positions b_1, \dots, b_k .

Output: The plaintext m .

- 1: Compute the polynomial $f(X) = \prod_{i=1}^{2^\ell} ((m_0 + y_i + X)^e - c) \bmod N$ of degree $e2^\ell$, with coefficients modulo N , using Alg. 2.
 - 2: Compute the evaluation of $f(X)$ at the points $x^{(1)}, \dots, x^{(2^{k-\ell})}$, using sufficiently many times Alg. 3.
 - 3: Find the unique i such that $f(x^{(i)}) = 0$.
 - 4: Deduce from $x^{(i)}$ the bits $b_{\ell+1}, \dots, b_k$.
 - 5: Find the remaining bits b_1, \dots, b_ℓ by exhaustive search.
-

By definition of ℓ , we have: $\sqrt{2^k/e} \leq 2^\ell \leq 2 \times \sqrt{2^k/e}$ and $\text{sqrte}2^k/2 \leq 2^{k-\ell} \leq 2 \times \sqrt{e2^k}$.

It follows that the overall complexity of Alg. 10. is $\tilde{O}(\sqrt{e2^k})$ polynomial-time operations, which is the “square root” of exhaustive search if e is constant.

5.2 Unknown Positions

In the previous section, we showed how to adapt our noisy factoring algorithm with known positions (Alg. 7) to the RSA case. Similarly, our noisy factoring algorithm with unknown positions (Alg. 9) can also be adapted. If the plaintext m is known except for exactly k unknown bit positions, then one can recover m using on the average $\tilde{O}(\ell\sqrt{ke})$ polynomial-time operations, where $\ell = \binom{n/2}{k/2}$ is roughly $\sqrt{\binom{n}{k}}$.

5.3 Variants

Our technique was presented to decrypt stereotyped low-exponent RSA ciphertexts, but the same technique clearly applies to a slightly more general setting, where the RSA equation is replaced by an arbitrary univariate low-degree polynomial equation. More precisely, instead of $c = m^e \bmod N$, we may assume that $P(m) \equiv 0 \pmod{N}$ where P is a univariate integer polynomial of degree e . This allows to adapt various attacks [6] on low-exponent RSA, such as randomized padding across several blocks.

References

- [1] D. Boneh and G. Durfee. Cryptanalysis of RSA with private key d less than $N^{0.292}$. *IEEE Transactions on Information Theory*, 46(4):1339, 2000.
- [2] A. Bostan, P. Gaudry, and E. Schost. Linear recurrences with polynomial coefficients and application to integer factorization and Cartier-Manin operator. *SIAM Journal on Computing*, 36(6):1777–1806, 2007.
- [3] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *Cryptology ePrint Archive*, Report 2011/344, 2011. <http://eprint.iacr.org/>.
- [4] Y. Chen and P. Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *Proc. ASIACRYPT ’11*, Lecture Notes in Comput. Sci. Springer, 2011.
- [5] H. Cohn and N. Heninger. Approximate common divisors via lattices. *Cryptology ePrint Archive*, Report 2011/437, 2011.
- [6] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptology*, 10(4):233–260, 1997.
- [7] J.-S. Coron, A. Joux, A. Mandal, D. Naccache, and M. Tibouchi. Cryptanalysis of the rsa subgroup assumption from TCC 2005. In *Public Key Cryptography - Proc. PKC 2011*, volume 6571 of *Lecture Notes in Computer Science*, pages 147–155. Springer, 2011.
- [8] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi. Fully homomorphic encryption over the integers with shorter public-keys. In *Advances in Cryptology - Proc. CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*. Springer, 2011.
- [9] J.-S. Coron, D. Naccache, and M. Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *EUROCRYPT*, pages 446–464, 2012.
- [10] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. STOC ’09*, pages 169–178. ACM, 2009.
- [11] C. Gentry and S. Halevi. Public challenges for fully-homomorphic encryption. Available at https://researcher.ibm.com/researcher/view_project.php?id=1548. The implementation is described in [13], 2010.
- [12] C. Gentry and S. Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. *Cryptology ePrint Archive*, Report 2011/279, 2011. <http://eprint.iacr.org/>.
- [13] C. Gentry and S. Halevi. Implementing Gentry’s fully-homomorphic encryption scheme. In *Advances in Cryptology - Proc. EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.
- [14] J. Groth. Cryptography in subgroups of Z_n . In *Proc. TCC 2005*, volume 3378 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.
- [15] D. Harvey and D. S. Roche. An in-place truncated fourier transform and applications to polynomial multiplication. In *Proc. ISSAC ’10*, pages 325–329. ACM, 2010.
- [16] N. Howgrave-Graham. Approximate integer common divisors. In *Proc. CaLC ’01*, volume 2146 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2001.
- [17] T. Mateer. *Fast Fourier Transform Algorithms with Applications*. PhD thesis, Clemson University, 2008.
- [18] A. May. Using LLL-reduction for solving RSA and factorization problems: A survey. 2010. In [22].

- [19] P. L. Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization*. PhD thesis, University of California Los Angeles, 1992.
- [20] P. Q. Nguyen. Public-key cryptanalysis. In I. Luengo, editor, *Recent Trends in Cryptography*, volume 477 of *Contemporary Mathematics*. AMS–RSME, 2009.
- [21] P. Q. Nguyen and D. Stehlé. LLL on the average. In *ANTS*, pages 238–256, 2006.
- [22] P. Q. Nguyen and B. Vallée, editors. *The LLL Algorithm: Survey and Applications*. Information Security and Cryptography. Springer, 2010.
- [23] J. M. Pollard. Theorems on factorization and primality testing. *Proc. Cambridge Philos. Soc.*, 76:521–528, 1974.
- [24] G. Qiao and K.-Y. Lam. RSA signature algorithm for microcontroller implementation. In *Proc. CARDIS '98*, volume 1820 of *Lecture Notes in Computer Science*, pages 353–356. Springer, 2000.
- [25] D. S. Roche. Space- and time-efficient polynomial multiplication. In *Proc. ISSAC '09*, pages 295–302. ACM, 2009.
- [26] V. Shoup. Number Theory C++ Library (NTL) version 5.4.1. Available at <http://www.shoup.net/ntl/>.
- [27] D. R. Stinson. Some baby-step giant-step algorithms for the low hamming weight discrete logarithm problem. *Math. Comput.*, 71(237):379–391, 2002.
- [28] V. Strassen. Einige Resultate über Berechnungskomplexität. *Jber. Deutsch. Math.-Verein.*, 78(1):1–8, 1976/77.
- [29] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology - Proc. EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.
- [30] J. von zur Gathen and J. Gerhard. *Modern computer algebra* (2nd ed.). Cambridge University Press, 2003.

A Practical Assessment of the Cohn-Heninger Attack

At the rump session of Crypto 2011, Cohn and Heninger [5] announced a lattice-based approach for solving PACD and GACD, using Coppersmith’s small-root technique [6], by exploiting several approximate divisors. In theory, this can be applied to all the FHE-challenges of Coron *et al.* [8]: for each challenge, the problem is reduced to running the LLL algorithm on a lattice basis of high dimension and large entries. Accordingly, Cohn and Heninger [5] gave for each FHE-challenge a set of parameters which should work, but they did not give any estimate on the running time of the LLL algorithm for these input matrices, so it was unclear how their method compared with our method and exhaustive search in practice.

In this section, we provide the first running time estimates for the Cohn-Heninger attack on the FHE-challenges. According to our estimates, the CH attack is actually slower than exhaustive search on the challenges, and therefore much slower than our attack.

It is not trivial to estimate the running time of LLL in practice, especially floating-point variants, because even though LLL is known to be polynomial time, the exact polynomial depends on many factors, including the geometry of the input basis. We estimated the running time of LLL reduction on the FHE challenges using two different methods:

1. The first method is based on the complexity analysis of LLL, following the results of Nguyen and Stehlé [21] on LLL on the average. The number of LLL iterations can be heuristically estimated to be proportional to the log of the ratio between the initial LLL potential (which can be computed from the input matrix), and the expected final LLL potential (which can be computed from the usual behaviour of LLL [21], where the Gram-Schmidt norms follow a geometric progression of known parameter). And since the input matrix has huge coefficients, the bit-complexity of each LLL iteration is expected to be at least the average bit-size of the entries, multiplied by the lattice dimension: this is the cost of performing elementary operations on the basis.

2. The second method is more experimental. We run experiments using lattice of similar structure (which depends on the exact parameters chosen by the CH attack), but with different maximal bit-size and dimensions, then we look at the evolution of the running time T when these parameters vary. Experimentally, it seems that T is independent of the size of secret key p and the size of the noise. The toy-challenge lattice has a different structure than the other ones. But for the other challenges, the same parameters t and k (defined by the CH attack) are used, namely $t = 2$ and $k = 1$, which allows for extrapolations. Based on our experiments for these parameters, the least square method gives the following relationship

$$\log(T) = 3.16 \log(D) + 1.68 \log(N) - 24.72,$$

where T is in seconds, N is the maximal bit-size of the entries and D is the lattice dimension.

This gives rise to Table 2, where running time estimates for the CH attack are given, as well as additional information: the parameters m , t and k are defined in [5] and impact the input lattice basis. The row “Estimate 1” used the first method to lower bound the running time. The row “Estimate 2” used the second method to estimate the running time in seconds. One can see that both estimates match rather well (by comparing the row ”Estimate 1” with the row “Security level”): they

Table 2: Time required to break the FHE-challenges by Coron *et al.* [8] using the Cohn-Heninger attack [5].

Name	Toy	Small	Medium	Large
Expected security level from [8]	≥ 42	≥ 52	≥ 62	≥ 72
Lattice dimension	165	595	2211	9591
m	8	33	65	137
t	3	2	2	2
k	1	1	1	1
Maximal bit-size of the input basis	4.8×10^5	1.72×10^6	8.4×10^6	3.8×10^7
Total size of the input basis	26 Mb	230 Mb	4.3 Gb	86 Gb
Estimate 1 of the number of “operations”	2^{50}	2^{59}	2^{66}	2^{75}
Estimate 2 of the running time in seconds	2.1×10^5 2.5 days	1.4×10^8 4.4 year	1.565×10^{11} 5.0×10^3 year	2.46×10^{14} 7.8×10^6 years
Security level with Estimate 2	48	58	68	79

give essentially the same security level, which adds confidence to the order of magnitude of our predictions. However, for the Large-challenge, it can be noted that Estimate 1 is a bit more optimistic than Estimate 2. Yet, the conclusion is clear: for the FHE challenges, we expect the CH attack to be slower than exhaustive search, and therefore much slower than our attack. The CH attack does not contradict the security claims of Coron *et al.* [8], as opposed to our attack. But the CH attack is interesting from a theoretical point of view, and of course, our estimates might have to be revised if the CH attack or LLL algorithms are improved.