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

Yuanmi Chen and Phong Q. Nguyen

[1] ENS, Dept. Informatique, 45 rue d'Ulm, 75005 Paris, France
http://www.eleves.ens.fr/home/ychen/
[2] INRIA, France and Tsinghua University, Institute for Advanced Study, China
http://www.di.ens.fr/~pnguyen/

Abstract. At EUROCRYPT '10, van Dijk *et al.* 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 *et al.* 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. Interestingly, our main technique can also be applied to other settings, such as noisy factoring and attacking low-exponent RSA.

1 Introduction

Following Gentry's breakthrough work [11], 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 [11,29,9,3,13], 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 [17]. 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 $x_0, \ldots, x_m$ 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, [9] 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 [9], 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 [9], 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 [29,9].

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 [29,9]. 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 [9], which were assumed to have comparable security to the FHE challenges proposed by Gentry and Halevi in [12]: 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 [9] have a much lower security level than those of Gentry and Halevi [14].

Table 1. Time required to break the FHE challenges by Coron *et al.* [9]. Size in bits, running time in seconds for a single 2.27GHz-core with 72 Gb. 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	$8.3 * 10^6$	$1.96 * 10^{10}$		$1.8 * 10^{13}$	
	40 mins	96 days	623 years		569193 years	
Concrete security level	≈ 42	≈ 54	≈ 65		≈ 75	
Running time of the new attack	99	25665	$1.64 * 10^7$	$6.6 * 10^6$	$6.79 * 10^{10}$	$2.9 * 10^8$
	1.6 min	7.1 hours	190 days	76 days	2153 years	9 years
Parameters	$d = 2^8$	$d = 2^{12}$	$d = 2^{13}$	$d = 2^{15}$	$d = 2^{10}$	$d = 2^{19}$
Memory	$\leq$ 130 Mb	$\leq$ 15 Gb	$\leq$ 72 Gb	$\approx$ 240 Gb	$\leq$ 72 Gb	$\approx$ 25 Tb
Speedup	24	324	1202	2977	264	62543
New security level	≤ 37.7	≤ 45.7	≤ 55	≤ 54	≤ 67	≤ 59

Interestingly, we can also apply our technique to different settings, such as noisy factoring, fault attacks on RSA-CRT, 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 differing 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 (*e.g.* the Pollard-Strassen factorization algorithm [23,28] or in ECM speedup [20]), in the folklore square-root attack on RSA with small CRT exponents (mentioned by [1] and described in [24,21]), as well as in the recent square-root attack [8] by Coron, Joux, Mandal, Naccache and Tibouchi on Groth's RSA Subgroup Assumption [15]. But this does not imply that our attack is trivial, especially since the authors of [9] form a subset of the authors of [8]. In fact, in most cryptanalytic applications (including [8]) of multipoint evaluation, one is 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 [6] 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 provide evidence in the full version [5] that for the FHE challenges of [9], 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.* [9]. Finally, we apply our main technique to different settings: noisy factoring (Sect. 4) and attacking low-exponent RSA (Sect. 5). More information can be found in the full version [5].

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 [9, Sect 6.1]):

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

which holds with overwhelming probability for the parameters of [9]. 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 [9] 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:

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

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:

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

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 [30,18]) 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 $\rho$.
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
   $0, 2^{\rho'}, \ldots, 2^{\rho'}(2^{\rho'+(\rho \bmod 2)} - 1)$, 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 [30,18]):

- 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.

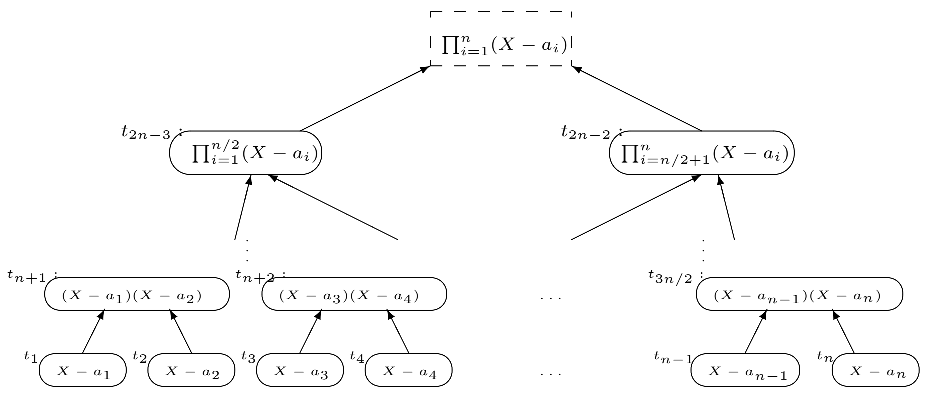
$\prod_{i=1}^{n}(X - a_i)$
t_{2n-3} : $\prod_{i=1}^{n/2}(X - a_i)$
t_{2n-2} : $\prod_{i=n/2+1}^{n}(X - a_i)$
t_{n+1} : $(X - a_1)(X - a_2)$
t_{n+2} : $(X - a_3)(X - a_4)$
$t_{3n/2}$: $(X - a_{n-1})(X - a_n)$
t_1 $X - a_1$
t_2 $X - a_2$
t_3 $X - a_3$
t_4 $X - a_4$
t_{n-1} $X - a_{n-1}$
t_n $X - a_n$

Fig. 1. Polynomial product tree $T = \{t_1, \ldots, t_{2n}\}$ for $\{a_1, \ldots, a_n\}$

Algorithm 2. $[T, D] \leftarrow$`TreeProduct`(A)

```
Input: A set of $n = 2^l$ numbers $\{a_1, \ldots, a_n\}$.
Output: The polynomial product tree $T = \{t_1, \ldots, t_{2n-1}\}$, corresponding to the evaluation of
    points $A = \{a_1, \ldots, a_n\}$ as shown in Figure 1.
    $D = [d_1, \ldots, d_{2n-1}]$ descendant indices for non-leaf nodes or 0 for leaf node.
1: for $i = 1 \ldots n$ do
2:     $t_i \leftarrow X - a_i$; $d_j \leftarrow 0$ {Initializing leaf nodes}
3: end for
4: $i \leftarrow 1$; $j \leftarrow n + 1$ {Index of lower and upper levels}
5: while $j \leqslant 2n - 1$ do
6:     $t_j \leftarrow t_i \cdot t_{i+1}$; $d_j \leftarrow i$; $i \leftarrow i + 2$; $j \leftarrow j + 1$
7: end while
```

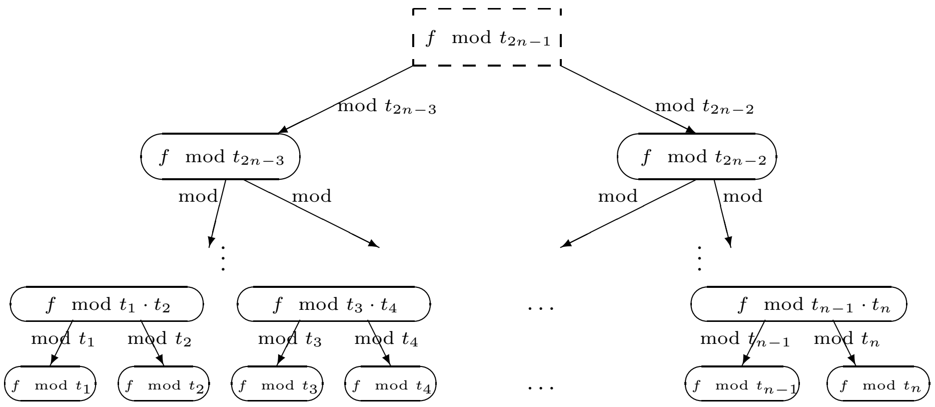

Fig. 2. Evaluation on the polynomial tree $T = \{t_1, \ldots, t_{2n-1}\}$ for $\{a_1, \ldots, a_n\}$

Algorithm 3. $V \leftarrow$ `RecursiveEvaluation`(f, t_i, D)

```
Input: A polynomial $f$ of degree $n$.
    A polynomial product tree rooted at $t_i$, and whose leaves are $\{X - a_k, \ldots, X - a_m\}$
    An array $D = [d_1, \ldots, d_{2n-1}]$ descendant indices for non-leaf nodes or 0 for leaf node.
Output: $V = \{f(a_k), \ldots, f(a_m)\}$
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:     $g_1 \leftarrow f \mod t_{d_i}$ {left subtree}
5:     $V_1 \leftarrow$ RecursiveEvaluation$(g_1, t_{d_i}, D)$
6:     $g_2 \leftarrow f \mod t_{d_i+1}$ {right subtree}
7:     $V_2 \leftarrow$ 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 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 $\mathbf{d}(\alpha,\beta,d) = \beta\cdot 2\ldots d\cdot(\alpha-d\beta)\ldots(\alpha+d\beta)$. And suppose also that the inverse of $\mathbf{d}(\alpha,\beta,d)$ is known. Let $F(\cdot)\in\mathbb{P}[X]$ of degree at most d and $x_0\in\mathbb{P}$. There exists an algorithm `ShiftPoly` which, given as input $F(x_0)$, $F(x_0+\beta)$, …, $F(x_0+d\beta)$, outputs $F(x_0+\alpha)$, $F(x_0+\alpha+\beta)$, …, $F(x_0+\alpha+d\beta)$ 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 $E(k_1,\ldots,k_j)$ 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 $\{k_1,\ldots,k_j\}$. Given a set A and an element and p, $A+p$ is defined as$\{a+p, a\in A\}$. Then we have

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

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 $F_i(X) = \prod_{Y\in E(k_1,\ldots,k_i)} f(X+Y)$ on $E(b_{k-i},\ldots,k_\rho)$ 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 $i = 1,\ldots,\lfloor\rho/2\rfloor$, the evaluation of $F_i(X)$ on points $X\in E(k_{\rho-i+1},\ldots,k_\rho)$
Output: the evaluation of $F_{i+1}(X)$ on points $X\in E(k_{\rho-i},\ldots,k_\rho)$
1: $F_i(X)$ for $X\in E(k_{\rho-i+1},\ldots,k_\rho)+2^{k_{\rho-i}} \leftarrow$ ShiftPoly($F_i(X), X\in E(k_{\rho-i+1},\ldots,k_\rho)$)
2: $F_i(X)$ for $X\in E(k_{\rho-i},\ldots,k_\rho)+2^{k_{i+1}} \leftarrow$ ShiftPoly($F_i(X), X\in E(k_{\rho-i},\ldots,k_\rho)$)
3: $F_{i+1}(X) = F_i(X)\cdot F_i(X+2^{k_{i+1}})$, for all $X\in E(k_{\rho-i},\ldots,k_\rho)$
```

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 to all $(x_0 - r_0, x_1)$ where r_0 ranges over $\{0, \ldots, 2^\rho - 1\}$.

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 factors 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 was noted with [10] that one can heuristically beat the GACD bound $\tilde{O}(2^{3\rho/2})$ using more samples x_i, by removing the "smooth part" of $\gcd(y_1, \ldots, y_s)$ 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 instance, 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 lower 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 [9]: 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. 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:

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

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 $\rho$, 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.

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 $(X - a_1)(X - a_2) = X^2 - (a_1 + a_2)X + a_1 a_2$, 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:

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

where

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

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 $0, -d, -2d, \dots, -d(d-1)$.

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 phenomenons, 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 $f'_{d,k}(x)$.

Precomputations. Now that we use (5), we change several times the polynomial $f'_{d,k}(x)$, but we keep the same evaluation points $0, -d, -2d, \ldots, -d(d-1)$, 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 $m \geqslant n$, we define the m-degree polynomial rev(f, m) as rev$(f, m) = f(1/X) \cdot X^m$. In Alg. 6, Line 1 is

Algorithm 6. Remainder using Newton's method (see [18, 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 $h = f \mod g$
1: $\bar{g} \leftarrow \text{Inverse}(\text{rev}(g, n)) \mod X^n$
2: $s \leftarrow \text{rev}(f, 2n-1) \cdot \bar{g} \mod X^n$
3: $h \leftarrow f - \text{rev}(s, n-1) \cdot g$
```

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.

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: a better 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 [25,16] 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.* [9], which come in four flavours: Toy, Small, Medium and Large. The security level ℓ is defined in [9] is defined 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 [9].

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 [9].

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 [9]. 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 used 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 $\geq$ 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 [9] fall short of the expected security level.

Hence, one needs to increase the parameters of the FHE scheme of [9], which makes it less competitive with the FHE implementation of [14]. It can be noted that the new security levels of the challenges of [9] are much lower than those given by [4] on the challenges of Gentry and Halevi [14], 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 [7] 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 [19] 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.

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 $0 \leqslant i \leqslant n - 1$.

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

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

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 $p_{b_1}, \ldots, p_{b_k}$ can be regrouped into two sets $\{p_{b_1}, \ldots, p_{b_\ell}\}$, and $\{p_{b_{\ell+1}}, \ldots, p_k\}$ of roughly the same size $\ell = \lfloor k/2 \rfloor$:

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

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

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

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 $p_0, \ldots, p_{n-1}$ of $p$, except the $k$ bits $p_{b_1}, \ldots, p_{b_k}$,
    where the bit positions $b_1 \leq b_2 \leq \cdots \leq b_k$ are known.
Output: The secret factor $p = \sum_{i=0}^{n-1} p_i 2^i$ of $N$.
1: Compute the polynomial $f(X) = \prod_{i=1}^{2^\ell} \left(X + y^{(i)}\right) \bmod N$ of degree $2^\ell$, with coefficients
    modulo $N$, using Alg. 2.
2: Compute the evaluation of $f(X)$ at the points $\{x^{(1)}, \ldots, x^{(2^{k-\ell})}\}$, using $1 + (k \bmod 2)$
    times Alg. 3 with $2^\ell$ points.
3: return $p \leftarrow \gcd\left(N, \prod_{i=1}^{2^{k-\ell}} \left(f(x^{(i)})\right) \bmod N\right)$
```

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

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

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

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 $i = 1, \ldots, \lfloor k/2 \rfloor$ 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 $X \in E(b_{\lfloor k/2\rfloor+2}, \ldots, b_k) + 2^{b_{\lfloor k/2\rfloor+1}}$
         $\leftarrow \texttt{ShiftPoly}(F_{\lfloor k/2\rfloor}(X), X \in E(b_{\lfloor k/2\rfloor+2}, \ldots, b_k), 2^{b_{\lfloor k/2\rfloor+1}})$
7: end if
8: $p'' = \gcd\left(N, \prod_{X\in E(b_{\lfloor k/2\rfloor+1},\ldots,b_k)} \left(F_{\lfloor k/2\rfloor}(X)\right)\right)$
9: return $p''$
```

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.

Pick a random subset S of $\{0, \ldots, n-1\}$ 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 $1 \leqslant i \leq \ell$ 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 $1 \leqslant i \leq \ell$ 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:

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

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 $\{0, \ldots, n-1\}$ containing exactly $n/2$ elements.
3:    Compute the integers $x^{(i)}$ and $y^{(i)}$ for $1 \leqslant i \leq \ell = \binom{n/2}{k/2}$.
4:    Compute the polynomial $f(X) = \prod_{j=1}^{\ell}(X + y^{(j)}) \mod N$.
5:    Compute the evaluation of $f(X)$ at the $\ell$ points $\{x^{(1)}, \ldots, x^{(\ell)}\}$.
6:    $p'' \leftarrow \gcd\left(N, \prod_{i=1}^{\ell}\left(f(x^{(i)})\right) \mod N\right)$
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].

5 Applications to Low-Exponent RSA

Our previous algorithms for noisy factoring easily apply to fault attacks on RSA-CRT signatures where few bits of the message are unknown. There, one retrieves the factorization by computing $\gcd(s^e - m \pmod N), N)$ where s is a faulty RSA signature of a message m, This requires the full knowledge of the message m, but our attack applies when not all bits are known, just like in noisy factoring. In this section, we show that our algorithms can also be adapted to attacks on 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 exactly the same two cases as for noisy factoring. This setting is usually called stereotyped RSA encryption [7]: there are well-known lattice attacks [7,19] against stereotyped RSA, but they require that the unknown bits are consecutive, or split across extremely few blocks.

Known Positions. Assume that m is a plaintext of n bits, among which only k bits are unknown, whose (arbitrary) positions are $b_1, \ldots, b_k$. Let $c = m^e \mod N$ 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 $k > 0$. Let m_0 be derived from m by keeping all the known $n - k$ bits, and setting all the k unknown bits to 0. For $1 \leqslant i \leqslant 2^{k-\ell}$, let the x_i's enumerate all the integers when $(b_{\ell+1}, \ldots, b_k)$ ranges over $\{0,1\}^{k-\ell}$. Similarly, for $1 \leqslant j \leqslant 2^\ell$, let the y_j's enumerate all the integers when $(b_1, \ldots, b_\ell)$ ranges over $\{0,1\}^\ell$.

Thus, 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 $b_{\ell+1}, \ldots, b_k$, then $f(x_t) = 0$. Hence, if we evaluate $f(X)$ at $x_1, \ldots, x_{2^{k-\ell}}$, we would be able to derive the $k - \ell$ higher bits $b_{\ell+1}, \ldots b_k$, 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 $b_1, \ldots, b_k$.
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 $x^{(1)}, \ldots, x^{(2^{k-\ell})}$, using sufficiently many times Alg. 3.
3: Find the unique $i$ such that $f(x^{(i)}) = 0$.
4: Deduce from $x^{(i)}$ the bits $b_{\ell+1}, \ldots, b_k$.
5: Find the remaining bits $b_1, \ldots, b_\ell$ by exhaustive search.
```

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

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 $\sqrt{\binom{n}{k}}$.

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 [7] on low-exponent RSA, such as randomized padding across several blocks.

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.

References

1. Boneh, D., Durfee, G.: Cryptanalysis of RSA with private key d less than $N^{0.292}$. IEEE Transactions on Information Theory 46(4), 1339 (2000)
2. Bostan, A., Gaudry, P., Schost, E.: Linear recurrences with polynomial coefficients and application to integer factorization and Cartier-Manin operator. SIAM Journal on Computing 36(6), 1777–1806 (2007)
3. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. Cryptology ePrint Archive, Report 2011/344 (2011), http://eprint.iacr.org/
4. Chen, Y., Nguyen, P.Q.: BKZ 2.0: Better Lattice Security Estimates. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 1–20. Springer, Heidelberg (2011)
5. Chen, Y., Nguyen, P.Q.: Faster Algorithms for Approximate Common Divisors: Breaking Fully-Homomorphic-Encryption Challenges over the Integers. Cryptology ePrint Archive, Report 2011/436 (2011), http://eprint.iacr.org/
6. Cohn, H., Heninger, N.: Approximate common divisors via lattices. Cryptology ePrint Archive, Report 2011/437 (2011)
7. Coppersmith, D.: Small solutions to polynomial equations, and low exponent RSA vulnerabilities. J. Cryptology 10(4), 233–260 (1997)
8. Coron, J.-S., Joux, A., Mandal, A., Naccache, D., Tibouchi, M.: Cryptanalysis of the RSA Subgroup Assumption from TCC 2005. In: Catalano, D., Fazio, N., Gennaro, R., Nicolosi, A. (eds.) PKC 2011. LNCS, vol. 6571, pp. 147–155. Springer, Heidelberg (2011)
9. Coron, J.-S., Mandal, A., Naccache, D., Tibouchi, M.: Fully Homomorphic Encryption over the Integers with Shorter Public Keys. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 487–504. Springer, Heidelberg (2011)
10. Coron, J.-S., Naccache, D., Tibouchi, M.: Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers. Cryptology ePrint Archive, Report 2011/440 (2011), http://eprint.iacr.org/

11. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proc. STOC 2009, pp. 169–178. ACM (2009)
12. Gentry, C., Halevi, S.: Public challenges for fully-homomorphic encryption. The implementation is described in [12] (2010), `https://researcher.ibm.com/researcher/view_project.php?id=1548`
13. Gentry, C., Halevi, S.: Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. Cryptology ePrint Archive, Report 2011/279 (2011), `http://eprint.iacr.org/`
14. Gentry, C., Halevi, S.: Implementing Gentry's Fully-Homomorphic Encryption Scheme. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 129–148. Springer, Heidelberg (2011)
15. Groth, J.: Cryptography in Subgroups of $\mathbb{Z}_n^*$. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 50–65. Springer, Heidelberg (2005)
16. Harvey, D., Roche, D.S.: An in-place truncated fourier transform and applications to polynomial multiplication. In: Proc. ISSAC 2010, pp. 325–329. ACM (2010)
17. Howgrave-Graham, N.: Approximate Integer Common Divisors. In: Silverman, J.H. (ed.) CaLC 2001. LNCS, vol. 2146, pp. 51–66. Springer, Heidelberg (2001)
18. Mateer, T.: Fast Fourier Transform Algorithms with Applications. PhD thesis, Clemson University (2008)
19. May, A.: Using LLL-reduction for solving RSA and factorization problems: A survey. In: [21] (2010)
20. Montgomery, P.L.: An FFT Extension of the Elliptic Curve Method of Factorization. PhD thesis, University of California Los Angeles (1992)
21. Nguyen, P.Q.: Public-key cryptanalysis. In: Luengo, I. (ed.) Recent Trends in Cryptography. Contemporary Mathematics, vol. 477. AMS–RSME (2009)
22. Nguyen, P.Q., Vallée, B. (eds.): The LLL Algorithm: Survey and Applications. Information Security and Cryptography. Springer, Heidelberg (2010)
23. Pollard, J.M.: Theorems on factorization and primality testing. Proc. Cambridge Philos. Soc. 76, 521–528 (1974)
24. Qiao, G., Lam, K.-Y.: RSA Signature Algorithm for Microcontroller Implementation. In: Schneier, B., Quisquater, J.-J. (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 353–356. Springer, Heidelberg (2000)
25. Roche, D.S.: Space- and time-efficient polynomial multiplication. In: Proc. ISSAC 2009, pp. 295–302. ACM (2009)
26. Shoup, V.: Number Theory C++ Library (NTL) version 5.4.1, `http://www.shoup.net/ntl/`
27. Stinson, D.R.: Some baby-step giant-step algorithms for the low hamming weight discrete logarithm problem. Math. Comput. 71(237), 379–391 (2002)
28. Strassen, V.: Einige Resultate über Berechnungskomplexität. Jber. Deutsch. Math.-Verein. 78(1), 1–8 (1976/1977)
29. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully Homomorphic Encryption over the Integers. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. Springer, Heidelberg (2010)
30. von Zur Gathen, J., Gerhard, J.: Modern computer algebra, 2nd edn. Cambridge University Press (2003)