Breaking RSA May Be Easier Than Factoring

(Extended Abstract)

Dan Boneh
dabo@cs.stanford.edu
Stanford University

Ramarathnam Venkatesan
venkie@microsoft.com
Microsoft Research

Abstract

We provide evidence that breaking low-exponent RSA cannot be equivalent to factoring integers. We show that an *algebraic* reduction from factoring to breaking low-exponent RSA can be converted into an efficient factoring algorithm. Thus, in effect an oracle for breaking RSA does not help in factoring integers. Our result suggests an explanation for the lack of progress in proving that breaking RSA is equivalent to factoring. We emphasize that our results do not expose any specific weakness in the RSA system.

Keywords: RSA, Factoring, Straight line programs, Algebraic circuits.

1 Introduction

Two longstanding open problems in cryptography are to prove or disprove that breaking the RSA system [10] is as hard as factoring integers and that breaking the Diffie-Hellman protocol [3] is as hard as computing discrete log. Although some recent progress has been made on the second problem [8, 9, 1] very little progress has been made on the first. A harder version of the first problem asks whether breaking low exponent RSA (LE–RSA) is as hard as factoring. Such reductions are desirable since they prove that the security of the RSA system follows from the intractability of factoring integers. In this paper we take a step towards disproving the equivalence of factoring and breaking low exponent RSA.

One way of disproving the equivalence is to present an algorithm for breaking LE–RSA that does not seem to provide a factoring algorithm. This is not our approach. Instead, we wish to show that if one could give an efficient reduction from factoring to breaking LE–RSA then the reduction can be converted into an actual efficient factoring algorithm. This proves that unless factoring is easy, the two problems cannot be equivalent. We make progress towards achieving this goal by showing that any efficient *algebraic* reduction from factoring to breaking LE–RSA can be converted into an efficient factoring algorithm. Thus, breaking LE–RSA cannot be equivalent to factoring under algebraic reductions (unless factoring is easy). Essentially, algebraic reductions are restricted to only perform arithmetic operations. They are not allowed to aggressively manipulate bits, e.g. given $x, y \in \mathbb{Z}_N$ they cannot compute $x \oplus y$. A precise definition of this notion is presented in Section 3.

To give a more concrete description of our results we consider the problem of breaking RSA when the public exponent is $e = 3$. In the body of the paper we allow any low public exponent (i.e. less than some fixed constant). Let $N = pq$ be a product of two large primes with $\gcd(\varphi(N), 3) = 1$. The encryption of a plain-text $x \in \mathbb{Z}_N$ is $x^3 \bmod N$. Breaking the system amounts to computing cube roots modulo N. To prove that breaking this system is equivalent to factoring one has to present a polynomial time oracle algorithm $\mathcal{A}$ that given $N = pq$ and a cube root oracle modulo N, factors N. We show that any such *algebraic* oracle algorithm $\mathcal{A}$, that does not make too many oracle calls, can be converted into a non-oracle algorithm $\mathcal{B}$ that factors the same set of integers as $\mathcal{A}$. In other words, if one can prove that taking cube roots modulo N is as hard as factoring N then the proof will provide a "real" factoring algorithm (that does not make use of an oracle). Hence, under these conditions a cube root oracle does not help in factoring N. We note that when $\gcd(\varphi(N), 3) \neq 1$ it is well known that taking cube roots is as hard as factoring. However, in this case 3 cannot be used as an RSA encryption exponent. For this reason, throughout the paper we only concern ourselves with the case where $\gcd(\varphi(N), e) = 1$.

Our results apply to large e as well – they apply whenever e is a smooth integer. We discuss this extension at the end of the paper.

Understanding our result

We emphasize that our result does not point to any weakness of the RSA system. Instead, it provides some evidence that breaking LE–RSA may be easier than factoring. Even if breaking LE–RSA is indeed easier than factoring, nothing in this work contests that it is likely to be intractable.

The class of algebraic reductions is not overly restrictive. For example, it encompasses some number theoretic and factoring algorithms. These are often simply polynomials evaluated modulo N at various inputs. A factorization is obtained once the polynomial evaluates to a non-zero non-invertible element modulo N (if $0 \neq x \in \mathbb{Z}_N$ is not invertible then $\gcd(N, x)$ gives a non-trivial factor of N). Both Pollard's $p - 1$ factoring [6] and Elliptic curve factoring [7] can be viewed as such (one evaluates the polynomial $x^B - 1$ for some smooth integer B, the other evaluates the B'th division polynomial of a random elliptic curve at a random point). It is natural to ask whether an oracle for breaking low exponent RSA can aid this type of factoring algorithms? Our results show that the answer is no as long as the algorithm does not make too many oracle calls.

Our methods leave it open that reductions using bit manipulations (i.e. non-algebraic operations as described in Section 3) on the outputs of an RSA oracle may reduce factoring to breaking RSA. However, we note that current attacks on low public exponent RSA [4, 2] decrypt a message without factoring the modulus. Our results suggest that this is no accident, since breaking LE–RSA may be easier than factoring.

2 Straight line programs

Our results make use of *straight line programs* for polynomials. In this section we define this notion and prove some properties of it. Throughout the section we let R be a ring with a cyclic additive group. The reader should think of R as the field $\mathbb{F}_p$ or the ring $\mathbb{Z}_N$ for an RSA composite $N = pq$.

Definition 2.1 *A straight line program (SLP) for a polynomial $f \in R[x_1, \ldots, x_k]$ is a sequence of polynomials $f_0, f_1, f_2, \ldots, f_m \in R[x_1, \ldots, x_k]$ such that $f_m = f$ and for all $i = 1, \ldots, m$ the polynomial f_i is either the constant 1, a variable x_j or $g_i = g_k \circ g_l$ where $k, l < i$ and $\circ \in \{+, -, *\}$.*

Examples of polynomials with low straight line complexity are univariate sparse polynomials (i.e. polynomials whose degree is much larger than the number of their terms). An SLP of length L for a polynomial f defines a method for evaluating f using exactly L arithmetic operations. An SLP is represented as a sequence of triplets $(i, \circ, j)$ where $\circ \in \{+, -, *\}$. The k'th such triplet implies that f_k, the k'th polynomial in the program, is equal to $f_i \circ f_j$. An SLP can compute more than one polynomial: we say that an SLP computes the polynomials $g_1, \ldots, g_r$ if these polynomials appear in the last r steps of the SLP.

We note that one may view SLP's as algebraic circuits (circuits whose gates compute $+, -, *$). The difference between the two notions is that the complexity of an SLP is measured by its size. An algebraic circuit is measured by both its size and depth. Since in this paper we ignore circuit depth we restrict our attention to SLP's.

2.1 Euclid's algorithm and SLP's

We next prove some properties of straight line programs. The reader may skip to Section 3 and return to the following lemmas when referenced.

Let f, g be two polynomials in $\mathbb{Z}_N[x]$ where $N = pq$. Let d_p be the degree of $\gcd(f, g)$ when f and g are reduced modulo p and let d_q be the degree of the gcd when they are reduced modulo q. Suppose $d_q \neq d_p$. Then when one tries to apply Euclid's algorithm to f and g in $\mathbb{Z}_N$ the factorization of N is leaked since at some point Euclid's algorithm must generate a polynomial whose leading coefficient is not invertible modulo N. This coefficient must have a non-trivial gcd with N, thus leaking the factorization. Note that since $\mathbb{Z}_N$ is not an integral domain Euclid's algorithm is not well defined in $\mathbb{Z}_N[x]$. In fact, the notation $f \bmod g$ is not well defined. When the leading coefficient of g is in $\mathbb{Z}_N^*$ we define the polynomial $f \bmod g$ as the output of the standard polynomial division algorithm. Otherwise we say that $f \bmod g$ is *undefined.* When $f \bmod g$ is undefined, the leading coefficient of g reveals the factorization of N.

Now, suppose f and g are given as SLP's in the variables $x, z_1, \ldots, z_k$. We view both f and g as polynomials in x whose coefficients are polynomials in the z_i's. We ask whether it is still possible to carry out Euclid's algorithm and obtain an analogous result to the one discussed above. The next lemma provides a positive answer to this question provided the degree of g in x is small. We use the following notation: given a polynomial $f \in \mathbb{Z}_N[x]$ we denote by f_p the polynomial f reduced modulo p where p is a prime factor of N.

Lemma 2.1 *Let $N = pq$ and $f \in Z_N[x, z_1, \ldots, z_k]$ be a polynomial given as an* SLP *of length L. Let $g(x, z_1, \ldots, z_k) = x^m - h(z_1, \ldots, z_k)$ where h is given as an* SLP *of length L. Both polynomials f and g are regarded as polynomials in x with coefficients in $\mathbb{Z}_N[z_1, \ldots, z_k]$. Then there exists a polynomial time algorithm (in L and 2^m) that given f, g outputs 2^m* SLP*'s in $z_1, \ldots, z_k$ satisfying the following:*

1. *The length of each* SLP *is bounded by $2m^2L + m^3$.*
2. *For any $\bar{c} = (c_1, \ldots, c_k) \in \mathbb{Z}_N^k$ satisfying*

$$deg\Big(\gcd\left(f_p(x, \bar{c}),\, g_p(x, \bar{c})\right)\Big) \neq deg\Big(\gcd\left(f_q(x, \bar{c}),\, g_q(x, \bar{c})\right)\Big)$$

   at least one of the 2^m programs on input $c_1, \ldots, c_k$ produces a non-zero non-invertible element of $\mathbb{Z}_N$.

The proof of the lemma is a bit tedious. Essentially we apply Euclid's algorithm to the polynomials f and g. The 2^m programs generated in the lemma correspond to coefficients of polynomials generated during the execution of Euclid's algorithm. Note that all these coefficients are polynomials in $z_1, \ldots, z_k$. When evaluated at appropriate values $c_1, \ldots, c_k$ (namely the ones satisfying condition two of the lemma) one of these coefficients must evaluate to a non-zero and non-invertible element in $\mathbb{Z}_N$. The first step is to build the polynomial $f' = f \bmod g$. The following lemma shows how to build an SLP that computes the coefficients of f' (each of these coefficients is a polynomial in $z_1, \ldots, z_k$). The lemma is quite easy. For completeness we sketch its proof in the appendix.

Lemma 2.2 *Let f and g be polynomials as in Lemma 2.1. Then there exists a polynomial time algorithm (in L and m) that outputs an SLP of length at most $2m^2L$ in which the last m steps are the coefficients of $f \bmod g$.*

We can now complete the proof of Lemma 2.1.

Proof of Lemma 2.1 Having built an SLP for the coefficients of $f' = f \bmod g$ we need to continue Euclid's algorithm and compute $g \bmod f'$. Since each of the m coefficients of f' is itself a polynomial in $z_1, \ldots, z_k$ we cannot determine the exact degree of f' (for different settings of $z_1, \ldots, z_k$ the polynomial f' will have different degrees). We cannot build an SLP for the coefficients of $g \bmod f'$ without knowing the degree of f'. To solve this problem we build an SLP for each of the possible m values for the degree of f'. Thus, for each $r = 0, \ldots, m-1$ we obtain a program that computes the coefficients of $g \bmod f'$ assuming the degree of f' is r. By allowing the programs to generate an invertible constant multiple of $g \bmod f'$ we can avoid the use of division. We iterate this process until the Euclidean algorithm is completed. At each stage of the algorithm, when computing $f^{(j-1)} \bmod f^{(j)}$ all possible values for the degree of $f^{(j)}$ are explored.

Recall that our objective is to create an SLP for the leading coefficient of all polynomials generated by Euclid's algorithm during the computation of $\gcd(f, g)$. Normally there would only be m such polynomials. However, since we try all possible degree values for intermediate polynomials we end up with at most 2^m SLP's for leading coefficients. We prove that at most 2^m SLP's are generated by induction on m. For $m = 1$ (i.e. g linear in x) the claim is trivial. If the claim holds for all $r \leq m-1$ then the number of programs generated when g has degree m in x is at most $\sum_{r=1}^{m-1} 2^r < 2^m$ (each of the m values $r = 0, \ldots, m-1$ for the degree of f' generates at most 2^r programs). Hence, Euclid's algorithm with symbolic coefficients generates at most 2^m programs. Each program has length at most $2m^2L + m^3$ as required. □

2.2 Eliminating division from straight line programs

Our definition of straight line programs does not allow for division. The reason is that division can be avoided altogether. Division turns out to be problematic for what we have in mind; the ability to avoid it is very helpful. We say that the evaluation of a division-SLP at a point $\bar{x} \in \mathbb{F}_p^k$ completes successfully if there are no divisions by zero along the way.

Lemma 2.3 *Let $f \in \mathbb{F}_p[x_1, \ldots, x_k]$ be a polynomial given as a division–SLP of length L. Then in linear time in L one can generate two SLP's g and h each of length $4L$ such that $f = g/h$. Furthermore, let $\bar{x} \in \mathbb{F}_p^k$ be an input for which the evaluation of f completes successfully. Then $\bar{x}$ is a root of f if and only if it is a root of g.*

We include a proof of the lemma in the final version of the paper. The lemma shows that we can always convert a division-SLP into a division free SLP while maintaining the same roots. Hence, if a division-SLP can be used to factor, it can be converted into a division free SLP that can also be used to factor.

3 Main results

Our method of transforming a "factoring to RSA" reduction into a real factoring algorithm applies whenever the reduction algorithm belongs to a certain "natural" class of algorithms. In this section we precisely define our notion of "natural" and prove our results. We begin by showing how to transform a straight line reduction into a factoring algorithm and then in Section 3.2 describe our full result.

Since we are mostly interested in factoring numbers that are a product of two large primes we define the following set:

$$\mathbb{Z}_{(2)}(n) = \{N \mid N < 2^n \ , \ N = pq \ , \ p > q > 2^{n/4} \ , \ p, q \,\text{prime}\}$$

We say that an algorithm $\mathcal{A}$ factors a non-negligible fractions of the integers in $\mathbb{Z}_{(2)}(n)$ if there exists a constant c such that infinitely often $\mathcal{A}$ factors $1/n^c$ of the integers in $\mathbb{Z}_{(2)}(n)$.

3.1 Removing an RSA oracle from straight line programs

Factoring algorithms are often simply straight line programs evaluated modulo N at various inputs. A factorization is obtained once the straight line program outputs a non-zero non-invertible element modulo N. Both Pollard's $p-1$ factoring [6] and Elliptic curve factoring [7] can be viewed as straight line factoring algorithms. In this section we show that an oracle for breaking low exponent RSA cannot aid straight line factoring algorithms as long as the algorithm doesn't make too many oracle calls. The following definition captures the notion of an SLP combined with an oracle for breaking LE–RSA. We denote the maximum allowable encryption exponent by ω and regard it as an absolute constant.

Definition 3.1 *Let ω be a fixed constant.*

- *A straight line* RSA *program (*RSA–SLP*) P is a sequence of algebraic expressions $1, c_1, c_2, \dots, c_m$ such that for all $i = 1, \dots, m$ the expression c_i is either $c_i = c_k \circ c_l$ for some $k, l < i$ and $\circ \in \{+, -, *\}$ or $c_i = \sqrt[e]{c_k}$ for some $k < i$ and $e < \omega$.*
- *The program can be successfully evaluated modulo N if all steps of the form $c_i = \sqrt[e]{c_k}$ with $k < i$ satisfy $\gcd(\varphi(N), e) = 1$. We refer to these steps of the program as radical steps.*

An RSA–SLP is an algebraic circuit in which gates perform arithmetic operations as well as take e'th roots (for small e). Next, we define the notion of a *straight line reduction* from factoring to breaking LE–RSA. Essentially, the reduction must factor elements of $\mathbb{Z}_{(2)}(n)$ only using RSA–SLP's.

Definition 3.2

- *An* RSA–SLP *P is said to factor N if it can be successfully evaluated modulo N and it evaluates to a non-zero non-invertible element. A set of* RSA–SLP*'s is said to factor N if one of the programs in the set factors N.*
- *A straight line reduction is a randomized algorithm $\mathcal{A}$ that on input n outputs a set of* RSA–SLP*'s $\{P_1, \dots, P_k\}$. Denote the output set by $\mathcal{A}(n)$. For a non-negligible fraction of the $N \in \mathbb{Z}_{(2)}(n)$ the set $\mathcal{A}(n)$ must factor N (with probability at least $\frac{1}{2}$ over the random bits of $\mathcal{A}$).*

An expected polynomial time straight line reduction $\mathcal{A}$ would *prove* that breaking low exponent RSA is as hard as factoring. The main result of this section shows that such a reduction can be converted into a real polynomial time factoring algorithm. Hence, an RSA breaking oracle does not help a straight line factoring algorithm. Alternatively, factoring is not reducible to breaking LE–RSA using straight line reductions, unless factoring is easy.

Theorem 3.1 *Suppose there exists a straight line reduction $\mathcal{A}$ whose running time is $T(n)$. Further suppose that each of the RSA–SLP's generated by $\mathcal{A}$ on input $N \in \mathbb{Z}_{(2)}(n)$ contains at most $O(\log T(n))$ radical steps. Then there is a real factoring algorithm $\mathcal{B}$ whose running time is $T(n)^{O(1)}$ and factors all $N \in \mathbb{Z}_{(2)}(n)$ that $\mathcal{A}$ does.*

The main tool used in the proof of Theorem 3.1 is presented in the next lemma. The statement of the lemma requires that we precisely define the degree of a polynomial $g(x) = \sum_{i=0}^{d} a_i x^i \in \mathbb{F}_p[x]$. The polynomial has degree d if $d > 0$ and $a_d \neq 0$. A non-zero constant polynomial is said to have degree 0. The zero polynomial is said to have degree -1.

Lemma 3.2 *Let $f \in \mathbb{F}_p[x]$ be some polynomial and m a positive integer satisfying $\gcd(m, p-1) = 1$. Then for any constant $0 \neq c \in \mathbb{F}_p$ the polynomial $\gcd(f(x),\, x^m - c^m)$ has odd degree if and only if x is a root of $f(x)$.*

Proof We know(see [5]) that when $c \neq 0$:

$$x^m - c^m = \prod_{d|m} c^{\varphi(d)}\ \Phi_d(\frac{x}{c}) \pmod p$$

where $\Phi_d(x)$ is the d'th cyclotomic polynomial. It's degree is $\varphi(d)$ and it is irreducible over $\mathbb{F}_p$. Observe that $\varphi(d)$ is even for all odd integers $d > 1$. Since m is odd all its divisors are odd and hence all irreducible factors of $x^m - c^m$ except $x - c$ have even degree. It follows that if c is not a root of $f(x)$ then $x - c$ does not divide the gcd implying that the gcd must have even degree. Conversely, if c is a root of $f(x)$ then $x - c$ does divide the gcd and hence its degree must be odd. □

Corollary 3.3 *Let $m \in \mathbb{Z}$ be a positive integer and let $N \in \mathbb{Z}_{(2)}(n)$ satisfy $\gcd(\varphi(N), m) = 1$. Let $f \in \mathbb{Z}_N[x]$ be a polynomial and let f_p, f_q be the reduction of f modulo p and q respectively where $N = pq$. Then for any constant $c \in \mathbb{Z}_N^* \cup \{0\}$ if $f(c)$ is a non-zero non-invertible element of $\mathbb{Z}_N$ then*

$$\deg(\gcd(f_p,\, x^m - c^m)) \neq \deg(\gcd(f_q,\, x^m - c^m))$$

Proof Since $f(c)$ is non-zero non-invertible we know that c is a root of f modulo exactly one of the primes p, q. When $c = 0$ the corollary is trivial. When $c \in \mathbb{Z}_N^*$ the previous lemma implies that one gcd has odd degree while the other has even degree. □

The above corollary shows that if $f \in \mathbb{Z}_N[x]$ is a polynomial such that $f(c)$ is non-zero non-invertible element of $\mathbb{Z}_N$ then $\gcd(f,\, x^m - c^m)$ behaves modulo p differently than it does modulo q. The difference in behavior enables one to factor N (simply apply Euclid's algorithm in $\mathbb{Z}_N$ to f and $x^m - c^m$). Thus, the corollary shows that if $f(c)$ reveals the factorization of N then one can factor N given only $c^m \bmod N$ (and f).

Proof of Theorem 3.1 Given an integer $N \in \mathbb{Z}_{(2)}(n)$ algorithm $\mathcal{B}$ factors it by first running algorithm $\mathcal{A}$ to produce k RSA–SLP's $P_1, \ldots, P_k$. We know that when evaluated modulo N (using the RSA breaking oracle) one of these programs produces a non-zero non-invertible element of $\mathbb{Z}_N$. Call this program P. We show how algorithm $\mathcal{B}$ can use the program P to generate a non-zero non-invertible element without using an RSA breaking oracle. Note that since $\mathcal{B}$ does not know which of the k programs is the right one, it tries them all.

To emphasize the steps in which P uses the RSA breaking oracle we write P as follows:

$$\begin{aligned}
\alpha_1 &= \sqrt[e_1]{f_0(1)} \\
\alpha_2 &= \sqrt[e_2]{f_1(\alpha_1)} \\
\alpha_3 &= \sqrt[e_3]{f_2(\alpha_2,\alpha_1)} \\
&\vdots \\
\alpha_r &= \sqrt[e_r]{f_{r-1}(\alpha_{r-1},\ldots,\alpha_1)} \\
\alpha_{r+1} &= f_r(\alpha_r,\ldots,\alpha_1)
\end{aligned}$$

where for all i, $\alpha_i \in \mathbb{Z}_N$ and α_{r+1} is non-zero non-invertible. The polynomials $f_0,\ldots,f_r$ all have straight line complexity smaller than the length of P. Note that the polynomial f_i may only depend on some of the α_j, $j < i$. Every line in the above list corresponds to one application of the RSA oracle. Recall that by assumption all the e_i are less than some absolute constant ω. Also, by assumption $r < O(\log T(n))$. We may assume $\alpha_r \in \mathbb{Z}_N \cup \{0\}$ since otherwise $\alpha_r^{e_r}$ is a non-zero non-invertible element of $\mathbb{Z}_N$ and the program may as well end there.

Consider the polynomial f_r as a polynomial in the variables x and $z_1,\ldots,z_{r-1}$. Setting $x = \alpha_r$ and $z_i = \alpha_i$ for $i = 1,\ldots,r-1$ causes f_r to evaluate to a non-zero non-invertible element of $\mathbb{Z}_N$. Let $g(x) = f_r(x,\alpha_{r-1},\ldots,\alpha_1) \in \mathbb{Z}_N[x]$. Then by Lemma 3.3, the degree of $\gcd(g, x^{e_r} - \alpha_r^{e_r})$ modulo p is different from its degree modulo q. We intend to apply Euclid's algorithm to $g(x)$ and $x^{e_r} - \alpha_r^{e_r}$ to reveal the factorization of N. The point is that $\alpha_r^{e_r}$ can now be expressed as a polynomial in $\alpha_1,\ldots,\alpha_{r-1}$.

Unfortunately at this point the values $\alpha_1,\ldots,\alpha_{r-1}$ are still unknown. So, we treat them as indeterminates $z_1,\ldots,z_{r-1}$. Working symbolically, we must apply Euclid's algorithm (with respect to x) to the polynomials f_r and $x^{e_r} - f_{r-1}$. We do so using Lemma 2.1. The lemma produces 2^m SLP's over $z_1,\ldots,z_{r-1}$ whose length is at most $\mathrm{len}(P)e_r^3$. The lemma guarantees that when evaluated at $z_1 = \alpha_1,\ldots,z_{r-1} = \alpha_{r-1}$ at least one of these programs must evaluate to a non-zero non-invertible element of $\mathbb{Z}_N$. Let P' be this program. Algorithm $\mathcal{B}$ does not know which is the right one and so it tries them all. Let $h(z_1,\ldots,z_{r-1})$ be the polynomial computed by P'. Then the following RSA–SLP factors N:

$$\begin{aligned}
\alpha_1 &= \sqrt[e_1]{f_0(1)} \\
\alpha_2 &= \sqrt[e_2]{f_1(\alpha_1)} \\
\alpha_3 &= \sqrt[e_3]{f_2(\alpha_2,\alpha_1)} \\
&\vdots \\
\alpha_{r-1} &= \sqrt[e_{r-1}]{f_{r-2}(\alpha_{r-2},\ldots,\alpha_1)} \\
\alpha_r &= h(\alpha_{r-1},\ldots,\alpha_1)
\end{aligned}$$

We obtained an RSA–SLP making one less oracle call than the original program. The total length of the RSA–SLP went up by at most ω^3 and it is one of $k2^\omega$ RSA–SLP's that algorithm $\mathcal{B}$ must evaluate. We can iterate this process of removing oracle calls until finally we obtain a collection of RSA–SLP's that never use radicals; they can all be evaluated without the use of an oracle. One of them yields the factorization of N. The total number of these SLP's is at most $k(2^\omega)^r$ and the length of each one is at most $\mathrm{len}(P)(\omega^3)^r$. Since ω is a constant, $r < O(\log T(n))$ and $\mathrm{len}(P) < T(n)$ the total running time of algorithm $\mathcal{B}$ is bounded by $T(n)^{O(1)}$. It factors all integers that algorithm $\mathcal{A}$ factors and makes no use of an oracle breaking LE–RSA. □

The result we just proved is a bit stronger than stated in the theorem. All the steps of the program P up until the first use of the RSA breaking oracle can be arbitrary. That is, our conversion process works even if $f_0(1)$ is computed using non-algebraic operations. This is an important observation since some factoring algorithms based on sieving fall into this category.

3.2 Removing an RSA oracle from an algebraic reduction

In this section we show how to convert a "factoring to RSA" reduction to a real factoring algorithm for a more general class of reductions. We refer to these as *algebraic reductions*. Unlike the straight line reductions of the previous section, algebraic reductions may include branches (decisions) based on values returned by the RSA oracle. Hence, algebraic reductions appear to be more general.

Definition 3.3 *An algebraic reduction $\mathcal{A}$ factors an element $N \in \mathbb{Z}_{(2)}(n)$ with the help of a special oracle $\mathcal{O}$. From time to time $\mathcal{A}$ stops and presents an* RSA–SLP *to $\mathcal{O}$. The oracle then says "yes" or "no" according on whether the* RSA–SLP *evaluates to zero in $\mathbb{Z}_N$. Eventually $\mathcal{A}$ stops and outputs a set of* RSA–SLP*'s $\{P_1, \ldots, P_k\}$ one of which factors N with probability at least $\frac{1}{2}$ (over the random bits of $\mathcal{A}$).*

If a polynomial time algebraic reduction exists then breaking low exponent RSA is as hard as factoring. As in the previous section we suggest that this is unlikely since an algebraic reduction (with a bounded number of oracle calls) can be converted into a real factoring algorithm.

Theorem 3.4 *Suppose there exists an algebraic factoring algorithm $\mathcal{A}$ whose running time is $T(n)$. Further suppose that each of the* RSA–SLP*'s generated by $\mathcal{A}$ on input $N \in \mathbb{Z}_{(2)}(n)$ contains at most $O(\log T(n))$ radical steps. Then there is a real factoring algorithm $\mathcal{B}$ whose running time is $T(n)^{O(1)}$ and factors all $N \in \mathbb{Z}_{(2)}(n)$ that $\mathcal{A}$ does.*

The difficulty here is in answering $\mathcal{A}$'s queries to the oracle $\mathcal{O}$. We show how given an RSA–SLP P it is possible to test if P evaluates to zero in $\mathbb{Z}_N$ *without* the help of an RSA breaking oracle. In the following lemma we use the same notion of gcd in $\mathbb{Z}_n[x]$ as the one discussed in the beginning of Section 2.1. The following lemma shows that to determine if $c \in \mathbb{Z}_N$ is a root of $f \in \mathbb{Z}_N[x]$ it suffices to observe the degree of $\gcd(f,\, x^m - c^m)$.

Lemma 3.5 *Let $m \in \mathbb{Z}$ be a positive integer and let $N \in \mathbb{Z}_{(2)}(n)$ satisfy $\gcd(\varphi(N), m) = 1$. Let $f \in \mathbb{Z}_N[x]$ be a polynomial. Let $c \in \mathbb{Z}_N$ be a value for which $h(x) = \gcd(f(x),\, x^m - c^m)$ is well defined. Then $c \neq 0$ is a root of $f(x)$ if and only if $h(x)$ has odd degree. $c = 0$ is a root of $f(x)$ if and only if the degree of $h(x)$ is greater than 0.*

Proof When $c = 0$ the lemma is trivial. Assume $c \neq 0$. Let $N = pq$ with p, q prime. Let f_p be the polynomial f reduced modulo p and f_q the polynomial reduced modulo q. If c is a root of f (in $\mathbb{Z}_N$) then it must also be a root of f_p and f_q. Hence, by Lemma 3.2, assuming $h \in Z_N[x]$ is well defined, its degree must be odd.

Conversely, suppose the degree of $h(x)$ is odd. Then, assuming the leading coefficient of h is invertible in $\mathbb{Z}_N$, the degree of $h(x)$ when reduced modulo p must be odd and the same holds modulo q. Hence, by Lemma 3.2, c must be a root of both f_p and f_q. It follows that c is a root of f in $\mathbb{Z}_N$. □

Proof of Theorem 3.4 On input $N \in \mathbb{Z}_{(2)}(n)$ algorithm $\mathcal{B}$ runs algorithm $\mathcal{A}$. If $\mathcal{B}$ could answer $\mathcal{A}$'s oracle queries correctly then eventually $\mathcal{A}$ will generate a set of RSA–SLP's $\{P_1, \ldots, P_k\}$ that factor N.

Algorithm $\mathcal{B}$ could then use the result of Theorem 3.1 to convert these RSA–SLP's into a real factoring algorithm (that makes no oracle calls). Hence, we must only show how $\mathcal{B}$ can answer $\mathcal{A}$'s oracle queries to $\mathcal{O}$.

At some point during the execution of $\mathcal{A}$ it outputs an RSA–SLP P and then stops and waits for an answer to whether P evaluates to zero in $\mathbb{Z}_N$. As before we write the program P in a way that emphasizes the oracle calls:

$$\begin{array}{rcl} \alpha_1 & = & \sqrt[e_1]{f_0(1)} \\ \alpha_2 & = & \sqrt[e_2]{f_1(\alpha_1)} \\ \alpha_3 & = & \sqrt[e_3]{f_2(\alpha_2, \alpha_1)} \\ & \vdots & \\ \alpha_r & = & \sqrt[e_r]{f_{r-1}(\alpha_{r-1}, \ldots, \alpha_1)} \\ \alpha_{r+1} & = & f_r(\alpha_r, \ldots, \alpha_1) \end{array}$$

P evaluates to α_{r+1} in $\mathbb{Z}_N$. We show how to test if $\alpha_{r+1} = 0$ in $\mathbb{Z}_N$ *without* the use of an oracle for breaking LE–RSA. Let $g(x) = f_r(x, \alpha_{r-1}, \ldots, \alpha_1) \in \mathbb{Z}_N[x]$. By Lemma 3.5 we know that if $\gcd(g,\, x^{e_r} - \alpha_r^{e_r})$ is well defined, its degree (in x) will tell us if $g(\alpha_r) = 0$. If the gcd is not well defined then Euclid's algorithm must have encountered a polynomial whose leading coefficient is not invertible in $\mathbb{Z}_N$ and hence the factorization of N is already revealed.

Since $\alpha_{r-1}, \ldots, \alpha_1$ are unknown at this point we treat them as indeterminates $z_1, \ldots, z_{r-1}$. The computation of $\gcd(g,\, x^{e_r} - \alpha_r^{e_r})$ reduces to computing the degree (in x) of

$$\gcd_x \Big(f_r(x, z_{r-1}, \ldots, z_1),\; x^{e_r} - f_{r-1}(z_{r-1}, \ldots, z_1) \Big)$$

Let $g_0 = x^{e_r} - f_{r-1}$. We construct a recursive algorithm for this problem as follows:

1. By Lemma 2.2 we can build an SLP for the coefficients of $g_1 = f_r \bmod g_0$. These coefficients are polynomials in the z's. Let $P_0, \ldots, P_m$ be the SLP's for these coefficients.

2. To determine the degree of g_1 we must determine the largest i for which $P_i(\alpha_{r-1}, \ldots, \alpha_1)$ is non-zero in $\mathbb{Z}_N$. This is a zero-testing problem like the one we are trying to solve except that the program P_i contains only $r-1$ oracle calls. Hence, we can recursively solve this question and determine the degree of g_1. Note that the length of the program P_i is at most ω^3 times the length of P.

3. To ensure that the gcd is well defined we must check that the leading coefficient of g_1 is invertible in $\mathbb{Z}_N$. To do this we apply Theorem 3.1 to the leading coefficient of g_1, namely to $P_i(\alpha_{r-1}, \ldots, \alpha_1)$. If the leading coefficient is not invertible, the factorization of N is found and algorithm $\mathcal{B}$ terminates.

4. Next we compute an SLP for the coefficients of $g_2 = g_0 \bmod g_1$. To avoid using division we actually compute g_2 multiplied by an invertible constant. We apply the same steps as before to determine the degree of g_2 and to ensure that its leading coefficient is invertible in $\mathbb{Z}_N$.

5. We iterate this procedure until Euclid's algorithm terminates. At which time we find the degree of x in $\gcd(g(x),\, x^{e_r} - \alpha_r^{e_r})$. By Lemma 3.5 the degree determines whether $g(\alpha_r) = 0$ in $\mathbb{Z}_N$.

The recursion depth is bounded by r, the number of oracle calls made by the RSA–SLP P. Hence, the total running time is $O(\text{len}(P)(\omega^3)^r) = T(n)^{O(1)}$ since $r = O(\log T(n))$ and $\text{len}(P) < T(n)$. $\square$

As in the case of Theorem 3.1 the theorem is slightly stronger than stated. In fact, we can allow all the RSA–SLP's produced by algorithm $\mathcal{A}$ to perform arbitrary operations (including aggressive bit manipulations) up until the first time the RSA oracle is invoked. Once the RSA oracle is used the programs must only use algebraic operations in $\mathbb{Z}_N$. The reason for this extra freedom is that it makes no difference how $f_0(1)$ is calculated. All that matters is that it is an element of $\mathbb{Z}_N$ which algorithm $\mathcal{B}$ can construct.

4 Conclusions and open problems

Our main objective is to study the relationship between breaking low exponent RSA and factoring integers. We show that under certain types of reductions (straight line reductions and algebraic reductions with bounded oracle queries) the two problems cannot be equivalent, unless factoring integers is easy.

Since our results may suggest that breaking LE–RSA is not as hard as factoring it is interesting to note that attacks on low public exponent RSA due to Hastad [4] and Coppersmith [2] break the RSA system (i.e. decrypt messages without the private key) but do not factor the modulus. In conjunction with our results this suggests further evidence that breaking LE–RSA is easier than factoring. It is important to keep in mind that even though it may be easier than factoring, breaking low exponent RSA is still most likely to be intractable.

We note that both our main results, Theorems 3.1 and 3.4, are a bit stronger than stated. In both cases the RSA–SLP's produced by the reductions are allowed to perform arbitrary operations (including aggressive bit manipulations) up until the first time the RSA oracle is invoked. Once the RSA oracle is used the programs must only use algebraic operations in $\mathbb{Z}_N$. Hence, for instance, before invoking the RSA oracle the reduction may perform arbitrary sieving. Our results are strong enough to convert such reductions into real factoring algorithms.

There are still several open problems that remain to be solved until we have a complete proof that RSA cannot be equivalent to factoring (unless factoring is easy). The first is to remove the restriction that the reduction must be algebraic. That is, given a factoring algorithm presented as a boolean circuit using RSA gates (i.e. gates breaking LE–RSA) convert it into a real factoring algorithm. This may be possible by first converting the boolean circuit into an arithmetic circuit (one using only arithmetic gates) using standard techniques and then applying our method to the resulting arithmetic circuit.

The second open problem is to strengthen our results regarding algebraic reductions. Currently our conversion process works only when the given RSA–SLP makes at most $O(\log T(n))$ RSA oracle queries, where $T(n)$ is the running time of the reduction algorithm[1]. If we assume factoring cannot be done in time $L_\epsilon(n)$ for some $\epsilon > 0$ then the reduction may take time $L_\epsilon(n)$ and consequently the RSA–SLP's it outputs may make n^ϵ oracle queries. It is an interesting open question to strengthen our results and allow the algebraic reduction to make unrestricted oracle calls.

As a final note we point out that our results apply to smooth public exponents in some limited

[1]Since the reduction algorithm outputs a number of RSA–SLP's, the total number of oracle queries is unbounded. The only restriction is that each RSA–SLP make at most $O(\log T(n))$ queries.

sense. When e is smooth, an oracle for taking e'th roots can be simulated using $\log e$ LE–RSA oracle calls. Hence, as long as e is smooth and $e < n^{\epsilon}$ our conversion process can be applied. Consequently, any algebraic reduction using a *costant* number of e'th root oracle calls can be converted into a real factoring algorithm. In this restricted sense, our results apply to more than just low exponent RSA.

References

[1] D. Boneh, R. Lipton, "Black box fields and their application to cryptography", Proc. of Crypto '96, pp. 283–297.

[2] D. Coppersmith, "Finding a small root of a univariate modular equation", Proc. of Eurocrypt '96, pp. 155–165.

[3] W. Diffie, M. Hellman, "New directions in cryptography", IEEE Transactions on Information Theory, vol. 22, no. 6, pp. 644–654, 1976.

[4] J. Hastad, "Solving simultaneous modular equations of low degree", SIAM Journal of Computing, vol. 17, pp 336–341, 1988.

[5] S. Lang, "Algebra", Addison-Wesley, 1993.

[6] A. Lenstra, H.W. Lenstra, "Algorithms in Number Theory", Handbook of Theoretical Computer Science (Volume A: Algorithms and Complexity), Elsevier and MIT Press, Ch. 12, pp. 673–715, 1990.

[7] H. W. Lenstra, "Factoring integers with elliptic curves", Annals of Math., Vol. 126, pp. 649–673, 1987.

[8] U. Maurer, "Towards proving the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms", Proc. of Crypto '94, pp. 271–281.

[9] U. Maurer, S. Wolf, "Diffie-Hellman oracles", Proc. of Crypto '96, pp. 268–282.

[10] R. Rivest, A. Shamir, L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", Communications of the ACM, vol. 21, pp. 120–126, 1978.

Appendix

Proof of Lemma 2.2 First, observe that since g is monic (as a polynomial in x) the gcd is well defined in $\mathbb{Z}_N$. Let P_f be the SLP for f. We prove the lemma by induction on L, the length of P_f. When $L = 1$ the claim is trivial. We assume the claim for $L - 1$ and prove for a program of length L. Suppose the last step of f applies one of $\{+, -, *\}$ to the i'th and j'th steps. By induction, there exist SLP's P_i and P_j for the m coefficients of $f_i \bmod g$ and $f_j \bmod g$ respectively (the last m steps of the program P_i are the coefficients of $f_i \bmod g$ and a similar condition holds for P_j). If the last step of P_f takes the sum (or difference) of f_i and f_j then P'_f (the program for the coefficient of $f \bmod g$) includes the programs for P_i and P_j and adds m more steps adding (or subtracting) the m last steps of P_i to those of P_j. By induction, the length of the combined programs for P_i and P_j is at most $2m^2(L-1)$. Hence, the total length for P'_f is $2m^2(L-1) + m < 2m^2 L$.

If the last step takes the product of f_i and f_j then as before the program P'_f includes P_i and P_j and adds a number of steps to that. First P'_f computes all $2m$ coefficients of the product of $f_i \bmod g$ and $f_j \bmod g$. Then using the simplest division algorithm it reduces the product modulo $g = x^m - h$. Since $x^m - h$ is monic this step does not require any divisions. The last m steps of the reduction contain the desired m coefficients of $f \bmod g$. This procedure adds at most $2m^2$ steps. The total length is now less than $2m^2(L-1) + 2m^2 = 2m^2 L$ as required. $\square$