On Factoring Integers and Evaluating Discrete Logarithms

A thesis presented by

JOHN AARON GREGG

to the departments of
Mathematics and Computer Science

in partial fulfillment of the honors requirements
for the degree of Bachelor of Arts

Harvard College
Cambridge, Massachusetts

May 10, 2003

ABSTRACT

We present a survey of the state of two problems in mathematics and computer science: factoring integers and solving discrete logarithms. Included are applications in cryptography, a discussion of algorithms which solve the problems and the connections between these algorithms, and an analysis of the theoretical relationship between these problems and their cousins among hard problems of number theory, including a new randomized reduction from factoring to composite discrete log. In conclusion we consider several alternative models of complexity and investigate the problems and their relationship in those models.

Acknowledgements

Many thanks to William Stein and Salil Vadhan for their valuable comments, and Kathy Paur for kindly reviewing a last-minute draft.

Contents

1. Introduction

Problem 1.1 (Factoring) *Given a positive composite integer N, to find an integer x, with $1 < x < N$, such that x divides N.*

Problem 1.2 (Discrete Logarithm) *Given a prime integer p, a generator g of $(\mathbf{Z}/p\mathbf{Z})^*$, and an element $y \in (\mathbf{Z}/p\mathbf{Z})^*$, to find an integer a such that $g^a = y$.*

In the recent history of applied mathematics and computer science, the two problems above have attracted substantial attention; in particular many have assumed that solving them is sufficiently difficult to base security upon that difficulty. This paper will analyze these two relevant problems and consider the relationships between them.

The first is the problem of finding the prime factorization of an integer N, considered particularly in the most difficult and relevant case where $N = p \cdot q$ for large primes p and q. The second is the discrete logarithm problem (or just "discrete log"), to find, given an element y of a ring $(\mathbf{Z}/p\mathbf{Z})^*$ constructed by raising a generator g to a secret power a (that is, $y = g^a \bmod p$), the logarithm a.

Both problems are challenges of inversion. New problem instances are trivial to create easily, by the easy tasks of multiplying integers or modular exponentiation respectively, but neither of these tasks has yet admitted an efficient method of being reversed, and this property has led to the recent interest in these problems.

From a pure mathematical perspective, neither problem is impossible to solve definitely in a finite amount of time (and such problems certainly exist, e.g., the halting problem of computational theory or Hilbert's 10th problem—finding integer solutions to diophantine equations). Both factoring and solving a discrete logarithm can be accomplished with a finite search, through the $\sqrt{N}$ possible divisors and the $p - 1$ possible exponents respectively.

However, in the real world, such solutions are unacceptably inefficient, as the number of algorithmic steps required to carry them out is exponential in the size of the problem. We mean this as follows: to write down the integer N in binary takes $\log_2 N$ bits, so we say that the size of N is $n = \log_2 N$. To find a factor of N by the trivial method described already will take $N^{1/2}$ trial divisions, or on the order $(2^n)^{1/2} = (\sqrt{2})^n$ steps, which is exponential in n, the size of the problem.

The research I will consider in this thesis is on the efforts to improve upon these solutions. The ultimate goal of such efforts would be to find a solution which runs in time polynomial in n, but as of yet no such solutions have been discovered, and much of cryptography is based on the assumption that no polynomial time solutions exist.

1.1 Notation

Because we have a mixed audience of mathematicians and computer scientists, it will be worth a few extra sentences about some notation conventions and some facts we will assume without proof.

- When in the context of algorithm input or output, the symbol 1^k (sim. 0^k) represents a string of k 1's (0's) over the alphabet $\{0, 1\}$. It does not indicate a k-fold multiplication.
- Vertical bars $| \cdot |$ are generally abused herein, and will have one of the following meanings determined by context. If X is a set, then $|X|$ is the number of elements in X. If x is a bit string, or some other object encoded as a string, then $|x|$ is the length of the string in bits; specifically if N is an integer encoded as a bit string, then $|N| = \lceil \log_2 N \rceil$. If a is a real number not in the context of string encoding, then $|a|$ is the absolute value of a (this is used rarely).
- We also will be liberal about our use of the symbol $\leftarrow$. In the context of an algorithm, it is an assignment operator. Thus the statement $x \leftarrow x + 1$ means "increment x by 1" as an instruction. If S is a set, we write $x \overset{\mathrm{R}}{\longleftarrow} S$ or sometimes just $x \leftarrow S$ to say that x is an element of S chosen uniformly at random. If $\mathcal{M}$ is a distribution (that is, a set S and a probability measure $\mu : S \to [0, 1]$ such that $\sum_{s \in S} \mu(s) = 1$), then $x \leftarrow \mathcal{M}$ means to choose x out of S according to $\mathcal{M}$ so that the probability for any particular $s \in S$ that $x = s$ is $\mu(s)$. When we wish to present $\mathcal{M}$ explicitly, we will often do it by presenting a randomized algorithm which chooses an element; the set S and the measure μ is then implicitly defined by the random choices of the algorithm.
- For any integer N, $\mathbf{Z}/N\mathbf{Z}$ (the integers modulo N) is the set of equivalence classes of the integers under the equivalence relation $a \sim b \iff N \mid a - b$. $(\mathbf{Z}/N\mathbf{Z})^*$ is the group of units of $\mathbf{Z}/N\mathbf{Z}$; equivalently, the group of elements of $\mathbf{Z}/N\mathbf{Z}$ with multiplicative inverses; equivalently, the set $\{a \in \mathbf{Z} : \gcd(a, N) = 1\}$ modulo the above equivalence relation.
- $\pi(N)$ is the number of positive primes $p \leq N$. We know $\pi(N) \sim N/\log N$ for large N.
- $\phi(N)$ is the Euler phi-function (or totient function), whose value is the number of integers $0 < a < N$ relatively prime to N, i.e., $\phi(N) = |(\mathbf{Z}/N\mathbf{Z})^*|$. For p prime $\phi(p) = p - 1$; for p, q relatively prime $\phi(pq) = \phi(p)\phi(q)$; and it is known that $\phi(N) \geq N/6 \log \log N$ for sufficiently large N.

1.2 Computational Complexity Theory

The broader goal of this paper is to consider (a small aspect of) the question *how much "hardness" exists in mathematics?* Over the past decades we have built real-world systems, several of them to be discussed later, which rely on the hardness of mathematical problems. Are these hardnesses independent of one another, or are they simply different manifestations of a single overriding hard problem? If an efficient factoring algorithm were found tomorrow, how much of cryptography would have to be thrown out? Only those cryptosystems which rely specifically on the intractability of factoring, or others as well?

One goal of complexity theory is to think about ways to sort mathematical and computation problems into classes of difficulty, and thus take steps towards understanding the nature of this hardness. In order to provide a backdrop for the discussion of our two problems, we first present a brief introduction to the basics of complexity theory.

Though some would strongly object in general, for our purposes the specific model of computation is not particularly important. To be perfectly specific, we would want to build up the theory of computation and Turing machines to provide a very rigorous definition of "algorithm," and to some degree we will do so here in order to support specific results which are particularly relevant to the main thread of this paper.

For our purposes, an *algorithm* is any finite sequence of instructions, or operations, which may take finite input and produce finite output. In different settings, we will both describe the operations of a given algorithm explicitly, and implicitly consider such an algorithm. The complexity of a given algorithm is the number of operations performed as a function of the length of the input expressed as a string in some alphabet. For brevity of description, we often specify the input of an algorithm as a member of some arbitrary set, with the implicit understanding that such input can be encoded as a string over the binary alphabet $\{0, 1\}$, and over this alphabet is its length considered. This understanding is intuitively unproblematic for any countable input set we might want.

We also will sometimes express the output of an algorithm as an element of an arbitrary set, and we do so with the implicit understanding that forcing algorithms to output a single bit $\{0, 1\}$ is sufficient to express all algorithms of finite output—to output n bits, specify an n-tuple of algorithms each providing a bit of the output. For the initial stages of developing this theory, we will consider such algorithms which output only single bits.

Definition 1.3 *A* **language** *is a set of strings over the alphabet* $\{0, 1\}$.

As discussed, most any interesting collection of mathematical objects can be considered to be a language.

Definition 1.4 *A language L is* **decided** *by an algorithm A if A outputs 1 on input $x \in L$ and A outputs 0 on input $x \notin L$.*

Definition 1.5 *An algorithm A is* **polynomially bounded** *if there exists a polynomial function $p(x)$ such that when A is run on input x it outputs a value after no more than $p(|x|)$ operations. Recall that $|x|$ denotes the length of x in bits.*

In this paper we apply the word "efficient" to algorithms with the same meaning as "polynomially bounded." We also use the word "feasible" to describe a problem for which there exists an efficient algorithm solving it.

For the sake of example, and to facilitate discussion of several results and relevant applications of the discrete log problem, we present a useful and efficient number theoretic algorithm for a problem which at first glance can appear daunting.

Proposition 1.6 *There exists a polynomial-time algorithm to compute the modular exponentiation* $a^m \bmod n$*, where* $a, m, n \in \mathbf{Z}$.

Proof. Consider the following algorithm:

```
ModExp($a, m, n$):
  1. Write $m$ in binary as $m = B_\ell B_{\ell-1} \cdots B_1 B_0$.
  2. Set $M_0 = m$, and for each $i = 1, \ldots, \ell$, let $M_i = M_{i-1}^2 \bmod n$.
  3. Output $\prod_{i:B_i=1} M_i$ (working mod $n$).
```

The successive squaring requires $\ell = \log_2(m)$ multiplications, and the final multiplication of the powers corresponding to the on bits of m requires at most ℓ more multiplications. Therefore the algorithm is clearly efficient, with operations bounded by a polynomial in $|m| < |(a, m, n)|$. ■

Note that without working modulo n, there is *no* efficient algorithm for computing exponentiation in general—just giving an algorithm to write down 2^n (which has $\log 2^n = n$ binary digits) would require at least n steps, which is not polynomially in the length of the input.

The collection of languages which are decided by efficient algorithms earns substantial attention from theoretical computer scientists, and is given the name $\mathcal{P}$.

Definition 1.7 *A language is* **in $\mathcal{P}$** *if it is decided by a polynomially bounded algorithm.*

In addition to $\mathcal{P}$, there is another common class of decidable languages often studied by computer scientists, called $\mathcal{NP}$. There are several equivalent ways of expressing membership in $\mathcal{NP}$, but the key feature is that $\mathcal{NP}$ contains languages decided by *non-deterministic* algorithms. But what is such a thing? So far we have only allowed algorithms to perform specified operations; now we must also allow algorithms to make random choices. There are several ways to formulate this, for our purposes we will give the algorithm access to an "oracle" which flips a fair coin; A thus has a source of truly random bits which we assume to be entirely separate from A.

To be able to think of A's operations in a functional sense, we denote $A(x; r)$ as the output of A on input x if the random oracle gives it the sequence r of coin tosses. We require that the algorithm terminate after a finite number of steps regardless of the random choices, and we say that such an algorithm A decides a language if for every $x \in L$, there is some sequence r such that $A(x; r) = 1$ and for every $x \notin L$, for all r we have $A(x; r) = 0$. The notation $A(x)$ thus stands for a random variable over the probability space of possible r, or "over the coin tosses of A."

Definition 1.8 *A language is* **in $\mathcal{NP}$** *if it is decided by a polynomially bounded nondeterministic algorithm.*

Though the above is standard, we give another equivalent characterization that will be more helpful for our purposes, that $\mathcal{NP}$ is the class of languages which have concise, easily verifiable proofs of membership. That is to say $L \in \mathcal{NP}$ if there exists another language $W \in \mathcal{P}$ and a polynomial p such that

$$x \in L \iff \text{there exists } (x, w) \in W \text{ with } |w| < p(|x|)$$

The element w is called the witness, or proof. Its conciseness is captured in the requirement $|w| < p(|x|)$, so that the length of the proof is bounded by a polynomial in the length of the thing being proved, and the easy verification is captured in the condition that $W \in \mathcal{P}$.

Problem 1.9 *Does $\mathcal{P} = \mathcal{NP}$?*

This open problem remains one of the central questions of theoretical computer science. Of course, the answer is widely believed to be no. The crux of the equation lies in the so-called $\mathcal{NP}$-complete problems, $\mathcal{NP}$ problems to which all other problems in $\mathcal{NP}$ reduce. If *any* of these problems were shown to be in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$ would follow. We can thus identify $\mathcal{NP}$-completeness as a single independent "hardness," and the many $\mathcal{NP}$-complete problems as non-independent manifestations of it.

Returning to the problems at hand—as already stated there is no known polynomial time algorithm for factoring integers, so we do not know whether the problem is in $\mathcal{P}$. Also, factoring integers has neither been shown nor "disshown" to be $\mathcal{NP}$-complete. However, while it is not really a language, it can be considered an $\mathcal{NP}$ problem, with the following characterization:

Proposition 1.10 *If $\mathcal{P} = \mathcal{NP}$, then there exists a polynomial-time algorithm for factoring integers.*

Proof . We begin with a lemma. I claim that the language

$$L = \{(N, x) : N \text{ has a positive proper divisor less than } x\}$$

is in $\mathcal{NP}$.

For now, suppose this lemma is true, and that $\mathcal{P} = \mathcal{NP}$. Then $L \in \mathcal{P}$, so let A be a polynomial-time algorithm which decides it. We can find a factor of N by doing a binary search using L:

```
F(N):
  1. Initialize $B_L = 0, B_U = \lfloor\sqrt{N}\rfloor$
  2. Repeat the following until $B_L = B_U$:
       • Let $X = \lceil(B_L + B_U)/2\rceil$.
       • Let $b = A(N, X)$.
       • If $b = 1$, set $B_U = X$.
       • If $b = 0$, set $B_L = X - 1$.
  3. Output $B_L$.
```

Since the search interval is cut in half on each iteration, we expect the total number of iterations to be on the order of $\log_2(N^{1/2}) = 1/2 \log_2(N)$, and so the total running time should be on the order of $\log_2(N) \cdot T(A(N))$, where T is the running time of A. But we know that $T(A)$ is polynomial in $|N| = O(\log_2(N))$, therefore F is polynomial-time as well.

We must now only prove our claim that $L \in \mathcal{NP}$. The witness language is quite easy to find, since the proof of the existence of a divisor is the divisor itself. Thus

$$W = \{((N, x), y) : 1 < y < x \text{ and } y \text{ divides } N\}.$$

Clearly $W \in \mathcal{P}$, since trial division, or even checking that $\gcd(y, N) = y$ would be very efficient. The size of the witness $w = ((N, x), y)$ is no more than twice $|(N, x)|$, so the proof is certainly concise, and the definitions easily lead to the condition $\exists y[((N, x), y) \in W] \iff (N, x) \in L$. ■

We can produce an analogous result for our second problem, discrete logarithm, showing that this problem shares the same realm of $\mathcal{P}$ vs. $\mathcal{NP}$ complexity with factoring.

Proposition 1.11 *If $\mathcal{P} = \mathcal{NP}$, then there exists a polynomial-time algorithm for evaluating discrete logarithms over $(\mathbf{Z}/p\mathbf{Z})^*$ as defined in Problem 1.2.*

Proof . As above, we begin by proving that a language we would like to use for binary searching is in $\mathcal{NP}$. Let this language be

$$L = \{(p, g, g_1, x) : \text{there exists a non-negative integer } y < x \text{ such that } g^y \equiv g_1 \bmod p\}.$$

We can give an easy language of proofs in the same way as above.

$$W = \{((p, g, g_1, x), y) : \ 0 \leq y < x \text{ and } g^y \equiv g_1 \bmod p\}.$$

The fact that $W \in \mathcal{P}$ follows immediately from the fact that modular exponentiation is efficient (Proposition 1.6); the length of an element of W is less than twice the length of an element of L, so the proofs are concise; lastly the definitions give the equivalence $\exists(x, y) \in W \iff x \in L$.

Since $L \in \mathcal{NP}$, the hypothesis $\mathcal{P} = \mathcal{NP}$ would again yield a binary search algorithm over all possible exponents in the range $0, \ldots, p - 1$, and checking L at each stage would be efficient, yielding an efficient solution of the discrete log problem. ■

We have therefore succeeded in placing our two problems in the same general difficulty zone—harder (so far as we know) than being in $\mathcal{P}$, but efficient in the case that $\mathcal{P} = \mathcal{NP}$. Certainly no equivalence between the two problems follows from this, but as in our binary search algorithms, we have helpfully narrowed the bounds.

1.3 Modes of Complexity and Reductions

At times we perceive the complexity of an algorithm in different senses: worst-case complexity, a question of how long we will have to wait to for our algorithm to halt no matter what input we give it; and average-case complexity, a question of how long we expect our algorithm to take on a random input, taking into account a distribution over the possible inputs.

The complexity notion which figures into the distinction between $\mathcal{P}$ and $\mathcal{NP}$ is the former, worst-case complexity. For example, the factoring problem is not “in $\mathcal{P}$” (quotes because it is not a language) only because there exist *some* large products pq which are hard to factor. But consider a random integer x—with probability $1/2$ we will have 2 a factor of x, and with probability $2/3$ x has either 2 or 3 as a factor, etc. Therefore, in the *average* case over all composite N, factoring is not hard.

Similarly, we will often consider reductions between problems. That is, we will make the statement that a solution for problem X implies a solution to problem Y, or equivalently, problem Y *reduces* to problem X. This statement also has several interpretations. One, analogous to the “worst-case” complexity notion, is that if we have a polynomial-time algorithm A which always solves problem X on any input, then we can construct an algorithm B which, given the ability to call A on any input it chooses (we often say B has “oracle access to A”), can solve problem Y on any input.

Just as average-case complexity is more relevant to the applications of factoring and discrete log, for these problems and their cryptographic relatives we are often more interested in an “average-case,” or a *probabilistic* reduction. That is, we suppose that we have an algorithm A which solves problem X with probability ε—taken over some distribution of problem instances and the randomness of A. Then we wish to construct an algorithm B which, given oracle access to A, can solve problem Y with probability δ, where δ is polynomially related to ε.

We must therefore be more precise about how we want to consider the difficulty of factoring. An important first step is to establish a distribution of problem instances. We will do this by defining the distribution $\mathcal{F}_k$ of k-bit factoring instances for any k to be the random output of the following algorithm $\mathcal{F}$ on input 1^k.

$\mathcal{F}(1^k)$:

- Select two k-bit primes p and q at random.
- Output $N = p \cdot q$.

Example. For example if $k = 3$, then the only k-bit primes are $5 = 101_2$ and $7 = 111_2$. It follows that $\mathcal{F}_k$ has the three possible outputs $5 \cdot 5 = 25$, $5 \cdot 7 = 7 \cdot 5 = 35$, and $7 \cdot 7 = 49$, which occur with probability $1/4$, $1/2$, and $1/4$ respectively. Naturally, for larger k the size of the distribution space is substantially larger. ◇

We now give an average-case version of the factoring problem:

Problem 1.12 (Factoring with probability ε) *Given a function $\varepsilon = \varepsilon(k)$, to give a probabilistic polynomial-time algorithm A such that for all k*

$$\Pr[A(X) \textit{ is a factor of } X] \geq \varepsilon(k),$$

where the probability is taken over all $X \xleftarrow{\mathrm{R}} \mathcal{F}_k$ and the coin tosses of A.

We can construct a parallel definition for the discrete log problem, defining the following instance generator. We let $\mathcal{D}_k = \mathcal{D}(1^k)$ be the random output of the following algorithm on input 1^k.

$\mathcal{D}(1^k)$:

- Select a k-bit prime p at random.
- Select at random a generator g of $(\mathbf{Z}/p\mathbf{Z})^*$.
- Select a random $a \in \mathbf{Z}/(p-1)\mathbf{Z}$, and calculate $y = g^a \bmod p$.
- Output (p, g, y).

The corresponding probabilistic problem can be formulated in terms of this distribution:

Problem 1.13 (Solving discrete logarithm with probability ε) *Given a function $\varepsilon = \varepsilon(k)$, to give a probabilistic polynomial-time algorithm A such that for all k*

$$\Pr[g^{A(p,g,y)} \bmod p = y] \geq \varepsilon(k),$$

where the probability is taken over all $(p, g, y) \leftarrow \mathcal{D}_k$ and the coin tosses of A.

1.4 Elliptic Curves

Elliptic curves appear in most of the sections of this paper in one form or another, both in the creation of and in the attacks upon the cryptographic applications we demonstrate for factoring and discrete log. Indeed, elliptic curves find interactions with both problems, and so here we present a minimal development of the basics of elliptic curves. For a more thorough treatment, consider any of the books by Silverman and Tate: [47], [48], [49].

We consider here only elliptic curves over finite fields, for example over the field $\mathbf{F}_p$ for p prime. Such an elliptic curve is defined by two field elements a, b which are used as coefficients in the equation $y^2 = x^3 + ax + b$, such that the discriminant $4a^3 + 27b^2 \neq 0$. We denote[1] this curve as $E_{a,b}$, or when not ambiguous simply E, and we define the set of points on the curve over a field K by $E(K)$. For reasons arising in the algebraic geometry used to construct these curves formally, we define this set of points as a subset of the projective plane $\mathbf{P}^2(K)$ over the field, which consists of equivalence classes of non-zero ordered triples $(x, y, z) \in K^3$, with two triples equivalent if one is a constant multiple of another. The equivalence class of (x, y, z) is denoted $(x : y : z)$. We then define

$$E(K) = \{(x : y : z) \in \mathbf{P}^2(K) : y^2 z = x^3 + axz^2 + bz^3\}. \tag{1.1}$$

The only point on E with $z \neq 1$ is the point at infinity, $(0 : 1 : 0)$, denoted by $\mathcal{O}$, which satisfies the elliptic curve equation for all a and b. The point $\mathcal{O}$ has an important role when we note that

[1]The notation is in accordance with Lenstra's paper [25], and later (section 5.3.1), Maurer's paper [27].

$E(K)$ has an abelian group structure, with additive identity $\mathcal{O}$. This claim specifies the group law completely when $z \neq 1$, for the other elements we can consider the normal picture of E as a curve in the x, y-plane and define the group law geometrically as follows: for two points P and Q on E, draw the line connecting them and find the third point where this line meets E.

Define $-(P+Q)$ to be this point, and then $(P+Q)$ is the reflection in the x-axis of this third point. This addition can be carried out efficiently (say, by a computer), with the following algebraic manifestation. If $P = (x_1 : y_1 : 1)$ and $Q = (x_2 : y_2 : 1)$, let $m = (y_2 - y_1)/(x_2 - x_1)$ if $P \neq Q$ and $m = (3x_1^2 + a)/2y_1$ if $P = Q$. (m is the slope of the line between the points); let $n = y_1 = mx_1$. We then define $P + Q$ as the point $R = (x_3 : y_3 : 1)$ with $x_3 = m^2 - x_1 - x_2$ and $y_3 = -(mx_3 + n)$. This can be seen geometrically in Figure 1.1.

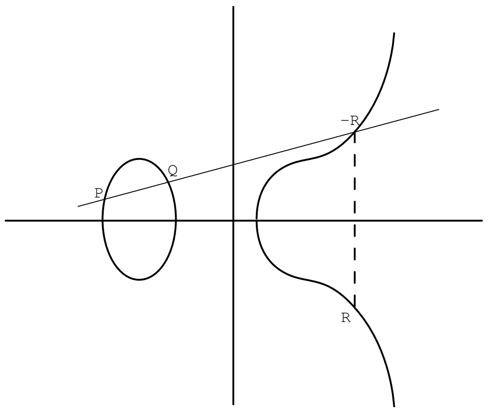


Figure 1.1: Adding points: $P + Q = R$.

We will also need to consider elliptic curves over $\mathbf{Z}/N\mathbf{Z}$ instead of $\mathbf{F}_p$, even though we do not have a field. We can construct an analogous domain $\mathbf{P}^2(\mathbf{Z}/N\mathbf{Z})$ as the set of orbit of

$$\{(x, y, z) \in (\mathbf{Z}/N\mathbf{Z})^3 : x, y, z \text{ generate the unit ideal of } \mathbf{Z}/N\mathbf{Z}\}$$

under the action of $(\mathbf{Z}/N\mathbf{Z})^*$ by $u(x, y, z) \mapsto (ux, uy, uz)$. As before, we denote the orbit of (x, y, z) by $(x : y : z)$. We can then define $E_{a,b}(\mathbf{Z}/N\mathbf{Z})$ exactly as in equation (1.1) by replacing K with $\mathbf{Z}/N\mathbf{Z}$. The group structure will hold provided that the discriminant $6(4a^3 + 27b^2)$ is relatively prime to N; only in that case to we actually call $E = E_{a,b}$ an *elliptic curve*. We leave until section 3.3 a more thorough discussion of addition of points on such a curve, since the fact that N is not prime raises complications which Hendrik Lenstra [25] demonstrated can be exploited to factor N. Furthermore, in section 5.3.1, we show how Ueli Maurer [27] used Lenstra's technique to bound the complexity of factoring in a specific complexity model.

2. Applications: Cryptography

As already discussed, these problems are only interesting from a real-world perspective. Solving them mathematically is not difficult in the strongest sense, since algorithms exist to do just that. Our interest is in measuring the degree of difficulty of implementing such solutions, and for the security of the cryptographic applications presented in this section we rely on the assumption that this degree is very high.

There are two standard ways to present cryptography: the first is to demonstrate independent secure protocols, and the second is to establish definitions of secure cryptographic primitives and then work towards creating specific objects which satisfy the definitions. Both methods make broad reliance at times on the assumptions that factoring and/or discrete log are difficult.

2.1 Cryptographic Protocols

We begin with public-key encryption. Though the idea of public-key cryptography is relatively recent, the idea of encryption has been around for centuries, and is the canonical task of cryptography, though not the only one.

Definition 2.1 *A* **public-key encryption scheme** *consists of three algorithms:*

1. *a randomized key generation algorithm* $\mathrm{Gen}(1^k)$ *which takes as input the number* k *encoded in unary as a* k*-bit string of all 1s, and produces a pair of keys* PK*, which is made public, and* SK *which is kept secret;*
2. *a randomized encryption algorithm* Enc *which, given the public key* PK *and a message* m *produces a ciphertext* c*;*
3. *a deterministic decryption algorithm* Dec *which, given the secret key* SK *and a ciphertext* c *returns the original message* m*.*

Attached to the scheme is a message space $\mathcal{M}$*, which may be allowed to vary according to the public key* PK*. To be a correct encryption scheme, we require that* $\mathrm{Dec}_{SK}(\mathrm{Enc}_{PK}(m)) = m$ *for all* $m \in \mathcal{M}$ *and for all pairs* (PK, SK) *which can be generated by the key generation algorithm.*

We can think of the key generation algorithm, which takes the 1^k argument (called the *security parameter*), as analogous to the instance generators discussed in the preceding chapter for hard problems. To feel secure in these encryption schemes, we want to make them unbreakable for the *average* instance (or key), and not just for some particularly difficult keys. Therefore the following definitions can serve for our model of security.

Definition 2.2 (Breaking an encryption scheme with probability ε) *Given a function $\varepsilon(k)$, a probabilistic polynomial-time algorithm A with single-bit output breaks a public-key encryption scheme $(\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$ with probability ε if for any k, and any two messages $m_0, m_1 \in \mathcal{M}$*

$$\Big| \Pr[A(PK, \mathrm{Enc}_{PK}(m_0)) = 1] - \Pr[A(PK, \mathrm{Enc}_{PK}(m_1)) = 1] \Big| \geq \varepsilon(k)$$

the probability taken over all $(PK, SK) \leftarrow \mathrm{Gen}(1^k)$ and the coin tosses of A.

Intuitively, the definition connects breaking the scheme to the goal of *distinguishing* between two different messages. We can think of the goal of A as to output 0 or 1 indicating that it thinks it sees an encryption of m_0 or m_1 respectively (although the definition is stronger, and actually allows A to attempt to guess any Boolean function of m_0 and m_1).

Note that while A must be able to distinguish over a random choice of the key, it may select the messages which will be distinguished ahead of time—this models the fact that an adversary may have external information about the distribution on the message space, and may even know that the secret message is one of only two possible values.

Definition 2.3 (Negligible function) *A function $\delta : \mathbf{Z} \rightarrow [0, 1]$ is* **negligible** *if, for any polynomial $p(x)$, there exists a k_0 such that for $k > k_0$, $\delta(k) < 1/p(k)$. That is, δ goes to 0 faster than $p(x)^{-1}$ for any polynomial p.*

The most common candidate is $\varepsilon(k) = 2^{-k}$, which is clearly negligible.

Definition 2.4 (Security) *A public key cryptosystem is* **secure** *if there does not exist a probabilistic polynomial-time algorithm which breaks it with probability $\varepsilon(k)$, for ε a non-negligible function.*

The definition of security as negligible indistinguishability is actually very strong compared to other possible definitions; in general cryptography tends toward such conservative positions. For the purposes of relating the protocols to factoring and discrete log, we will be satisfied with weaker and more intuitive properties, such as

- An encryption scheme is *insecure* if exists an efficient algorithm A which, given PK and $\mathrm{Enc}_{PK}(m)$, can recover m with high probability over all m, PK.

- An encryption scheme is *insecure* if there exists an efficient algorithm A which given PK, can recover SK with high probability over all pairs (PK, SK).

2.1.1 RSA encryption

The RSA method, generally the most well-known and commonly used public-key cryptosystem, was developed in 1976-77 by Rivest, Shamir, and Adelman at MIT's Laboratory for Computer Science. It was first seen in print in 1978 [43], but not until after Rivest and MIT worked out a deal with the National Security Agency to allow the new method to be presented. Whether or not this suggests that the NSA had previously discovered the technique, it has been recently declassified that the British cryptographer Clifford Cocks had discovered a similar method as early as 1973 [7], but the finding was classified by the UK Communications-Electronics Security Group.

The scheme of Rivest, Shamir, and Adelman is as follows. Key pairs are generated by the Gen algorithm:

$\mathrm{Gen}(1^k)$:

1. Select random k-bit prime numbers p and q, and let $N = p \cdot q$.
2. Select at random $e < \phi(N)$ so that $\gcd(e, \phi(N)) = 1$.
3. Solve $ed \equiv 1 \pmod{\phi(N)}$ for d.
4. Output $PK = (N, e); SK = (N, d)$.

The space of possible messages for a given public key (N, e) is all integers $m \in (\mathbf{Z}/N\mathbf{Z})^*$, and the encryption operates as follows:

$\mathrm{Enc}_{(N,e)}(m)$: Output $m^e \bmod N$.

Decryption for RSA is the same operation as encryption, using the secret key as the exponent instead of the public one.

$\mathrm{Dec}_{(N,d)}(c)$: Output $c^d \bmod N$.

Proof of correctness. We first prove that the key generation algorithm is correct. It follows from elementary number theory that $ax \equiv 1$ has a solution modulo n if and only if $a \in (\mathbf{Z}/n\mathbf{Z})^*$. Therefore d can always be found given e with $\gcd(e, \phi(N)) = 1$.

As for correct decryption, we rely on Fermat's Little Theorem, although Fermat certainly never could have imagined his work would find its way into implementations of secrecy. Fermat's Theorem asserts that for any $a < N$, $a^{\phi(N)} \equiv 1 \bmod N$. For RSA, we have $\mathrm{Dec}_{SK}(\mathrm{Enc}_{PK}(m)) = (m^e)^d = m^{ed}$. But we know by construction that $ed = k \cdot \phi(N) + 1$ for an integer k, therefore $m^{ed} = m \cdot (m^{\phi(N)})^k = m$. ■

Since the only operation of the encryption and decryption functions is modular exponentiation, we know that these functions are polynomial time by Proposition 1.6.

We also note that the RSA encryption algorithm given here is deterministic—this actually leads to problems (such as the ability to recognize when the same message is sent twice) which make so-called "Plain RSA" insecure under our indistinguishability criterion, for an algorithm given access to PK can easily generate his own encryptions of m_0 and m_1, compare them to the ciphertext input, and know which message he is seeing. Nevertheless, this is the canonical theoretic version of the RSA method, and incorporating random padding into the message is a painless and common way of overcoming the determinism problem.

Examining the algorithm, we see that the public key consists of a random number e, and the composite modulus $N = pq$, which is distributed according to $\mathcal{F}_k$. Since, if we already know e, the knowledge of p and q would allow an adversary to simulate the operation of the Gen algorithm, clearly the secret key SK is no more secret than p and q themselves. This gives us the following result.

Proposition 2.5 *If there is an efficient algorithm for factoring integers with high probability, then an efficient algorithm exists to recover RSA keys with high probability. Therefore in this event, RSA is insecure.* ∎

In fact finer arguments can be made: RSA is insecure if $\phi(N)$ can be computed from N with high probability over $N \leftarrow \mathcal{F}_k$, or if eth roots can be extracted over $(\mathbf{Z}/N\mathbf{Z})^*$. But fundamentally, the structure of RSA keys links its security to the factoring assumption, since these other goals reduce to factoring.

Our proposition tells us that breaking RSA is *no more* difficult than factoring, but we do not have the desirable converse statement that RSA is in fact *no easier* to break than factoring, the sort of claim that the marketers of any cryptosystem would love to make.

Even without this claim, however, the RSA method is the most prevalent public-key cryptography system in commercial use today. Early uses of the method were rare after MIT patented the technique in 1983 and granted the only license to the RSA Security company. At that point acquiring licenses to use the algorithms were extremely expensive. However RSA released the algorithms into the public domain in September 2000, just a few weeks before their patent would expire. Now RSA is used for many applications, including the common privacy software package PGP (Pretty Good Privacy), the Internet Secure Socket Layer protocol (SSL), and the Secure Electronic Transactions (SET) protocol used by major credit card companies.

Since RSA is no more difficult that factoring integers, if that problem were found to be tractable the damage to existing cryptosystems would be substantial.

2.1.2 Rabin ($x^2 \bmod N$) Encryption

As we saw above, RSA is broken if factoring is lost as a hard problem, but we cannot yet prove that nothing less than factoring will cause its collapse. Just a few years after RSA was published, Michael Rabin, also at MIT, proposed a new method [40], similar to RSA, but which *was* able to make this claim. He showed that not only does breaking his scheme reduce to factoring, but vice-versa.

The structure of the key generation and encryption functions is strongly reminiscent of what we have just seen, but by using a fixed encryption exponent $e = 2$, Rabin's scheme not only decreases the computational workload of the actual encrypting machines, but allows the expanded theoretical knowledge surrounding quadratic residues in finite fields to bring the desired provable results.

The tradeoff is a more complex decryption routine, for we know already that since $\gcd(2, \phi(pq))$ will never be 1, there is no decryption exponent d which will undo the squaring operation as is possible in RSA. However, sufficient mathematical work has been done on the topic of quadratic residuosity to deal with this problem.

Definition 2.6 *The quadratic residue symbol $\left(\frac{a}{p}\right)$ has value $+1$ if a is a square in $(\mathbf{Z}/p\mathbf{Z})^*$, and value -1 if a is a non-square in $(\mathbf{Z}/p\mathbf{Z})^*$. If $a \equiv 0 \bmod p$, we say $\left(\frac{a}{p}\right) = 0$. We extend the definition to $\left(\frac{a}{n}\right)$ for n composite by the formula $\left(\frac{a}{p_1p_2\cdots p_k}\right) = \left(\frac{a}{p_1}\right)\left(\frac{a}{p_2}\right)\cdots\left(\frac{a}{p_k}\right)$.*

Elementary number theory gives us the following facts:

$$\text{i. } \left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right) \qquad\qquad \text{ii. } \left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \bmod p \tag{2.1}$$

This second fact gives us the following result which will become most helpful when we wish to decrypt, that is, to take square roots mod N:

Proposition 2.7 *Let p be a prime such that $p \equiv 3 \bmod 4$. Then there is an efficient algorithm which, given $\left(\frac{y}{p}\right) = +1$, generates x such that $x^2 \equiv y \bmod p$.*

Proof. Let $x = y^{(p+1)/4} \bmod p$. Then $x^2 = y^{(p+1)/2} = y^{1+(p-1)/2} = y \cdot \left(\frac{y}{p}\right) = y$. Since modular exponentiation is efficient, this algorithm is efficient. ■

Of interest however, is that there is no efficient deterministic algorithm known to generate a square root of y modulo $p \equiv 1 \bmod 4$, although there is an efficient randomized one. Because of this, and because we require an unambiguous way of distinguishing between the 4 square roots of an quadratic residue mod N, the typical implementation of Rabin's encryption is the Blum variant, where the primes p and q satisfy $p \equiv q \equiv 3 \bmod 4$. (Integers $N = pq$ composed of such primes are called Blum integers.)

Definition 2.8 *The***Rabin public-key encryption scheme** *is composed of the following 3 algorithms:*

$\mathrm{Gen}(1^k)$*:*

1. *Select k-bit prime numbers p and q with $p \equiv q \equiv 3 \bmod 4$.*
2. *Compute $N = p \cdot q$.*
3. *Output $PK = N; \quad SK = (p, q)$.*

Like RSA, the message space for a given public key N is all integers $m \in (\mathbf{Z}/N\mathbf{Z})^*$.

$\mathrm{Enc}_N(m)$*: Output $m^2 \bmod N$.*

To decrypt, use the secret factors of N to find roots modulo p, q, which we know can be done efficiently, then combine those roots into a root modulo N using the Chinese Remainder Theorem.

$\mathrm{Dec}_{(p,q)}(c)$:

1. *Find x_p such that $x_p^2 \equiv c \bmod p$.*
2. *Find x_q such that $x_q^2 \equiv c \bmod q$.*
3. *Use Chinese Remainder Thm. to find an integer x such that $x \equiv x_p \bmod p$ and $x \equiv x_q \bmod q$.*

Here we run into the problem of ensuring that $\mathrm{Dec} \circ \mathrm{Enc}$ is in fact the identity map, which is the problem of making sure that when using the Chinese Remainder Theorem to construct x, the Dec algorithm chooses the same x which was actually encrypted. The fact that $p \equiv q \equiv 3 \bmod 4$ ensures that this is possible through the following observation.

Lemma 2.9 *If $N = pq$ and $p \equiv q \equiv 3 \bmod 4$, any quadratic residue $y \in (\mathbf{Z}/N\mathbf{Z})^*$ has a unique square root x with the properties that $\left(\frac{x}{N}\right) = +1$, and $x < N/2$ when lifted into $\{0, \ldots, N-1\}$.*

Proof. By (2.1-ii), we can determine that when $p \equiv 3 \bmod 4$, we have $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2} = -1$, therefore -1 is not a square modulo p. By (2.1-i), it follows that for any a, exactly one of a and $-a$ is a quadratic residue modulo p. Therefore when we find the square roots of y mod N, and we first find the two roots $u_1, u_2 = -u_1$ of y mod p and the two roots $v_1, v_2 = -v_1$ of y mod q, exactly one of u_1 and u_2 is a square, and exactly one of v_1, v_2 is a square. Without loss of generality, let us say the squares are u_1 and v_1.

By the definition of the quadratic residue symbol, $\left(\frac{a}{N}\right) = +1$ if and only if $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) \neq 0$. Therefore if we label the 4 square roots of y mod N by x_{ij}, each corresponding to the integer computed by the Chinese Remainder Theorem on the pair (u_i, v_j), we know that x_{11} and x_{22} have quadratic residue symbol $+1$ mod N, but x_{12} and x_{21} do not. We also know that $\left(\frac{-1}{N}\right) = \left(\frac{-1}{p}\right)\left(\frac{-1}{q}\right) = (-1)(-1) = +1$. It follows that since one of the x_{ij} must be $-x_{11}$, it must be the one with the same quadratic residue symbol. Thus we must have $x_{11} = -x_{22}$. It follows that of x_{11} and x_{22}, the two square roots of y with $\left(\frac{x}{N}\right) = +1$, only one is less than $N/2$ (under the canonical lift into $\{0, \ldots, N-1\}$). This proves the lemma. ■

It follows that the encryption scheme can be made unambiguous by requiring messages m to satisfy $\left(\frac{m}{N}\right) = +1$ and $m < N/2$. Both properties can be efficiently tested for a given m and N (the first by using quadratic reciprocity). When decrypting, our algorithm will actually find all 4 roots, then return the only root with these properties, and will therefore always recover the original message.

The security of Rabin's scheme rests on the inability of an adversary to calculate square roots modulo N without knowing the factorization of N.

Proposition 2.10 *Breaking Rabin's scheme and factoring are of equivalent difficult, in the sense that each reduces to the other. More specifically, if an algorithm A exists which decrypts Rabin* *ciphertexts with probability ε, an algorithm A' exists which solves the factoring problem (Prob. 1.12) with probability $\varepsilon/2$.*

Proof . One direction is trivial : if we can factor N, then given the public key we can determine the secret key, and clearly the scheme is broken by possession of the secret key.

For the second direction, we employ some randomness in our construction of A'.

```
$A'(N)$:
  1. Select $u \neq \pm 1$ at random from $(\mathbf{Z}/N\mathbf{Z})^*$.
  2. Calculate $y = x^2 \bmod N$.
  3. Calculate $v = A(N, y)$.
  4. Output $\gcd(u - v, N)$.
```

Suppose that A was successful in decrypting the ciphertext y, that is to say it returned an element whose square is y (a possible message for which y is the ciphertext). Then we have $u^2 \equiv v^2 \bmod N$, or $(u-v)(u+v) \equiv 0 \bmod N$. If we have $u \neq \pm v$, it follows that N divides this product but neither of the factors. Therefore $\gcd(u - v, N)$ will be a proper factor of N. Consider the probability that $u = \pm v$ given that $u^2 = v^2$ in the above algorithm. Since u is randomly selected, it will be uniformly distributed over the 4 square roots of v^2 modulo N no matter which v is returned by A. Therefore $\Pr[u \neq \pm v] = 1/2$. So if A is successful with probability ε, A' is successful with probability $\varepsilon/2$. ■

2.1.3 Diffie-Hellman Key Exchange

Though we present it after RSA and Rabin's schemes, the method of Diffie and Hellman is actually considered the first example of public-key cryptography. Their landmark paper in 1976 "New Directions in Cryptography" [14] was the first to propose that a secure cryptosystem might allow some parameters to be made public.

The goal of the Diffie-Hellman protocol is not directly encryption, although it can be extended to support encryption. Its purpose is to perform a *secure key exchange*, a protocol by which two parties can agree upon a secret key over an insecure channel. The key can then be used as the secret key in a symmetric-key encryption scheme, but is more often used for the purpose of authentication. It is also the least complicated and most prevalent example of assuming the difficulty of the discrete log problem for security.

The protocol operates as follows: Let the two parties be Alice and Bob. They agree, in the open, upon a prime p and a generator g of $(\mathbf{Z}/p\mathbf{Z})^*$. Alice chooses a random $a \in (\mathbf{Z}/p\mathbf{Z})^*$ and computes $\kappa_A = g^a \bmod p$. Bob similarly chooses $b \in (\mathbf{Z}/p\mathbf{Z})^*$ and computes $\kappa_B = g^b \bmod p$. They then send these values to one another over an insecure channel. Alice, upon receiving κ_B, computes $K_A = \kappa_B^a$; Bob similarly computes $K_B = \kappa_A^b$. It follows that

$$K_A = (\kappa_B)^a = (g^b)^a = g^{ab}; \quad K_B = (\kappa_A)^b = (g^a)^b = g^{ab},$$

so the two parties have indeed agreed on the same key.

To say that an adversary, observing this exchange, is unable to figure out the key, reduces to what is called the Diffie-Hellman Computation problem, which states that no efficient adversary is able to compute g^{ab} in $(\mathbf{Z}/p\mathbf{Z})^*$ given g^a and g^b (as well as g and p). In fact, usually a stronger problem is assumed, which states that not only can an adversary not determine the key, but he cannot even identify it if it is shown to him. This problem, the Diffie-Hellman Decision problem, states that no efficient algorithm can distinguish between a triple of the form (g^a, g^b, g^{ab}) and one of the form (g^a, g^b, g^c) for a random c.

While the assumption that these problems are difficult is not exactly the same as assuming that the discrete log problem is difficult, it is clear that if a polynomial-time algorithm for taking discrete logs were to be discovered, both of the Diffie-Hellman problems would immediately become feasible, and therefore (though we do not define security in this case):

Proposition 2.11 *If there exists a polynomial time algorithm to solve the discrete log problem, then the Diffie-Hellman key exchange is not secure.* ■

2.1.4 Generalized Diffie-Hellman

The above protocol deals with 2 parties agreeing on a key, but we could easily conceive of extending it to work for $k > 2$ parties. In this protocol, we have the parties $P_1, \ldots, P_k$, and each P_i selects at random an element a_i from $(\mathbf{Z}/p\mathbf{Z})^*$ (or whatever group with generator g has been agreed upon), and computes $\kappa_i = g_i^a$. They initially pass around these κ_i, and can iteratively compute messages of the form (I, κ_I) where I is a *proper* subset of $\{1, \ldots, k\}$, and $\kappa_I = g^{(\prod_{i \in I} a_i)}$, and they also pass around these messages. After enough passing, each P_i will be able to compute the secret key $K = \kappa_S = g^{(\prod_{i=1}^k a_i)}$.

If we want the key to be secure, we wish for the following problem to be infeasible, which we call the Generalized Diffie-Hellman Problem. Like above, it can be considered computationally or decisionally; the computational version is to find, given g a generator of a fixed group G, and for some set S, the value of $K = g^{(\prod_{i \in S} a_i)}$ given access to $\kappa_I = g^{(\prod_{i=I} a_i)}$ for any proper subset $I \subsetneq S$.

Clearly the security of this scheme has the same dependence on the hardness of discrete log that the 2-party version did, but it will turn out that it also has an unexpected relationship to factoring, which we will show in Section 4.3.

2.1.5 ElGamal Encryption

We can also harness the hardness of discrete log directly for encryption using the ElGamal public key cryptosystem [15], given by the following three algorithms:

Gen:

- Select a random prime p and a generator g of $(\mathbf{Z}/p\mathbf{Z})^*$.
- Select a random element $a \in (\mathbf{Z}/p\mathbf{Z})^*$, and compute $y = g^a \bmod p$.

- Output $PK = (p, g, y)$; $SK = a$.

The dependence on discrete log complexity is clear from the keys; if discrete logs were efficiently computable, the secret key could be immediately deduced from the public key. To encrypt a message $m \in \mathbf{Z}/p\mathbf{Z}$ with this public key cryptosystem, proceed as follows:

$\mathrm{Enc}_{PK}(m)$:

- Select a random $b \in (\mathbf{Z}/p\mathbf{Z})^*$.
- Output $c = (g^b \bmod p, m \cdot y^b \bmod p)$.

The message is therefore padded in a sense by $y^b = g^{ab}$, and the random string b is sent along to help in the decryption, although it must be sent in the hidden form g^b because clearly b alone would be enough to extract m from y and $m \cdot y^b \bmod p$. To decrypt using the secret key,

$\mathrm{Dec}_{SK}(c = (h, z)$:

- Compute h^a (note that this is g^{ab}), and compute the inverse $(h^a)^{-1}$.
- Output $z \cdot (h^a)^{-1}$.

It is clear that $z \cdot (h^a) - 1 \equiv m \cdot g^{ab} \cdot g^{-ab} \equiv m$, so the decryption is valid.

This cryptosystem has been of substantial importance since the National Institute of Standards and Technology released the Digital Signature Standard [16], which is based on ElGamal.

2.1.6 Elliptic Curve Cryptography

A method similar to ElGamal can be performed over elliptic curves, and such a technique is accepted as a variant of the standard Digital Signature Algorithm [20].

Our key generator now selects a prime p and two elements a and b of $\mathbf{Z}/p\mathbf{Z}$ such that $6(4a^3 + 27b^2)$ is not divisible by p. It follows that $E_{a,b}$ given by $y^2 = x^3 + ax + b$ is an elliptic curve, and we can consider the set $E_{a,b}(\mathbf{F}_p)$ of ordered pairs $(x, y) \in (\mathbf{Z}/p\mathbf{Z}) \times (\mathbf{Z}/p\mathbf{Z})$ which satisfy this equation. We saw in chapter 1 that this set of points has a natural group structure with operations that can be efficiently computed. Therefore we can easily conceive of creating instances of the discrete log problem over this group of points.

Our generator is some point $P \in E(\mathbf{F}_p)$ with high order, and this is made public as in ElGamal. Each user of the system creates a secret key which is an integer a, computes the point $P_1 = a \cdot P$ and makes this public. (The group operation on elliptic curves is written additively, but this is the equivalent to g^a in ElGamal.) To send the message m, we encode m as some point M in $E(\mathbf{F}_p)$, then choose a random $k < p$ and send the pair $(k \cdot P, k \cdot P_1 + M)$.

Decryption is analogous to the previous section: on receipt of the pair (H, Z), we use our knowledge of the secret key a to compute

$$Z - a \cdot H = k \cdot P_1 + M - a \cdot k \cdot P = k \cdot (a \cdot P) + M - a \cdot k \cdot P = M.$$

The problem assumed to be difficult for the security of this cryptosystem is specifically that of computing from P and $P_1 = a \cdot P$, the power a. In general, this is currently thought to be more difficult than the discrete log problem in group modulo p. In part, this is because our current level of knowledge about elliptic curves does not give us the same power to exploit the representation of group elements that we have when those representations are integers. Therefore the best algorithms available are the so-called "generic" ones, and these algorithms have established lower bounds on their complexity, which present later (see section 5.1.1).

2.2 Cryptographic Primitives

Theoretical cryptographers often consider the perspective of rigorous definitions rather than individual protocols like those listed above. By demonstrating various provable constructions, we can build such secure objects as *one-way functions*, *pseudo-random generators* (introduced by Blum and Micali in 1982, published in 1984 [3]) and *pseudo-random functions* (introduced by Goldreich, Goldwasser, and Micali in 1984, published 1986 [18])

These primitives, particularly the last two—pseudo-random generators and functions— imply naturally the existence of secure encryption schemes. Given any finite message m, we know that if we choose a random r of the same bit length, the bitwise exclusive $r \oplus m$ completely conceals m from an information-theoretic perspective (a technique known as the one-time pad). The difficulty in using this fact for encryption is that in order to decrypt, r must be a secret known by both parties, and having a key the same length as your message is very restrictive. What these pseudo-random primitives allow is the creation of a lot of near-random bits from a small number of truly random ones. Pseudo-random generators are a way of creating a long sequence of near-random bits from a short secret "seed" which is the shared key; pseudo-random functions allow shared randomness of arbitrary size as well, where the shared key is parameters into some family of functions which are indistinguishable from a random function.

To avoid being either incomplete or unnecessarily lengthy, we avoid presenting these notions in full. For a complete presentation of cryptography building up these definitions, consider Oded Goldreich's book [17].

While the one-time pad is not a number-theoretic cipher on its own, the constructions of the pseudo-random objects which generate these pads are often based on assumptions of number-theoretic difficulty. Along this line, Naor, Reingold, and Rosen [32] demonstrated an efficient pseudo-random function secure under the difficulty of the Generalized Diffie-Hellman problem (therefore assuming the difficulty of taking discrete logarithms), and Dedic, Reyzin, and Vadhan [12] constructed a pseudo-random generator based on the difficulty of factoring.

3. The Relationship Between the Problems I – Algorithms

3.1 Background Mathematics

3.1.1 Smoothness

We begin by defining a class of integers which are easy to factor. These integers will appear repeatedly in the methods that follow, as searching for them by one or another means has proven valuable towards factoring more difficult integers, and also towards solving discrete logarithms.

Definition 3.1 *An integer x is B-***smooth** *if all prime numbers dividing x are less than or equal to B. An integer x is B-***powersmooth** *if all prime powers dividing x are less than or equal to B.*

Since we intend to search for these values, it will serve us to present some analysis of their frequency. We define the de Bruijn function $\psi(x, B)$ as the number of B-smooth positive integers less than or equal to x. Following Wagstaff [50], for $t \in [0, 1]$ and $x > 2$ define

$$p(x,t) = \Pr_{0<N\leq x}[\text{the largest prime factor of } N \text{ is less than } x^t],$$

the probability taken over all positive $N \leq x$. It follows that $p(x, t) = \psi(x, x^t)/x$. Define $F(t) = \lim_{x\to\infty} p(x, t)$, the Dickman function, after the mathematician who gave, in 1930, the following heuristic argument that this limit exists for all t.

Consider s in the interval $0 < s < 1$. For any x, the number of integers whose largest prime factor lies between x^s and x^{s+ds} for small ds is $x \cdot F'(s)\, ds$. By the prime number theorem, the number of primes between x^s and x^{s+ds} is (see [50] p. 55)

$$\pi(x^{s+ds}) - \pi(x^s) \approx \frac{x^s + (\ln x)x^s\, ds}{\ln x^s} - \frac{x^s}{\ln x^s} = \frac{x^s\, ds}{s}.$$

For each of these primes p, the number of n such that $pn \leq x$ and n has no prime factor greater than p is the same as the number of $n \leq x^{1-s}$ (since $p \leq x^s$), whose greatest prime factor is less than $x^s = (x^{1-s})^{s/(1-s)}$. But the number of such n is $x^{1-s}F(s/(1-s))$. Therefore, heuristically we have

$$x \cdot F'(s)\, ds = \frac{x^s\, ds}{s} \cdot x^{1-s}\, F(s/(1-s)).$$

The x terms all cancel, and then by integrating we get Dickman's functional equation

$$F(t) = \int_0^s F\Big(\frac{s}{1-s}\Big)\frac{ds}{s}.$$

Typically when searching for smooth numbers we will want t to be very near zero, so we consider $\rho(u) = F(1/u)$. It is not hard to derive the functional equation $\rho'(u) = -\rho(u-1)/u$ from the above functional equation for F. Dickman's heuristic argument shows that $\rho(u)$ defined by this equation and the condition $\rho(u) = 1$ for $u \leq 1$ (necessarily all $n < x$ are x^t-smooth if $t \geq 1$) is the limit $\lim_{x\to\infty} \psi(x, x^{1/u})/x$. This theorem was proved rigorously by Ramaswami in 1949 [41].

Playing with the definition of ρ leads to the result $\rho(u) < \rho(u-1)/u$ for all $u > 1$, and it has been shown that $\rho(u)$ is closely modeled by u^{-u} for sufficiently large u. Thus $\psi(x, x^{1/u}) \approx u^{-u}$, and even if we allow u to vary with x, it has been shown that as long as $u < (1-\varepsilon)\log x/(\log\log x)$, the approximation is valid. To approximate $\psi(x, B)$, as we will often want to do, we note that if we let $u = (\log x)/(\log B)$, we have $\ln B = \ln x/u$, so $B = x^{1/u}$. Therefore $\psi(x, B) \approx xu^{-u}$, with $u = (\log x)/(\log B)$.

3.1.2 Subexponential Complexity

Though we know of no polynomial time algorithms to solve factoring and discrete log, it would not be correct to say that all known algorithms have *exponential* running times. Several of the modern methods we present do better than exponential. Although we present for the sake of historical continuity the exponential algorithms first (the ones Cohen [8] sorts under "Factoring in the Dark Ages"), here we introduce some standard notation which will facilitate the eventual analysis of the more modern algorithms.

Definition 3.2 *For any* $0 \leq \alpha \leq 1$, *define the function*

$$L_x[\alpha; c] = \exp(c(\log x)^\alpha(\log\log x)^{1-\alpha}).$$

This L function lets us identify running times along the continuum from polynomial to exponential. For example, an algorithm with running time $L_N[0; c]$ is $O((\log N)^c)$, and therefore polynomial time. An algorithm with running time $L_N[1; c]$ is $O(N^c)$, so exponential. If an algorithm has running time $L_N[\alpha, c]$ for some $0 < \alpha < 1$, as we will show for several factoring and discrete log methods here, it is called *subexponential*.

Because it occurs quite often in multiple roles, we abbreviate one of these such functions simply by $L(x)$, with the definition

$$L(x) = L_x[1/2; 1] = \exp((\log x)^{1/2}(\log\log x)^{1/2}).$$

3.2 Standard Techniques Applied to Both Problems

3.2.1 Pollard's $p-1$ method

Our first algorithm for factoring hopes to find a factor p of N by taking advantage of the (possible) smoothness of $p-1$. The algorithm dates to 1974, and was discovered by John M. Pollard [36], a number theorist and cryptographer working for British Telecom (and more recently the 1999 recipient of the RSA Award in Mathematics given by RSA Data Security, Inc. for his contributions to cryptography).

Proposition 3.3 (Pollard) *If N has a prime divisor p such that $p-1$ is B-powersmooth, where B is bounded by a polynomial in $n = \log N$, then p can be found efficiently with arbitrarily high probability.*

Proof. The proof of this is Pollard's algorithm, which operates as follows:

```
POLLARD($N$):
  1. Let $Q = \mathrm{lcm}(1, 2, \ldots, B)$, where $B$ is the smoothness bound.
  2. Select at random an integer $x \in (\mathbf{Z}/N\mathbf{Z})^*$. (Do this by selecting out of $\mathbf{Z}/N\mathbf{Z}$
     and checking the $\gcd(x, N)$. If $\gcd > 1$, we have a factor.)
  3. Compute $d = \gcd(x^Q - 1, N)$. If $1 < d < N$, output $d$. Else fail.
```

We assert that the algorithm is correct. For if $p-1$ is B-powersmooth, then all its factors divide Q. Therefore $p - 1 \mid Q$. It follows from Fermat that $x^Q \equiv 1 \bmod p$, therefore $p \mid x^Q - 1$. So if $d = \gcd(x^Q - 1, N)$, then $p \mid d$.

The algorithm will only fail if $d = N$. However in this event $x^Q \equiv 1 \bmod N$, which indicates the bound B, and therefore Q, was too large, and we can try again with a lower B.

Consider the running time of the algorithm. The main operations are computing Q and raising x to the power Q, and the second dominates the running time. So to analyze this algorithm we need to consider how large Q is. We can give the following alternate definition: $Q = \prod_{p \le B} p^e(p)$ over all primes $p \le B$, where $e(p) = \max\{e : p^e \le B\}$. So Q is the product of all the largest possible powers of primes less than B. It follows that $\log Q = \sum_{p \le B} e(p) \log p$. Since we could redefine $e(p) = \max\{e : e \log p \le \log B\}$, it follows that $e(p) \log p < \log B$. Therefore $\log Q \le \sum_p \log B = \pi(B) \log B = O(B)$.

So POLLARD performs $O(B)$ modular multiplications, for a total complexity of $O(B \log N)$. ■

Note that there is an alternate presentation of the algorithm (for example, in [29]) which puts the weaker requirement on $p-1$ that it be B-smooth instead of B-powersmooth. In this version Q is constructed the same way over all $\prod_{p \le B} p^{e(p)}$, but now $e(p) = \max\{e : p^e \le N\}$. There is a corresponding tradeoff in complexity, for now $\log Q = \pi(B) \cdot \log N = O(B \log N / \log B)$ multiplications are required instead of only $O(B)$.

Example. We use Pollard's $p-1$ method to factor $N = 9557780739229$. It appears daunting, but we might hope that there is a factor p with smooth $p-1$—we will try first $B = 20$, very optimistically.

- We compute $Q = \mathrm{lcm}(1, \ldots, 20) = 232792560$.
- We choose at random $x = 3$, and compute $y = x^Q \bmod N = 5068864611225$

- We compute $\gcd(y-1, N)$, which is unfortunately 1.

We have therefore failed, but perhaps we were too optimistic. So we can try again with a larger B, say $B = 30$.

- We compute $Q = \operatorname{lcm}(1, \ldots, 30) = 2329089562800$.
- We choose at random $x = 5$, and compute $y = x^Q \bmod N = 2792592641549$
- We compute $\gcd(y-1, N)$, which is fortunately $p = 12601$. We note that $p-1 = 2^3 \cdot 3^2 \cdot 5^2 \cdot 7$ is quite smooth.

We see that there is a significant tradeoff between the power of using a larger smoothness bound B in order to be able to factor more integers, and the size of the values we must operate with. $\diamond$

Adapting Pollard $p-1$ to Discrete Log

We have just witnessed how the presence of a factor p with $p-1$ smooth enables this factor to be efficiently found. We can see an immediate connection to discrete log over a prime modulus p, and demonstrate a way to compute discrete logs easily in the case that $p-1$ is smooth.

The algorithm is due to Pohlig and Hellman [35], who without specifically discussing smoothness present an algorithm for solving discrete logarithms over a cyclic group of order n given the factorization of n. This clearly applies, since the smoothness of $p-1 = |(\mathbf{Z}/p\mathbf{Z})^*|$ implies that the factorization can be quickly found.

In general, since we wish to compute an exponent a which exists modulo $p-1$, if we have a factorization $p-1 = q_1^{e_1} \cdots q_s^{e_s}$, we can compute a modulo each $q_i^{e_i}$ and then use the Chinese Remainder Theorem to reconstruct the full exponent $a \in \mathbf{Z}/p\mathbf{Z}$. The smoothness thus comes into play in the computation of discrete log modulo $q_i^{e_i}$, which can be done by brute force search if the modulus is small.

This characterization is intuitive if we say $p-1$ must be B-powersmooth for a low bound; we can reduce to the less restrictive B-smooth property with the algorithm in its entirety:

```
POHLIG-HELLMAN($p, g, y$):
  1. Compute the factorization of $p - 1 = q_1^{e_1} \cdots q_s^{e_s}$, each $q_i$ prime and each $e_i \geq 1$.
  2. For each $i = 1, \ldots, s$, compute $a_i = a \bmod q_i^{e_i}$ as follows:
     (a) Set $h = g^{(p-1)/q_i}$. Note that the order of $h$ is $q_i$.
     (b) Initialize $\gamma = 1$, $l_{-1} = 0$.
     (c) For each $j = 0, \ldots, e_i - 1$:
          i. Set $\gamma \leftarrow \gamma g^{l_{j-i} q_i^{j-1}}$, $z \leftarrow (y/\gamma)^{(p-1)/q_i^{j+1}}$.
         ii. Compute $l_j$ such that $h^{l_j} = z$ (e.g., by brute force).
     (d) Set $a_i = l_0 + l_1 q_i + \cdots + l_{e_i - 1} q_i^{e_i - 1}$.
```

3. Combine the a_i into a using Chinese Remainder Theorem.

The algorithm is sufficiently complex to warrant some justification. First, since as indicated the order of the generator h in the ith step is q_i; so if we assume that $p-1$ is B-smooth, we require no more than $O(B)$ operations to compute logarithms to the base h by brute force. Now, consider the jth iteration of the inner loop. At this point we have $\gamma = g^{l_0+l_1q_i+\cdots+l_{j-1}q_i^{j-1}}$. Recall that we are searching for the p-ary representation of $a_i = l_0+l_1q_i+\cdots+l_{e_i-1}q_i^{e_i-1}$., where $y=g^a$. Therefore

$$\begin{aligned}
z &= (y/\gamma)^{(p-1)/q_i^{j+1}} \\
&= (g^{a-l_0-l_1q_1-\cdots l_{j-1}q_i^{j-1}})^{n/q_i^{j+1}} \\
&= (g^{n/q_i^{j+1}})^{a-l_0-l_1q_i-\cdots-l_{j-1}q_i^{j-1}} \\
&= (g^{n/q_i^{j+1}})^{l_jq_i^j+\cdots+l_{e_i-1}q_i^{e_i-1}} \\
&= (g^{n/q_i})^{l_j+\cdots+l_{e_i-1}q_i^{e_i-1-j}} \\
&= h^{l_j}\cdot h^{q_i(\cdots)} = h^{l_j}.
\end{aligned}$$

It follows that setting l_j equal to the discrete log of z to the base h is exactly what we should do to generate a completely. As mentioned, each inner loop takes $O(B)$ operations, and the number of such loops is $\sum e_i$, or the total number of prime factors of $p-1$ counting multiplicity, which is naturally less than $\log_2(p-1) < \log_2 p$ Therefore if $p-1$ is B-smooth, we can evaluate discrete logs modulo p in $O(B\log p)$ time.

The two above algorithms thus show the first practical correlation between the two problems: a certain smoothness condition which leads to polynomial time solutions to each. We note that this fact is not particularly useful for the applications of Chapter 2, for there we assume that the problems are hard over a particular distribution of instances, and the instances are engineered to avoid things like smoothness, usually by selecting primes p such that $p=2q+1$ for another prime q. Such a p is known as a Sophie Germain prime after the pioneering female number theoretician of the turn of the 19th century, and $p-1$ is clearly not at all smooth.

3.2.2 Pollard's rho Method

We arrive at a second method which demonstrates commonalities between the two problems. The specifics of the rho method, although it is specialized and (for factoring) only provides efficient access to *small* factors, apply to solving both factoring and discrete log.

Both applications of the technique are due to Pollard and were published in the 1970s not long after his $p-1$ method. Because they operate by simulating a random walk over the finite field, he called them "Monte Carlo" methods, which refers to the class of randomized algorithms to which rho belongs. The method for factoring appeared in 1975 [37] and the method for discrete log in 1978 [38].

The rho Method : Factoring.

To find a factor p of N, we make use of the recursive sequence defined by $x_0 = 2$, $x_n = f(x_{n-1}) = x_{n-1}^2 + 1 \bmod N$. Without knowing p, we can imagine this sequence reduced modulo p. Since this reduction preserves multiplication and addition it is clear that the reduced sequence $\bar{x}_0, \bar{x}_1, \ldots$ also satisfies the recursive relation $\bar{x}_n = \bar{x}_{n-1}^2 + 1 \bmod p$. Therefore we have an infinite sequence of elements drawn from a finite set, so eventually $\bar{x}_i = \bar{x}_j$, and the sequence must begin to cycle.

To avoid the necessity of storing the entire sequence in order to find a collision, we make use of the following lemma:

Lemma 3.4 (Floyd's cycle-finding method) *If $x_0, \ldots, x_n$ is a recursive sequence defined by $x_n = f(x_{n-1})$ over a finite set, then there is some m such that $x_m = x_{2m}$.*

Proof of lemma. We know that the sequence must eventually collide and begin repeating. Let λ' be the least integer such that there exists $\lambda < \lambda'$ and $x_\lambda = x_{\lambda'}$. Then the sequence is periodic with period $\rho = \lambda' - \lambda$. It follows that if m is any multiple of ρ and $m > \lambda$, we will have $x_m = x_{2m} = x_{m+k\rho}$. [From [29]: Letting $m = \rho(1 + \lfloor \lambda/\rho \rfloor)$ provides the smallest such m.] ■

Therefore, we only need consider pairs (x_i, x_{2i}), which can be easily (and with minimal storage space) derived from the previous pair (x_{i-1}, x_{2i-2}), and look for collisions. And when we do find a pair x_i, x_j such that the reductions $\bar{x}_i, \bar{x}_j$ collide, it means $x_i \equiv x_j \bmod p$, or equivalently $p \mid \gcd(x_i - x_j, N)$. Therefore, unless this gcd is N itself, a collision will mean we have found a factor of N.

The algorithm thus takes the following form:

```
RHO($N$):
  1. Initialize $x = y = 2$.
  2. For $i = 1, 2, 3, \ldots$, do the following:
     • Update $x \leftarrow f(x)$, $y \leftarrow f(f(y))$.
     • Compute $d = \gcd(x - y, N)$. If $1 < d < N$, output $d$.
     • If $d = N$, algorithm fails.
```

Consider the running time of this algorithm on input N. While we cannot prove a certain order of complexity, we can model $f(x)$ as simulating a random walk. With this heuristic assumption, we can invoke the "birthday problem"[1], and thus we expect the period of a random walk over a finite set of size s to be $\sqrt{\pi s/8} = O(\sqrt{s})$, and the number of terms occurring before the periodicity begins (the "tail") to also be $\sqrt{\pi s/8}$. (The name rho originates in this "tail"/"loop" shape of a sequence, nicely represented by the character ρ.) Therefore (from the note at the end of the proof of Floyd's method in Lemma 3.4 above), we expect (again, heuristically) the first duplicate to be found after $O(\sqrt{p}) = O(n^{1/4})$ iterations. So compared to trial division, which requires $O(n^{1/2})$ computations, we have made significant progress—but we are nowhere near efficient.

[1]That is, the combinatorial fact that we need only about $\sqrt{365}$ (uniformly random) people in a room to expect two of them to share a birthday.

The rho method : Discrete Log

Interestingly enough, the same approach can be used to solve the discrete log problem. This is not particularly surprising, since the critical fact above we used was the finite size of $\mathbf{Z}/p\mathbf{Z}$ and thus finding a collision in a sequence over it. For factoring N, the group $\mathbf{Z}/p\mathbf{Z}$ is somewhat indirectly attached; for discrete logs over a prime, $\mathbf{Z}/p\mathbf{Z}$ is part of the essence of the problem—thus the rho algorithm should intuitively fit well there as well.

In order to actually get access into the exponent, the system is slightly more complex than for factoring. We partition $\mathbf{Z}/p\mathbf{Z}$ into three sets S_1, S_2, S_3 by defining

$$S_i = \{x \in \mathbf{Z}/p\mathbf{Z} : \tilde{x} \equiv i \bmod 3 \text{ for } \tilde{x} \text{ the canonical lift of } x \text{ into } \{0, \ldots, p-1\}\}.$$

The actual partition is not as important as that membership be easy to check and that the partitions be of roughly equal size. We recall that our goal is, given $p, g,$ and $y = g^a$, to recover a. Using these, we define our recursive function by $x_0 = 1$, and

$$x_n = f(x_{n-1}) = \begin{cases} y \cdot x_{n-1}, & \text{if } x_{n-1} \in S_1 \\ x_{n-1}^2, & \text{if } x_{n-1} \in S_2 \\ g \cdot x_{n-1}, & \text{if } x_{n-1} \in S_3 \end{cases}$$

Therefore we can think of this iterative function as operating "behind-the-scenes" as follows: each x_i is of the form $g^{\alpha_i} y^{\beta_i}$, where the sequences $\{\alpha_i\}, \{\beta_i\}$ are induced by the function f to satisfy $\alpha_0 = \beta_0 = 0$ and the recursive relations

$$\alpha_n = \begin{cases} \alpha_{n-1}, & \text{if } x_{n-1} \in S_1 \\ 2\alpha_{n-1}, & \text{if } x_{n-1} \in S_2 \\ \alpha_{n-1} + 1, & \text{if } x_{n-1} \in S_3 \end{cases}, \quad \beta_n = \begin{cases} \beta_{n-1} + 1, & \text{if } x_{n-1} \in S_1 \\ 2\beta_{n-1}, & \text{if } x_{n-1} \in S_2 \\ \beta_{n-1}, & \text{if } x_{n-1} \in S_3 \end{cases} \tag{3.1}$$

We use the same general method as we did for factoring above to find a matching pair $x_m = x_{2m}$. It follows that $g^{\alpha_i} y^{\beta_i} = g^{\alpha_{2i}} y^{\beta_{2i}}$, thus $g^{\alpha_i - \alpha_{2i}} = y^{\beta_{2i} - \beta_i} = g^{a(\beta_{2i} - \beta_i)}$. It follows that

$$a \equiv (\alpha_i - \alpha_{2i})/(\beta_{2i} - \beta_i) \pmod{p-1}. \tag{3.2}$$

Therefore, provided that $\beta_{2i} \not\equiv \beta_i \pmod{p-1}$ (which occurs negligibly often), we can solve for a. It is clear that we can efficiently compute x_i, x_{2i} for each i, maintaining the α and β sequences as we go according to (3.1), and when we eventually do find $x_i = x_{2i}$, compute a according to (3.2)—it being so clear, we omit the actual algorithm.

As above, if we conjecture that this function behaves as a random walk, then we expect to find the first i such that $x_i = x_{2i}$ somewhere around $i = 2\sqrt{\pi p/8})$, so our algorithm runs in time $O(\sqrt{p})$. Just as the rho method did for factoring, it has reduced the complexity of a naive discrete log search algorithm by a power of $1/2$.

And so we see for the second time that the same idea for an algorithm contributes to solving both of our problems; the idea behind rho being to take a random walk around a finite space and take advantage of the collision which must result in order to get the answer. We note without further investigation that this method is, in general, the best available algorithm to solve the discrete log problem on elliptic curves mentioned in Section 2.1.6.

3.3 The Elliptic Curve Method

A derivative of the Pollard $p-1$ method, the Elliptic Curve Method (or ECM) was discovered by H.W. Lenstra [25] and published in 1987, thirteen years after Pollard's $p-1$ method. It remains one of the fastest general purpose factoring methods (that is, applicable to arbitrary integers and not just those of a special form), particularly for finding factors with size about $10^{20}-10^{40}$. It is less effective than some of the sieving methods presented later for finding larger factors, but is often used within those algorithms when intermediate values of medium size need to be factored.

In a sense, it is the odd-ball of the factoring methods presented here, since currently there is no known way to adapt this method for solving discrete logs as well. However in some sense it is the natural bridge between the preceding "exponential" methods (since it follows so directly from the $p-1$ method) and the class of subexponential ones to follow, of which it counts itself a member.

Even though the ECM does not have a discrete log variant, elliptic curves still do play a role in the lives of both of these problems, though ironically in different directions—from what we know at this point, elliptic curves make factoring easier and discrete log harder!

Recall that the aim of Pollard's $p-1$ method was to find prime factors p of N such that $p-1$ is smooth. However, in the average case it is very improbable that such a p will exist, and as mentioned it is feasible to specifically construct N which do not have any such factors. Lenstra's ECM method allows us to factor N if there is a prime factor p such that there is some smooth s *near* p, not necessarily $s=p-1$. Here the density of smooth numbers comes into play, but under certain heuristic assumptions the likelihood of finding such an s is high enough to make ECM viable.

The algorithm is based on the observation that for any elliptic curve $y^2=x^3+ax+b$, which we denote $E_{a,b}$ or simply E, if N factors as $p_1^{e_1}\cdots p_k^{e_k}$, then

$$E(\mathbf{Z}/N\mathbf{Z})=E(\mathbf{Z}/p_1^{e_1}\mathbf{Z})\times E(\mathbf{Z}/p_2^{e_2}\mathbf{Z})\times\cdots\times E(\mathbf{Z}/p\mathbf{Z}_k^{e_k})$$

Therefore, if the order of one of the groups on the right is smooth, for example suppose $|E(\mathbf{Z}/p_1^{e_1}\mathbf{Z})|$ is B-powersmooth, then (mirroring Pollard's $p-1$) by taking a large multiple of a point $P\in E(\mathbf{Z}/N\mathbf{Z})$, say $m=\operatorname{lcm}\{1,\ldots,B\}$, we must get that the $m\cdot\bar{P}\equiv\mathcal{O}=(0:1:0)$, where $\bar{P}$ is the projection of P into $E(\mathbf{Z}/p_1^{e_1}\mathbf{Z})$. In that case it follows that $m\cdot P\equiv(0:1:0)\bmod p$, and so it must be that the third coordinate of $m\cdot P$ must be divisible by p. Equivalently, we attempt to compute $m\cdot P$ using the algebraic addition formulas introduced in section 1.4, and if we ever find that x_1-x_2 is not invertible modulo N, we will have a factor of N.

To implement this method, the curves typically used are of the class $y^2z=x^3+axz^2+z^3$ for a random a, mainly for the reasons that there is only a single parameter and there is always a convenient point $(0:1:1)$ on the curve. Letting P equal this point, we choose a smoothness bound B and compute $m=\operatorname{lcm}\{2,\ldots,B\}$ as in Pollard's method. Then we compute $(x:y:z)=m\cdot P$ on the curve $E_a(\mathbf{Z}/N\mathbf{Z})$. Note that this computation will always be possible as a projective point—the failure may occur when we try to rewrite this value as $(x/z:y/z:1)$ modulo N, in the event that z is not a unit modulo N. If we find $\gcd(z,N)>1$, output this factor of N.

It is clear that if $E_a(\mathbf{Z}/p\mathbf{Z})$ is B-powersmooth, then the method will succeed with very high probability. The only source of failure is that $m\cdot P$ might be equal to $(0:1:0)$, which is certainly congruent to $\mathcal{O}$ modulo p, but does not lead to a factor. This will only happen if $E_a(\mathbf{Z}/q\mathbf{Z})$ is B-powersmooth for all $q \mid N$, and therefore simply suggests that our B was too large. See [25] for a full description.

The following well-known result is a theorem of Hasse which is extremely relevant to the search for smooth values of $E_a(\mathbf{Z}/p\mathbf{Z})$: For any elliptic curve E, if $n = |E(\mathbf{Z}/p\mathbf{Z})|$, then $p + 1 - 2\sqrt{p} \leq n \leq p + 1 + 2\sqrt{p}$. Deuring showed in 1941 that for any n in that range (called the Hasse interval), there exists an elliptic curve E such that $|E(\mathbf{Z}/p\mathbf{Z})| = n$, and moreover gave a formula for the number of curves which have that order [13].

Since we will only be able to analyze the algorithm assuming the choice of *random* curves, fixing $b = 1$ and $P = (1 : 0 : 1)$ does not really work, although it is very often done in practice. A better way is to choose a random a, and then a random point $P = (x_0 : y_0 : 1)$, and then let $b = y_0^2 - x_0^3 - ax_0$. That way we have a truly random curve $E_{a,b}$ and a random point on it. This will give us a roughly random value for $|E(\mathbf{Z}/p\mathbf{Z})|$ in the Hasse interval.

We must now make a conjectural leap. By the results from the introduction to this chapter, we have some knowledge about the probability that a random integer less than p will be B-smooth; in order to apply that formula here we must assume that the distribution of smooth integers in the Hasse interval matches the overall distribution.

Once we assume this conjecture, the probability that this group order is B-smooth (and will thus lead to a factor of N) is u^{-u} for $u = \log p / \log B$ by earlier analysis. Therefore we expect to have to try u^u curves to be successful. As in the analysis of Pollard's $p - 1$ method, the number of group operations necessary to compute $m \cdot P$ on any one curve is roughly $\pi(B) \cdot \log B = B$. We therefore want to choose a bound B to minimize the total work required, given by Bu^u. Recall the definition $L(x) = \exp((\log x)^{1/2}(\log\log x)^{1/2})$ from earlier, and define $a = \log B / \log L(p)$. Thus $B = L(p)^a$, and $\log B = a(\log p)^{1/2}(\log\log p)^{1/2}$. Therefore

$$u = (\log p)/(\log B) = (1/a)(\log p)^{1/2}(\log\log p)^{-1/2}.$$

and

$$\log u = \log(1/a) + (1/2)\log\log p - (1/2)\log\log\log p \approx (1/2)\log\log p,$$

since the other terms are relatively small. It follows that

$$u^u = \exp(u \log u) = \exp((1/2a)(\log p)^{1/2}(\log\log p)^{1/2}) = L(p)^{1/2a}.$$

It follows that to minimize our total work $Bu^u = L(p)^a L(p)^{1/2a}$, it is enough to minimize $a + 1/2a$. Differentiation yields the optimal choice of $a = 1/\sqrt{2}$, so the optimal B is $L(p)^{1/\sqrt{2}}$, and the minimum total work is $L(p)^{\sqrt{2}}$.

We note one very interesting aspect of this complexity, which is that it depends on the size of the prime factor p to be found, not on N itself.

3.4 Sieving Methods for Factoring

We assume for this chapter that N, the integer to be factored, is odd, greater than 1, and is not a prime power. Our goal in general is to use the following fact:

Proposition 3.5 *Given $x \in (\mathbf{Z}/N\mathbf{Z})^*$ with $x^2 = 1$, and $x \neq \pm 1$, a nontrivial factor of N can be found.*

Proof . The existence of such a square root of 1 modulo N follows from the Chinese remainder theorem. Given such an x, with $x^2 - 1 \equiv 0$, it follows that N divides $(x-1)(x+1)$, but since $x \neq \pm 1$, N does not divide either factor. Therefore we can simply evaluate $\gcd(x-1, N)$ and will have a factor of N. ■

In general, consider the subgroup $\{x : x^2 = 1\}$ of $(\mathbf{Z}/N\mathbf{Z})^*$. We can consider this as a vector space over the 2-element field $\mathbf{F}_2$, and its dimension, by the Chinese remainder theorem, is equal to the number of prime factors of N. Therefore listing the generators of this subgroup is equivalent to fully factoring N.

We now describe a general sieving process:

Step 1 : Choose a factor base. Let P be a finite index set, and choose a collection of elements $\{\alpha_p\}_{p \in P}$ ranging over this set, with each $\alpha_p \in (\mathbf{Z}/N\mathbf{Z})^*$. Let $\mathbf{Z}^P$ denote the set of integer vectors $(v_p)_{p \in P}$ ranging over P; and let $f : \mathbf{Z}^P \to (\mathbf{Z}/N\mathbf{Z})^*$ be the group homomorphism mapping $(v_p)_{p \in P} \mapsto \prod_{p \in P} \alpha_p^{v_p}$.

Step 2 : Collect relations. We can think of each vector $v = (v_p)$ in $\ker(f)$ as a *relation* among the α_p, in that $\prod_{p \in P} \alpha_p^{v_p} = 1$. In this step we search for such relations until we have at least $|P|$ of them, and we then hope that the collection we have found is sufficient to generate $\ker(f)$.

Step 3 : Search for dependencies among the relations. Let V be the collection of relations. For each $v \in V$, reduce it modulo 2 coordinate-wise and let $\bar{v} \in \mathbf{F}_2^P$ be the resulting vector. Since we ensured $|V| > |P|$, the resulting vectors $\bar{v}$ cannot be linearly independent over $\mathbf{F}_2$, so we wish to explicitly find the dependencies among them using linear algebra. The matrix is of course, very large and sparse, but traditional Gaussian elimination is usually employed with some optimizations.

We have the convenience of being able to omit coefficients working over $\mathbf{F}_2$, and therefore can write down any relation as a subset $R \subseteq V$ such that $\sum_{v \in R} \bar{v} = 0$. It follows that each coefficient of $r = \sum_{v \in R} v$ is even, and therefore $r/2 \in \mathbf{Z}^P$. Also, since r is a linear combination within V, $r \in \ker(f)$. We can therefore consider the element $x = f(r/2) \in (\mathbf{Z}/N\mathbf{Z})^*$. We have $x^2 = f(r) = 1$.

It may be, of course that $x = \pm 1$, and we have found only a trivial factorization. It is thus our hope that we will have generated enough relations and found enough dependencies between them to completely factor N.

3.4.1 The Rational Sieve

In this algorithm we select a bound B, and choose our factor base to be all primes under B. So we have $P = \{p \leq B : p \text{ is prime}\}$, and choose $\alpha_p = p \bmod N$ for each $p \in P$. We note now that our original desire was to have each α_p a unit of $\mathbf{Z}/N\mathbf{Z}$, and the selection method just given does not guarantee this. However, if we ever do get an $\alpha_p \notin (\mathbf{Z}/N\mathbf{Z})^*$, we have found a nontrivial factor of N, so we do not worry about this possibility.

We then search for integers b such that both b and $N+b$ are B-smooth. Having these, we will have the factorizations of both b and $N+b$ in terms of the α_p. Since $b \equiv N+b \bmod N$, we can use these factorizations to generate a vector $v \in \mathbf{Z}^P$ in the kernel of our function $f : \mathbf{Z}^P \to (\mathbf{Z}/N\mathbf{Z})^*$, thus each such b yields a relation for step 2 above.

Example. Let's use the Rational Sieve to factor the number $N = 1517$. We'll set $B = 35$, which thus gives $P = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31\}$ and since we have $B < N$, the α_p are the same as the elements of P; conveniently (through coaxing of N) each of the α_p is in fact a unit!

By inspection, we find some b such that b and $N+b$ are 35-smooth:

- $b = 2; N + b = 1519 = 7^2 \cdot 31.$
  This gives the kernel vector $(-1, 0, 0, 2, 0, 0, 0, 0, 0, 1)$.

- $b = 3; N + b = 1520 = 2^4 \cdot 5 \cdot 19.$
  This gives the kernel vector $(4, -1, 1, 0, 0, 0, 0, 1, 0, 0, 0)$.

- etc.

We need to find some collection of more than 11 such vectors to guarantee dependencies, and we can do this using $b = 2, 3, 4, 13, 19, 22, 30, 31, 33, 51, 56, 58$; these respectively give the following matrix of kernel vectors

$$\begin{array}{cccccccccc}
-1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 \\
4 & -1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
-2 & 2 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\
1 & 2 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
9 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
-1 & 4 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\
-1 & -1 & -1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
1 & -1 & 2 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\
5 & -1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
-3 & 0 & 0 & -1 & 2 & 1 & 0 & 0 & 0 & 0 \\
-1 & 2 & 2 & 1 & 0 & 0 & 0 & 0 & -1 & 0
\end{array}
\xrightarrow{\bmod 2}
\begin{array}{ccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}$$

We needn't search too hard for dependencies, seeing that the third vector is itself 0. Therefore we consider $f(v_3) = 2^{-2} \cdot 3^2 \cdot 13^2 = 1$, and we have $x = f(v_3/2) = 778$. We have $\gcd(x-1, N) = 37$, and have found a factor of N. $\diamond$

Several other methods of extracting these relations are used in practice today.

3.4.2 The Continued Fraction Method

We have long known how to approximate accurately a given real number with rational numbers by constructing continued fractions. Morrison and Brillhart [31] developed a factoring algorithm based on this ability.

Recall that to evaluate a continued fraction $[a_0, a_1, a_2, \ldots]$ (where $a_i \in \mathbf{Z}$ and $a_i > 0$ for $i > 0$) one proceeds as follows: Initially set $P_{-1} = 1; \quad Q_{-1} = 0; \quad P_0 = a_0; \quad Q_0 = 1$; and then proceed through the continued fraction, iteratively computing

$$P_k = a_k P_{k-1} + P_{k-2}; \quad Q_k = a_k Q_{k-1} + Q_{i-2}.$$

It can then he shown that if $[a_0, a_1, \ldots]$ represents the real number y,

$$\left| \frac{P_k}{Q_k} - y \right| \leq \frac{1}{Q_k^2}. \tag{3.3}$$

Since the Q_i are a strictly increasing sequence of integers, this result guarantees us arbitrarily close rational approximations of low height for any real number.

Returning to the problem of factoring, suppose we are given N to factor. Consider approximating $\sqrt{N}$ using continued fractions. Since we assume N is not a perfect square, we have a continued fraction of an irrational number, which will not terminate but will eventually repeat. Using a variant of the typical algorithm to compute the continued fraction terms a_i from $\sqrt{N}$ which does not rely on knowing $\sqrt{N}$ to extreme precision, we can maintain a third sequence of integers R_i such that for all i,

$$P_i^2 - N \cdot Q_i^2 = (-1)^i R_i; \quad \sqrt{R_i} < 2\sqrt{N}.$$

But from this we have $P_i^2 \equiv (-1)^i R_i \pmod{N}$—a quadratic relation. The term on the right being bounded by $2\sqrt{N}$, we might hope to factor it over a small factor base of primes, and then use the same linear algebra mod 2 to find dependencies, each of which provides a congruence $x^2 \equiv y^2$ (or equivalently in the language of the above description $(xy^{-1})^2 = 1$) which provides a chance to factor N.

3.4.3 The Quadratic Sieve

A cousin of the continued fraction method, the quadratic sieve also searches for relations $x^2 = q$ for q small and factorable over a factor base. However it is more efficient in that it attempts to factor these q using quadratic polynomials, instead of trial division.

Define $f(x) = x^2 - N$, and let $s = \lceil \sqrt{N} \rceil$. Consider now the sequence $f(s), f(s+1), f(s+2), \ldots$, and suppose that we could completely factor each of these. If a prime p divides $f(x)$, we know that $p | x^2 - N$, and therefore unless p divides N, $N \equiv x^2$ is a quadratic residue modulo p. Therefore if we select a factor base be the set S of all the primes p such that each $p < B$ and each such that $\left(\frac{N}{p_i}\right) = +1$ (such that N is a quadratic residue modulo p_i), and if we can find $R > |S|$ values $f(x)$ which are B-smooth, it follows that we can factor these values among the p_i. Then the linear algebra techniques modulo 2 will again give quadratic relations which might factor N.

We would like to be able to factor $f(s+i)$ using a sieve – that is, to be able to identify which p will divide it without trial division. So how can we know if $p \mid f(s+i)$? Since $f(x) = x^2 - N$, this happens if and only if $(s+i)^2 \equiv N \pmod{p}$. But all positive solutions to $x^2 \equiv N \pmod{p}$ lie in one of two arithmetic sequences $\{r_1 + kp\}_{k \geq 0}, \{r_2 + kp\}_{k \geq 0}$, where r_1 and r_2 are the principal square roots of N mod p (that is, the ones in $\{0, 1, \ldots, p-1\}$), which we know exist because $\left(\frac{N}{p}\right) = +1$.

These principal roots can be found efficiently (since p is prime), and therefore we can predetermine all the i for which $f(s+i)$ will be divisible by p. We thus proceed through our factor base, dividing out each p_j from the $f(s+i)$ which it divides. Trial division is therefore avoided—we know a priori which divisions will be successful.

If we perform the sieve over i in some range $a < i < b$, (for $a > s$) we expect that number of divisions required for each prime p is therefore about $(2/p)(b-a)$, since 2 out of every p numbers are square roots of N modulo p. So if we sieve out all primes less than B, it will take us $(b - a) \cdot \sum_{p<B}(2/p)$, and this sum is $O(\log\log B)$, substantially less than the $O(B/\log B)$ required for trial division in the continued fraction method.

Wagstaff [50] notes that this process can be made more efficient by storing the logarithms of $f(a+i)$ in an array instead of the values themselves. Then division by p becomes the more efficient subtraction by $\log p$. At the end, we search the array for small values (not quite zero because of some other optimizations— not actually wasting our time sieving small primes but instead sieving higher powers of them). When we do see a value below whatever threshold we decide upon (and this can be updated dynamically depending on our success), we can reconstruct $f(a+i)$ and actually factor it, using our knowledge of the square roots of N mod p to assist. If we do factor it completely over our factor base, we have a relation, and we repeat and proceed as above to factor N with dependencies among these relations modulo 2.

Consider the running time of the algorithm outlined above. Following Wagstaff [50], we estimate the size of the polynomial values $f(s+i)$ from which we would like to find B-smooth numbers. If $i < M \ll \sqrt{N}$, we have

$$f(s+i) = (s+i)^2 - N = s^2 + 2si + i^2 - N \approx 2si + i^2 \approx 2si < 2M\sqrt{N}.$$

We recall that in the continued fraction method we hoped to find $R_i < 2\sqrt{N}$ which were smooth; for the quadratic sieve we are searching for numbers M times larger.

By the analysis of section 3.1.1, if assume that the $f(s+i)$ are roughly the size of $\sqrt{N}$, then the probability that $f(s+i)$ is B-smooth is approximately u^{-u}, where $u = (\log\sqrt{N})/(\log B) = \log N/2\log B$. We would like to choose a value for B which minimizes the total work required to find these smooth $f(s+i)$. We noted earlier that the time on average it takes test each i is roughly $\log\log B$. Since the probability that i leads to a B-smooth value is u^{-u}, we expect to have to try $u^u(K+1)$ different i to succeed. Let $K = |S|$ be the size of our factor base, and since we choose this base to be all primes $p < B$ with $\left(\frac{N}{p}\right) = +1$, $K \approx \pi(B)/2 = O(B/\log B)$. Therefore if we let (as in Crandall and Pomerance [9]) $T(B)$ be the expected amount of work using bound B,

$$T(B) = u^u(K+1)\log\log B, \ \rightarrow \ \log T(B) = u\log u + \log B - \log\log B + \log\log\log B,$$

and we will assume that as B gets large enough this is dominated by $S(B) = u \log u + \log B$. It follows that we can minimize the work by find a zero of

$$S'(B) = \frac{-\log N}{2B(\log B)^2}(\log \log N - \log \log B - \log 2 + 1) + \frac{1}{B} \quad \text{(from [9])}.$$

This leads to the optimal choice of B of $\exp(\frac{1}{2}(\log N)^{1/2}(\log \log N)^{1/2}) = L(N)^{1/2}$, and the running time is about $B^2 = L(N) = \exp((\log N)^{1/2}(\log \log N)^{1/2})$.

We note that the analysis and the eventual result here are both very similar to the ECM method described earlier, however we reiterate the distinction that ECM's complexity is in terms of p, and here we are in terms of N. In practice, the quadratic sieve is useful for factoring much larger integers than the ECM, especially those formed of 2 prime factors of similar size, although the latter method may get quite lucky at factoring large integers with medium sized prime factors.

3.4.4 The Number Field Sieve and Factoring

Though the number field sieve is most commonly presented as a natural sequel to the quadratic sieve, we can conceive of it perhaps more accurately as a generalization of the rational sieve introduced at the beginning of this section. In the quadratic sieve, we seek modulo N congruences of the form $x^2 \equiv Q$, where Q is something we can easily factor over our factor base. In the rational sieve we sought relations of the form $p \equiv Q$, where p was a small prime, thus already factored over the factor base, as was Q. We can generalize to seeking relations of the most general form $P \equiv Q$, where both P and Q are easily factored over the factor base. It is clear that this will be enough to generate relations and dependencies to factor N.

The number field sieve is due to Pollard who first proposed the idea in 1988, much later than his earlier exponential offerings of the $p-1$ and rho methods. For an account of the development, see [21] and the other papers in the same volume edited by Lenstra and Lenstra.

In the number field sieve, we attempt to find such relationships by leaving the $\mathbf{Z}/N\mathbf{Z}$ world (temporarily) and moving into an extension field $K = \mathbf{Q}(\alpha)$. In order to keep ourselves grounded in integers, and specifically to be able to write relations modulo N, we will need a ring homomorphism $h : \mathcal{O}_K \to \mathbf{Z}/N\mathbf{Z}$ on the ring of integers of K. Then by working with irreducible algebraic integers β such that we can easily factor $h(\beta)$ over the integers, we will be able to convert relations over K involving integers expressed as a product of such β into useful relations modulo N which we can use to factor.

To construct $K = \mathbf{Q}(\alpha)$, we need a monic irreducible polynomial

$$f(X) = X^d + c_{d-1}X^{d-1} + \cdots + c_1 X + c_0$$

with integer coefficients, and we let α be a complex root of f. We then have $\mathbf{Q}(\alpha)$, and we will also make use of $\mathbf{Z}[\alpha]$, a subring of the ring $\mathcal{O}_K$ of algebraic integers. To define our homomorphism, we would like to know a root r of f modulo N (that is an r such that $f(r) \equiv 0 \pmod{N}$), and then if we define $h(\alpha) = r$, we will indeed induce a homomorphism, and so for any $\beta = \sum_{i=0}^{d-1} b_j \alpha^j$ in $\mathbf{Z}[\alpha]$, we have $h(\beta) = \sum_{i=0}^{d-1} b_j r^j \pmod{N}$.

Typically, to select such a polynomial, we select a degree d, and then let r be an integer near $N^{1/d}$ (but below it). We then write N in base r, $N = \sum_{i=0}^{d} c_i r^i$. It follows that $f(X) = \sum_{i=0}^{d} c_i X^i$ has r as a root modulo N. [This makes the polynomial easy to find in cases where N is of a special form $s^e + t$ for small s and $|t|$, in which case this algorithm can be optimized and is called the Special Number Field Sieve.] We can assume that $f(X)$ is irreducible, for if not $N = f(r) = g(r)h(r)$ gives a nontrivial factor.

Reminding us of our goal, it follows that (as a simple example) if we can find a and b such that $a - br = x^2$ is a square modulo N *and* $a - b\alpha = \gamma^2$ is a square in $\mathcal{O}_K$, then if we let $y = h(\gamma)$ it follows that

$$y^2 \equiv h(\gamma)^2 \equiv h(\gamma^2) \equiv h(a - b\alpha) \equiv a - br \equiv x^2 \pmod N$$

and we have a chance of factoring N.

In general, we choose a factor base of "small" algebraic integers of the form $a_i - b_i\alpha$, and our search for relations is a search for *sets* S of pairs a, b such that the generalized above example holds, that is so that $\prod_{(a,b)\in S}(a - b\alpha)$ is a square in $\mathbf{Z}[\alpha]$ and $\prod_{(a,b)\in S}(a - br)$ is a square modulo N. Such examples will occur as modulo 2 linear dependencies, if we can find enough a, b pairs such that $a - b\alpha$ and $a - br$ can be written as a product of elements in our factor base.

To do this, we extend the notion of smoothness to $\mathcal{O}_K$, and we say that an algebraic integer γ is B-smooth if its rational norm $N(\gamma)$ is B-smooth. Wagstaff [50] demonstrates how to easily compute the norm of an algebraic integer $a + b\alpha$. He notes that if $\alpha_1, \ldots, \alpha_d$ are the d complex roots of f, they are the conjugates of α, and thus $a - b\alpha_1, \ldots, a - b\alpha_d$ are conjugates of $a - b\alpha$. Therefore

$$N(a - b\alpha) = \prod i = 1^d (a - b\alpha_i) = b^d \prod_{i=1}^{d} (a/b - \alpha_i) = b^d f(a/b).$$

It follows that if we define $F(x, y) = y^d f(x/y)$, which is the homogeneous polynomial

$$x^d + c_{d-1}x^{d-1}y + \cdots + c_1 x y^{d-1} + c_0 y^d,$$

then $N(a - b\alpha) = F(a, b)$. It follows that we can sieve this polynomial in order to find pairs a, b such that $a - b\alpha$ has a smooth norm.

Since we have seen sieving already in some detail, we will not go any deeper into the specifics of how the algorithm proceeds. The literature on the number field sieve is extensive, and the process has many unexpected complications, none of which would offer particular insight upon elaboration into the relationship between factoring and discrete log. For reference into these specific issues, consider [21] and the relevant chapters of [50], [9], or [8].

The one unmentioned issue which will highlight the relationship is the complexity of the number field sieve. Our analysis parallels Crandall and Pomerance's in [9]. As in the quadratic sieve, our main concern is in the probability that the numbers we want to be smooth are smooth. We saw above that the smoothness bound $B = L_{1/2}[N]$ leads to the optimal situation, and this can actually be proven under the heuristic assumption that the values we test for smoothness are random. In the quadratic sieve these were the values $f(s + i)$; here they are the values $F(a, b)$ from above and also $G(a, b) = a - br = h(a - b\alpha)$. Thus we wish to find smooth values $F(a, b)G(a, b)$.

Now assume that the root r is no larger than $N^{1/d}$ (d is the degree of f), and that the coefficients of f are equally bounded by $N^{1/d}$, and that we are searching for a, b-pairs in the range $|a|, |b| \leq M$. It follows that

$$|F(a,b)G(a,b)| \leq 2(d+1)M^{d+1} \cdot N^{2/d}.$$

If we call this quantity X, then by Pomerance's theorem under our heuristic assumptions, we would expect to require testing $L_{1/2}[X]^{\sqrt{2}+o(1)}$ a, b-pairs to succeed. We can see exactly that number of pairs if we make our bound M such that $M^2 = L_{1/2}[X]^{\sqrt{2}+o(1)}$. Then substituting into the above definition of X and taking the log of both sides,

$$\log X = \log 2 + \log(d+1) + \frac{2}{d}\log N + (d+1)\sqrt{\frac{1}{2}\log X \log\log X}.$$

The first two terms on the right are dwarfed by the last, so we ignore them. Now let us let $N \to \infty$ and assume that $d \to \infty$ as well. This leaves us with

$$\ln X = \frac{2}{d}\log N + d\sqrt{\frac{1}{2}\log X \log\log X},$$

and taking the derivative with respect to d gives

$$\frac{X'}{X} = \frac{-2}{d^2}\log N + \sqrt{\frac{1}{2}\log X \log\log X} + \frac{dX'(1+\log\log X)}{4X\sqrt{\frac{1}{2}\log X \log\log X}},$$

which gives $X' = 0$ for $d = (2\log N)^{1/2}((1/2)\log X \log\log X)^{-1/4}$. Substituting back in gives $\frac{3}{4}\log\log X \sim \frac{1}{2}\log\log N$, or

$$\log X \sim (64/9)^{1/3}(\log N)^{2/3}(\log\log N)^{1/3}.$$

The running time of the algorithm is then

$$L_{1/2}[X]^{\sqrt{2}+o(1)} = L_{1/3}[c; N],$$

with $c = (64/9)^{1/3} + o(1)$.

3.5 Sieving for Discrete Logarithms

It turns out that both the general concept of sieving and some of the specific variants just seen for factoring apply well to solving the discrete log problem as well.

3.5.1 The Index Calculus Method

The basic construction of a sieve which we saw on page 29 can be adapted easily to find discrete logarithms instead of factors. The resulting algorithm is called the *index-calculus* method, which dates to ideas in the work of Kraitchik in the 1920s and in current form is due to multiple individuals including Cunningham and Adleman.

The form is as follows. We present it over the group $(\mathbf{Z}/p\mathbf{Z})^*$, but note that it could be used generally to solve logarithms over any cyclic group. The high-level goal of the algorithm is to build up knowledge about the logarithms of some elements of the group which generate a significant portion of the group, and use these logarithms to solve easily for the logarithm we don't know.

Step 1 : Choose a factor base. Let $S = \{p_1, \dots, p_s\}$ be a subset of $(\mathbf{Z}/p\mathbf{Z})^*$, which we choose in hopes that it will be enough to generate all of $(\mathbf{Z}/p\mathbf{Z})^*$.

Step 2 : Collect relations among the logarithms of the p_i. To do this, we create a random element whose logarithm we know, and hope it is generated by S. Specifically, choose $b \in (\mathbf{Z}/p\mathbf{Z})^*$ at random, and compute g^b. Attempt to write g^b as a product of elements of the factor base:

$$g^b = \prod_{i=1}^{s} p_i^{e_i}, \quad e_i \geq 0.$$

If this was successful, then we have a linear relation on the logarithms: $b = \sum e_i \log_g p_i$. Since this is taking place in the exponents, it is a congruence modulo $p-1$, or generally in the order of the cyclic group being used. If we were unsuccessful, we try again for a new b. Since we are trying to solve for each of the $\log_g p_i$, we will be working in s unknowns; therefore we should repeat until we have a collection of more than s relations (say $s+k$) of the above form.

Step 3 : Solve for the $\log_g p_i$. We do this as hinted above by solving the linear system of equations modulo $p-1$ given by the relations collected in step 2. We hope that the $s+k$ relations will not be dependent, and therefore we can get a unique answer.

Step 4 : Use the known logarithms to solve $y = g^a$. To do so, select b at random from $(\mathbf{Z}/p\mathbf{Z})^*$. Then calculate $y \cdot g^b$, and hope that it is generated by elements of S whose logarithms we know. If it is not, then repeat with a different b, if it is we can write $y \cdot g^b = \prod_i p_i^{c_i}$, which implies the linear congruence

$$a + b \equiv \sum_i (d_i \log_g p_i) \pmod{p-1}.$$

Knowing all these elements except a, we can solve for a and we have the logarithm.

Example. Let $p = 307$, and $g = 5$, which is a generator of $(\mathbf{Z}/307\mathbf{Z})^*$. Suppose we are given $y = 214$, and we will use the index-calculus method to find a such that $y = g^a \pmod{307}$.

We will begin by choosing our factor base to be the first few prime numbers: $S = \{2, 3, 5, 7, 11, 13\}$. Let l_p denote the logarithm of p to the base g, which we initially do not know (except $l_5 = 1$).

1. Now we choose some random b, say $b = 30$. We have $g^b = 54 = 2 \cdot 3^3$, so we have the relation $30 \equiv l_2 + 3l_3 \pmod{306}$. Similarly,

2. $b = 55 \rightarrow g^b = 120 = 2^3 \cdot 3 \cdot 5 \rightarrow 55 \equiv 3l_2 + l_3 + l_5$.
3. $b = 39 \rightarrow g^b = 128 = 2^7 \rightarrow 39 \equiv 7 \cdot l_2$.
4. $b = 118 \rightarrow g^b = 175 = 5^2 \cdot 7 \rightarrow 118 \equiv 2l_5 + l_7$.
5. $b = 156 \rightarrow g^b = 182 = 2 \cdot 7 \cdot 13 \rightarrow 156 \equiv l_2 + l_7 + l_{13}$.
6. $b = 259 \rightarrow g^b = 198 = 2 \cdot 3^2 \cdot 11 \rightarrow 259 \equiv l_2 + 2l_3 + l_{11}$.

and while we don't have any more than s relations, we might feel pretty good about them, and in fact we can do some manual linear algebra to solve this system: from (3): $l_2 \equiv 39/7 \pmod{306} = 93$; from (1): $l_3 = 81$; from (2): $l_5 = 1$; from (4): $l_7 = 116$; from (6): $l_{11} = 4$; from (5): $l_{13} = 253$.

With these in hand, we try to actually take the log of y. So choose at random $b = 155$ (it took me 3 guesses to find one that would work), and we have $g^b \cdot y = 176 = 2^4 \cdot 11$. It follows that, modulo 306, $155 + a \equiv 4l_2 + l_{11} \equiv 4(93) + 4 \equiv 70$, so $a = 221$. $\diamond$

3.5.2 The Number Field Sieve and Discrete Log

The above index calculus discrete method closely parallels the quadratic sieve method for factoring; similarly the modification of the factoring sieve to use number fields also has a discrete log corollary. Here there is no particular name, and the method retains the title of number field sieve. The principal work in this area was presented by Daniel Gordon in 1993 [19] and improved by Schirokauer in 1999 [44].

The algorithm proceeds very much like its factoring predecessor. Given a prime p and a generator g which are the parameters of our problem, we choose an irreducible monic polynomial $f(x)$ over $\mathbf{Z}$ with a known root $r \in \mathbf{Z}$ such that $f(r) \equiv 0 \pmod{p}$. We also require that p not divide the discriminant of f.

Now let α be a complex root of f, define $K = \mathbf{Q}(\alpha)$ and let $\mathcal{O}_K$ be the ring of integers in K. Since will be looking to "factor" in K, we invoke the following theorem ([19] Prop. 1)

Proposition 3.6 *If q is a prime number not dividing the index $[\mathcal{O}_K : \mathbf{Z}[\alpha]] = |\mathcal{O}_K/\mathbf{Z}[\alpha]|$, and if $f(x)$ factors modulo s as the product of distinct, irreducible monic polynomials $f(x) = \prod g_i(x)^{e_i} \pmod{s}$, then the ideal (q) factors as $(q) = \prod \mathfrak{s}_i^{e_i}$ for distinct prime ideals $\mathfrak{s}_i = (s, g_i(\alpha))$, which have norm $N(\mathfrak{s}_i) = s_i^d$, where d_i is the degree of g_i.*

It follows from this that the ideal $\mathfrak{p} = (p, \alpha - r)$ divides (p), since $x - r$ divides $f(x)$ modulo p. Since it has degree 1, $N(\mathfrak{p}) = |\mathcal{O}_K/\mathfrak{p}| = p$, and in fact $\mathcal{O}_K/\mathfrak{p} \cong \mathbf{F}_p$. We say that a "good" prime ideal is one whose norm does not divide the index $[\mathcal{O}_K : \mathbf{Z}[\alpha]]$, and there exist efficient ways of determining if (p) is a good prime ideal.

We will want our factor base to consist of such good prime ideals of low norm, and Gordon gives us a way of finding ideals which will factor over this factor base.

Proposition 3.7 ([19] - Prop. 2) *If c and d are integers with $\gcd(c, d) = 1$ and $N(c + d\alpha)$ is relatively prime to the index $[\mathcal{O}_K : \mathbf{Z}[\alpha]]$, then the ideal $(c+d\alpha)$ in $\mathcal{O}_K$ can be written as a product of good prime ideals of degree 1.*

Therefore, to take discrete logarithms over $\mathbf{F}_p$, we build a factor base with two parts. One part B_0, consists of integer primes less than our smoothness bound B, and a second part, B_K consists of good prime ideals of degree 1 and with norm less than B.

We then sieve over pairs of integers (c, d) such that both $c + dr$ and $(c + d\alpha)$ can be written as a products of terms in our factor bases B_0 and B_K respectively. To do the latter we actually sieve to find $N(c + d\alpha)$, for suppose that

$$c + dr = \prod_{p_i \in B_0} p_i^{e_i} \quad \text{and} \quad |N(c + d\alpha)| = \prod_{p_i \in B_0} p_i^{e'_i}.$$

Then by the above proposition for each i with $e'_i > 0$ we can find a unique ideal $\mathfrak{p}_i$ containing p_i and dividing $c + d\alpha$. Define $e'_{\mathfrak{p}_i} = e'_i$ for this ideal and 0 for all other ideals of norm p_i. Then the above becomes

$$c + dr = \prod_{p_i \in B_0} p_i^{e_i} \quad \text{and} \quad (c + d\alpha) = \prod_{\mathfrak{p}_i \in B_K} \mathfrak{p}_i^{e'_{\mathfrak{p}_i}}.$$

We are know well poised to search for dependencies by solving a linear system of the exponent vectors e_i and $e'_{\mathfrak{p}_i}$ once we have slightly more than $|B_0| + |B_K|$ pairs c, d with the above property. Such a dependency will be a set S of pairs c, d such that

$$\prod_{(c,d)\in S} (c + d\alpha)^{e(c,d)} = u, \ u \text{ a unit} \quad \text{and} \quad \prod_{(c,d)\in S} (c + dr)^{e(c,d)} \text{ is } B\text{-smooth}.$$

With enough such sets, we can find some union $\mathcal{S} = S_1 \cup \cdots \cup S_j$ such that the units u cancel out and we have $\prod_{(c,d)\in\mathcal{S}}(c+d\alpha)^{e(c,d)} = 1$, and of course the corresponding product of the $(c+dr)$ will still be B-smooth. Recalling the homomorphism $h : \mathcal{O}_K \to \mathbf{Z}/p\mathbf{Z}$ which sends $\alpha \mapsto r$, we have

$$\prod_{(c,d)\in\mathcal{S}} (c + dr)^{e(c,d)} \equiv \prod_{(c,d)\in\mathcal{S}} h(c + d\alpha)^{e(c,d)} \equiv 1 \pmod{p}.$$

Since we know how to factor $c+dr$ over B_0 (this was how we selected the c, d pairs), we have some relation

$$\prod_{p\in B_0} p^{s(p)} \equiv 1 \pmod{p}.$$

Therefore, taking the discrete log both sides with respect to the generator g,

$$\sum_{s\in B} s(p) \log_g p \equiv 0 \pmod{p - 1}.$$

Therefore with sufficiently many sets $\mathcal{S}$, we can solve a linear system modulo $p - 1$ and determine all the $\log_g p$ for $p < B$. Steps 3 and 4 of the above index calculus method can then be applied to find $\log_g y$.

The running time for this version of the number field sieve parallels that of the factoring algorithm; Gordon [19] shows that optimal values for the bound B and the size of the root m are $L_p[1/3; c]$ for some values of c, and then derives the optimal running time of $L_p[1/3, 3^{2/3}]$. We note that since the algorithm Gordon proposes relies on the elliptic curve method to find smooth integers near p, this running time is based on the same heuristics Lenstra assumed that the distribution of B-smooth integers in the Hasse interval around p is the same as the general distribution $\psi(p, B)$.

3.6 Conclusions

The topic of this paper was originally motivated by the observations just seen in the preceding chapter that the same ideas can be applied to solve two seemingly disjoint problems in number theory, and that when programmed as algorithms, the running times are the same. This remains remarkable, given that no concrete mathematical reason to expect this occurrence has yet been proved.

We have seen in Chapter 2 the importance that the difficulty of these problems plays in modern systems, and as suggested in Chapter 1 one of the reasons to consider the relationships of these problems is out of concern that our security assumptions may not be sufficiently diversified among independent hard problems.

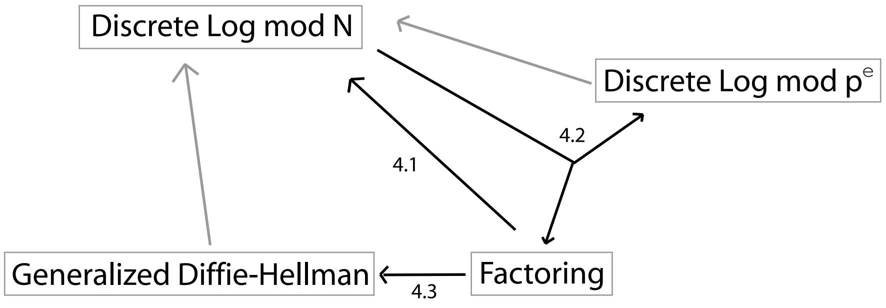
Practically, from the algorithmic point of view, the evidence presented in this chapter leads us to conjecture that the hardnesses of factoring and discrete log are not independent, and that if in one year's time an even faster factoring method were discovered, running in say time $L_N[1/5, c]$, not long from that date would the $L_p[1/5, c]$ discrete log algorithm be unveiled.

These observations of comparable running time, based purely on the subset of yet-discovered algorithms, should not be confused with the claim that factoring and discrete log have been proven to have equivalent complexity, though this confusion is not uncommon in the world of practiced cryptography. (Or, in the related form "If factoring were tractable, all number-theoretic cryptography would be lost" is a common misconception.) Let it remain clear that this statement has not been proven.

4. The Relationship Between the Problems II – Theoretical

Having now seen evidence of a historical and empirical nature that the efficiency of these two problems is closely correlated, we turn to the mathematics behind the problems in search of a rigorous connection.

We will show, in turn, the three reductions figured below. In the diagram, an arrow from $X \to Y$ indicates that problem X reduces to problem Y, which is in general to say that it is possible to solve X given the ability to solve Y, or that "Y implies X". As mentioned in the opening chapter, this can take several different specific meanings depending on context, and we leave the discussion of that issue to each individual section. The branching arrow $X \to (Y, Z)$ is possibly non-standard, and we wish for it to mean that the problem X is solvable if both Y and Z are solvable.



The two light arrows indicate trivial reductions between these problems, and are included only to make our graph complete; we will not specifically prove them here. For a more complete picture of reductions between number theoretic problems other than factoring and discrete log (and specifically easier than them), see Heather Woll's 1987 paper [51].

4.1 Composite Discrete Log implies Factoring

Problem 4.1 (Composite Discrete Log) *Given an integer N, not necessarily prime, an element g of $(\mathbf{Z}/N\mathbf{Z})^*$, and an element $y \in (\mathbf{Z}/N\mathbf{Z})^*$ such that $y = g^a$ for some a, to find the integer a.*

Parallel to the discussion of section 1.3, we can also frame this problem probabilistically, by defining an instance generator and the notion of ε-solving the problem according to those instances.

$\mathcal{D}^C(1^k)$:

- Generate $N \leftarrow \mathcal{F}_k$.
- Choose g with high order in $(\mathbf{Z}/N\mathbf{Z})^*$.
- Choose a at random from $(\mathbf{Z}/\phi(N)\mathbf{Z})^*$, and compute $y = g^a \bmod N$.
- Output (N, g, y).

Some care must be taken over the meaning of "high order" as applied to elements of $(\mathbf{Z}/N\mathbf{Z})^*$. Given just $N \leftarrow \mathcal{F}_k$, we do not know how to find such a g. So we assume that during the operation of $\mathcal{D}^C$ we have access to the intermediate values used in the selection of N from $\mathcal{F}$, specifically the prime factors p and q of N and the factorizations of $p-1$ and $q-1$. It is easy to find generators g_1 of $(\mathbf{Z}/p\mathbf{Z})^*$ and g_2 of $(\mathbf{Z}/q\mathbf{Z})^*$ these intermediate values, and the Chinese Remainder Theorem gives us an element g in $(\mathbf{Z}/N\mathbf{Z})^*$ which is of maximal order $\operatorname{lcm}(p-1, q-1)$. So we take this as our meaning of "high order". See [29] section 4.6 for details.

Problem 4.2 (Solving Composite DL with probability ε) *Given a function $\varepsilon(k)$, to construct a probabilistic polynomial-time algorithm A such that for any integer k,*

$$\Pr[g^{A(N,g,y)} \bmod N = y] \geq \varepsilon(k),$$

taken over all $(N, g, y) \leftarrow \mathcal{D}^C(1^k)$.

Although I name it "Composite Discrete Log," we do not require in the definition that N be composite. Therefore, considering that when p is prime, every element of $(\mathbf{Z}/p\mathbf{Z})^*$ is a generator, and therefore every $y \in (\mathbf{Z}/p\mathbf{Z})^*$ is some power of g, this problem is trivially no easier in the worst case than the discrete log problem using a prime modulus which we defined in Problem 1.2 and have been discussing all along.

However, we can establish our first theoretical connection to factoring, which is slightly less expected, that factoring can be accomplished by taking discrete logs. This result was shown by Eric Bach [1] in 1984, extending earlier work of Gary Miller from 1976 [30]. The proof below, which is a worst case reduction, is expanded from that in Bach's paper. We follow it with our own randomized reduction, using an adaptation of Bach's method.

Proposition 4.3 (Bach, Miller, et al.) *If there exists a polynomial-time algorithm A to solve the Composite Discrete Log problem on all inputs, then a polynomial-time algorithm exists which solves the Factoring problem with arbitrarily high probability.*

Proof of Proposition. We assume that N is odd and not a prime power. These conditions can be efficiently tested and, if they fail to hold, lead immediately to a factor of N.

Consider the structure of $(\mathbf{Z}/N\mathbf{Z})^*$. If N has prime factorization $p_1^{e_1} \cdots p_s^{e_s}$, then

$$(\mathbf{Z}/N\mathbf{Z})^* = (\mathbf{Z}/p_1^{e_1}\mathbf{Z})^* \times \cdots \times (\mathbf{Z}/p_s^{e_s}\mathbf{Z})^*.$$

This group has order $\phi(N) = \phi_1 \cdots \phi_s$, each $\phi_i = \phi(p_i^{e_i}) = p_i^{e_i-1}(p_i - 1)$, however it is not cyclic, and there is no element of order $\phi(N)$. However, if we raise any $a \in (\mathbf{Z}/N\mathbf{Z})^*$ to a power e such that each $\phi_i \mid e$, it follows that $a^e \equiv 1 \bmod p_i^{e_i}$ for each i, and therefore $a^e \equiv 1 \bmod N$. So each element of $(\mathbf{Z}/N\mathbf{Z})^*$ has order dividing $L = \mathrm{lcm}(\phi_1, \ldots, \phi_s)$.

We prove our result in two stages. First we show that given the ability to find the order of elements in $(\mathbf{Z}/N\mathbf{Z})^*$, we can factor N. Then we demonstrate how to compute orders by taking discrete logs mod N.

Our first goal is to factor N by determining the order of an element. We've established that $x^L \equiv 1$ for all x, but as mentioned earlier, since N is composite we know there are square roots of 1 other than ± 1. Therefore consider the subgroup K of $(\mathbf{Z}/N\mathbf{Z})^*$:

$$K = \{x \in (\mathbf{Z}/N\mathbf{Z})^* : x^{L/2} \equiv \pm 1 \pmod N\}.$$

We can show that $K \neq (\mathbf{Z}/N\mathbf{Z})^*$ as follows. First, define the symbol $e_2(k)$ for $k \in Z$ to be the exponent of 2 in the prime factorization of k, or equivalently the highest power of 2 which divides k. Now sort the p_i using this symbol, such that $e_2(\phi_1) \geq e_2(\phi_i)$ for all i. It follows that $e_2(\phi_1) = e_2(L)$. Since each group $(\mathbf{Z}/p_i^{e_i}\mathbf{Z})^*$ is cyclic, choose generators $g_1, \ldots, g_s$. Then let $a \in (\mathbf{Z}/N\mathbf{Z})^*$ have coordinates $(g_1, g_2^2, \ldots, g_s^2)$. It follows that

$$a^{L/2} = (g_1^{L/2}, g_2^L, \ldots, g_s^L) = (g_1^{L/2}, 1, \ldots, 1)$$

is not ± 1, since g has order ϕ_1, of which $L/2$ cannot be a multiple since $e_2(L/2) = e_2(\phi_1) - 1$. Therefore $a \notin K$. Since K is thereby a proper subgroup it follows that at least half of the elements of $(\mathbf{Z}/N\mathbf{Z})^*$ are outside of K.

Now suppose that $x \in (\mathbf{Z}/N\mathbf{Z})^*$ but $x \notin K$, and that we have some m such that $x^m \equiv 1$. Consider the sequence $x^m, x^{m/2}, x^{m/4}, \ldots, x^{m/2^{e_2(m)}}$. I claim that for some k, we must have $x^{m/2^k} \equiv 1$, but $x^{m/2^{k+1}} \not\equiv \pm 1$. Suppose the actual order of x in $(\mathbf{Z}/N\mathbf{Z})^*$ is r. It follows that $L = c_0 \cdot r$ for an integer c_0. Furthermore, since $x \notin K$, we know $x^{L/2} \not\equiv 1$, therefore c_0 must be odd. Also, $m = c \cdot r$, and we can pull out all the 2s in c to get $m = 2^{e_2(c)} \cdot c_1 \cdot r$ for c_1 odd. Therefore if we make $k = e_2(c)$, we see $x^{m/2^k} \equiv x^{c_1 r} \equiv 1$, and consider $x^{m/2^{k+1}}$. Observe now that for any odd d, whenever $z^2 \equiv 1 \pmod N$, then we must have $z \equiv \pm 1 \pmod{p_i^{e_i}}$ for each i, and since both ± 1 are their own dth power, $z^d \equiv z \pmod{p_i^{e_i}}$, and so $z^d \equiv z \pmod N$ as well. It follows that $x^{L/2} \equiv x^{c_0 \cdot r/2} \equiv x^{r/2}$ (since c_0 is odd and $x^{r/2}$ is a square root of 1). Likewise $x^{m/2^{k+1}} = x^{c_1 r/2} \equiv x^{r/2}$. Therefore

$$x^{m/2^{k+1}} \equiv x^{L/2} \not\equiv \pm 1.$$

This proves the claim. [1]

[1] Since this claim holds only because N is composite, the sequence can also be used to check primality; the resulting method is known as the Miller-Rabin primality test. See [30].

It follows that $x^{m/2^{k+1}}$ is a non-trivial square root of 1, therefore we can factor N by taking $\gcd(x^{m/2^{k+1}} \pm 1, N)$ as in Proposition 3.5.

We are only left with demonstrating that an "exponent"[2] of an element can be found with an oracle for composite discrete log. We note that it is not enough to ask for the discrete logarithm of 1 to the base x, since $m = 0$ is a valid exponent which will not lead to a usable sequence in the above method, 0 having a not-well-defined number of 2s dividing it.

But suppose that p is a prime such that $\gcd(p, \phi(N)) = 1$. If so, there would exist an integer solution q to the congruence $pq \equiv 1 \bmod \phi(N)$. Then by Fermat, $x^{pq} \equiv x \bmod N$, and so we could ask for the discrete log of x to the base x^p, get a valid answer q, and then take $m = pq - 1$ as our exponent of x.

Of course, we do not even know $\phi(N)$, so we cannot specifically test this condition on p; however algorithmically we can test by asking for $\log_{x^p} x$ for p and seeing if the q we get back satisfies $x^{pq} \equiv x$. Since $\phi(N) < N$, we can guarantee that there must be a prime p which is relatively prime to $\phi(N)$ among the first $\log N + 1$ primes. Therefore we can complete a search for a p such that $\log_{x^p} x$ exists in polynomial time.

Since we know $x \notin K$ with probability $\geq 1/2$ for a random selection of x, it follows that our algorithm, if repeated ℓ times, will fail to provide a factor with probability $\leq 1/2^\ell$. Therefore we have an arbitrarily high probability of success. ■

We repeat the observation that this proof as given by Bach [1] is a worst-case reduction, not probabilistic, first because we allow N to be any integer with any prime factorization, and also because we assume the ability to take arbitrary discrete logs to any base modulo N. However we can extend it without too much trouble into a probabilistic reduction in terms of our instance generators $\mathcal{F}$ and $\mathcal{D}^C$.

Proposition 4.4 *If A is a probabilistic polynomial time algorithm which solves the Composite Discrete Log problem (4.2) with non-negligible probability ε, then there exists a probabilistic polynomial algorithm B which solves the Factoring problem (1.12) with non-negligible probability.*

Note that the two problems referred to above draw the composite integers N from the same distribution $\mathcal{F}_k$.

Proof. We condense our proof of proposition 4.3, with some modifications, into an algorithm.

```
$B(N)$:
  1. Select $x \xleftarrow{R} (\mathbf{Z}/N\mathbf{Z})^*$.
  2. Let $H = N^2 - 2N\sqrt{N} + N$, and select $r \xleftarrow{R} \{2, \ldots, H\}$.
  3. Let $y = x^r \bmod N$
  4. Use $A$ to compute $s = A(N, y, x)$
```

[2]Terminology is Bach's [1]: an *exponent* of x is any y such that $x^y \equiv 1$.

5. If $x^{rs-1} \not\equiv 1$, fail.
6. Let $m = rs - 1$.
7. Compute the sequence $x^m, x^{m/2}, \ldots$ (all mod N) until impossible, or until for some t, $x^{m/2^{t+1}} \not\equiv \pm 1$ while $x^{m/2^t} \equiv 1$.
8. If such a t was found, output $d = \gcd(x^{m/2^{t+1}} - 1, N)$.

As in the above proof, we know that if $x \notin K$, once we have $x^m \equiv 1$ we are home free, and guaranteed to get a non-trivial factor of N. We asserted that for a random $x \in (\mathbf{Z}/N\mathbf{Z})^*$, $\Pr[x \notin K] \leq 1/2$, so we are left with the probability that A will lead us to a valid m. We know that this happens if $x^{rs} = x$; therefore if A successfully finds $s = \log_{x^r} x$, and we know that this s exists whenever $\gcd(r, \phi(N))$ is 1. Since the instance generator $\mathcal{D}^C$ always creates instances where the exponent a is relatively prime to $\phi(N)$, if $\gcd(r, \phi(N))$ is not 1, we present A with invalid input—note the fact that A solves CDL with probability ε does not imply any behavior on invalid input, all we know is that A is correct with probability ε given a random input according to $\mathcal{D}^C$ (that is, an input where N, the *generator* and the *exponent* were chosen at random). It is clear that since are showing B solves factoring with probability ε, then N will already be random according to $\mathcal{F}$.

We now limit the consideration to r relatively prime to $\phi(N)$. Let $N = p \cdot q$, since it came from $\mathcal{F}$. In this case, it follows that r is relatively prime to $\phi(p)$ and to $\phi(q)$. Thus r has an inverse $s_p \in (\mathbf{Z}/p\mathbf{Z})^*$, and so every element $y \in (\mathbf{Z}/p\mathbf{Z})^*$ is of the form x^r for some unique x, namely $x = y^s$. Likewise in $(\mathbf{Z}/q\mathbf{Z})^*$. Therefore in these two groups, if x is random in $(\mathbf{Z}/N\mathbf{Z})^*$ and r is relatively prime to p and to q, x^r is random in each group $(\mathbf{Z}/p\mathbf{Z})^*$ and $(\mathbf{Z}/q\mathbf{Z})^*$, and therefore in their product $(\mathbf{Z}/N\mathbf{Z})^*$. It follows that the generator of our problem instance is random whenever $\gcd(r, \phi(N)) = 1$. Since for the input to A to be valid we require our generator to have high order modulo N, we must now consider the probability that a random y will have high order.

First, note that if x is a generator modulo p, and r is relatively prime to $\phi(p) = p - 1$, then x^r is also a generator. It follows that there are $\phi(\phi(p))$ generators modulo p [out of $\phi(p)$ total elements in $(\mathbf{Z}/p\mathbf{Z})^*$]. Let $\mathrm{ord}_p(x)$ denote the order of x modulo p. So if $\mathrm{ord}_p(x) = p-1$ and $\mathrm{ord}_q(x) = q-1$, then

$$\mathrm{ord}_N(x^r) = \mathrm{lcm}(\mathrm{ord}_p(x_p^r), \mathrm{ord}_q(x_q^r)) = \mathrm{lcm}(p-1, q-1),$$

in which case our generator y will have high order as defined in $\mathcal{D}^C$. Since our algorithm generates $x \bmod p$ and $x \bmod q$ independently at random, it follows that the probability of this success is

$$\frac{\phi(\phi(p))}{\phi(p)} \frac{\phi(\phi(q))}{\phi(q)}. \tag{4.1}$$

Also, we know for any s relatively prime to $\phi(N)$, there is a unique r relatively prime to $\phi(N)$ such that s is the solution to $rs \equiv 1 \bmod \phi(N)$. Equivalently, taking inverses in $(\mathbf{Z}/\phi(N)\mathbf{Z})^*$ is a permutation. Therefore if we have a random r which is relatively prime to $\phi(N)$, then we have a random exponent $s \in (\mathbf{Z}/\phi(N)\mathbf{Z})^*$ in our instance of CDL.

So while, as Bach argues correctly, we do know that there exists an $r < \log N + 1$ that must be relatively prime to $\phi(N)$, we must go about selecting such an r at random in order to invoke what we know about A's probability of success, thus step 2 above looks as it does. We note that since $N = pq$ and p and q are roughly the same size (each k bits), we can approximate $\phi(N) = (p-1)(q-1) = N - p - q + 1 \approx N - 2\sqrt{N} + 1$. Our bound H in step 2 of the algorithm is thus designed to approximate closely $N\phi(N)$.

Based on this assumption, H is near a multiple of $\phi(N)$, so that if r is drawn at random from $\{2, \ldots, H\}$and $\gcd(r, \phi(N)) = 1$, then $r \bmod \phi(N)$ should be uniformly distributed in $(\mathbf{Z}/\phi(N)\mathbf{Z})^*$. Also under this assumption the probability that r is relatively prime to $\phi(N)$ is

$$\frac{\phi(\phi(N))}{\phi(N)} \tag{4.2}$$

In summary, B factors successfully in the event that the random choices of r and x form a valid random problem instance in $\mathcal{D}^C$, conditions which hold with independent probabilities given in (4.2) and (4.1) respectively; *and* in the event that A is successful on the instance, which occurs with probability $\geq \varepsilon$. Therefore

$$\begin{aligned} \Pr[B(\mathcal{F}_k) \text{ divides } \mathcal{F}_k] &\geq \frac{\phi(\phi(N))}{\phi(N)} \frac{\phi(\phi(p))}{\phi(p)} \frac{\phi(\phi(q))}{\phi(q)} \cdot \varepsilon \\ &\geq \frac{\varepsilon}{6^3 \cdot \log\log\phi(N) \cdot \log\log\phi(p) \cdot \log\log\phi(q)}, \end{aligned}$$

which is non-negligible if ε is non-negligible. This completes our proof. ■

4.2 Factoring and Prime DL imply Composite DL

We have seen that we can factor integers efficiently given an algorithm to solve discrete log modulo N efficiently. The converse of this statement is currently not known to be true, but if we are given the ability both to factor and to take discrete logs modulo a prime, then discrete log modulo N becomes feasible.

Proposition 4.5 *Suppose that efficient algorithms exist to solve both factorization and discrete log modulo a prime p. It follows that there exists an efficient algorithm to solve discrete log modulo any N.*

We preface this proof with an important lemma.

Lemma 4.6 *For any prime p, if given the ability to efficiently solve discrete log modulo p, it is possible to efficiently solve discrete log modulo p^e as well for any e.*

Proof of Lemma. The converse of the lemma is trivial, for it is clear that if discrete logs modulo p^e are computable, then so are discrete logs modulo p. For the direction which interests us, observe that the structure of the groups involved is

$$(\mathbf{Z}/p^e\mathbf{Z})^* \cong (\mathbf{Z}/p\mathbf{Z})^* \times (\mathbf{Z}/p^{e-1}\mathbf{Z}),$$

where the group operation of the last term is addition, not multiplication. The isomorphism from left to right is given by $x \mapsto (\pi(x), \psi(x))$; the projection π is reduction modulo p, which can clearly be evaluated efficiently, while the projection ψ is slightly more complicated, and we construct it as follows.

We can, by considering a quotient, consider the structure of $(\mathbf{Z}/p^e\mathbf{Z})^*$ as

$$(\mathbf{Z}/p^e\mathbf{Z})^* \cong (\mathbf{Z}/p\mathbf{Z})^* \times U,$$

where U is the subgroup $\{x \in (\mathbf{Z}/p^e\mathbf{Z})^* : x \equiv 1 \pmod{p}\}$ (following [1]) . It is therefore easy to project onto u by raising any element to the power $p-1$ (modulo p^e); by Fermat the result is congruent to 1 modulo p, and this operation (call it θ) is clearly a homomorphism. We can thus give an isomorphism $\eta : U \rightarrow \mathbf{Z}/p^{e-1}\mathbf{Z}$ and we will have ψ as the composition of η and θ.

Consider

$$\eta(x) = \frac{(x^{p^{e-1}} - 1)}{p^e} \bmod p^{e-1}.$$

We want show that this is an isomorphism, but first we show that it is well-defined on U, which requires that $p^e \mid a^{p^{e-1}} - 1$ whenever $a \equiv 1 \pmod{p}$. Let $e_p(x)$ denote the highest power of p which divides x (as in $e_2(x)$ earlier). More generally, Bach [1] asserted that for any integers a and b both relatively prime to p, if $e_p(a-b) \geq 1$ then for all $k \geq 1$ then

$$e_p(a^{p^k} - b^{p^k}) = e_p(a-b) + k,$$

and what we need for well definition follows from setting $b = 1$, from which we naturally get $e_p(a-b) \geq 1$ under the assumption that $a \equiv 1 \pmod{p}$.

So it is well-defined. It is a homomorphism because

$$\eta(ab) = \frac{(ab)^{p^{e-1}} - 1}{p^e} = \frac{(a^{p^{e-1}} - 1) + (b^{p^{e-1}} - 1) + (a^{p^{e-1}} - 1)(b^{p^{e-1}} - 1)}{p^e}$$

But we know from above that p^e must divide each of $(a^{p^{e-1}} - 1)$ and $(b^{p^{e-1}} - 1)$, therefore the product of these terms is divisible by p^{2e}, and so even after evaluating the fraction this term will be divisible by p^e and vanish modulo p^{e-1}. Therefore we have

$$\eta(ab) \equiv \frac{(a^{p^{e-1}} - 1)}{p^e} + \frac{(b^{p^{e-1}} - 1)}{p^e} = \eta(a) + \eta(b) \pmod{p^{e-1}},$$

therefore this is indeed a homomorphism into the additive group $\mathbf{Z}/p^{e-1}\mathbf{Z}$.

To show that it is an isomorphism, note that if we take $a \in U$ such that $e_p(a-1) = 1$, then by the above $e_p(a^{p^{e-1}} - 1^{p^{e-1}}) = e_p(a^{p^{e-1}} - 1) = e$. Therefore when we take $\eta(a) = (a^{p^{e-1}} - 1)/p^e \bmod p^{e-1}$ we get an integer which p does not divide. Therefore $\eta(a)$ is a unit mod p^{e-1}, and it follows that it generates all of $\mathbf{Z}/p^{e-1}\mathbf{Z}$. Since we have already shown η is a homomorphism , we have that η is surjective, which implies that it is an isomorphism.

We note that as written η is not polynomially computable, since we observed earlier that exponentiation must be modular in order to be efficient. However since the value of the numerator only matters modulo $p^e \cdot p^{e-1}$, we can take the exponentiation mod p^{2e-1} and be okay.

Now we have shown that we can efficiently move between representations of elements in $(\mathbf{Z}/p^e\mathbf{Z})^*$ and in $(\mathbf{Z}/p\mathbf{Z})^* \cdot \mathbf{Z}/p^{e-1}\mathbf{Z}$. Since discrete log in the additive group $\mathbf{Z}/p^{e-1}\mathbf{Z}$ is trivial, assuming the ability to solve it in $(\mathbf{Z}/p\mathbf{Z})^*$ immediately yields solutions in $(\mathbf{Z}/p^e\mathbf{Z})^*$. Specifically, suppose that we are given $y \equiv g^a$ modulo p^e. Rewrite $y = (y_1, y_2)$ and $g = (g_1, g_2)$ in terms of the group product above. It follows that $y_1 \equiv g_1^{a_1} \bmod p$, and that $y_2 = g_2 \cdot a_2 \bmod p^{e-1}$. We can find a_2 just by dividing, and a_1 using our assumed ability. Therefore recombine $a = (a_1, a_2)$ using our isomorphisms. ∎

With this lemma in place, we proceed into our proof of the above proposition.

Proof of Proposition 4.5. We are faced with the problem of calculating a such that $g^a = y \bmod N$ for some g, y, and N given us; we have at our disposal the abilities to factor integers and solve discrete logs modulo p. So first, we factor

$$N = p_1^{e_1} \cdots p_r^{e_r}, \quad \text{each } p_i \text{ prime.}$$

Since the corresponding isomorphism

$$(\mathbf{Z}/N\mathbf{Z})^* = (\mathbf{Z}/p_1^{e_1}\mathbf{Z})^* \times \cdots \times (\mathbf{Z}/p_r^{e_r}\mathbf{Z})^*$$

follows from this, we know that if $y = g^a$ as elements of $(\mathbf{Z}/N\mathbf{Z})^*$, then the projections into the r groups on the right must also hold; thus if $y_i = y \bmod p_i^{e_i}$ is the projection into the ith coordinate, and likewise for g_i, we must have $y_i = g_i^{a_i}$, where a_i is the reduction of a modulo $\phi_i = \phi(p_i^{e_i}) = (p_i - 1)p_i^{e_i-1}$. (We could also in truth say $y_i = g_i^a$, but it is important to realize that the logarithm within the ith coordinate only exists as an element of $\mathbf{Z}/\phi_i\mathbf{Z}$.)

It is easy to project and determine the y_i and g_i which correspond, and by the above lemma we can use our ability to solve discrete logs modulo a prime to solve for the a_i.

We then have equations of the form $y \equiv g^{a_i} \pmod{p_i^{e_i}}$, and we must combine them into a single a such that $y \equiv g^a \pmod{N}$.

We can then combine them back into a using the Chinese Remainder Theorem such that for each i, a is congruent to a_i modulo the order of g in $(\mathbf{Z}/p_i^{e_i}\mathbf{Z})^*$. Note that the Chinese Remainder Theorem does not actually guarantee a solution in this case, because we do not have that these orders are relatively prime; however since we know such an a exists we can use CRT methods to find it.

We must show how to find these orders, which we denote $\mathrm{ord}_i(g)$. Trivially, since this divides $\phi(p_i^{e_i})$,

$$\mathrm{ord}_i(g) = \prod_{q \text{ prime}, \, q | \phi(p_i^{e_i})} q^{e_q(\mathrm{ord}_i(g))},$$

and Bach [1] shows that we can compute

$$e_q(\mathrm{ord}_i(g)) = \min\{k : g^{\phi(p_i^{e_i})/q^k} \equiv 1 \pmod{p_i^{e_i}}\}.$$

Note that these last steps require the factorizations of $\phi(p_i)$, but we assume a factoring oracle as part of the reduction. ■

This completes the reduction from composite discrete log to a combination of prime discrete log and factoring. We note that this reduction, like the first Bach reduction of the previous section, is worst-case, in that it assumes the availability of arbitrary factorizations in the final steps. It would thus be more difficult to attempt a probabilistic version, since the factorization problem instances are not necessarily coming from the same distribution as the composite modulus of the discrete log problem.

4.3 Generalized Diffie-Hellman mod *N* implies Factoring

The Generalized Diffie-Hellman problem, or GDH, was introduced in section 2.1.4 as an application of the discrete log problem; in particular we can consider this problem over a composite modulo N. It is clear that a solution of composite discrete log will imply a solution to GDH by the setup of the protocol; we now show that a solution to GDH implies a solution to factoring when N is a Blum integer ($N = p \cdot q$, where p and q are primes congruent to 3 mod 4).

The result we show was first proven by Shmuely in 1985, and was strengthened by Biham, Boneh, and Reingold in 1999 [2]. We state the GDH assumption precisely as follows:

Definition 4.7 *Let N be any Blum integer which is a product of n-bit primes and g any quadratic residue in $(\mathbf{Z}/N\mathbf{Z})^*$. Let $\mathbf{a} = (a_1, \ldots, a_k)$ be any sequence of elements in $\mathbf{Z}/N\mathbf{Z}$. Define $f_{N,g,\mathbf{a}}(x)$, such that for any k-bit string $x = x_1 \cdots x_k$,*

$$f_{N,g,\mathbf{a}}(x) = g^{\prod_{x_i=1} a_i} \bmod N.$$

Define $\tilde{f}$ to be the restriction of f to all k-bit strings except for 1^k (the string where all $x_i = 1$.)

In the above definition, the a_i are the secret values selected by each party, and the $f(x)$ values are the messages sent among the parties in order to determine the secret key—x indicates which parties' keys are involved. The secret key itself is therefore $f(1^k)$. Therefore an adversary would be able to see any of the values of $\tilde{f}$, and wishes to compute the value $f(1^k)$.

Definition 4.8 *A probabilistic polynomial-time algorithm A* **solves the GDH problem** *for $k = k(n)$ users with probability $\varepsilon = \varepsilon(n)$ if for all sufficiently large n,*

$$\Pr[A^{\tilde{f}_{N,g,\mathbf{a}}}(N, g) = f(1^k)] > \varepsilon(n),$$

taken over the coin tosses of A, the random selection of N and g, and the random selection of $a_1, \ldots, a_k$. [NB: the superscript notation indicates oracle access to $\tilde{f}$—the ability to evaluate the function on any input with a single instruction.]

Proposition 4.9 *If there exists a polynomial time algorithm which solves the GDH problem with non-negligible probability, then there exists an algorithm which factors with non-negligible probability.*

Proof . So suppose A is an algorithm which solves the GDH problem. We propose the algorithm B which operates as follows:

1. Choose v at random from $(\mathbf{Z}/N\mathbf{Z})^*$. Then compute $g = v^{2^k} \bmod N$.

   Note that since N is a Blum integer, we know $\phi(N) = (p-1)(q-1) = (4\alpha+2)(4\beta+2) = 4(4\alpha\beta + 2\alpha + 2\beta + 1)$. Therefore 4 divides $\phi(N)$, but no higher power of 2 divides $\phi(N)$, since the term in parentheses is clearly odd. Therefore if ℓ is the order of g, we know that $2^k \cdot \ell$ divides $\phi(N)$, but since $k \geq 2$ it follows that ℓ is odd. Thus $\gcd(2, \ell) = 1$, which implies the existence of a multiplicative inverse of 2 modulo ℓ, which we will denote $\mathbf{1/2}$ [its value is actually just $(\ell+1)/2$.] Therefore $g^{\mathbf{1/2}} = g^{\frac{\ell+1}{2}}$ is a square root of g modulo N.

2. Select $r_1, \ldots, r_k$ uniformly at random from the range $1, \ldots, N$, and for each i let a_i be the value $r_i + \mathbf{1/2} \bmod \ell$. Let $\mathbf{a}$ be the sequence $(a_1, \ldots, a_k)$. Note that since B does not know ℓ, it also doesn't know the value of $\mathbf{1/2}$, therefore it does not know the a_i.

3. Now run the algorithm A on input (N, g). Even though B does not know the a_i, whenever A requests the value $\tilde{f}(x)$ from its oracle, we show below that B can respond with the correct answer in polynomial time.

4. Provided that A responds with $f(1^k)$ (that is, successfully breaks GDH in this situation), use this to compute $u = g^{(\mathbf{1/2})^k}$. As will be shown below, it follows that $u^2 = v^2 \pmod N$. If $u \neq \pm v$, we have a factor of N. Otherwise fail.

We are left with three things to show:

i. For any k-bit string $x \neq 1^k$, B can compute $\tilde{f}_{N,g,\mathbf{a}}(x)$ in polynomial time.

We make a subclaim: for all $i = 1, \ldots, k-1$, we have $g^{(\mathbf{1/2})^i} = v^{2^{k-i}}$. (Recall that by definition $v = g^{2^k}$). We know that g is a quadratic residue, and therefore any power of g is as well. For $i = 1$, we know that $g^{(\mathbf{1/2})}$ and $v^{2^{k-1}}$ are both quadratic residues which square to g. Since squaring is an injective function (in fact, a permutation), on the quadratic residues modulo N for N a Blum integer, it follows that $g^{(\mathbf{1/2})} = v^{2^{k-1}}$. Induction on i proves our subclaim.

It follows that for $i < k$, we can compute $g^{(\mathbf{1/2})^i}$ in polynomial time.

$$f(x) = g^{\prod_{x_i=1} a_i} = g^{\prod_{x_i=1}(r_i + \mathbf{1/2})}.$$

Consider now the expression $\prod_{q_i=1}(r_i + y)$ as a polynomial in y. We can expand in polynomial time into the expression $\sum_{j=0}^{k-1} c_j y^j$ for integer coefficients c (we know the degree is at most $k-1$ because $x \neq 1^k$, therefore we have at most $k-1$ factors in the product.) Therefore we can rewrite the above as

$$f(x) = g^{\prod_{x_i=1} a_i} = g^{\prod_{x_i=1}(r_i+\mathbf{1/2})} = g^{\sum_{j=0}^{k-1} c_j (\mathbf{1/2})^j} = v^{\sum_{j=0}^{k-1} c_j 2^{k-j}},$$

which we can compute in polynomial time.

ii. Given $f(1^k)$, $u = g^{(\mathbf{1/2})^k}$ can be computed in polynomial time.

We can show this similarly. We know that

$$f(1^k) = g^{\prod_{i=1}^k a_i} = g^{\prod_{i=1}^k (r_i + \mathbf{1/2})} = g^{(\mathbf{1/2})^k} \cdot g^{\sum_{j=0}^{k-1} b_j (\mathbf{1/2})^j} = g^{(\mathbf{1/2})^k} \cdot v^{\sum_{j=0}^{k-1} b_j 2^{k-j}},$$

where here the coefficients b_j come from the expansion of $(\prod_{i=1}^k (r_i + y)) - y^k$. Therefore, since the power of v which appears at the end of the above equation is computable in polynomial time, if we know $f(1^k)$ we can divide modulo N to find u efficiently.

iii. Defined as above, $u^2 \equiv v^2 \pmod N$.

We know that $u^2 = g^{(\mathbf{1/2})^{k-1}}$; above we showed that $g^{(\mathbf{1/2})^i} = v^{2^{k-i}}$, so if we let $i = (k-1)$ we see that $u^2 = v^2$.

Consider now the probability that B is successful in factoring N. It is clear that

$$\Pr[B \text{ factors } N] = \Pr[(u \neq \pm v) \wedge A \text{ successfully computed } f(1^k)].$$

First we notice that since $g = v^{2^k}$ for $k > 2$, and the a_i were all chosen independent of g, the values of $f_{N,g,\mathbf{a}}(x)$ are invariant if v is replaced by v^* such that $v^2 = (v^*)^2$—that is, f only varies with v^2 and not with v. It follows that $\Pr[(u \neq \pm v)] = 1/2$.

It follows that if A solves the GDH problem with probability $\varepsilon(n)$, that B factors with probability $\varepsilon(n)/2$. ■

5. Alternative Computational Models

We have so far considered how these two problems relate in the most traditional setting of computation, "algorithms", in the intuitive sense. In a way, we have often deviated from the most conservative model of computation by adding the ability to choose random values at will. This allowance is made in accordance with physical assumptions that sources of sufficiently "high-quality"—that is, near to uniform—randomness can be found in the real-world. Our experience has convinced us that these assumptions are reasonable, and in fact some ongoing research has suggested evidence that, theoretically, randomness actually adds no power to algorithms—that the class of languages solvable by probabilistic polynomial time Turing machines (called $\mathcal{BPP}$) equals the class $\mathcal{P}$.

In this section we extend our notion of computability significantly farther, considering notions of complexity which do not rest on overwhelmingly believable real-world assumptions. They are, however, not entirely unreasonable, each for its own reasons, and they do pose interesting theoretical questions which can be asked about the two problems at hand.

5.1 Generic Algorithms

Let us first consider a very limited model, a strict mathematical setting of computation in which actual proofs of hardness can be established.

5.1.1 Generic Discrete Log Algorithms

We consider now a computational machine which has no access to any special information about the encodings of the objects it manipulates, but can make oracle queries to a mathematical "black box" to make the manipulations. Sometimes called the Generic Group Model, this vision of computation was first analyzed for its implications on the discrete logarithm problem by Nechaev in 1994 [33], and these results were extended by Victor Shoup in 1997 [46].

Consider an algorithm which works over an abelian group G. Let S be a set of bit strings with at least as many elements as G. We consider the strings of S to be an encoding of the elements of G, and we call any injective function $\sigma : G \to S$ an *encoding function* of G on S.

A *generic algorithm* A for G on S is a probabilistic algorithm defined by the following properties:

- A takes as input as list of encoding strings $(\sigma(x_1), \ldots, \sigma(x_k))$ where each $x_i \in G$ and σ is an encoding function of G on S.
- A has access to an oracle function which takes as input two indices i and j out of the encoding list, as well as a sign bit, and returns $\sigma(x_i + x_j)$ or $\sigma(x_i - x_j)$ depending on the sign bit ($+$ here indicates the group operation). This value is then added to the list of encodings.

We denote the output of A, which is a bit string, as $A(\sigma; x_1, \ldots, x_k)$ [although note that A never gets access to the x_i, only their encodings.] This model captures the idea of an algorithm which knows nothing about the group it is working in, only that it has certain elements with certain relations among them. Not knowing the encoding function used to give the bit strings in its list, it cannot introduce arbitrary group elements into the calculation at all—it can only perform the group operation on elements it has already seen.

The key result is that in such a situation, no generic algorithm can recover an element from its encoding. We present this theorem and proof from [46].

Proposition 5.1 (Shoup - Theorem 1) *Let n be an integer with p its largest prime factor. Let S be a set of binary strings with at least n elements. If A is a generic algorithm for $\mathbf{Z}/n\mathbf{Z}$ on S that makes at most q queries to the group oracle, then*

$$\Pr[A(\sigma; 1, x) = x] = O(q^2/p),$$

where the probability is taken over the coin tosses of A, and all possible encoding functions $\sigma : \mathbf{Z}/n\mathbf{Z} \hookrightarrow S$ and all $x \in \mathbf{Z}/n\mathbf{Z}$ chosen at random.

We note that here the algorithm A is given the encodings of the group identity element and of another element and asked to produce that element. It follows that if we consider the map $\sigma_g : x \mapsto g^x \bmod p$ (in binary), then σ_g is an encoding of the group $(\mathbf{Z}/p\mathbf{Z})^* \cong \mathbf{Z}/(p-1)\mathbf{Z}$ on $S = \{0,1\}^m$ where $m = \lceil \log_2(p) \rceil$.

What the theorem tells is that for any algorithm C which solves the discrete log problem without taking advantage of the encoding of group elements, there is some encoding σ for which the probability of success is $O(q^2/p)$, where q is the number of group operations performed. Therefore if we want the probability of success P to be strictly greater than some constant $c > 0$, we must have q proportional to $\sqrt{p}$. So C must have running time $O(\sqrt{p})$.

Proof of Proposition. The following Lemma will be needed:

Lemma 5.2 *If $f(X_1, \ldots, X_k)$ is a non-zero polynomial with coefficients in $\mathbf{Z}/p^t\mathbf{Z}$, with total degree d, then the probability that $f(x_1, \ldots, x_k) = 0$ for a random k-tuple $(x_1, \ldots, x_k) \in (\mathbf{Z}/p^t\mathbf{Z})^k$ is at most d/p.*

Proof of lemma. We consider two cases, $t = 1$ and $t > 1$.

$t = 1$: Prove by induction on k. Suppose $k = 1$. Then since over a field f can have only d roots, the probability that a random element of $\mathbf{Z}/p\mathbf{Z}$ is a root is at most d/p. If $k > 1$, $f(X_1, \dots, X_k)$ can be re-written as a polynomial in X_1 with polynomial coefficients in $\mathbf{Z}/p\mathbf{Z}[X_2, \dots, X_k]$. Let $f_1(X_2, \dots, X_k)$ be the coefficient of this polynomial corresponding to some term $X_1^{d_1}$ which yields the total degree d. By induction, it follows that since f_1 is a $(k-1)$-variable polynomial with total degree $d - d_1$, $f_1 = 0$ with probability at most $(d - d_1)/p$, or equivalently, that f_1 has at most $p^{k-1}(d-d_1)/p$ roots. If $(y_2, \dots, y_k)$ is such a root, then $f(X_1, y_2, \dots, y_k)$ might be either zero everywhere; if not, or if $(y_2, \dots, y_k)$ is not a root then the one-variable polynomial $f(X_1, y_2, \dots, y_k)$ has at most d_1 roots. Therefore the total number of roots of f is

$$p \cdot (p^{k-1}(d - d_1)/p) + d_1 \cdot (p^{k-1}) = d \cdot p^{k-1}.$$

It follows that the probability that a random k-tuple is a root of f is $(d \cdot p^{k-1})/p^k = d/p$.

$t > 1$: Divide the equation $f = 0$ by the highest power of p which occurs, and then reduce modulo p. The result is a polynomial of no higher degree over $\mathbf{Z}/p^t\mathbf{Z}$ with $t = 1$. Since randomly choosing elements modulo p^t yields random reductions modulo p, apply the above result for $t = 1$, and the lemma holds.

■

At each stage, A's oracle interactions give it the encoding $\sigma(x_i)$ of some $x_i \in \mathbf{Z}/N\mathbf{Z}$. Since $\mathbf{Z}/N\mathbf{Z} \cong \mathbf{Z}/p^t\mathbf{Z} \times \mathbf{Z}/s\mathbf{Z}$, we can equivalently think of each unique x_i as representing a unique ordered pair (x_i', x_i'') over this group product.

We assume that A knows the order N of the group and even its prime factorization. So if A is able to recover x, then it must be able to recover x', the reduction of x modulo p^t.

If we therefore assume that A restricts its interest to the mod p^t components of the group elements, at each stage A can maintain a list $F_1, \dots, F_k$ of linear polynomials with coefficients in $\mathbf{Z}/p^t\mathbf{Z}$ such that $x_i = F_i(1, x)$—these polynomials can be derived recursively from previous queries and its original inputs $\sigma(1)$ and $\sigma(x)$.

Though we write it as a function of two variables, since the first is always 1, each polynomial is really in a single variable X. Initially $F_1(X) = 1$ and $F_2(X) = X$.

I claim that the only way A can get any information at all about the group is if it gets two encodings which are the same, $\sigma(x_i) = \sigma(x_j)$, but $F_i \neq F_j$, which gives A a linear relationship on 1 and x. For suppose for all i, j such that $F_i \neq F_j$ we have $\sigma(x_i) \neq \sigma(x_j)$. Then A has learned the encodings of distinct elements; since σ is a random encoding function, these $\sigma(x_i)$ are independently random strings in S, and it follows that A learns nothing from them.

Clearly there is no way to find x without finding such a relation (for otherwise A will never get any non-random information), and therefore we can bound the probability that A will recover x by the probability that A can find i and j with $F_i \neq F_j$ but $\sigma(x_i) = \sigma(x_j)$.

But if it can do this, then consider the polynomial $G = F_i - F_j$, which is a polynomial in $\mathbf{Z}/p^t\mathbf{Z}[X]$ of total degree no more than 1. Since $F_i \neq F_j$ we have $G \neq 0$, so by the previous lemma we know that the probability that $G(x) = 0$ is at most $1/p$ for a random $x \in \mathbf{Z}/p^t\mathbf{Z}$. But since $\sigma(x_i) = F_i(x)$ and $\sigma(x_j) = F_j(x)$, if we do have $\sigma(x_i) = \sigma(x_j)$ then x is a root of G.

Since there are q queries, there are q^2 possible pairs i, j. It therefore follows that the probability A can find any pair i, j with the properties described, given the encoding of a random x, is at most q^2/p. ∎

5.1.2 Generic Algorithms and Factoring

This being a paper about the relationship between discrete logs and factoring, we would like at this point to be able to present the equivalent proof that no efficient generic algorithm can solve the factoring problem. However this sort of problem takes place more in a ring than a group, and several issues on generic ring algorithms remain open. One relevant result by Boneh and Lipton [5] showed that in a "generic field model," an algorithm *is* able to determine an element from its encoding given the order of the field, but it is unknown how this result changes when field becomes a ring and the order remains concealed (as it naturally must for factoring).

The closest we appear to be on this issue is a very recent paper by Damgård and Koprowski [11] which showed that root extraction, equivalent to RSA decryption, is provably hard for a generic algorithm which does not know the order of the group but does know bounds on the order as well as the distribution within those bounds.

Lacking the bidirectional implication between RSA decryption and factoring, this result is not quite germane, but it does suggest that provable generic bounds might not be discovered for the factoring problem.

The usefulness of the generic model lies in two areas. First, it allows us to recognize that certain algorithms are in a sense "optimal" for their class, so to speak. For example, Pollard's rho algorithm for discrete log is generic, as is the Pohlig-Hellman algorithm. However the index calculus method is not, since it specifically uses the fact that group elements are encoded as integers. The fact that the rho algorithm has $O(p^{1/2})$ running time, which we know from Shoup is the lower bound, tells us in one sense that trying to improve on rho is not a good use of our time.

The second use of the model is in motivating the search for contexts in which it has deeper meaning. More specifically, it leads us to find groups where we know of no way to exploit the representation of elements, and therefore the best algorithms we have to solve discrete log in these groups must necessarily be generic. This reason leads us to use discrete log over elliptic curves for digital signature schemes, because at present the best algorithms for taking these logarithms are generic, and therefore the complexity bounds apply.

5.2 Quantum Computers

The existence of a quantum computer in any practical sense is only theoretical—those built to date have been capable of processing inputs of only a small number of bits. However even at such small scales the proof-of-concept is worth taking note of, particularly for cryptographers, whose very existence—embodied in the security of the protocols presented in section 2—is threatened by main result we present here: that quantum computers are capable of efficiently solving both factoring and the discrete logarithm. We present a brief introduction to quantum computing for the purposes of discussing Shor groundbreaking result; for a good and more complete introduction to quantum computing refer to the text by Michael Nielsen and Issac Chuang [34].

5.2.1 The Fundamentals

While we think of a normal computer being a collection of bits, each of which is in one of two possible states, either 1 or 0. In a quantum computer the fundamental unit of information is not so restricted. Known as a *qubit*, it is a quantity which also has a state, but there are infinitely many possible states. The standard notation for a state in a quantum computer, called Dirac notation, is $|\cdot\rangle$. So while a qubit can be in the standard states $|0\rangle$ and $|1\rangle$, more generally we allow it to be in any state which is a linear combination of these basis states:

$$|s\rangle = \alpha|0\rangle + \beta|1\rangle, \quad \alpha, \beta \in \mathbf{C}, \text{such that } |\alpha|^2 + |\beta|^2 = 1$$

This mixed state, or *superposition*, only holds until the qubit is observed (sometimes “measured”), at which point it is forced into the physical world where it can only have value 0 or 1. If the qubit is in state $|s\rangle$ as above, then when observed it will have the value 0 with probability $|\alpha|^2$ and the value 1 with probability $|\beta|^2$.

In one sense, quantum computing gives us comparable power to a computer with infinite precision, since each individual quantum bit is capable of holding any of an infinite number of values. However, just as infinite precision is memory is infeasible, so is measuring a qubit beyond 1 bit of precision.

We can consider a quantum computer as a state machine with n qubits, and represent the entire state of the machine as a superposition of the 2^n basis states. For example if $n = 3$, the basis states are $|000\rangle$, $|001\rangle$, $|010\rangle$, $|011\rangle$, etc., and the state of the machine is a complex linear combination of these values, $\sum_i \alpha_i|S_i\rangle$ such that $\sum |\alpha_i|^2 = 1$. Observing this machine is just like as with one bit: each S_i will be the observed state with probability $|\alpha_i|^2$.

Just as a quantum computer is in all of its states simultaneously (with some probability per state), the transitions between states all occur simultaneously, each path of computation having a certain complex *probability amplitude*. Under this view, if a superposition is thought of as a vector in the complex vector space V of possible superpositions, a transition from one configuration to another can be viewed as an linear transformation $A : V \to V$, which operates on some *finite subset* of the bits of the configuration from one distribution to another. To ensure that the sum of the state-probabilities remains 1, it turns out that the only requirement is that $A^{-1} = \bar{A}^T$, i.e., that A be a unitary matrix (see [45] or [34]).

A useful such transformation is the *Fourier transform matrix* A_q which sends state $|a\rangle$, for each a with $0 \leq a < q$ to the distribution

$$\sum_{b=0}^{q-1} \frac{e^{2\pi iab/q}}{\sqrt{q}} \cdot |b\rangle.$$

Thus the probability of transitioning from state $|a\rangle$ to state $|b\rangle$ is $e^{2\pi iab/q}/q^{1/2}$. We will use this matrix often in the proof of the main theorem, Shor [45] shows how A_q can be constructed in polynomial time for q even of exponential size provided q is smooth.

In addition to these transform matrices, we have all the standard tools of classical algorithms, for it can be shown that if f is a classically computable function, then f can be computed by a quantum computer as well, provided it can be reversed (or that the argument remains on the tape). Thus we can transition from a superposition $\sum \alpha \cdot |a\rangle$ to $\sum \alpha \cdot |a, f(a)\rangle$.

5.2.2 The Impact

While there is substantial mathematics going on in the proof and analysis, the overall intuitive effect of quantum computation on problems such as factoring and discrete log is quite logical—these problems are difficult only in that the number of tests to be performed is huge, but each of the tests is quite easy; in a situation where all the tests could be done at once, it should be efficient.

We thus arrive at the main relevant result of quantum computing:

Proposition 5.3 (Shor) *The problems of factoring and discrete logarithm are solvable in polynomial time on a quantum computer.*

Proof (for factoring). We have seen already that a factor of N can be found if we can determine the order m of an element x modulo N, such that $x^m \equiv 1 \pmod{N}$. We demonstrate now how to do this in polynomial time using a quantum computer.

First, find a number q in the range $2N^2 < q < 4N^2$ which is smooth in the sense that for some fixed c, no prime factor of q is greater than $(\log N)^c$; this bound will assure us that working with q and specifically the construction of the transform matrix A_q mentioned above will always be possible in polynomial time.

Now, we place our quantum computer in the uniform superposition over states representing each element a modulo q. This state is written

$$\frac{1}{q^{1/2}} \sum_{a=0}^{q-1} |a\rangle,$$

and for readability we do not write the complete state of the machine, which would actually be $|N, x, q, a\rangle$, since all but the last entry will remain constant throughout the operation of the machine.

We now calculate $x^a \pmod{N}$. This occurs in whatever state we find the machine, so the resulting superposition is

$$\frac{1}{q^{1/2}} \sum_{a=0}^{q-1} |a, x^a \pmod{N}\rangle.$$

We now apply the Fourier transform A_q to the first part of our state; recall that this maps $a \to b$ with probability amplitude $\frac{1}{q^{1/2}} \exp(2\pi iab/q)$ for each $b \in \mathbf{Z}/q\mathbf{Z}$. Therefore our machine is placed in the superposition

$$\frac{1}{q}\sum_{a=0}^{q-1}\exp(2\pi iab/q)|b, x^a \pmod N\rangle.$$

At this point we observe the machine, seeing the values of b and of $x^a \pmod N$. Consider the probability that we end in a particular state $|b, x^k \pmod N\rangle$. To be in this state at the end, we clearly must have been in some state $|a, x^a \pmod N\rangle$ before the transformation A_q, where $x^a \equiv x^k \pmod N$. For each such state, we know the probability of moving from a to b, and therefore counting over all such ways to reach the state, the probability of ending in $|b, x^k \pmod N\rangle$ is

$$P = \left|\frac{1}{q}\sum_{a:x^a\equiv x^k}\exp(2\pi iab/q)\right|^2.$$

Since we are searching for m such that $x^m \equiv 1$, we can equivalently think of this sum as over all a satisfying $a \equiv k \pmod m$. Therefore write $a = tm + k$ and we can rewrite

$$P = \left|\frac{1}{q}\sum_{t=0}^{\lfloor(q-k-1)/r\rfloor}\exp(2\pi i(tm+k)b/q)\right|^2$$

The $\exp(2\pi ikb/q)$ terms do not involve t and therefore can be factored out; since they have magnitude 1 they can be ignored as well. Also, since the $\exp(2\pi ix)$ function is periodic with unit period, it follows that we can replace mb with any residue mod q, so let us fix $\overline{mb}$ as the residue which lies in the interval $(-q/2, q/2]$.

We now show that this probability P is large if $\overline{mb}$ is small, for the probability amplitudes will then be aligned in the same direction. If we have $\overline{mb}$ small with respect to q then using the change of variables $u = t/q$, we can approximate P by

$$\left|\int_0^{\frac{1}{q}\lfloor(q-k-1)/m\rfloor}\exp(2\pi i\,\overline{mb}\,u)\,du\right|^2.$$

If $-m/2 \leq \overline{mb} \leq m/2$, then it turns out this integral is bounded below by $4/\pi^2m^2$. Thus the probability of observing a given state $|b, x^k \pmod N\rangle$ is thus at least $4/\pi^2m^2 > 1/3m^2$ as long as $-m/2 \leq \overline{mb} \leq m/2$, or equivalently if there exists some d such that $-m/2 \leq mb - dq \leq m/2$, or by rearranging the terms and dividing by mq:

$$|b/q - d/m| \leq 1/2q.$$

Recall that we know b and q after observing the machine, but are searching for m and d. However, since $q \geq 2n^2$, there is only one such d/m at most which satisfies this inequality with denominator $m < N$. Thus if we round b/q to the nearest fraction with denominator less than N (say, using continued fractions), we can find this fraction d/m. If a qualifying fraction in lowest terms can be found (that is, $m < N$ and $\gcd(d, m) = 1$, we will have found the order m.

Consider the number of states $|b, x^k \pmod N\rangle$ which will give us such a fraction. We know that there are $\phi(m)$ possible numerators d, which will give a fraction in lowest terms, and each of these corresponds to a particular b so that d/m is close to b/q in the above inequality. So there are $\phi(m)$ possibilities for b; for x^k there are m possibilities, since the order of x is m. This gives us $m \cdot \phi(m)$ states each of which allow us to compute m if observed. Since we have seen that the probability of each of these states is at least $1/3m^2$, the probability of observing a successful state is at least $m \cdot \phi(m)/3m^2 = \phi(m)/3m$.

We have a theorem which asserts $\phi(n) \geq n/6 \log\log n$, which bounds our probability of success to $\Theta(1/\log\log m)$, so we expect success of factoring to be extremely likely after only $O(\log\log m)$ repetitions. ∎

Proof (for discrete log). The proof is quite detailed for the general case of discrete log, and the details are less important for our purposes here than the existence itself, so we present a sketch here. Recall that our purpose is to find a such that $g^a = y \bmod p$, given g, y, p.

Given p, find a smooth q in the interval $p \leq q \leq 2p$. Shor demonstrates how to do this. Now choose b and c uniformly at random from $\mathbf{Z}/p-1\mathbf{Z}$ and compute $g^b y^{-c} \bmod p$. The machine is then in the superposition

$$\frac{1}{p-1}\sum_{a=0}^{p-2}\sum_{b=0}^{p-2}|b, c, g^b x^{-c} \bmod p\rangle.$$

Apply the Fourier transform A_q which independently manipulates the first two positions of the state, sending $b \mapsto d, c \mapsto e$ for each c and e in $\mathbf{Z}/q\mathbf{Z}$ with amplitude $1/q \exp(2\pi i(bd+ce)/q)$, leaving the machine in the superposition

$$\frac{1}{(p-1)q}\sum_{b,c=0}^{p-2}\sum_{d,e=0}^{q-1}\exp(\frac{2\pi i}{q}(bd+ce))|d, e, g^b x^{-c} \bmod p\rangle.$$

In the "easy case" where $p-1$ is smooth, then we can let $q = p-1$, and Shor shows that the probability of observing a particular state $|d, e, y\rangle$ with $y \equiv g^k \bmod p$ is

$$\left|\frac{1}{(p-1)^2}\sum_{c=0}^{p-2}\exp(\frac{2\pi i}{p-1}(kd + c(e+ad)))\right|^2.$$

If $e + ad \not\equiv 0 \bmod p-1$, it then follows that this is a sum over a full set of roots of unity, thus has probability 0. If $e + ad \equiv 0$ then the sum is over the same root of unity $(p-1)$ times, leading to a probability of $1/(p-1)^2$. Since there are $(p-1)$ values of d and $(p-1)$ non-zero values of y, it follows that there are $(p-1)^2$ such d, e pairs. Thus this calculation will *always* produce a pair d, e such that $e \equiv -ad \pmod{p-1}$. So as long as $\gcd(d, p-1) = 1$, we have the exponent a. Since all possible ds occur with equal probability, our chance of this occurrence is $\phi(p-1)/(p-1)$, and as in the above proof we can get a very high probability of success by repeating our quantum experiment $O(\log^c p)$ times.

For full analysis of the general case when $p-1$ is not assumed smooth, see [45]. ∎

The significance of these results for us is that they have once again grouped factoring and discrete log into the same complexity class, sometimes called $\mathcal{BQP}$, of problems solvable by probabilistic polynomial-time algorithms on a quantum computer. As already mentioned, the practicality of these results has not yet been seen—in fact it is just one of several new computational paradigms which seek to harness the massive computation power of natural phenomena, among the most researched is DNA-based computers [4], which are believed capable of solving $\mathcal{NP}$-complete problems in polynomial time.

5.3 Oracle Complexity

We now turn to a seemingly all-powerful model of computation, in which algorithms have access to an infinite font of knowledge. We imagine this source of knowledge as an "oracle" capable of instantly answering any yes/no question, and we consider the number of questions necessary to solve a particular problem.

In some settings, particularly cryptographic ones, this model bears some resemblance to reality despite its fantastic nature—we can use the oracle model to answer questions such as "if an adversary gains access to the low-order bits of my p and q, can they recover the full p and q efficiently?" We consider both the restriction of our oracle to actual bits of the secret, and the operation of the oracle on arbitrary yes/no questions; the latter is naturally a more theoretical mode of analysis, but it does provide an alternative way of considering how much security is contained in a certain kind of secret.

5.3.1 Oracle Complexity of Factoring

Here we present two results on the oracle complexity of factoring. The first was shown by Rivest and Shamir in 1985. As two thirds of the team which created the RSA cryptosystem (shown in Chap. 2 to depend on the inability to recover p and q from $N = pq$), they were naturally concerned with the security of p and q even in the event that some kind of "side information" was leaked to an adversary.

For introduction, we show the trivial method of factoring integers using an all-powerful oracle.

Proposition 5.4 *A composite integer N can be factored in polynomial time using $n/2$ oracle questions, where $n = \lceil \log_2 N \rceil$.*

Proof. For $i = 1, \ldots, n/2$, ask the oracle "what is the ith bit of the smallest prime factor of N?", and let the result be β_i. Then output the binary number $\beta_{n/2} \cdots \beta_2\beta_1$.

We know that if p is the smallest prime factor, $p \leq \sqrt{N}$, so the binary representation of p requires no more than $\log_2 \sqrt{N} = n/2$ bits. Therefore $n/2$ questions will always suffice. ■

Rivest and Shamir [42] improved on this result as follows.

Proposition 5.5 *A composite integer $N = pq$ can be factored in polynomial time using $k = n/3$ oracle questions.*

Proof . Ask for the top k bits of the smallest factor p. We can then write $p = p_1 2^m + p_0$, where $m = n/6$; $p_1 \leq 2^k$ is known and $p_0 \leq 2^m$ is not known. We can represent the other factor q similarly as $q = q_1 2^m + q_0$; while both quantities are initially unknown, we can compute q_1 as the k highest bits of $N/(p_1 2^m)$.

Therefore we have

$$N = (p_1 2^m + p_0)(q_1 2^m + q_0) = p_1 q_1 2^{2m} + (p_1 2^m) q_0 + (q_1 2^m) p_0 + p_0 q_0.$$

To isolate our unknowns, we abbreviate $X = N - p_1 q_1 2^{2m}, A = p_1 2^m, B = q_1 2^m$, and our equation becomes

$$X = A p_0 + B q_0 + p_0 q_0.$$

At this point we can try to approximate X as best as possible by a linear combination of A and B, the remaining term $p_0 q_0$ (which is relatively small) is our approximation error. We therefore have a 2-dimensional integer programming problem of minimizing $X - Ap_0 - Bq_0$ subject to the constraints $0 \leq p_0, q_0 \leq 2^m$. H.W. Lenstra proved [24] that this problem can be solved in polynomial time, the details of which we omit because it is quite involved. ■

The Rivest/Shamir result shows that knowledge of the high-order bits of one factor of N leads in polynomial time to the entire factorization, and indeed the same proof applies when considering knowledge of the low-order bits. In this sense, the question of oracle complexity is relevant, for we can conceive of an adversary who, with some direct access to the computer hardware or some temporary storage might be able to piece together parts of the actual values used in computation. However, if we generalize back into the world of an oracle capable of providing bits which answer *any* question, not simply bits which at one point physically existed, Maurer showed that the number of questions can be made less than εn for arbitrarily small ε, although a small probability of failure is introduced.

The result depends on manipulating the parameters of the elliptic curve factoring method described in Section 3.3.

Proposition 5.6 (Maurer) *For any $\varepsilon > 0$, a sufficiently large integer N (with $n = \log_2 N$ bits) can be factored in polynomial time using at most $\varepsilon \cdot n$ oracle questions. The probability of error is at most $N^{-\varepsilon/2}$ under plausible conjectures relating to the elliptic curve factoring algorithm.*

Proof . We assume for this section that N is not divisible by 2 or 3, and that N is not a prime power; in these cases a factor can be easily found.

Now, with ε given, choose an arbitrary positive $\delta < \varepsilon$ and let $c = 1/(\varepsilon - \delta)$, $W = n^c$. Now let $h = \prod q^{e(q)}$ for all prime $q \leq W$, where $e(q)$ is the greatest integer e such that $r^e \leq N^{1/2} + 2N^{1/4} + 1$. We note that this is an upper bound on the number of elements on an elliptic curve over $\mathbf{F}_p$ for any prime factor p of N, based on Hasse's result that the order of a curve over $\mathbf{F}_p$ is bounded above by $p + 2\sqrt{p} + 1$.

Let $\mathcal{F}$ represent the finite field with 2^{3n} elements, and choose s and t uniformly at random from $\mathcal{F}$. Choose a natural enumeration of the elements of $\mathcal{F}$ as $\alpha_1, \ldots, \alpha_{2^{3n}}$, and a natural representation of the elements of $\mathcal{F}$ as triples of n-bit integers (a, x, y); let (a_k, x_k, y_k) be the triple which corresponds to the element $s \cdot \alpha_k + t$; define

$$b_k = y_k^2 - x_k^3 - a_k x_k \bmod N.$$

For motivation, we note that (x_k, y_k) is then a point on the modulo N elliptic curve $y^2 = x^3 + a_k x + b_k$.

We then ask the oracle the following $\lfloor \varepsilon \cdot n \rfloor$ questions. For $i = 1, \ldots, \lfloor \varepsilon \cdot n \rfloor$:

- If there exists an integer k with $|k| < \varepsilon \cdot n$ such that the following two conditions hold under the above definitions:

  1. For p the smallest prime factor of N, $4a_k^3 + 27b_k^2 \not\equiv 0 \pmod{p}$, and the order of the elliptic curve $E : y^2 = x^3 + a_k x + b_k$ over $\mathbf{F}_p$ is W-smooth.
  2. For some prime factor $q > p$ of N, $4a_k^3 + 27b_k^2 \not\equiv 0 \pmod{q}$, and the order of the elliptic curve $y^2 = x^3 + a_k x + b_k$ over $\mathbf{F}_q$ is not divisible by the largest prime factor of the order of the point (x_k, y_k) on E over $\mathbf{F}_p$.

  then output the ith bit of the smallest such k; otherwise output 0.

If the oracle returns 0, then we have failed. If not, we have a value k. We can compute a_k, b_k, x_k, y_k as defined above, and then proceed as in the ECM for factoring described in Section 3.3. Consider the curve $y^2 = x^3 + a_k x + b_k$ modulo N. We know that the point $P = (x_k, y_k)$ is on it, so try to compute the point $h \cdot P$. The conditions on k guarantee for us the fact that $h \cdot P \equiv \mathcal{O} \pmod{p}$, but for some other factor q of N we have $h \cdot P \not\equiv \mathcal{O} \pmod{q}$. It follows that we thus will have found a factor of N, since when $h \cdot P$ is written as $(\alpha : \beta : \gamma)$ in projective coordinates, γ is a multiple of p but not of N. Refer to Section 3.3 for the details.

Consider the running time of our algorithm. The formal addition of points on an elliptic curve in the ECM is $O(n^2)$, and (using the same trick as seen in the modular exponentiation algorithm of Proposition 1.6) we can compute $h \cdot P$ with approximately $2 \log h$ additions. This is polynomial in n since

$$\log h = \sum e(r) \log r \leq w \log w = O(n^c \log n^c).$$

The probability of success requires extending Lenstra's conjecture about the distribution of smooth integers in the Hasse interval, and a complete derivation of Maurer's claim can be found in [27]. ■

Practically, Maurer's result is not particularly damning for cryptographers, since while the oracle necessary for Rivest and Shamir seems potentially real, this oracle is unlikely to ever find a real-world manifestation. Maurer is only able to ask so few questions because the oracle is being asked to search through a huge directory of possible elliptic curves and find one which will work.

5.3.2 Oracle Complexity of Discrete Log

Rivest and Shamir suggest the additional problem of considering the oracle complexity of the discrete logarithm, although they do not address it at all, nor does the current literature contain much on this point.

I conjecture that discrete log will behave differently than factoring under this complexity model because discrete log is significantly more malleable than factoring. That is, given one problem instance (say $y = g^a \bmod p$) we can easy construct related problem instances (say $g^{2a} \bmod p = y^2$) without breaking the problem.

Because of this observation, it seems to be possible to take an algorithm A which solves discrete logs based on partial information about the exponent and build an algorithm which solves full discrete logs using this subroutine. In order to structure the proof, we do not actually have A make queries to an oracle; rather we allow ourselves to simulate the role of the oracle and, knowing the questions that A will ask (in this case, for the low order bits of the exponent a), provide those answers ourselves in the form of an additional input to the algorithm.

Conjecture 5.7 *Suppose that there exists an algorithm A, which on inputs p^e, g, y, b, where $y = g^a \bmod p^e$, outputs a in polynomial time provided that b is the $k = k(n)$ lowest bits of a for some function $k(n) < n$, where $n = \log_2 p$ is the size of the modulus. It follows that there exists a polynomial time algorithm which on inputs p^e, g, y of the same form outputs a.*

We would like to convert our input p^e, g, y into a new input $p'^{e'}, g', y' = g'^{a'}$, where in some way a can be recovered from a' and we know the k lowest bits of a'. From this, we can use our subroutine A to find a', and then recover a. Naively we could generate $a' = 2^k \cdot a$ by letting $y' = y^{2^k}$, and then we would know that the lowest k bits of a' are 0. Unfortunately this does not succeed, since a' only exists mod $\phi(p^e)$, and therefore multiplying by 2^k will cause some “wrap-around” and the k low order bits will be scrambled.

To correct for this we might try letting $p' = p$ and $e' = 2e$. This gives us more space to work with in the exponent, and it’s likely that multiplying by 2^k will not cause wraparound. However we are left with finding g' which has high order modulo p^{2e}, and doing this successfully implies the ability to multiply by a, which we cannot do.

So we leave the question open.

Bibliography

[1] Eric Bach, *Discrete logarithms and factoring*, Provided by author., June 1984.

[2] Eli Biham, Dan Boneh, and Omer Reingold, *Breaking generalized Diffie-Hellman modulo a composite is no easier than factoring*, Information Processing Letters **70** (1999), 83–87.

[3] Manuel Blum and Silvio Micali, *How to generate cryptographically strong sequences of pseudorandom bits*, SIAM J. Comput. **13** (1984), no. 4, 850–864.

[4] Dan Boneh, Christopher Dunworth, Richard J. Lipton, and Jiri Sgall, *On the computational power of DNA*, DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science **71** (1996).

[5] Dan Boneh and Richard J. Lipton, *Algorithms for black-box fields and their application to cryptography*, Lecture Notes in Computer Science **1109** (1996), 283–297.

[6] E. R. Canfield, Paul Erdős, and Carl Pomerance, *On a problem of Oppenheim concerning "factorisatio numerorum"*, J. Number Theory **17** (1983), no. 1, 1–28.

[7] Clifford Cocks, *A note on non-secret encryption*, Available online from www.cesg.gov.uk/publications/, 1973.

[8] Henri Cohen, *A course in computational algebraic number theory*, Springer-Verlag, 1993.

[9] R. Crandall and C. Pomerance, *Prime numbers: a computational perspective*, Springer-Verlag, New York, 2001.

[10] Richard Crandall and Carl Pomerance, *Prime numbers*, Springer-Verlag, New York, 2001, A computational perspective.

[11] Ivan Damgrd and Maciej Koprowski, *Generic lower bounds for root extraction and signature schemes in general groups*, to appear in Eurocrypt 2002.

[12] Nenad Dedic, Leonid Reyzin, and Salil Vadhan, *An improved pseudorandom generator based on hardness of factoring*, Cryptology ePrint Archive, Report 2002/131, 2002, http://eprint.iacr.org/.

[13] Max Deuring, *Die Typen der Multiplikatorenringe elliptischer Funktionenkörper*, Abh. Math. Sem. Hansischen Univ. **14** (1941), 197–272.

[14] Whitfield Diffie and Martin E. Hellman, *New directions in cryptography*, IEEE Trans. Information Theory **IT-22** (1976), no. 6, 644–654.

[15] Taher ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Trans. Inform. Theory **31** (1985), no. 4, 469–472.

[16] National Institute for Standards and Technology, *NIST FIPS PUB 186, Digital Signature Standard.*, 1994.

[17] Oded Goldreich, *Foundations of cryptography*, Cambridge University Press, Cambridge, 2001, Basic tools.

[18] Oded Goldreich, Shafi Goldwasser, and Silvio Micali, *How to construct random functions*, J. Assoc. Comput. Mach. **33** (1986), no. 4, 792–807.

[19] Daniel M. Gordon, *Discrete logarithms in* $\mathrm{GF}(p)$ *using the number field sieve*, SIAM J. Discrete Math. **6** (1993), no. 1, 124–138.

[20] D. Johnson and A. Menezes, *The elliptic curve digital signature algorithm (ecdsa*, 1999.

[21] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard, *The number field sieve*, The development of the number field sieve, Lecture Notes in Math., vol. 1554, Springer, Berlin, 1993, pp. 11–42.

[22] A.K. Lenstra, H.W. Lenstra, M.S.Manasse, and J.M. Pollard, *The factorization of the ninth fermat number*, Mathematics of Computation **61** (1993), no. 203, 319–349.

[23] A.K. Lenstra and H.W. Lenstra, Jr., *Algorithms in number theory*, 1987.

[24] H. W. Lenstra, Jr., *Integer programming with a fixed number of variables*, Math. Oper. Res. **8** (1983), no. 4, 538–548.

[25] ———, *Factoring integers with elliptic curves*, Ann. of Math. (2) **126** (1987), no. 3, 649–673.

[26] Harry R. Lewis and Christos H. Papadimitrou, *Elements of the theory of computation*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[27] Ueli M. Maurer, *On the oracle complexity of factoring integers*, Computational Complexity **5** (1995), no. 3-4, 237–247, Used preprint version.

[28] Kevin S. McCurley, *The discrete logarithm problem*.

[29] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of applied cryptography*, CRC Press Series on Discrete Mathematics and its Applications, CRC Press, Boca Raton, FL, 1997, With a foreword by Ronald L. Rivest.

[30] Gary L. Miller, *Riemann's hypothesis and tests for primality*, J. Comput. System Sci. **13** (1976), no. 3, 300–317, Working papers presented at the ACM-SIGACT Symposium on the Theory of Computing (Albuquerque, N.M., 1975).

[31] Michael A. Morrison and John Brillhart, *A method of factoring and the factorization of F_7*, Math. Comp. **29** (1975), 183–205, Collection of articles dedicated to Derrick Henry Lehmer on the occasion of his seventieth birthday.

[32] Moni Naor, Omer Reingold, and Alon Rosen, *Pseudorandom functions and factoring*, SIAM J. Comput. **31** (2002), no. 5, 1383–1404 (electronic).

[33] V.I. Nechaev, *Complexity of a determinate algorithm for the discrete logarithm*, Mathematical Notes **55** (1994), 165–172.

[34] Michael A. Nielsen and Isaac L. Chuang, *Quantum computation and quantum information*, Cambridge University Press, Cambridge, 2000.

[35] Stephen C. Pohlig and Martin E. Hellman, *An improved algorithm for computing logarithms over $\mathrm{GF}(p)$ and its cryptographic significance*, IEEE Trans. Information Theory **IT-24** (1978), no. 1, 106–110.

[36] J. M. Pollard, *Theorems on factorization and primality testing*, Proc. Cambridge Philos. Soc. **76** (1974), 521–528.

[37] ______, *A Monte Carlo method for factorization*, Nordisk Tidskr. Informationsbehandling (BIT) **15** (1975), no. 3, 331–334.

[38] ______, *Monte Carlo methods for index computation $\pmod p$*, Math. Comp. **32** (1978), no. 143, 918–924.

[39] Carl Pomerance, *Factoring*, vol. 42, American Mathematical Society, 1990, pp. 27–47.

[40] M. Rabin, *Digitalized signatures and public-key functions as intractable as factorization*, MIT/LCS/TR-212, MIT Technical Memo (1979).

[41] V. Ramaswami, *The number of positive integers $\leq x$ and free of prime divisors $> x^c$, and a problem of S. S. Pillai*, Duke Math. J. **16** (1949), 99–109.

[42] Ronald L. Rivest and Adi Shamir, *Efficient factoring based on partial information*, Advances in Cryptology - EUROCRYPT '85, 1986, pp. 31–34.

[43] Ronald L. Rivest, Adi Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Comm. ACM **21** (1978), no. 2, 120–126.

[44] Oliver Schirokauer, *Using number fields to compute logarithms in finite fields*, Mathematics of Computation **69** (1999), no. 231, 1267–1283.

[45] Peter W. Shor, *Algorithms for quantum computation: Discrete logarithms and factoring*, IEEE Symposium on Foundations of Computer Science, 1994, pp. 124–134.

[46] Victor Shoup, *Lower bounds for discrete logarithms and related problems*, Advances in cryptology—EUROCRYPT '97 (Konstanz), Lecture Notes in Comput. Sci., vol. 1233, Springer, Berlin, 1997, pp. 256–266.

[47] Joseph H. Silverman, *The arithmetic of elliptic curves*, Graduate Texts in Mathematics, vol. 106, Springer-Verlag, New York, 1986, Corrected reprint of the 1986 original.

[48] ______, *Advanced topics in the arithmetic of elliptic curves*, Graduate Texts in Mathematics, vol. 151, Springer-Verlag, New York, 1994.

[49] Joseph H. Silverman and John Tate, *Rational points on elliptic curves*, Undergraduate Texts in Mathematics, Springer-Verlag, New York, 1992.

[50] Samuel S. Wagstaff, Jr., *Cryptanalysis of number theoretic ciphers*, CRC Press, 2002.

[51] H. Woll, *Reductions between number theoretic problems*, Information and Computation **72** (1987), 167–179.