MT3660/4660/5466
Cryptography II

(formerly *Public Key Cryptography*)

Royal Holloway
2020-21

1 An Introduction to Algorithms and Complexity

In Public Key Cryptography, we are generally concerned with constructing mathematical problems and then considering the computational difficulty in solving these problems. In certain circumstances, such a "difficult" problem can potentially be used as the basis for a Public Key Cryptosystem. To illustrate these issues, we give an outline of public key cryptography, and consider factoring and the RSA cryptosystems.

The technical approach to the issues of problem difficulty and so on in Public Key Cryptography is to set up a framework in which *computers*, *problems* and *algorithms* are specified and their properties considered. This technical approach, and some basic examples forms the latter part of this Section.

1.1 An Outline of Public Key Cryptography

The two main goals of cryptography are to provide **privacy** or **confidentiality** of communication and to provide **authentication** of one party to another. Encryption and decryption provides a solution to this first confidentiality problem. The encryption algorithm takes a message and an encryption key and outputs a ciphertext. Conversely, the decryption algorithm takes a ciphertext and a decryption key and outputs the message. Digital Signatures provide a solution to the latter authentication problem. The signing algorithm takes a message and a signing key and outputs a signature. The verification algorithm takes a message, a signature and a verification key and determines whether or not the signature is valid.

Symmetric cryptography is the oldest form of cryptography and has been practiced in various forms for centuries. It is also arguably the most important type of cryptography in the "real world", being used to secure billions of dollars of financial transactions and billions of mobile phone calls. However, symmetric cryptography cannot address all cryptographic issues satisfactorily. In particular, there are three main problems with symmetric cryptography:

- Each pair of users needs to have a shared key.
- These keys must be transmitted securely to the users.
- It is difficult to provide non-repudiation.

It was the key management problem in particular that led to the development of Public Key Cryptography (PKC), also know as asymmetric

cryptography. The ideas were first developed publicly by Ralph Merkle, Whit Diffie and Marty Hellman in the mid-1970s. Similar ideas had also been developed a few years earlier at GCHQ by James Ellis and Cifford Cocks. One of the earliest and most important public key cryptosystems was invented by Leonard Adleman, Ron Rivest and Adi Shamir in 1977, and is now known as the RSA cryptosystem.

1.2 The Textbook RSA Cryptosystem

We now describe the basic variant of the RSA Cryptosystem, often termed Textbook RSA. For RSA to be deployed practically in the real world, it is necessary to add some further enhancements to this basic RSA variant. It is common practice to refer to the the communicating parties as "Alice" and "Bob", who are sometimes joined by "Eve" the eavesdropper and others.

RSA Set-up. Let p and q be primes and $N = pq$. Let $e \in \mathbb{N}$ be coprime to $\varphi(N) = (p-1)(q-1)$, where $\varphi(N) = \#\{1 \leq x \leq N | x, N \text{ co-prime}\}$ is the Euler φ-function, and let d satisfy

$$ed = 1 \mod \varphi(N).$$

(In fact, it is sufficient to have $ed = 1 \mod \lambda(n)$ where $\lambda(n)$ is the Carmichael function.) The *RSA public key* is the pair of integers (N, e) and the *RSA private key* is the integer d.

RSA Encryption. If Bob wishes to send an encrypted message to Alice using RSA, Bob takes the following steps.

1. Obtain an authentic copy of Alice's public key (N, e).
2. Encode the message as an integer m such that $1 < m < N$.
3. Compute and transmit the ciphertext

$$c = m^e \mod N.$$

RSA Decryption. Alice takes the following steps to decrypt ciphertext c.

1. Compute

$$m = c^d \mod N.$$

2. Decode m to obtain the message.

The following Lemma shows that RSA process gives a proper cryptosystem.

Lemma. RSA decryption is the inverse of RSA encryption.

Proof. If the RSA encryption of $m \in \mathbb{Z}_N^*$ is $c = m^e \bmod N$, then Fermat's Little Theorem states that $m^{\varphi(N)} = 1 \bmod N$, so

$$c^d = (m^e)^d = m^{ed} = m^{1+k\varphi(N)} = m \cdot (m^{\varphi(N)}) = m \mod N.$$

1.3 The Textbook RSA Digital Signature

Digital signatures are an analogue of written signatures. They are used to prove ownership or authorship of a document, or to provide authorisation or a guarantee or a commitment to some task. Unlike handwritten signatures, digital signatures must depend on the document being signed. We now give the basic textbook form of a digital signature RSA process

RSA Signature Set-up. The set-up is identical to the Textbook RSA encryption given above, so we have primes p and q with $N = pq$, together with the public value e and private value d satisfying $ed = 1 \bmod \varphi(N)$. For digital signatures, the private value d is often referred to as the *signing* key and the public value e is referred to as the *verification* key. A cryptographic hash function H taking values in $\{1, \ldots, N\}$ is also required.

RSA Signature Generation. Alice takes the following steps to sign a message with her private signing key d.

1. Hash the message m to give $H(m)$ satisfying $1 < H(m) < N$.
2. Compute
$$s = H(m)^d \mod N$$
and transmit (m, s) as the digital signature.

RSA Signature Verification. Given a message m and Alice's putative signature s for m, Bob takes the following steps to verify Alice's signature.

1. Obtain an authentic copy of Alice's public key (N, e).
2. Hash the message m to give $H(m)$ satisfying $1 < H(m) < N$.
3. Test whether $H(m)$ equals $s^e \bmod N$ in order to determine whether s is a valid signature for m.

1.4 Issues in Public Key Cryptography

For Public Key Cryptography to be genuinely useful, a number of practical issues must be addessed and a partial list is given below.

1. Can public keys be efficiently generated?
2. Is the cryptosystem efficient in the sense of computational time and ciphertext size?
3. How does Bob know that Alice's public key is authentic?
4. Is the scheme secure?
5. What does 'security' mean anyway?

We can give a flavour of these issues by describing three attacks on RSA.

Textbook RSA Attack 1. Suppose the RSA cryptosystem is being used for an online election in such a way that people send votes encrypted under the public key of the electoral office. A major security goal in this case might be that each person does not want anyone else to know who they voted for. Suppose the eavesdropper Eve is monitoring internet traffic from Alice's computer. When Eve detects Alice has sent her vote she can perform a very simple attack. Alice's vote is the encryption of one of the names on the list of candidates. Eve simply encrypts each of the candidate's names in turn under the public key of the electoral office and compares with the ciphertext sent by Alice. When a match is found then Eve knows who Alice has voted for. This sort of attack will always be possible when the encryption is deterministic. Later we will show how RSA encryption can be randomised.

Textbook RSA Attack 2. To improve on the computational efficiency of encryption it is tempting to use small encryption exponents, such as $e = 3$. Suppose Alice has public key $(N, 3)$ and suppose Bob is only sending a very small message $m < N^{1/3}$ to Alice. Then $c = m^3$ in $\mathbb{N}$, so no modular reduction has taken place. If Eve knows that m will be very small then she can recover the message m from the ciphertext by taking cube roots. This may seem artificial: why would Bob send such a short message, and how would Eve know that he did? But public key cryptography is often used simply to send a key for a symmetric encryption scheme. These keys are short (e.g., 128 bits) compared with RSA moduli (at least 1000 bits). Hence, in practice, very short messages are sent. So we need to avoid this

attack. A signature forgery attack related to this issue was proposed by Daniel Bleichenbacher in 2006.

Textbook RSA Attack 3. A good encryption scheme should allow an adversary to learn absolutely nothing about a message from the ciphertext (except possibly the length of the message). However, the Textbook RSA cryptosystem the value of Jacobi symbol $\left(\frac{m}{N}\right) = \left(\frac{c}{N}\right)$, that is to say one bit of information about the message m as $\left(\frac{c}{N}\right)$ can be calculated. We discuss the Jacobi symbol at length later.

1.5 Computers

The general properties of what is considered to be a "computer" can be formally modelled by the theoretical concept of a *Turing machine*, which can be further categorised as Deterministic or Randomised. However, we note that theoretical Turing machines have infinite memory, whereas real computers of course do not. For our purposes, we consider that a computer is a device with the following properties.

- A computer always has as much memory as it needs.
- The memory consists of bits (something that can be either 0 or 1). The bits are typically arranged in blocks of 16, 32 or 64 which are called 'words'.
- All computational operations are broken down into a sequence of basic *bit operations* like 'and', 'or', 'not', 'nand', 'xor', 'add', branch etc that operate on words.
- A microprocessor has a 'clock' and typically performs one such operation every 'clock cycle'.
- We usually assume that a computer can read the value of a bit from memory instantly. (Unless we are doing lots of loads/stores that are not next to each other.)
- The computer follows some pre-loaded program that tells it what and how to compute.
- The computer starts by loading an input into its memory.
- The computer then runs for a while, doing computations and producing output.

- After a while the computer may halt.
- A computer is able to toss a random coin at any stage.

In order to make mathematical computations, a computer needs to able to represent and store mathematical objects. We list below the most basic mathematical objects and the representations usually used to represent them on a computer.

- Integers are usually written in binary.
- Real numbers are 'represented' by approximations.
- Vectors are usually written as a list or array of numbers.
- Matrices are written as a list of vectors.
- Rational numbers are represented as a list of two integers, or 'represented' by approximations.

The above list does not address certain practical issues, such as how to represent the end of one integer and the start of the next in its memory. We also have to consider issues of efficiently storing mathematical objects, for example by determining that it takes $\lceil \log_2 n \rceil$ bits to represent a positive integer n. More generally, we have to specify efficient representations of complicated mathematical objects. For example, a finite graph of vertices and edges (or alternatively a network of nodes) is generally represented by a 0-1 adjacency or incidence matrix.

1.6 Problems

A **problem** is defined by specifying an **input** (including its representation) and one or more corresponding valid **outputs**. Such a formal specification is required in order to make precise meaningful statements about the precise fundamental underlying difficulty of a problem. We now give two examples of formally specified problems about basic arithmetic and two examples about graphs (a more complicated mathematical structure). A *clique* in a graph is a subgraph in which all vertices are connected to each other, and a *Hamiltonian path* in a graph is a path through the graph that visits each vertex exactly once.

- **Problem**: BINARY ADDITION.
  Input: Two integers a and b, written in binary.
  Output: The integer $a + b$, written in binary.

- **Problem:** PRIME FACTOR.
  Input: An integer n (in binary), $n > 1$.
  Output: A prime integer p (in binary) where p divides n.

- **Problem**: CLIQUE.
  Input: A graph G (as an incidence matrix) and a positive integer k.
  Output: The answer to: Does G contain a clique of order k?

- **Problem:** HAMILTONIAN PATH.
  Input: A graph G (as an incidence matrix).
  Output: A Hamiltonian path (written as a list of vertices), or 'Graph not Hamiltonian' if no such path exists.

1.7 Algorithms

An **algorithm** is a finite sequence of steps to (correctly) solve a problem. An algorithm can be thought of as a computer program that always correctly solves a problem given enough time and memory. A computer that runs the algorithm outputs the answer to the problem and then halts. We can further classify algorithms into deterministic algorithms (often just termed algorithms) and probabilistic algorithms. A *deterministic algorithm* (often just termed an "algorithm") is one where no random choices are permitted, and a *probabilistic algorithm* is one where random choices are permitted. We now give some examples of algorithms.

- The BASE 10 ADDITION problem is solved by "school arithmetic".

- The BINARY ADDITION problem is solved by a modification of the obvious modification of this "school arithmetic".

- The GCD problem of finding the greatest common divisor of two integers is solved by the Euclidean algorithm.

- The HAMILTONIAN PATH problem can be solved as follows. For each permutation of the vertices of the graph, check if the corresponding path is Hamiltonian. If it is, output the path and halt. If no path is found, output 'Graph not Hamiltonian'.

The critical question in assessing the quality of an algorithm is to determine its computational efficiency. For example, the first three algorithms above are fairly efficient, but some improvements are possible. However, the Hamiltonian Path algorithm given above is hopelessly inefficient as it potentially involves checking $n!$ possible paths for a graph with n vertices.

1.8 Complexity

The foundation of much Public Key Cryptography involves assessing the complexity of basic operations on integers, such as addition and multiplication. We have seen that integers are represented on computers as sequences of words, that is to say ultimately as strings of bits. In particular, Public Key Cryptography can involve very large integers, often termed "big numbers" or "bignums", represented ultimately as very long bit strings. Thus simple operations like addition or multiplication in Public Key Cryptography typically correspond to many bit operations. We therefore typically assess the complexity of an operation in integer arithmetic by counting, or at least approximating in some sense, the number of bit operations required to perform the operation. Furthermore, such a method of determining the complexity does not depend on the computer used to carry out the computation.

In Public Key Cryptography, we are generally very concerned about the actual computational time required to perform various tasks, which we can in general consider as being approximately proportional to the number of bit operations required to carry out the task. However, obtaining very precise estimates is in practice usually too complicated, and such precise estimates are not always very illuminating. However, the integers involved in Public Key Cryptography are usually very large, so we usually only concern ourselves with asymptotic complexity estimates for the running time of algorithms.

The *computational complexity* of a computational problem is described in terms of the input (or output) size of a computational problem in bits. In order to give an asymptotic approximation for the computational cost in terms of input bits, we generally use the **Big O Notation**.

Big O Notation. Let $f(n)$ and $g(n)$ be functions from the natural numbers to the positive real numbers. We write $f = O(g)$ if there is a constant $c > 0$ and an integer N such that for all $n > N$

$$f(n) \leq cg(n)$$

The big O notation $f = O(g)$ means that the function f is eventually (asymptotically) dominated by a multiple of the function g. We use the big O notation when assessing computational complexity in order to focus on the most time consuming parts of the problem (asymptotically for large input sizes) to avoid getting bogged down in the generally irrelevant (from a complexity viewpoint) details which are unimportant for large input sizes. Given a function f, we can determine a function g such that $f = O(g)$ by

determining the biggest component of f asymptotically and then removing any co-efficient. For example, we have

$$3n^2 + 2n + 1 = O(n^2), \qquad O\left(\sqrt{n^3 + 9n - 3}\right) = O(n^{\frac{3}{2}}),$$
$$n^{100} + 2^n = O(2^n) \qquad \text{and} \qquad \log(n) + \log(\log(n)) = O(\log(n)).$$

We now consider the computational complexity of integer addition and multiplication in terms of this big O notation. The notation $\log(n)$ is generally used for $\log_2(n)$ in cryptography for convenience and also because the logarithm base is irrelevant for the big O notation as $\log_2 n = (\log_2 e)(\log_e n)$ is a multiple of $\log_e n$.

Addition. Consider the addition of two positive integers m and n. The integer m can essentially be represented with $\log(m)$ bits and the integer n with $\log(n)$ bits. Using a binary adaptation of school arithmetic, we can represent this sum using $\max(\log(m), \log(n))$ columns. Loosely speaking, each column can be processed with a finite and bounded of bit operations in addition, so the total complexity is essentially proportional to the number $\max(\log(m), \log(n))$ of columns. Thus the addition of integers m and n takes $O(\max(\log(m), \log(n)))$ bit operations. This complexity is also phrased in terms of the less precise formulation that the addition of two positive integers m and n with $m, n < N$ takes $O(\log N)$ bit operations with basic arithmetic.

Multiplication. Consider the multiplication of two positive integers m and n with $m, n < N$. The integers m and n can essentially be represented within a binary adaptation of school arithmetic (long multiplication) in $\log N$ columns. Loosely speaking, each column bit for m is multiplied by each column bit for n in the first and most time-consuming part of the operation, which takes $O((\log N)^2)$ bits operations. The final addition part of long multiplication takes $O(\log N)$ bit operations. Thus the multiplication of these integers m and n takes $O((\log N)^2) + O(\log N) = O((\log N)^2)$ bit operations with basic arithmetic.

1.9 Fast Multiplication Algorithms

The Karatsuba method of multiplication is a far faster method than the basic school long multiplication method described above. This method multiplies two k-bit integers a and b with $0 < a, b < 2^k = N$ to produce the $2k$-bit integer ab using much less than $O((\log N)^2) = O(k^2)$ operations.

We suppose without loss of generality that $k = 2^x$ is a power of 2 in order to simplify the description and we let $M = 2^{\frac{1}{2}k}$. We can regard the k-bit

integer a as consisting of the concatenation of two $\frac{1}{2}k$-bit strings a_1 and a_0, so $a = (a_1|a_0)$ as a bit string and $a = a_0 + Ma_1$ as an integer. Similarly, we can split k-bit b into the concatenation of two $\frac{1}{2}k$-bit strings b_1 and b_0, with $b = (b_1|b_0)$ as a bit string and $b = b_0 + Mb_1$ as an integer.

Our first step in calculating the product ab of two k-bit integers a and b is to calculate the three products

$$s_0 = a_0b_0,\ s_1 = a_1b_1 \text{ and } s_2 = (a_0 + a_1)(b_0 + b_1).$$

of $\frac{1}{2}k$-bit integers. We can then construct the product

$$\begin{aligned} ab &= a_0b_0 + M(a_0b_1 + a_1b_0) + M^2a_1b_1 \\ &= s_0 + M(s_2 - s_0 - s_1) + M^2s_1 \\ &= s_0 + 2^{\frac{1}{2}k}(s_2 - s_0 - s_1) + 2^k s_1 \end{aligned}$$

using three products of $\frac{1}{2}k$-bit integers, six additions and two multiplications by $M = 2^{\frac{1}{2}k}$, which being a power of 2 is simply a trivial bit string shift. We can then calculate each of these three products of $\frac{1}{2}k$-bit integers as three products of $\frac{1}{4}k$-bit integers, so we can calculate ab using nine products $\frac{1}{4}k$-bit integers and some additions and shifts, and so on.

The improvement in speed for the Karatsuba method essentially occurs as we replace a k-bit multiplication by three $\frac{1}{2}k$-bit multiplications, which take about $\frac{3}{4}$ of the time. More generally, f we repeat this process x times, we can find ab by using $3^x = 3^{\log_2 k} = 3^{\log_2(3)\log_3(k)} = k^{\log_2(3)} = k^{1.58}$ bit operations (multiplications of 1-bit integers), which can also shown to bound the number of bit operations required by the addition operations. The Karatsuba multiplication of k-bit integers therefore has a complexity of $O(k^{\log_2 3}) = O(k^{1.58})$ bit operations, a significant improvement on the quadratic $O(k^2)$ bit operations given by the basic long multiplication algorithm described above.

It is possible to construct multiplication algorithms with a lower asymptotic complexity than Karatasuba multiplication. The Schönhage-Strassen Fast Fourier Transform (FFT) algorithm for multiplying two k-bit positive integers numbers using $O(k\log(k)\log(\log(k)))$ bit operations.. Thus the complexity calculations indicate that the Schönhage-Strassen is ultimately the fastest of the three multiplication algorithms discussed here for large inputs, though the other two algorithms are faster for smaller inputs. The school long multiplication method is the faster than Karatsuba for integers up to around 300-500 bits (about 90-150 decimal digits), and for example no-one would use Karatsuba to calculate 11×13 efficiently. The

Schönhage-Strassen Fast Fourier Transform (FFT) becomes a faster method than Karatsuba for integers exceeding about 30000 bits (about 10000 decimal digits). For Public Key Cryptography, the integers used typically have size about 1000 bits, so Karatsuba is the fastest method for multiplying integers typically used in cryptography.

1.10 Modular Arithmetic

A similar analysis of "school" long division as was given above for "school" long multiplication shows that for positive k-bit integers $x, y < 2^k = N$, the calculation of the quotient and remainder for the division of x by y has a a computational complexity of $O(k^2) = O((\log N)^2)$. We can also obtain the following complexity results for modular arithmetic.

- The reduction of a modulo p has complexity $O((\log a)(\log p))$.
- The addition of residues modulo p has complexity $O(\log p)$.
- The multiplication of residues modulo p has complexity $O((\log p)^2)$.

It is also possible to conduct multiplication modulo N by conducting integer multiplication of integers of size approximately N, for example using Montgomery's method. Thus we can perform multiplication of residues modulo a k-bit integer p in a method with time complexity $O((\log p)^{1.58})$ using Karatsuba methods, or potentially even faster using FFT methods.

1.11 Polynomial Time Algorithms

The efficiency of an algorithm in Public Key Cryptography is generally assessed according to the algorithm's time complexity function in its big O formulation, with some caveats as discussed above. In particular, this leads a classification of an algorithm according to whether its time complexity function can be expressed as a polynomial function, as specified in the definitions below.

Definition. An algorithm A is **polynomial time** if its time complexity function for a k bit input is $O(k^d)$, where d is a fixed positive integer.

Definition. An algorithm A is **exponential time** if its time complexity function for a k bit input is bounded above by c^k and bounded below by $(c')^k$ for some constant $c > c' > 1$.

Algorithms that run in polynomial time are considered "easy" to implement (at least in principle) and exponential algorithms are considered

Input: a, n, m.
Output: $b = a^m \mod n$
1. $i = \lfloor \log(m) \rfloor - 1$
2. $b = a$
3. while $(i \geq 0)$ do
4. $\quad b = b^2 \mod n$
5. $\quad$ if ith bit of m is 1 then $b = ba \mod n$
6. $\quad i = i - 1$
7. Return b

Figure 1: **Square and Multiply Algorithm** to compute $a^m \bmod n$.

"hard" to implement. For example, all of the multiplication algorithms considered above are polynomial time (at worst quadratic in the input size). By contrast, exhaustively searching an n-bit key space has a time complexity of $O(2^n)$ so has exponential time complexity. We also note that that are algorithms with a "super-polynomial" time complexity that are not exponential time algorithms. However, we can regard the goal at a high level in constructing a public key cryptosystem is to ensure a legitimate user only has to use "easy" polynomial time algorithms, whilst ensuring that an illegitimate adversary has to use a "hard" (ideally exponential time) algorithm to extract useful information from the cryptosystem.

1.12 Modular Exponentiation

The calculation of $a^m \bmod n$ for integers a, m simply by multiplying a by itself m times and then performing a reduction modulo n is an example of an exponential algorithm. The input to the algorithm is n, a and m, with a total input length of largest $k = 3\lceil \log n \rceil$ bits, where each of n, a and m is a $(\frac{1}{3}k)$-bit integer. The repeated multiplication takes $m-1$ multiplications of $(\frac{1}{3}k)$-bit integers. This takes at least $(m-1)(\frac{1}{3}k) \approx 2^{\frac{1}{3}k}(\frac{1}{3}k)$ bit operations, that is to say an exponential number of bit operations. Another example of an exponential algorithm is trial division to find a non-trivial factor of an integer.

Square and Multiply is a far faster way of performing modular exponentiation, and is detailed in Figure 1. In the Square and Multiply algorithm, the exponent is processed bit-by-bit. The introduction of each new bit requires a squaring, followed by a multiplication if the new bit is a 1. For example,

in the computation of $2^{25} \bmod 11$, the exponent 25 is written as 11001 in binary, and the consequent Square and Multiply calculations to show that $2^{25} = 10 \bmod 11$ are given below.

$$\begin{array}{rcrcl} 2^1 & = & 2^1 & = & 2 \bmod 11 \\ 2^2 & = & 2^{10} & = & 2^2 = 4 \bmod 11 \\ 2^3 & = & 2^{11} & = & 2 \times 5 = 8 \bmod 11 \\ 2^6 & = & 2^{110} & = & 8^2 = 64 = 9 \bmod 11 \\ 2^{12} & = & 2^{1100} & = & 9^2 = 81 = 4 \bmod 11 \\ 2^{24} & = & 2^{11000} & = & 4^2 = 16 = 5 \bmod 11 \\ 2^{25} & = & 2^{11001} & = & 2 \times 5 = 10 \bmod 11 \end{array}$$

The main loop of the Square and Multiply algorithm is repeated once for every bit in the exponent, so this loop is repeated $O(\log(m))$ times. Every iteration involves a squaring modulo n, which can be performed in time $O((\log n)^2)$ and, if the bit is set to 1, a multiplication modulo n which is also time $O((\log n)^2)$ with basic multiplication, giving a total time complexity of $O(\log(m)(\log n)^2)$. Square and Multiply is therefore much faster than the naive "Repeated Multiplication" of computing a^m in $\mathbb{N}$ and then performing reduction modulo n. Such modular exponentiation is implemented in computer algebra packages. For example, the computation of $12^{34567890} \bmod 311$ to give 146 is implemented in Mathematica by

$$\texttt{PowerMod}[\texttt{12}, \texttt{34567890}, \texttt{311}] \quad < \texttt{SHIFT} - \texttt{RETURN} >.$$

There has been much research on speeding-up the process of exponentiating a number modulo n and there is an extensive literature on the topic. However, the methods found have the same asymptotic "big O" complexity, but the constants hidden in the "big O" complexity estimate can be reduced. For example, the exponent $m = 2^{16} + 1 = 65537$ is used in many RSA banking standards as it is a short integer with only two 1s, so modular exponentiation is much faster.

1.13 Probabilistic Computation

In Public Key Cryptography, we often need to use randomness in computations, for example for generating a private key randomly. Another example is the Miller–Rabin algorithm for testing whether an integer is a prime or a composite integer. We clearly have to adapt the deterministic approach in order to assess such a probabilistic algorithm.

We normally assess of a probabilistic algorithm by measuring its expected complexity. Suppose we have a probabilistic algorithm M that, on

any input x, halts with probability 1. The expected time before M halts on an input x is defined by

$$\sum_{t=0}^{\infty} t \cdot p_t(x),$$

where $p_t(x)$ is the probability that M halts after t steps. If this sum is always finite, then we can define the expected running time of this algorithm as

$$ET_M(k) = \max \left\{ t \;\middle|\; \begin{array}{l} \text{There exists an } k\text{-bit input } x \text{ such that} \\ \text{the expected time before } M \text{ halts is } t \end{array} \right\}.$$

Example: A probabilistic algorithm to choose an integer between 0 and n uniformly at random, where n is a k-bit integer.

- Generate k random bits $b_1, b_2, \ldots, b_k$.
- If $a = b_1 b_2 \ldots b_k$ belongs to $\{0, 1, \ldots, n\}$ then output a and halt.
- Otherwise repeat the process.

This algorithm might run for ever, but this is happens with probability 0. The worst case (for a fixed k) is when $n = 2^{k-1}$, in which case we have to repeat the generation step with probability $\frac{1}{2}$. The probability that the algorithm needs x attempts is given by

$$\left(\frac{(2^k - n - 1)}{2^k} \right)^{x-1} \frac{n+1}{2^k} < \frac{1}{2^{x-1}}.$$

Each attempt takes at most ck bit operations, where c is a constant, so o the expected time compexityis at most

$$\sum_{x=1}^{\infty} ckx \cdot 2^{-(x-1)} = 4ck = O(k).$$

We can generalise the idea of a polynomial time algorithm to this probabilistic setting. Thus a probabilistic algorithm M has *polynomial expected running time* if and only if there exists a polynomial $p(k)$ such that $ET_M(k) \leq p(k)$ for all $k \in \mathbb{N}$. Probabilistic algorithms with polynomial expected running time are regarded as "efficient", so the random number algorithm above is efficient.

2 Further Algorithms and Complexity

In this Section, we discuss some of the basic number theoretic functions used in Public Key Cryptography and show how to calculate these quantities efficiently with appropriate algorithms. The Section concludes with a discussion on use of Turing Reductions, which are a device for allowing us to discuss the relative difficulty of various Problems.

2.1 Euclidean Algorithm and Extended Euclidean Algorithm

The basic Euclidean algorithm calculates the greatest common divisor of two positive integers. This algorithm iteratively uses the Euclid division formula that for positive integers a and b with $a > b$ there exist positive integers q and r such that $a = bq + r$, where the non-negative "remainder" r does not exceed b, that is $0 \leq r < b$. This algorithm terminates after a finite number of steps as the non-negative remainder r decreases at each iteration. The Euclidean algorithm to determine $\gcd(a, b)$ is implemented in Mathematica by `GCD[a,b]`. A simple way to express Euclid's algorithm to calculate the greatest common divisor $\gcd(a, b)$ of positive integers a and b with $a \geq b$ is by the recursive expression

$$\gcd(a, b) = \gcd(b, a \bmod b), \qquad \text{with } \gcd(a, 0) = a \text{ at the conclusion.}$$

The greatest common divisor of 323 and 113 can thus be calculated as

$$\gcd(323, 113) = \gcd(113, 57) = \gcd(57, 19) = \gcd(19, 0) = 19.$$

The Extended Euclidean algorithm for positive integers a and b calculates their greatest common divisor $r = \gcd(a, b)$ and also integers s and t such that $sa + tb = r$, and is specified in Figure 2. The Extended Euclidean algorithm is used for example to find a private key d in RSA with modulus N and public key e satisfying $ed = 1 \bmod \varphi(N)$ by solving $de + \lambda\varphi(N) = 1$. The Extended Euclidean algorithm to determine $r = \gcd(a, b)$ and the multipliers s and t is implemented in Mathematica by `ExtendedGCD[a,b]`.

The speed of the basic Euclidean algorithm can be improved by using an adapted Euclid division formula $a = bq + r$ with $|r| \leq \frac{1}{2}|b|$, that is to say with potentially negative "remainders" of half the previous absolute size. Further improvements can be made by making extensive use of operations such as division by powers of 2 which use few bit operations. For such algorithms, the time complexity of a Euclidean algorithm for input positive integers $a, b < N$ is $O((\log N)^2)$. Similarly, computing a modular inverse $a^{-1} \bmod n$ has time complexity $O((\log n)^2)$.

Input: a, b
Output: $d = \gcd(a, b)$ and $s, t \in \mathbb{Z}$ such that $d = sa + tb$
1. $r_{-1} = a, s_{-1} = 1, t_{-1} = 0$
2. $r_0 = b, s_0 = 0, t_0 = 1$
3. $i = 0$
4. while $(r_i \neq 0)$ do
5. $\quad i = i + 1$
6. $\quad$ find (q, r_i) such that $0 \leq r_i < |r_{i-1}|$ and $r_{i-2} = qr_{i-1} + r_i$
7. $\quad s_i = s_{i-2} - qs_{i-1}$
8. $\quad t_i = t_{i-2} - qt_{i-1}$
9. Return $r_{i-1}, s_{i-1}, t_{i-1}$

Figure 2: **Extended Algorithm** to compute $r = \gcd(a, b)$ and integers s, t such that $sa + tb = r$.

2.2 Legendre Symbol

The Legendre symbol is a multiplicative function specifying whether or not a value a is a square, or more formally a *quadratic residue*, modulo a given odd prime p. The Legendre symbol is formally specified in the Definition below, but essentially the Legendre symbol of a with respect to p is 1 if nonzero a is a square modulo p and -1 if if nonzero a is a non-square modulo p. Thus the Legendre symbol of $1 = 1^2 = 4^2 \bmod 5$ and $4 = 2^2 = 3^2 \bmod 5$ with respect to the prime 5 is 1, and the Legendre symbol of 2 and 3 with respect to 5 is -1.

Definition. The *Legendre symbol* $\left(\frac{a}{p}\right)$ for an integer a and an odd prime p is defined to be

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } x^2 = a \bmod p \text{ has a solution.} \\ 0 & \text{if } p \mid a \\ -1 & \text{otherwise.} \quad \square \end{cases}$$

In this notation, $\left(\frac{1}{5}\right) = \left(\frac{4}{5}\right) = 1$ and $\left(\frac{2}{5}\right) = \left(\frac{3}{5}\right) = -1$, with $\left(\frac{0}{5}\right) = 0$. The following Theorem gives properties of the Legendre symbol.

Theorem. The *Legendre symbol* $\left(\frac{a}{p}\right)$ for an integer a and an odd prime p has the following properties.

(i) *Modular Reduction.* $\left(\frac{a}{p}\right) = \left(\frac{(a \bmod p)}{p}\right)$ and $\left(\frac{1}{p}\right) = 1$.

(ii) *Euler's criterion.* $\left(\frac{a}{p}\right) = a^{(p-1)/2} \bmod p$.

(iii) *Multiplicative property.* $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$.

(iv) *Squareness of* $-1 \bmod p$. $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$.

(v) *Squareness of* $-1 \bmod p$. $\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}$.

(vi) *Quadratic Reciprocity* $\left(\frac{p}{q}\right) = (-1)^{(p-1)(q-1)/4}\left(\frac{q}{p}\right)$ for odd primes p and q, so $\left(\frac{p}{q}\right) = \left(\frac{q}{p}\right)$ unless $p, q = 3 \bmod 4$.

The parts of this Theorem can be used to efficiently compute a Legendre symbol by repeatedly reducing this computation to the calculation of simpler Legendre symbols.

Example. The Legendre symbol of 391 with respect to the prime 859 is

$$\begin{aligned}\left(\frac{391}{859}\right) &= \left(\frac{17}{859}\right)\left(\frac{23}{859}\right) = -\left(\frac{859}{17}\right)\left(\frac{859}{23}\right) = -\left(\frac{9}{17}\right)\left(\frac{8}{23}\right) \\ &= -\left(\frac{3}{17}\right)^2\left(\frac{2}{23}\right)^3 = -\left(\frac{2}{23}\right) = -1,\end{aligned}$$

so 391 is a non-square modulo 853. In more detail, this calculation is given by

- Multiplicative property (iii) gives $\left(\frac{391}{859}\right) = \left(\frac{17}{859}\right)\left(\frac{23}{859}\right)$.
- Quadratic Reciprocity (vi) gives $\left(\frac{17}{859}\right) = \left(\frac{859}{17}\right)$ as $17 = 1 \bmod 4$.
- Quadratic Reciprocity property (vi) gives $\left(\frac{23}{859}\right) = -\left(\frac{859}{23}\right)$ as both $23 = 3 \bmod 4$ and $859 = 3 \bmod 4$.
- Modular reduction property (i) and multiplicative property (iii) gives $\left(\frac{859}{17}\right) = \left(\frac{9}{17}\right) = \left(\frac{3}{17}\right)^2 = 1$ as $\left(\frac{3}{17}\right) = \pm 1$.
- Modular reduction property (i) and multiplicative property (iii) gives $\left(\frac{859}{23}\right) = \left(\frac{8}{23}\right) = \left(\frac{2}{23}\right)^3 = \left(\frac{2}{23}\right)$ as $\left(\frac{2}{23}\right) = \pm 1$.

- Property (v) shows that $\left(\frac{2}{23}\right) = (-1)^{(23^2-1)/8} = (-1)^{66} = 1$.

This example demonstrates that a Legendre symbol can be calculated efficiently. More generally, $\left(\frac{a}{p}\right)$ can be calculated in polynomial time for $|a| < p$ using Euler's criterion (iii).

2.3 The Jacobi Symbol

The Jacobi symbol is a generalisation of the Legendre symbol from the case where the "denominator" is an odd prime to the "denominator" being a general composite odd positive integer, as specified in the Definition below.

Definition. The *Jacobi symbol* $\left(\frac{a}{n}\right)$ for an odd integer $n = \prod_{i=1}^{k} p_i^{e_i}$, where $p_1, \ldots, p_k$ are primes with corresponding Legendre symbols $\left(\frac{a}{p_i}\right) \ldots \left(\frac{a}{p_i}\right)$ is

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i}.$$

We first note that if a is a square or quadratic residue modulo n and coprime to n, then $\left(\frac{a}{n}\right) = 1$. However, the converse need not be true for composite n. For example, $\left(\frac{2}{15}\right) = 1$, but 2 is not a square modulo 15. More generally, the properties of the Jacobi symbol are given in the Theorem below.

Theorem. The *Jacobi symbol* $\left(\frac{a}{n}\right)$ for an integer a and an odd integer n has the following properties.

(i) Modular Reduction: $\left(\frac{a}{n}\right) = \left(\frac{(a \bmod n)}{n}\right)$ and $\left(\frac{1}{n}\right) = 1$.

(ii) Multiplicative property: $\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$.

(iii) $\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}$.

(iv) $\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8}$.

(v) Quadratic Reciprocity: $\left(\frac{m}{n}\right) = (-1)^{(m-1)(n-1)/4}\left(\frac{n}{m}\right)$ for odd integers m and n, so $\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right)$ unless $m, n = 3 \bmod 4$.

This Theorem shows that the Jacobi symbol satisfies the same criterion as the Legendre symbol except for Euler's criterion, so we cannot use Euler's criterion to compute Jacobi symbols. In general, we use quadratic reciprocity in a recursive way together with stripping out powers of 2 to compute $\left(\frac{m}{n}\right)$ where n is odd. To compute $\left(\frac{m}{n}\right)$, we first reduce m modulo n, and if this reduction is 0 or ± 1 we can directly obtain the Jacobi symbol. More generally, we can then write $m = 2^e q$ for odd q. and obtain the value $\left(\frac{2^e}{n}\right) = \left(\frac{2}{n}\right)^e$ We can then use quadratic reciprocity to relate $\left(\frac{q}{n}\right)$ with $\left(\frac{n}{q}\right)$, and then repeat the process.

Example. The Jacobi symbol $\left(\frac{303}{451}\right)$ can be calculated as

$$\begin{aligned}\left(\frac{303}{451}\right) &= \left(\frac{3}{451}\right)\left(\frac{101}{451}\right) = -\left(\frac{451}{3}\right)\left(\frac{451}{101}\right) = -\left(\frac{1}{3}\right)\left(\frac{47}{101}\right) \\ &= -\left(\frac{1}{3}\right)\left(\frac{101}{47}\right) = -\left(\frac{1}{3}\right)\left(\frac{7}{47}\right) = \left(\frac{1}{3}\right)\left(\frac{47}{7}\right) = \left(\frac{1}{3}\right)\left(\frac{5}{7}\right) \\ &= \left(\frac{1}{3}\right)\left(\frac{7}{5}\right) = \left(\frac{1}{3}\right)\left(\frac{2}{5}\right) = 1 \times -1 = -1.\end{aligned}$$

There is a connection between computing Jacobi symbols using quadratic reciprocity and Euclid's algorithm, related to continued fraction convergents. We can therefore show that the complexity for computing Jacobi symbols is $O((\log p)^2)$. The Jacobi symbol $\left(\frac{a}{n}\right)$ is implemented in Mathematica as `JacobiSymbol[a,n]`.

2.4 Chinese Remainder Theorem

The Chinese Remainder Theorem (CRT) allows us to solve the simultaneous linear modulo equations

$$x = c_1 \bmod m_1 \text{ and } x = c_2 \bmod m_2 \quad \text{for } \gcd(m_1, m_2) = 1,$$

and we note that the CRT generalises to more than two such linear modulo equations. The CRT asserts that this module equation system has a unique solution for x modulo $m_1 m_2$ given by

$$x = c_1 + (c_2 - c_1)\ (m_1^{-1} \bmod m_2)\ m_1.$$

Example. Consider $x = 1 \bmod 5$ and $x = 2 \bmod 7$, so $c_1 = 1$, $m_1 = 5$ and $c_2 = 2$, $m_2 = 7$. Thus $m_1^{-1} = 5^{-1} = 3 \bmod m_2 = 7$, and so we have $x = 1 + (2-1) \times 3 \times 5 = 16 \bmod m_1 m_2 = 35$. □

In a CRT problem, we can suppose that $0 \leq c_1 < m_1$ and $0 \leq c_2 < m_2$ with $m_1 < m_2$, so all numbers in the problem are at most m_2 and the input size is $O(\log m_2)$. Computing the modular inverse $m_1^{-1} \bmod m_2$ has time complexity $O((\log m_2)^2)$, and the multiplications can all be performed in time $O((\log m_2)^2)$, so the total time complexity of computing a CRT solution is $O((\log m_2)^2)$. The Chinese remainder theorem is implemented in Mathematica as `ChineseRemainder`$[\{\mathtt{c}_1, \mathtt{c}_2\}, \{\mathtt{m}_1, \mathtt{m}_2\}]$

2.5 Computation of Square Roots

It is a straightforward matter to compute an approximation to a square root in the real numbers $\mathbb{R}$ using Newton's method (also called the Newton-Raphson method). An integer value $\lfloor \sqrt{n} \rfloor$ can then be obtained if required by computing to an appropriate precision and then rounding. More sophisticated methods show that integer square roots of m-bit numbers can be computed in time proportional to the cost of a multiplication of two m-bit numbers. Similarly, other roots (such as cube roots) can be computed in polynomial time. Real-valued square roots can be computed in Mathematica as `Sqrt[a]//N`.

An important computational problem n Public Key Cryptography is the ability to compute square roots modulo a prime p, This is a particularly important problem in Public Key Cryptography, for example in Rabin decryption and Elliptic Curve cryptography. We first note that non-zero modular square roots occur in "plus-minus" pairs as $x^2 = y^2 \bmod p$ implies that $x = \pm y \bmod p$. It follows that half the integers $1 \leq a < p$ are squares modulo p. Furthermore, the decision problem "Is a a square modulo p?" is soluble in polynomial time, since we can compute the Legendre symbol $\left(\frac{a}{p}\right)$ in polynomial time. This fact does not necessarily imply though that the computational problem "Find a square root of a modulo p" is easy. However, finding a square root modulo p is straightforward when $p = 3 \bmod 4$, as the following Lemma shows.

Lemma. Suppose that p is a prime satisfying $p = 3 \bmod 4$. If a is a positive integer with Legendre symbol $\left(\frac{a}{p}\right) = 1$, then $x = a^{\frac{1}{4}(p+1)} \bmod p$ satisfies $x^2 = a \bmod p$.

Proof. By Euler's criterion, $x^2 = a^{\frac{1}{2}(p+1)} = aa^{\frac{1}{2}(p-1)} = a\left(\frac{a}{p}\right) = a \bmod p$.

2.6 Tonelli-Shanks Algorithm

Finding a square root modulo an odd prime p when $p = 1 \bmod 4$ takes more effort than when $p = 3 \bmod 4$. To do so, we repeatedly use the fact that if a has order rs modulo N, then a^r has order s modulo N and the following Lemmas.

Lemma. Suppose that $a \neq 0 \bmod p$ has order k for an odd prime p. If $q = lk$ is odd, then $x = a^{\frac{1}{2}(q+1)} \bmod p$ satisfies $x^2 = a \bmod p$.

Proof. We have $x^2 = a^{q+1} = a^q a = (a^k)^l a = a \bmod p$.

Lemma. Suppose that p be a prime, where $p - 1 = 2^e q$ and q is odd. If a is a square modulo p and $w = a^{\frac{1}{2}(q+1)} \bmod p$, then $w^2 = ab \bmod p$, where b has order dividing 2^{e-1}.

Proof. We have $w^2 = a^{q+1} = aa^q \bmod p$, so $b = a^q \bmod p$. However, a has order dividing $\frac{1}{2}(p-1) = 2^{e-1}q$, so $b = a^q \bmod p$ has order dividing 2^{e-1}.

To complete the computation of the square root of a modulo p, it is therefore sufficient to compute a square root of b. We will need the following lemma.

Lemma. If n is not a square modulo p and not divisible by p, so $\left(\frac{n}{p}\right) = -1$, then $y = n^q \bmod p$ has order 2^e.

Proof. If $y = n^q \bmod p$, then y has order dividing 2^e modulo p as $p-1 = 2^e q$. However, $y^{2^{e-1}} = n^{q2^{e-1}} = n^{\frac{1}{2}(p-1)/2} = -1 \bmod p$, so the order of y is 2^e.

If $\mathbb{F}_p$ denotes the finite field with p elements, then its multiplicative group $\mathbb{F}_p^*$ is a cyclic group in which y generates the full subgroup of elements of order dividing 2^e. Thus $b = y^i \bmod p$ for some i such that $1 \leq i \leq 2^e$. Furthermore, the order of b divides 2^{e-1} so i is even. We can therefore write $i = 2j$ and $x = wy^{-j} \bmod p$ to obtain $x^2 = w^2 y^{-2j} = aby^{-2j} = a \bmod p$ Thus if we can compute i then we can compute the square root of a.

If e is small then the value i can be found by a simple search. A more advanced method is to use a Pohlig-Hellman idea to solve the discrete logarithm of b to the base y (discussed when considering discrete logarithms) This idea leads to the Tonelli-Shanks algorithm for computing square roots modulo p, which is specified in Figure 3

Example. We compute the square root of $a = 3 \bmod p = 61$ by the Tonelli-Shanks algorith, We note that $\left(\frac{3}{61}\right) = 1$ and that $p - 1 = 60 = 2^2 \cdot 15$, so we take $e = 2$ and $q = 15$, and we note that $\left(\frac{2}{61}\right) = -1$. We therefore

Input: a, p such that $(\frac{a}{p}) = 1$
Output: x such that $x^2 = a \bmod p$
1. Write $p - 1 = 2^e q$ where q is odd.
2. Choose random integers $1 < n < p$ until $(\frac{n}{p}) = -1$
3. Set $y = n^q \bmod p$
4. Set $w = a^{\frac{1}{2}(q+1)} \bmod p$ and $b = a^q \bmod p$
5. Compute an integer i such that $b = y^i \bmod p$
6. Return $wy^{-\frac{1}{2}i} \bmod p$

Figure 3: **Tonelli-Shanks Algorithm** to compute a square root mod p.

set $y = 2^{15} = 11 \bmod 61$, $w = 3^{\frac{1}{2}(15+1)} = 3^8 = 34 \bmod 61$ and $b = 3^{15} = -1 \bmod 61$. Thus $x = wy^{-1} = 53 \bmod 61$ satisfies $x^2 = 3 \bmod 61$ as does $-x = -53 = 8 \bmod 6$.

This algorithm for computing square roots modulo p is a probabilistic algorithm with expected time complexity $O((\log p)^3)$, and is computed by the command `SqrtMod[a,p]` in Mathematica. If $p - 1$ is highly divisible by 2 then an algorithm due to Cipolla is more suitable, and there is also a deterministic algorithm due to Schoof.

2.7 Turing Reductions and Oracles

Turing Reductions are used in Public Key Cryptography to compare one Problem to another Problem by "reducing" the solution of one Problem to the solution of the other Problem, We illustrate the issue of Turing Reductions by revisiting the CLIQUE problem of Section 1.6, where a clique in a graph is a subgraph in which all vertices are connected to each other.

- **Problem**: CLIQUE.
  Input: A graph G (as an incidence matrix) and a positive integer k.
  Output: The answer to: Does G contain a clique of order k?

We can formally address the CLIQUE Problem by considering the related MAXCLIQUE Problem, stated below.

- **Problem**: MAXCLIQUE.
  Input: a graph G and an integer k
  Ouput: does the largest clique in G have order k?

If we can solve MAXCLIQUE for a graph and clique size or order k for different integers k, then we can use such MAXCLIQUE solutions to solve CLIQUE by solving MAXCLIQUE systematically for different values of k, as detailed below.

- **Process for solving** CLIQUE **using** MAXCLIQUE.
  Input: A graph with n vertices, and an integer k.
  Output: true if and only if G has a clique of order k
  Algorithm: For $i = k$ to n:
  if MAXCLIQUE(G, i) is true
  output true and halt.
  next i
  output false

Such a process for solving CLIQUE by using MAXCLIQUE is termed a *Turing reduction* from CLIQUE to MAXCLIQUE, and we use the notation CLIQUE$\leq_T$MAXCLIQUE to denote such a Turing reduction.

For a formal definition of a Turing reduction, we use the theoretical abstraction of an *oracle* $\mathcal{O}_B$ for a Problem B, which is a machine that can solve a Problem B in a single step. We can specify an oracle for any problem, even ones that we think are incredibly difficult. This abstraction to an Oracle $\mathcal{O}_B$ for a Problem B allows us to think about how we might use the solution to the Problem B without getting distracted the details of how to obtain such a solution for B. This allows us to "compare" the relative difficulties of problems in the manner and identify problems of equivalent difficulty, as specified below.

Definition. If A and B are problems, then Problem A is *Turing reducible* to Problem B if we can solve A in polynomial time when we have access to an oracle $\mathcal{O}_B$ for the Problem B. We write $A \leq_T B$ to denote that Problem A is Turing reducible to Problem B.

Definition. If A and B are problems where Problem A is Turing reducible to Problem B and vice versa, so $A \leq_T B$ and $B \leq_T$, then Problem A is *polynomial time equivalent* to Problem B.

The statement that "Problem A is Turing reducible to Problem B" or $A \leq_T B$ informally means that Problem A is "not significantly harder" than Problem B. The Problem A being Turing reducible to Problem B means we can solve Problem A with a "few" (polynomially many) invocations of the oracle $\mathcal{O}_B$ for Problem B and some negligible book-keeping, that is to

say we can solve Problem A given a "few" solutions to instances of Problem B. In particular, if $A \leq_T B$ and Problem B can be solved in polynomial time, then Problem A can be solved in polynomial time. Furthermore, we can potentially establish a chain of problems of "decreasing difficulty", as summarised in the Lemma below.

Lemma If A, B, C are problems with $A \leq_T B$ and $B \leq_T C$, then $A \leq_T C$.

To illustrate Turing reductions, we compare the CLIQUE problem and the INDEPENDENT SET problem. A graph G has an independent set of size k if it has a subgraph of size k such that no vertex in the subgraph is connected to any other vertex in the subgraph, or equivalently that the complementary graph G^c contains a clique of order or size k.

- **Problem**: CLIQUE.
  Input: A graph G (as an incidence matrix) and a positive integer k.
  Output: Answer to: Does G contain a clique of order k?

- **Problem**: INDEPENDENT SET.
  Input: A graph G and an integer k.
  Output: Answer to: Does G contain an independent set of order k?

We first show that INDEPENDENT SET is Turing reducible to CLIQUE. Given a graph G with n vertices specified by an $n \times n$ adjacency matrix A, we can specify the complementary graph G^c with $n \times n$ adjacency matrix $\bar{A}$, which is just the adjacency matrix A of G with the 1s and 0s swapped round, which takes n^2 bit operations. We can then use an Oracle $\mathcal{O}$ for the CLIQUE problem with input G^c and integer k to determine whether G^c has a clique of size k, which immediately solves the INDEPENDENT SET problem for the graph G and integer k. Thus we can solve INDEPENDENT SET in polynomial time given access to an Oracle for the CLIQUE problem, so

$$\text{INDEPENDENT SET} \ \leq_T \ \text{CLIQUE}.$$

A similar approach shows that CLIQUE $\leq_T$ INDEPENDENT SET, so these problems

CLIQUE and INDEPENDENT SET are polynomial time equivalent.

3 Introduction to Factoring-based Cryptosystems

This Section begins with a discussion of a formal approach to public key cryptography, which is potentially applicable to both factoring-based public key cryptosystems and other types of public key cryptosystem. The latter part of this Section discusses the factoring-based public key cryptosystems RSA and Rabin both within this formalism and more generally.

3.1 Formal Definitions for Public Key Cryptography

Public Key Cryptography has the following four main functionalities

- *Encryption* allows Bob to send a message to Alice in a private way so that no eavesdropper can learn the contents of the message.
- *Signatures* allows Alice to establish that a document was authorised by her.
- *Key Exchange* allows Alice and Bob to agree on some random data which can be used as a shared key to a symmetric cryptosystem.
- *Identification* allows Alice to prove that she is on-line or present at a physical location.

The challenge when constructing formal definitions is to capture these functionalities in a succinct and useful way. We also need to consider the properties required of each of these functionalities, so these can be captured in a succinct and useful way.

3.2 A Formal Specification for Public Key Encryption

The formal specification of a Public Key Encryption scheme uses a *security parameter* λ. For technical reasons, the security parameter is usually expressed in *unary*, that is to say as a string of 1s, where the length of the string specifies the security parameter λ. For Public Key Encryption, we can then specify the following spaces in terms of this security parameter λ.

- The *message space* M_λ is the space of all possible messages.
- The *public key space* PK_λ is the space of all possible public keys.
- The *secret key space* SK_λ is the space of all possible private keys.
- The *ciphertext space* C_λ is the space of all possible ciphertexts.

For Public Key Encryption, we then have to specify three algorithms in order to be able to encrypt and decrypt messages and ciphertexts.

- `KeyGen` is a randomised algorithm which takes the security parameter λ as input and outputs a public key $pk \in PK_\lambda$ and a private or secret key $sk \in SK_\lambda$. `KeyGen` runs in time $O(\lambda^c)$ for some constant, that is to say it has polynomial time in its input.

- `Encrypt` is a probabilistic algorithm which takes $m \in M_\lambda$ and $pk \in PK_\lambda$, runs in polynomial time and outputs $c \in C_\lambda$.

- `Decrypt` is an algorithm (not usually probabilistic) which takes $c \in C_\lambda$ and $sk \in SK_\lambda$, runs in polynomial time and outputs either $m \in M_\lambda$ or the invalid ciphertext symbol $\perp$. It is required that

  $$\texttt{Decrypt}(\texttt{Encrypt}(m, pk), sk) = m$$

  with very high probability if (pk, sk) are a matching key pair.

3.3 Notions of Security for Public Key Encryption

The concept of security varies according to the application. We generally assume that any attacker knows the specification of the cryptosystem in use (Kerckhoff's principle), so the security of the system ultimately depends on knowledge of the secret or private key sk. A *key recovery* attack or a *total break* is the ability of an adversary to compute the private key sk corresponding to a known public key pk.

There are other ways in which a Public Key Encryption scheme can be compromised that fall short of a total break but can still be highly damaging, such as obtaining some partial information about a message corresponding to a particular ciphertext. In order to capture this, we usually reason about the properties and abilities of an attacker or adversary.

Definition. An *Adversary* is a probabilistic algorithm which takes certain inputs and produces certain outputs. The inputs are typically a public key $pk \in PK_\lambda$ and a certain ciphertext $c \in C_\lambda$. The output is usually some information about the ciphertext c and its corresponding message $m \in M_\lambda$.

In order to assess whether an Adversary is able to obtain information about a ciphertext and its corresponding message, the Adversary is run with random inputs to see if the outputs produced by the Adversary are correct. Thus we essentially run experiments on the Adversary, usually termed *games*.

Depending on the circumstances, we may be interested in various different *security properties* or *attack goals*. Such security notions are usually phrased in terms of an Adversary succeeding with some *non-negligible* probability, where a negligible function is specified below

Definition. A function $\mu(x) : \mathbb{N} \to [0, 1]$ is *negligible* if $\mu(\lambda) < \frac{1}{\lambda^d}$ for every positive integer d when λ is sufficiently large.

The motivation for this definition is that if an event has negligible probability, then we would not expect a polynomial time algorithm to encounter this event when λ is sufficiently large. Conversely, we need to be potentially concerned if an event of interest happens with *non-negligible* probability, where a function μ is non-negligible if $\mu \geq \lambda^{-d}$ for some positive integer d for all sufficiently large λ. We now give three notions of security or *security properties* phrased in these terms.

- **One way encryption (OWE)**.
  Given a challenge ciphertext c an Adversary cannot compute the corresponding message m.

  The OWE property is usually phrased more technically in terms of the following game. We consider the Adversary A to be a polynomial time algorithm that takes ciphertext and public key (c, pk) as input and outputs a message $m \in M_\lambda$.

  - We run `KeyGen` on 1^λ to produce (sk, pk).
  - We generate a message $m \in M_\lambda$ uniformly at random and run `Encrypt` on (m, pk) to produce $c \in C_\lambda$.
  - We run the Adversary A on (c, pk) to produce m'.
  - We say that the Adversary A succeeds if $m = m'$.

  The cryptosystem satisfies the OWE property if no Adversary A is able to succeed in this game with non-negligible probability.

- **Semantic Security**.
  An Adversary learns no information at all about a message from its ciphertext, apart from possibly the length of the message.

  A more precise specification (Goldwasser-Micali) can be phrased in the following way. Suppose all messages in M_λ have the same length and that $f : M_\lambda \to \{0, 1\}$ is any function such that divides the message space in two, that is to say $\mathbf{P}(f(m) = 1 : m \in M_\lambda) = \frac{1}{2}$. We then

consider the Adversary A to be a polynomial time algorithm that takes ciphertext and public key (c, pk) as input and outputs a message $m \in M_\lambda$.

- We run `KeyGen` on 1^λ to produce (sk, pk).
- We generate a message $m \in M_\lambda$ uniformly at random and run `Encrypt` on (m, pk) to produce $c \in C_\lambda$.
- We run the Adversary A on (c, pk) to produce $a \in \{0, 1\}$.
- We say that the Adversary A succeeds if $f(m) = a$.

The cryptosystem satisfies the Semantic Security property if no Adversary A is able to succeed in this game for any such function f with probability $\frac{1}{2} + \epsilon$ where ϵ is non-negligible.

We note that it is always possible for an Adversary simply to guess the private key sk, so we require the Adversary to succeed with probability $\frac{1}{2} + \epsilon$ for some non-negligible ϵ rather than simply $\epsilon > 0$, so the Adversary is doing "better" than guessing. We also note that a more general definition of semantic security allows messages $m \in M_\lambda$ to be drawn according to any probability distribution rather than simply uniform.

- **Indistinguishibility (IND)**.
  An Adversary A cannot distinguish the encryption of any two messages m_0 and m_1 of the same length.

  The IND property can be more technically phrased in terms of a (two-part) game. We consider an Adversary A that can first produce two messages of equal length in M_λ and can then take ciphertext and public key (c, pk) and produce a single bit b.

  - We run `KeyGen` on 1^λ to produce (sk, pk).
  - The Adversary A produces two messages $m_0, m_1 \in M_\lambda$ of its choosing of equal length.
  - We choose a bit a and run `Encrypt` on m_a to give ciphertext c.
  - We run the Adversary A on (c, pk) to produce a bit $b \in \{0, 1\}$.
  - We say that the Adversary A succeeds if a equals b.

  The cryptosystem satisfies the IND property if no Adversary A is able to succeed in this game with probability at least $\frac{1}{2} + \epsilon$ where ϵ non-negligible. Loosely speaking, the Indistinguishability (IND) property

is that an Adversary A cannot tell whether a ciphertext c was generated from one of the two messages m_0 and m_1.

Example. Textbook RSA does not have the Semantic Security property as the Jacobi symbol of the ciphertext $(\frac{c}{N}) = (\frac{m}{N})$ is the Jacobi symbol of the message, so a Textbook RSA encryption leaks (at least) one bit of information about the plaintext.
Textbook RSA also does not have the Indistinguishability (IND) property. The Adversary A can simple encrypt both messages m_0 and m_1 to obtain ciphertexts c_0 and c_1. The Adversary A can then simply compare c_0 and c_1 with the test ciphertext c to determine which of the original messages m_0 and m_1 was used to produce the test ciphertext c. □

The above example for Textbook RSA illustrates that for a Public Key Encryption scheme to possess the Indistinguishability IND property the encryption process needs to include some randomisation. Thus Textbook RSA is usually modified in practical usage to include some randomisation as part of an encryption process. More generally, the Semantic Security property and the Indistinguishability IND property are equivalent under most attack scenarios. However, it is usually easier in practice to show that the Indistinguishability IND property holds.

3.4 Attack Models for Public Key Encryption

The attacks on Public Key Encryption that we have considered so far have been passive attacks, and such attacks can be thought of as roughly analogous with known plaintext attacks in symmetric cryptography. The chosen ciphertext attack models below are roughly analogous with the chosen plaintext attacks in symmetric cryptography attack, and are more sophisticated and powerful.

- **Passive Attack**.
  The Adversary is given the public key and challenge ciphertext only.
- **Lunchtime Attack (CCA1)**.
  The Adversary has the public key and can also ask for decryptions of ciphertexts of its choice during the first stage of the attack, that is before the challenge ciphertext is received.
- **Adaptive Chosen Ciphertext Attack (CCA2)**.
  The Adversary has the public key and is given access to a decryption oracle $\mathcal{O}$ which will provide decryptions of any ciphertext of its

choosing with the restriction that the challenge ciphertext cannot be submitted to $\mathcal{O}$.

The CCA2 scenario is a challenging theoretical setting for the use of a cryptosystem. If a cryptosystem can be shown to be secure even under CCA2, then we can has a lot of confidence in its security in real-world settings. However, the CCA2 attack scenario can also be realistic in practice. In 1998 Bleichenbacher showed a CCA2 attack against a standardised version of RSA encryption in a real-world system. The system has a server which transmits error information for certain incorrect SSL/TLS sessions.

One final detail which needs to be specified to determine the security of a Public Key Encryption system is the computational model of the Adversary, for example whether the Adversary has bounded or unbounded computational resources. More generally, it may be appropriate to specify whether we are using the standard model or the random oracle model in which hash functions are modelled as "random functions" in some sense. In this course we only consider a polynomial time Adversary, that is an Adversary which has appropriately bounded computational resources and in the standard model. The strongest notion of security then *indistinguishability* (IND) *under a* CCA2 *attack in the standard model*. In theoretical cryptography this notion of security is generally considered the minimal requirement for an encryption scheme. The first "practical" cryptosystem with this security level is an extension of El Gamal and was developed by Cramer and Shoup (1998).

3.5 A Formal Specification for Public Key Signatures

The motivation for a Digital Signature or a Public Key Signature is to produce a scheme that is analogous to a handwritten signature. For a digital signature, the generated signature must depend on the document being signed (unlike a handwritten signature) and must be publicly verifiable, and this digital signature is sent together with the document. A digital signature requires that it should only be possible to generate a valid signature should only be if the private or *signing* key of a public-private key pair is known, but that the signature should be publicly verifiable using the public or *verification* key. A signature algorithm has message, signature and key spaces, and a signature scheme is made up of the following algorithms.

- `KeyGen` is a randomised algorithm that generates a public and private key pair (pk, sk) and runs in probabilistic polynomial time.

- `Sign` is a randomised algorithm that takes as input a private key sk and a message m and outputs a signature s, and runs in probabilistic polynomial time.
- `Verify` is a (usually) deterministic algorithm that takes as input a public key pk, a message m and a signature s and outputs "valid" or "invalid", and runs in polynomial time. It is required that

  $$\texttt{Verify}(pk, m, \texttt{Sign}(sk, m)) = \text{Valid}$$

  with very high probability if (pk, sk) are a matching key pair.

3.6 Notions of Security for Public Key Signatures

Some notions of security to consider when analysing a Public Key or Digital Signature scheme are listed below.

- **Existential Forgery**.
  An Adversary chooses a message m and produces a signature s for that message m with knowledge of the public key pk.
- **Selective Forgery**.
  An Adversary receives a message m and produces a signature s for that message m with knowledge of the public key pk.
- **Universal Forgery**.
  An Adversary can produce a signature s for *any* message m with knowledge of the public key pk.
- **Key Recovery**.
  An Adversary can obtain the private key sk corresponding to the public key pk.

The notions of security are listed above in order of severity. For example, an existential forgery (knowledge of a single message-signature pair) is unlikely to be a serious security threat in practice. By contrast, a key recovery attack completely compromises a digital signature scheme. These notions of security for digital signatures can be formalised as before in terms of games.

3.7 Attack Models for Public Key Signatures

We can produce analogous attack models for Public Key Signatures to the attack models for Public Key Encryption, which are listed below.

- **Passive Attack**.
  The Adversary has to produce a forgery (a new signature) given access only to the public key.

- **Known Message Attack**.
  The Adversary has to produce a forgery given access to various sample message-signature pairs for a given public key.

- **Adaptive Chosen Message Attack**.
  The Adversary has to produce a forgery given access to Signature Oracle $\mathcal{O}$ which generates signatures for the given public key on messages of the Adversay's choosing (apart from any challenge message).

The strongest notion of security for signatures is security against *existential forgery under adaptive chosen message attack in the standard model*, which is usually consider the minimal level of security in theoretical cryptography.

3.8 Digital Signatures and Hash Functions

The naive approach to signing a large document would be to break it into small blocks and sign each small block individually. However, such a process would be very inefficient (as discussed above) as well as having other drawbacks. This a large document m is always "hashed" down to a small value $H(m)$ using an efficient hashing and public *hash* function H. The RSA signing process is then $s = H(m)^d \bmod N$. Thus we use an efficient hash function on the large document and a single implementation of say a 2048-bit RSA modular encryption in order to sign this document.

A hash function used in cryptography is more formally known as a *cryptographic hash function*, which is a deterministic algorithm H which maps arbitrary length bitstrings to bitstrings of a fixed length n (typically $n = 160$ or $n = 256$) with the properties listed below. There has been a long history of development of cryptographic hash functions. There is a tension in the design of cryptographic hash functions between the need for a fast algorithm and the need to satisfy these security properties, and many hash functions have been compromised. The U.S. National Institute of Standards and Technology (NIST) has recently standardised the cryptographic hash function SHA-3 (2015) as part of its Digital Signature Standard (DSS) after running a competition for hash function designs. A cryptographic hash function should possess the following security properties.

- **Preimage Resistance**.
  Given a n-bit string y it should be computationally infeasible to compute a bitstring x such that $H(x) = y$.

- **Second-preimage Resistance**.
  Given a bitstring x and a bitstring $y = H(x)$ it should be computationally infeasible to compute a bitstring $x' \neq x$ such that $H(x') = y$.

- **Collision Resistance**.
  It should be computationally infeasible to compute bitstrings $x \neq x'$ such that $H(x) = H(x')$.

A Digital Signature process actually "signs" the hash of a message, so any two messages that hash to the same value give rise to the same signature. The collision resistance property means that given a signature s for one message m it is computationally infeasible to compute another message m' with the same hash value and hence the same signature. If an attacker could find two messages m and m' with the same signature then they could potentially trick a genuine signer into signing the harmless message m and then insist they perform the action specified by m'. Similarly, a malicious signer could be repudiate (deny) a signature on m' by claiming that they had actually only actually signed the message m.

For a hash function with an n-bit output, the birthday paradox allows us to find collisions in time complexity $O(2^{\frac{1}{2}n})$. Thus it is possible to find a collision for a 160-bit hash function by making about 2^{80} hash function evaluations. This gives a rough comparison that finding collisions in a 160-bit hash function is roughly comparable in complexity to conducting a 80-bit key search of a block cipher, which is generally not currently thought to be feasible. However, if a signature scheme and hash function are secure, then a digital signature provides message integrity, message authentication and non-repudiation.

3.9 A Specification of the Textbook RSA Cryptosystem

We now give a specification of the Textbook RSA cryptosystem in terms of the formalism of Section 3.2, in which an algorithm's complexity is measured in terms of a security parameter λ. The message space and ciphertext space for textbook RSA are usually defined to be $C_\lambda = M_\lambda = \mathbb{Z}_N^*$, though technically we should take $C_\lambda = M_\lambda = \{\text{all } \lambda\text{-bit strings}\}$. The formal specifications for Textbook RSA algorithms are given below.

- `KeyGen`(λ). Generate two random $\frac{1}{2}\lambda$-bit primes p and q and set $N = pq$, so that N has λ bits. Choose a random λ-bit integer e coprime to $(p-1)$ and $(q-1)$ (or choose $e = 65537$), and set $d = e^{-1} \bmod \lambda(N)$, where $\lambda(N)$ is the Carmichael lambda function. The output is the public key $pk = (N, e)$ and the private key $sk = (N, d)$.
- `Encrypt`(pk, m). Compute $c = m^e \bmod N$ as output.
- `Decrypt`(sk, c). Compute $m = c^d \bmod N$ as output.

Enhancements to RSA to make it suitable for practical use include padding and randomisation, so the above specification would need to be modified for such versions of RSA. For completeness, we give below the specification of the signing and verification functions for the textbook version of an RSA digital signature.

- `Sign`(sk, m). Compute $s = m^d \bmod N$ as output.
- `Verify`(pk, m, s). Check whether m is $s^e \bmod N$ and output "valid" or "invalid" as appropriate.

In bulk encryption, we need to encrypt large documents, typically several megabytes. However, a practical implementation of RSA would for example have a 2048-bit modulus N, so we can only encrypt data blocks of 2048 bits (quarter of a kilobyte). Thus encrypting even a small 10 kilobyte document would require 40 RSA encryptions, so would be very slow. To obtain an efficient encryption system, public key encryption is usually only used as a *key transport* mechanism, that is to say to securely send a key for use in a symmetric algorithm. Such an encryption system requires a single use of a (relatively slow) public key cryptosystem such as RSA to ensure both parties share a common symmetric key. This common symmetric key is then used with a (fast) symmetric algorithm such as DES or AES in counter mode to encrypt the large document.

3.10 The RSA Cryptosystem in Practice

We have already seen that the deterministic Textbook RSA cryptosystem does not possess the Indistinguishability (IND) property. For this and other reasons, a message m that is to be encrypted using the RSA process would usually have further bits added before the RSA modular exponentiation from some *randomised padding scheme*. For example, if N is a typical 2048-bit RSA modulus used for the key transport of a symmetric 256-bit AES key,

Input: Public Key (N, e), Private Key (p, q) $[N = pq]$ and Ciphertext c.
Output: List of prime factors of n
1. Compute $d_p = e^{-1} \bmod p - 1$ and $d_q = e^{-1} \bmod q - 1$.
2. Compute $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$.
3. Compute m such that $m = m_p \bmod p$ and $m = m_q \bmod q$ with CRT.

Figure 4: **Fast RSA Decryption Algorithm** using CRT.

then most of the 2048-bit message block is such additional padding. More elaborate padding schemes will be described later.

The best factoring algorithms for an RSA-type number ("hardest" factoring problem) have subexponential complexity. The largest RSA-type numbers that have been factored are the challenge numbers RSA-768 (768 bits or 232 decimal digits) in 2009 and RSA-240 (795 bits or 240 decimal digits) in 2019. Thus RSA moduli are typically at least 2048 bits long (about 600 decimal digits) in practice.

The algorithms used in an RSA cryptosystem have polynomial time complexity. We have already seen that $\frac{1}{2}\lambda$-bit primes can be found in time complexity $O(\lambda^5)$, and RSA encryption and RSA decryption also clearly have polynomial time complexity. We can also make further improvements to the speed of the RSA cryptosystem. For example, the small public key exponent $e = 2^{16} + 1 = 65537$ (a prime) is in widespread use in many RSA cryptosystems in banking and elsewhere as exponentiation by e essentially takes 17 multiplications.

The Chinese Remainder Theorem can be used to speed up RSA `Decrypt` algorithm in the manner described in Figure 4. The motivation is that p and q are half the size of N and the exponents d_p and d_q is this CRT decryption are typically half the size of the corresponding RSA private key exponent d. If the original modular exponentiation modulo N takes time T, then each of the exponentiations at Step 2 take time $\frac{1}{8}T$, giving a total time of $\frac{1}{4}T$. As the CRT time complexity is negligible compared to modular exponentiation, we obtain an overall time complexity of about $\frac{1}{4}T$. This gives a 4-fold speed-up for decryption, which can potentially be further speeded up (perhaps by Karatsuba).

3.11 The Rabin Cryptosystem

The Rabin cryptosystem is essentially RSA with $e = 2$, and as we later discuss the security of Rabin is directly related to factoring (unlike RSA). The message space and ciphertext space for Rabin are $M_\lambda = C_\lambda = \mathbb{Z}_N^*$, and the Rabin algorithms are given below.

- $\texttt{KeyGen}(\lambda)$. Generate two random $\lambda/2$-bit primes p and q such that $p, q = 3 \bmod 4$ and set $N = pq$. The output is the public key $pk = N$ and the private key (p, q).
- $\texttt{Encrypt}(pk, m)$. Compute $c = m^2 \bmod N$ and output c and some associated added redundancy.
- $\texttt{Decrypt}(sk, c)$. Compute $m_p = c^{\frac{1}{4}(p+1)} \bmod p$ and $m_q = c^{\frac{1}{4}(q+1)} \bmod q$. Use the Chinese Remainer Theorem to find the four possibilities for m satisfying $m = \pm m_p \bmod p$ and $m = \pm m_q \bmod q$. Use the added redundancy to determine which of the four possibilities for m is the actual message and output this value.

The key generation requirement that the primes p and q are 3 modulo 4 is to allow us to find square roots modulo p and modulo q easily in the `Decrypt` algorithm. We can also construct a Rabin signature scheme with the algorithms below.

- $\texttt{Sign}(sk, m)$. Assume that $(\frac{m}{N}) = 1$, so either m or $-m$ is a square modulo N (follows from $p, q = 3 \bmod 4$). Compute the square root of m or $-m$ with the above method and output this square root as the signature s.
- $\texttt{Verify}(pk, m, s)$. Check whether m is $\pm s^2 \bmod N$ and output "valid" or "invalid" as appropriate.

The Rabin ciphertext c has four square roots, each of which could be the message without further information. The added redundancy is required to specify which of the four square roots is the actual message, and we give two redundancy schemes below.

- **Redundancy Scheme A**
  The additional information is the two bits of information given by the Jacobi symbol $(\frac{m}{N})$ and the bit $b = 0$ if $0 < m < \frac{1}{2}N$ and $b = 1$ if $\frac{1}{2}N < m < N$. This redundancy scheme works because $(\frac{-1}{N}) = 1$ and so m and $N - m$ have the same Jacobi symbol. The drawbacks of

this scheme are that the ciphertext contains some information about the message and that it is necessary to compute the Jacobi symbol to encrypt.

- **Redundancy Scheme B**
  We encrypt an integer m corresponding to a bitstring where the ℓ least significant bits (for $\ell > 2$) are a fixed pattern or the replication of the least ℓ bits of the message. If ℓ is big enough then it is very unlikely that two different choices of square root would have the right pattern in the ℓ bits. If none of the four square roots have the correct bit pattern then output the invalid ciphertext symbol $\perp$. The drawback of this method is that the potential message space is reduced.

The modulus N in a practical Rabin cryptosystem has at least 2048 bits for the reasons given for an RSA modulus N, as N has to be large enough to be resistant to factoring in both cases. The Rabin encryption system is typically used for key transport for similar reasons as RSA, so a Rabin message m is usually a symmetric key (processed through a padding scheme), and in a Rabin signature scheme it is usually a hash of the message that is actually signed. Rabin encryption (essentially modular squaring) is extremely fast. Rabin decryption uses the Chinese Remainder Theorem and is roughly the same speed as RSA decryption using the CRT. We also note that Williams encryption is another padding scheme which gives unique decryption, though (like Scheme A) it requires the computation of a Jacobi symbol.

3.12 Prime Generation and Testing

In the RSA and Rabin cryptosystems, as well as more generally in Public Key Cryptography, we need to be able to generate a "random" prime number of an appropriate size. We can potentially generate a random k-bit prime number by generating a random k-bit number, and then testing this generated number for primality by using the Miller-Rabin process, which has time complexity in terms of the length k of $O((\log 2^k)^4) = O(k^4)$. We can determine the probability that a k-bit number is prime by using the Prime Number Theorem, which essentially states that the prime number counting function $\pi(x)$ giving the number of primes less than or equal to x satisfies $\pi(x) \approx \dfrac{x}{\log_e(x)}$. Thus the probability that a k-bit number is prime is about $\dfrac{\pi(2^k)}{2^k} \approx \dfrac{1}{\log_e(2^k)} = O(k^{-1})$. We can therefore generate a random k-bit

prime number by randomly generating k-bit numbers and then testing for primality, giving an algorithm for generating a k-bit prime with an expected time complexity of $O(k^5)$.

A prime p for use in Public Key Cryptography is usually in practice issued with a a *certificate of primality*, which is a mathematical proof that establishes the primality of p and can be checked in polynomial time.

A simple algorithm to test whether a positive integer n is a prime number or a composite number is *trial division* specified in Figure 5, which actually finds the factors of n. However, trial division is an exponential algorithm, so it is not suited to be a general test of whether a number is prime or composite. In particular, trial division is not useful for factoring integers used in many aspects of cryptology, though a reduced form of trial division to remove all 'small' prime factors of n can be useful before trying more elaborate methods.

The Miller–Rabin test is the method by which an assessment as to whether an integer n is a composite number is made in practice. The idea of the Miller-Rabin test is to write $n - 1 = 2^b m$ where m is odd, choose a positive integer a coprime to n ($\gcd(a, n) = 1$), and then to consider the sequence $a_0, a_1, \ldots, a_b$ given by

$$a_0 = a^m \bmod n, \quad a_1 = a_0^2 = a^{2m} \bmod n, \quad \ldots, \quad a_b = a_{b-1}^2 = a^{n-1} \bmod n.$$

If n is prime then this sequence must take one of the following three forms: $(*, *, \ldots, *, -1, 1, \ldots, 1)$ or $(-1, 1, \ldots, 1)$ or $(1, \ldots, 1)$. Any deviation from these three possible forms means that the number n is composite. In this case, we can output *true* for the test of whether n is composite and halt; otherwise we output *false* (for the base a).

An analysis of this process shows that if n is odd and composite (and $n > 9$), then at least $\frac{3}{4}$ of the possible choices for the base a lead to the test to output *true*. By contrast, if n is prime, the test never outputs *true*. This process shows that the problem COMPOSITE (below) can be solved efficiently if we allow ourselves a little uncertainty. By repeating the this basic Miller-Rabin test for k randomly chosen values of a, we can achieve a high degree of knowledge about whether n is a composite integer or a prime integer. If each of the k tests returns *false*, then n is prime with probability at least $1 - (\frac{1}{4})^k$. By contrast, if any of the k tests returns *true*, then we know with certainty that n is a composite integer. We usually make $k \approx \log n$ tests, in which case a Miller-Rabin process for testing whether an integer n is composite or prime is a probabilistic algorithm with time complexity $O((\log n)^4)$.

Input: n
Output: List of prime factors of n
1. $m = \lfloor \sqrt{n} \rfloor$
2. $a = 2$
3. while $(a \leq m)$ do
4. while $(n = 0 \mod a)$ do
5. output a
6. $n = n/a$
7. $m = \lfloor \sqrt{n} \rfloor$
8. $a = a + 1$

Figure 5: **Trial Division Algorithm** to compute prime factors of n.

There are deterministic algorithms for testing compositeness and primality. The Agrawal, Kayal and Saxena (AKS) algorithm (2002) is a deterministic algorithm guaranteed to give the right answer that has time complexity $O((\log n)^{6+\epsilon})$, so is a polynomial time algorithm. However, the Miller-Rabin process is faster, so is generally used. Such a test are often termed a *primality test*, and one is implemented in Mathematica as `PrimeQ[p]`

3.13 Factoring Algorithms

A factoring algorithm takes a composite integer n as an input and returns a non-trivial factor d of n as output. Three examples of factoring algorithms that work well for certain types of input are given below.

- *Trial Division* is an exponential algorithm, so is only useful for very small integers.
- The *Pollard* $p-1$ algorithm factors n if n has a prime factor p where $p-1$ has only small factors.
- The *Pollard rho* algorithm factors n when n has a fairly small prime factor p. Pollard rho has expected time complexity $O(\sqrt{p}(\log n)^2)$.

In Public Key Cryptography, we are in general interested in factoring algorithms that work well for *RSA-type* numbers N, that is to say factoring algorithms that can find a factor (p or q) of N when $N = pq$ is the product of two primes p and q of comparable size. The state-of-the-art factoring

algorithms for factoring RSA-type numbers have a time complexity function that is specified in terms of the subexponential function below.

Definition. Let a and c be real constants such that $0 \leq a \leq 1$ and $c > 0$. The *subexponential function* $L_N(a, c)$ is defined to be

$$L_N(a, c) = \exp(c\ (\log_e N)^a\ (\log_e(\log_e(N)))^{1-a}).$$

We first note that taking $a = 0$ in this subexponential function actually gives a time complexity function $L_N(0, c) = \log_e(N)^c$ that is a polynomial function of the input length $\log N$, and that taking $a = 1$ gives a time complexity function $L_N(1, c) = N^c = \exp(\log_e N^c)$ that is an exponential function of the input length $\log N$. More generally, an algorithm with a time complexity function $L_N(a, c)$ for a value of a lying strictly between 0 and 1 is (loosely speaking) an algorithm that runs slower than polynomial time algorithms and faster than exponential time algorithms.

There has been much research into factoring algorithms for RSA-type numbers since the initial development of Public Key Cryptography in the 1970s, and we give two important factoring algorithms below.

- The *Quadratic Sieve* (QS) is typical of a number of algorithms were developed through the 1980s that all had the same time complexity function, essentially given by $L_N(\frac{1}{2}, 1)$ for large N.
- The *General Number Field Sieve* (GNFS) is typical of a number of algorithms was developed in the early 1990s and was a major breakthrough in factoring algorithms with a time complexity function essentially given by $L_N(\frac{1}{3}, 1.932)$ for large N.

The General Number Field Sieve has a much smaller time complexity than the Quadratic Sieve. This meant that requirements in Standards and elsewhere for a 512-bit (typically) RSA modulus N in the mid-1990s were rapidly changed following the development of GNFS to a 1024-bit (and now higher) modulus N for RSA. The current record for factoring an RSA modulus is an 829-bit (250 decimal digits) modulus in February 2020.

4 Further Factoring-based Cryptosystems

This Section begins with a discussion of the computational Problems that naturally arise when considering factoring-based cryptosystems and the relationships between these Problems, leading to an analysis of the hardness of the RSA and Rabin cryptosystems. We then consider some specific attacks on RSA and some of the padding schemes used to increase the security of RSA in practical situations.

4.1 Problems in Factoring-based Cryptography

There are many different computational Problems which naturally arise from the study of the RSA and Rabin cryptosystems, and we list some of these Problems below. However, this list does not give a formal specification of these Problems, for which a full description of the input and output would be required. The formal complexity of these algorithms would then be discussed in terms of the input size (and output size).

- FACTOR.
  Given a positive integer $N = pq$ the product of two unknown primes, find the primes p and q.
- RSA-PRIVATE-KEY.
  Given an RSA public key (N, e), compute the private key d.
- COMPUTE-PHI.
  Given an integer N, compute $\varphi(N)$.
- RSA or e-th ROOT
  Given an RSA public key (N, e) and an RSA ciphertext c, compute the Textbook RSA decryption of c.
- SQRT-MOD-N.
  Given an integer N and a value y modulo N, compute a solution to $x^2 = y \bmod N$ or output 'no solution' if y is not a square modulo N.
- SOLVE-QUADRATIC-MOD-N.
  Given four integers a, b, c, N, find a solution to the quadratic equation $ax^2 + bx + c = 0 \bmod N$ or output "no solution" if no solution exists.

4.2 Relationships between Problems

We discussed Turing reductions between Problems in Section 2.7. Loosely speaking, a *Turing reduction* from a Problem A to a Problem B is a demonstration that an Oracle $\mathcal{O}_B$ for solving Problem B can be transformed (in polynomial time) into an algorithm to solve Problem A, which we denote by $A \leq_T B$. Thus a Turing reduction from Problem A to Problem B means that solving Problem A has been "reduced" to solving Problem B, and so we can infer that Problem B is "at least as hard as" Problem A or that Problem A is "no harder than" Problem B. If there is a Turing reduction from Problem A to Problem B and another Turing reduction from B to A, then the Problems A and B are said to be *polynomial time equivalent*.

It is important to understand the relations between the computational Problems related to RSA and Rabin that were listed above, and we discuss some of these relationships below.

FACTOR and COMPUTE-PHI

We have already seen that knowledge of $\varphi(N) = (p-1)(q-1)$ for an RSA modulus $N = pq$ allowed us to factorise N into primes p and q in polynomial time by setting up a quadratic equation (in integers), so

$$\text{FACTOR} \ \leq_T \ \text{COMPUTE-PHI}.$$

Conversely, if we know the factorisation of $N = pq$ into primes p and q, then we can obviously compute $\varphi(N) = (p-1)(q-1)$ in polynomial time, so

$$\text{COMPUTE-PHI} \ \leq_T \ \text{FACTOR}.$$

It follows that FACTOR and COMPUTE-PHI are polynomial time equivalent Problems for "RSA moduli" inputs.

More generally, versions of the computational Problems FACTOR and COMPUTE-PHI can also be defined for all integers, not just RSA moduli. A more general method, similar to the one used below to show that FACTOR is equivalent to RSA-PRIVATE-KEY, shows that the versions of FACTOR and COMPUTE-PHI defined for all integers are also polynomial times equivalent Problems.

SQRT-MOD-N and SOLVE-QUADRATIC-MOD-N

Suppose we have an Oracle $\mathcal{O}$ for the SOLVE-QUADRATIC-MOD-N Problem so $\mathcal{O}$ can solve quadratic equations modulo N, where for simplicity, we

assume that N is odd and square-free, and a SQRT-MOD-N Problem with input (N, y). We can run $\mathcal{O}$ on the quadratic equation $x^2 - y \bmod N$ to find a square root of y, so

$$\text{SQRT-MOD-N} \leq_T \text{SOLVE-QUADRATIC-MOD-N}.$$

Conversely, suppose we have an Oracle $\mathcal{O}$ for the SQRT-MOD-N problem. A solution to the quadratic equation $ax^2 + bx + c = 0 \bmod N$ in the case $\gcd(a, N) = 1$ (the case $\gcd(a, N) \neq 1$ can also be handled) is then obtained by using the Oracle $\mathcal{O}$ to compute

$$x = (-b + \sqrt{b^2 - 4ac(\text{mod}N)})(2a)^{-1} \bmod N,$$

as a polynomial time computation is required to obtain x after extracting the modular square root, so

$$\text{SOLVE-QUADRATIC-MOD-N} \leq_T \text{SQRT-MOD-N}.$$

Thus SQRT-MOD-N and SOLVE-QUADRATIC-MOD-N are polynomial time equivalent Problems.

RSA and FACTOR

Suppose that we have an Oracle $\mathcal{O}$ for the FACTOR Problem. We can simply run $\mathcal{O}$ on an RSA modulus $N = pq$ to obtain the two prime factors p and q of N. We can then find the RSA private key d by solving the usual key set-up equation $ed = 1 \bmod \varphi(N) = (p-1)(q-1)$ in polynomial time. Having obtained the decryption private key d, we then can decrypt a Textbook RSA ciphertext in polynomial time, so

$$\text{RSA} \leq_T \text{FACTOR}.$$

This Turing reduction tells us that FACTOR is at least as hard as RSA, or equivalently that decrypting RSA is no harder than factoring. The natural question is whether a reverse Turing reduction from FACTOR to RSA exists. However, no such reduction from FACTOR to RSA is known, and there is some evidence to suggest that the RSA Problem cannot be as hard as factoring. This means that whilst the design of RSA was motivated by the difficulty of the factoring problem, there is no good theoretical security assurance for the RSA cryptosystem relating it directly to the difficulty of factoring.

FACTOR and RSA-PRIVATE-KEY

Suppose that we have an Oracle $\mathcal{O}$ for the FACTOR Problem and an RSA public key (N, e) input to the RSA-PRIVATE-KEY Problem. We can run $\mathcal{O}$ on N to obtain its factors p and q, and then find the RSA private key d in the usual way in polynomial time (as discussed above), so

$$\text{RSA-PRIVATE-KEY} \leq_T \text{FACTOR}.$$

Conversely, suppose that we have an Oracle $\mathcal{O}$ for the RSA-PRIVATE-KEY Problem and an input N for the FACTOR Problem. We can choose a random large prime e and run $\mathcal{O}$ on (N, e) (a simulated RSA public key) to obtain d, the "private key" corresponding to the "public key" e. We can therefore compute $M = ed - 1$ which is a multiple of $\lambda(N)$, and we can use a Miller-Rabin approach with M to try to factor N. We write $M = 2^r m$ for odd m, and for a random a with $\gcd(a, N) = 1$ we compute the sequence $a_0 = a^m \bmod N$, $a_1 = a_0^2 \bmod N, \ldots, a_{r-1} = a_{r-2}^2 \bmod N, a_r = a_{r-1}^2 = a^M = 1 \bmod N$. This sequence $a_0, a_1, \ldots, a_{r-1}, a_r$ is likely to contain a non-trivial square root of 1, that is to say that there exists a_i with $a_i \neq 1$ but $a_{i+1} = a_i^2 = 1 \bmod N$. Furthermore, $a_i \neq -1 \bmod N$ with probability at least $\frac{1}{2}$ with $a_i^2 - 1 = (a_i + 1)(a_i - 1) = 0 \bmod N$, in which case $\gcd(a_i + 1, N)$ gives a factor of N. Repeating this process a polynomial number of times give an algorithm for factoring N with overwhelming probability, so

$$\text{FACTOR} \leq_T \text{RSA-PRIVATE-KEY}.$$

Thus FACTOR and RSA-PRIVATE-KEY are polynomial time equivalent Problems.

A subtlety is that this second Turing reduction from FACTOR to RSA-PRIVATE-KEY was only a probabilistic algorithm, but Turing reduction formally requires the algorithm to be deterministic. Thus we are being a bit vague about whether our Oracles are perfect, that is they always work and give a correct answer, or whether they only work with a very high probability. The formal way to analyse reductions between Problems is to consider Oracles which work for a non-negligible proportion of Problem instances, but this formal approach adds an extra level of complexity. However, a deterministic reduction FACTOR $\leq_T$ RSA-PRIVATE-KEY for these Problems was given by Coron and May in 2004.

SQRT-MOD-N and FACTOR

Suppose that we have an Oracle $\mathcal{O}$ for the FACTOR Problem and inputs N and y, a square modulo N, to the SQRT-MOD-N Problem. We can run

$\mathcal{O}$ on N to find its prime factors. We can then compute the square root of y modulo each of these prime factors in polynomial time, and then we can combine these solutions using the Chinese Remainder Theorem to find a square root of y modulo N is polynomial time, so

$$\text{SQRT-MOD-N} \leq_T \text{FACTOR}.$$

Conversely, suppose we have an Oracle $\mathcal{O}$ for the SQRT-MOD-N Problem and an input N for the FACTOR Problem. We choose a random $x \in \mathbb{Z}_N^*$ and let $y = x^2 \bmod N$. We then run $\mathcal{O}$ on N and y to find a square root x' of y modulo N, so $y = x^2 = x'^2 \bmod N$. However there are four square roots of y modulo N, so $x' \neq \pm x \bmod N$ with probability $\frac{1}{2}$. In this case,

$$x^2 - x'^2 = (x + x')(x - x') = 0 \bmod N$$

and so $\gcd(N, x \pm x')$ gives a prime factor of N. Repeating this process a polynomial number of times allows us to factor N in polynomial time with overwhelming probability, so

$$\text{FACTOR} \leq_T \text{SQRT-MOD-N}.$$

Thus SQRT-MOD-N and FACTOR are polynomial time equivalent Problems. The equivalence of these Problems provides one of the motivations for the design of the Rabin cryptosystem (see below).

4.3 The Hardness of RSA and Rabin Cryptosystems

An understanding of relations between Problems means that we can make inferences about the hardness of breaking the RSA and Rabin cryptosystems in the case of passive attacks and without padding schemes. An algorithm which breaks the one-way encryption property of RSA under passive (or selective signature forgery under passive attacks) attacks is easily turned into an RSA Oracle. Thus we can infer that the RSA cryptosystem is no harder to break than factoring the RSA modulus. A major open Problem in cryptography is to show that breaking RSA with a passive attack is actually hard, but indirect evidence suggests that breaking RSA is not as hard as factoring.

In the Rabin cryptosystem, decryption is performed by first calculating square roots modulo N, which we have shown is a polynomial time equivalent Problem to factoring N, so there is a direct connection between the underlying Rabin cryptosystem and factoring the modulus N, a property that is not possessed by the RSA cryptosystem. However, in a Rabin decryption

we have to consider four square roots modulo N, so extra information must be used to decrypt correctly. We have addressed this issue by considering Redundancy Scheme A in which two extra bits are sent and Redundancy Scheme B which essentially restricts the message space to messages of a certain form. However, we can give a precise security reduction by addressing these two redundancy schemes in the two theorems below, and we note that the security result for Scheme A is stronger in some sense than the security result for Scheme B.

Theorem (Rabin A). Breaking the one-way encryption (OWE) security property of the Rabin cryptosystem with Redundancy Scheme A under passive attacks is polynomial time equivalent to factoring products $N = pq$ of primes of the form $p, q = 3 \bmod 4$. □

Proof. We have already established that factoring N allows us to break the Rabin cryptosystem in polynomial time. For the reverse reduction, we suppose that we have an Oracle $\mathcal{O}_A$ a Rabin public key N and a ciphertext c encrypted under Redundancy Scheme A and returns either the corresponding message m or an invalid ciphertext symbol $\perp$. We choose a random message $m \in \mathbb{Z}_N^*$ and set $c = m^2 \bmod N$, $b_1 = -(\frac{m}{N})$ and choose a random bit b_2. We then run $\mathcal{O}_A$ on the Rabin input (c, b_1, b_2) to obtain a message m' satisfying $(m')^2 = m^2 \bmod N$ or equivalently $(m' - m)(m' + m) = 0 \bmod N$. However, $m' \neq \pm m \bmod N$ since they have different Jacobi symbols, so calculating $\gcd(m' \pm m, N)$ gives a factor of N. □

Theorem (Rabin B). Breaking the one-way encryption (OWE) security property of the Rabin cryptosystem with Redundancy Scheme B where $l = O(\log \log N)$ under passive attacks is polynomial time equivalent to factoring products $N = pq$ of primes of the form $p, q = 3 \bmod 4$. □

Proof. As noted before, we have already established that factoring N allows us to break the Rabin cryptosystem in polynomial time. For the reverse reduction, we suppose that we have an Oracle $\mathcal{O}_B$, a Rabin public key N and a ciphertext c encrypted under Redundancy Scheme B. The Oracle $\mathcal{O}_B$ with input N and c returns either a value whose square modulo N is c or the invalid ciphertext symbol $\perp$. We choose a random $m \in \mathbb{Z}_N^*$ such that neither m nor $-m$ satisfy the padding scheme, that is to say the ℓ least significant bits are not correct. We then set $c = m^2 \bmod N$ and run the $\mathcal{O}_B$ on c to obtain either m' or $\perp$. If one of the four square roots of c modulo N has the correct ℓ least significant bits, then we have obtained a value $m' \neq \pm m \bmod N$ such that $m'^2 = m^2 \bmod N$ and so $\gcd(m' \pm m, N)$

gives a factor of N (as above). The probability that one of the four square roots of c modulo N does have the correct ℓ least significant bits is roughly $2 \times 2^{-\ell} = 2^{-(\ell-1)}$. Thus we would expect to be able to factor N if we can repeat this process about 2^{ℓ} times. If we set $\ell = O(\log(\log(N)))$ we would therefore a polynomial time reduction. □

The hardness guarantee of the Rabin cryptosystem is often stronger than for the RSA cryptosystem, at least under passive attacks. Thus the Rabin cryptosystem is very attractive as it has faster public key operations and a stronger security guarantee than RSA.

4.4 Attacks on the RSA and Rabin Cryptosystems

We now give a number of attacks on the RSA and Rabin cryptosystems and their relation to the various security properties and attack goals specified in Section 3.1.

Brute Force Attacks

Encryption is deterministic in Textbook RSA, so we can always try encrypting all possible messages in situations with a small effective message space (for example a short message). Such a brute force attack can be foiled by adding randomisation, as long as enough bits of randomisation are added, say 80 or 128 bits of randomisation.

Person in the Middle

This is a standard attack for any public cryptosystem. Suppose Bob intends to send sensitive information to Alice by encrypting it. To achieve this Bob must obtain Alice's public key (for example from her webpage). Suppose that Eve (the eavesdropper) can ensure that Bob instead receives a public key of her choosing for which Eve knows the private key, instead of Alice's public key. If Bob sends a message encrypted under what he believes is Alice's public key but is in fact encrypted under Eve's replacement public key, then Eve can decrypt the ciphertext and therefore obtain the message. Furthermore, Eve can then encrypt the message using Alice's real public key and send this encrypted version of the message on to Alice. Thus Eve can monitor the communication between Alice and Bob without Alice or Bob being aware that this is happening.

The Hastad Attack

The Hastad attack is essentially an attack in a broadcast scenario of encrypted versions of the same message when the various public encryption exponents are small. If the same message is encrypted with several different public keys, then the original message can be recovered using the Chinese Remainder Theorem and taking e-th roots of integers. Suppose one sender sends the same message M in encrypted form to three people P_1, P_2 and P_3 using $e = 3$ but different moduli N_1, N_2 and N_3. If the Adversary intercepts C_1 , C_2, and C_3, where $C_i = M^3 \bmod N_i$, then the Adversary can use the Chinese Remainder Theorem to compute $C \in \mathbb{Z}^*_{N_1N_2N_3}$ such that $C_i = C \bmod N_i$, so $C = M^3 \bmod N_1N_2N_3$. However, $M < N_1, N_2, N_3$, so $M^3 < N_1N_2N_3$ and we obtain $C = M^3$ over the integers, from which we can obtain M. This attack is also very effective against Rabin ($e = 2$).

The Hastad attack is foiled if e is set a little larger, which is one reason why modern systems use $e = 65537 = 2^{16} + 1$. A further way to prevent the attack is to use randomised encryption, so that the "same" message is not actually sent several times.

Algebraic Attacks

A number of adaptive attacks on Textbook RSA follow immediately from the fact that Textbook RSA encryption is *multiplicative* in the sense that

$$m_1^e m_2^e = (m_1 m_2)^e \bmod N.$$

Suppose we are given an Oracle $\mathcal{O}$ for Textbook RSA decryption and a challenge ciphertext c. We can generate a random $r \in \mathbb{Z}^*_N$ and compute $c' = cr^e \bmod N$. We then run $\mathcal{O}$ on c' to obtain m'. The actual message corresponding to the challenge ciphertext c is $m = m'r^{-1} \bmod N$ because $m^e = m'^e r^{-e} = cr^e r^{-e} = c \bmod N$. We can also forge signatures in a similar way with a signing Oracle for a Textbook RSA signature scheme. Such attacks are CCA2 attacks and require one query to an Oracle for each decryption or forgery. There are also variants of these attacks for the Rabin cryptosystem, where essentially $e = 2$.

The Desmedt-Odlyzko Attack

The Desmedt-Odlyzko Attack is a form of algebraic attack. In the signature context with a Signing Oracle $\mathcal{O}$, we run $\mathcal{O}$ on the first r prime numbers $p_1, \ldots, p_r$ to get the corresponding "signatures' $s_1, \ldots, s_r$. If a message $m =$

$\prod_{i=1}^{r} p_i^{f_i}$ is a product of powers of these first r primes, then the corresponding signature is $s = \prod_{i=1}^{r} s_i^{f_i}$. This attack is generally not feasible if messages are random elements between 1 and N as the probability of "smoothness" is too small, but it might be effective if messages in the system are small. An analogous attack can also be constructed in principle in an encryption context, but this requires ciphertexts to be "factored", so is in general not faster than factoring the modulus.

Related Message Attacks

This an attack on Textbook RSA (due to Franklin and Reiter) with a small exponent e, for example $e = 3$ or Rabin ($e = 2$). Suppose we obtain ciphertexts c_1 and c_2 for the related messages m and $m+a$ for some known integer a, and we wish to compute the underlying message m (and hence $m+a$). In this case, the two polynomials $f_1(x) = x^e - c_1$ and $f_2(x) = (x+a)^e - c_2$ satisfy $f_1(m) = f_2(m) = 0 \bmod N$ and so have a common factor $g(x) = x - m$ modulo N. Thus we can run Euclid's algorithm for polynomials on $f_1(x)$ and $f_2(x)$ modulo N and obtain this linear factor $g(x) = (x - m)$ with high probability. The complexity of the method is polynomial in the degree e of the polynomials (so exponential in $\log e$). This attack is therefore feasible only when e is rather small, for example $e < 2^{40}$.

Example. Consider a Textbook RSA encryption with a small public key exponent $e = 3$ and modulus $N = 2157212598407$. Suppose we have ciphertexts

$$c_1 = 1429779991932 \text{ and } c_2 = 655688908482$$

corresponding to the encryption of m and $m + 2^{10}$ respectively. To find m we construct the polynomials $f_1(x) = x^3 - c_1$ and $f_2(x) = (x + 2^{10})^3 - c_2$ and compute their gcd

$$g(x) = \gcd(x^3 - c_1, (x + 2^{10})^3 - c_2) = x - 1234567890 \quad \text{in } \mathbb{Z}_N[x].$$

We therefore deduce that $m = 1234567890$ and $m + 2^{10} = 1234568914$.

Small Private Exponent d Attack

It is possible obtain a secure RSA cryptosystem when using a small public exponent e in comparison with the modulus size N, for example the banking standard $e = 2^{16} + 1 = 65537$ and a modulus N of well over 1000 bits. It is there natural to ask whether it is possible to speed-up RSA decryption (or signature) by choosing the private exponent d to be a small integer. In

i	q	r_i	s_i	t_i
-1	–	86063528783122081	1	0
0	–	14772019882186053	0	1
1	5	12203429372191816	1	-5
2	1	2568590509994237	-1	6
3	4	1929067332214868	5	-29
4	1	639523177779369	-6	35
5	3	10497798876761	23	-134
6	60	9655245173709	-138	8075
7	1	842553703052	1409	-8209

Figure 6: Wiener attack for small private exponent RSA with modulus $N = 86063528783122081$ and public exponent $e = 14772019882186053$.

this setting, key generation is performed by first choosing such a d and then setting $e = d^{-1} \bmod \varphi(N)$. Thus choosing the decryption exponent d to be a small integer comes at the cost of RSA encryption speed as e is generally now an arbitrary value modulo $\varphi(N)$. If d is very small, then there is an attack by just trying each small value for d in turn and testing it by checking whether $(m^e)^d = m \bmod n$ for some m, so we should take $d > 2^{80}$ or perhaps $d > 2^{128}$.

We now present a more sophisticated attack due to Wiener. The defining equation for the public exponent is $ed = 1 \bmod \varphi(N)$, or equivalently

$$ed = 1 + k\varphi(N)$$

for some positive integer k. However, $e < \varphi(N)$ and so $k < d$. Furthermore, $\varphi(N) \approx N - 2\sqrt{N}$, so $ed \approx 1 + k(N - 2\sqrt{N})$ or equivalently

$$-ed + kN \approx (-1 + k2\sqrt{N}).$$

If d is smaller than $\sqrt{N}/2$, then the right hand side $-1+k2\sqrt{N}$ is smaller than N. In this case the above expression is reminiscent of an expression of the form

$$ad + bN = \gcd(d, N)$$

obtained by using the Extended Euclidean algorithm. The Wiener attack is simply to run the extended Euclidean algorithm on with input integers e and N to attempt to find d, k and the right hand side. A candidate $\pm d$ for the private key exponent can be tested by checking whether $(m^e)^d = m$

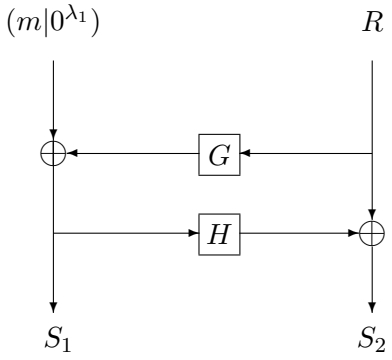

Figure 7: RSA-OAEP processing of n-bit message m in an $(n+\lambda_1)$-bit block and a λ_0-bit blockstrings to give λ-bit string $S = (S_1|S_2)$.

mod N. If a suitable d can found, then the system is easily broken. The following result indicates that the process is efficient for small d. We also note this approach has been extended by Boneh and Venkatesan using lattice methods to $d < N^{0.292}$.

Theorem. If $0 < d < 2^{-\frac{1}{2}}N^{\frac{1}{4}}$ then d can be found by the above Wiener method in polynomial time.

Proof. As mentioned above, we have $0 < e < \varphi(N) = N - u$ with $0 < u < 2\sqrt{N}$ and $0 < k < d$. Hence the equation $ed = 1 + k\varphi(N)$ becomes $-ed + kN = uk - 1$. The Euclidean algorithm finds all triples (d, k, r) satisfying $-ed + kN = r$ with $|dr| < N$ and $|kr| < e$. Hence, if $|duk| < N$ then the required solution will be found. If $0 < d < N^{1/4}/\sqrt{2}$ then

$$|duk| < d^2u < \frac{N^{1/2}}{2}2\sqrt{N} = N.$$

Example. Consider a Textbook RSA encryption with a public key exponent $e = 14772019882186053$ corresponding to a small private key exponent d and a modulus $N = 86063528783122081$. We run the Extended Euclidean algorithm with input N and e, so $r_{-1} = N$ and $r_0 = e$, and s_i, t_i are integers such that $r_i = s_iN + t_ie$, to obtain the Table of Figure 4.4. Thus we obtain possible values ± 8209 for d in only seven steps, and on checking we find $d = 8209$.

- Choose a random λ_0-bit string R.
- Set $S_1 = (m \| 0^{\lambda_1}) \oplus G(R)$, where $\|$ denotes concatenation and $\oplus$ XOR.
- Set $S_2 = R \oplus H(S_1)$.
- Form the bitstring $S = S_1 \| S_2$.
- The bitstring S is interpreted as an element of $\mathbb{Z}_N^*$ and raised to the power e modulo N.

Figure 8: RSA-OAEP Encryption of a n-bit message m

4.5 Side Channel and Fault Injection Attacks

A completely different class of attacks on cryptosystems can be applied when the attacker has some sort of access to the machine running it. For example, when RSA is implemented on a smart card, the attacker might get her hands on that smart card. The idea is that if such a device is captured then information about the private keys on it might be obtained by observing the time taken to perform various tasks or by monitoring the power usage during a protocol run. These sorts of attacks are generally known as *side channel attacks* and they are not considered in this course, though they are potentially important in any practical cryptosystem deployment).

The idea of a *fault injection* attack is to run a device such as a smartcard in an environment which may cause faults to occur during the computation, for example at temperature extremes or by bombarding the device with radiation at specific stages in the computation. For example, suppose the Chinese Remainder Theorem (CRT) is often used for RSA private key operations. For ciphertext c decryption takes place by first computing $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$ and then setting $m = CRT(m_p, m_q)$. If there is a fault in the second computation then we have $m^e - c = 0 \bmod p$ but $m^e - c \neq 0 \bmod q$, so $\gcd(m^e - c, N)$ will reveal a multiple of p.

- The computation $c^d \mod N$ is interpreted as a bitstring $S = (S_1|S_2)$ low λ_0-bits S_2 and the high $(n + \lambda_1)$-bits S_1.
- Compute $R = S_2 \oplus H(S_1)$.
- Compute $S_3 = S_1 \oplus G(R)$.
- Check whether the low λ_1 bits of S_3 are all zero, if not output the decryption failure symbol $\perp$ and halt.
- Obtain m as the top n bits of S_3.

Figure 9: RSA-OAEP Decryption of a ciphertext c

4.6 Padding Schemes for RSA

In response to attacks such as those mentioned above special randomised padding schemes have been introduced, with the following three main goals.

- Introduction of randomness into the message and the expansion of short messages to full size.
- Destruction of algebraic relationships among messages.
- To nsure that random elements of $\mathbb{Z}_N^*$ do not correspond to valid ciphertexts. This means that access to a decryption Oracle in CCA2 attacks is not useful.

RSA-OAEP (Optimal Asymmetric Encryption Padding) is a padding proposal proposal for RSA encryption due to Bellare and Rogaway. Suppose we are using λ-bit RSA moduli (so for example $\lambda = 2048$). We choose λ_0 and λ_1 to be positive integers so no Adversary can perform 2^{λ_0} or 2^{λ_1} operations, so for example $\lambda_0, \lambda_1 = 80$ or 128. We then set $n = \lambda - \lambda_0 - \lambda_1$ and consider messages to be n-bit strings. We then define the two collision-resistant hash functions G and H such that:

- G is a hash function mapping λ_0 bit strings to $n + \lambda_1$ bit strings,
- H is a hash function from $n + \lambda_1$ bit strings to λ_0 bit strings.

The pre-processing of a message m in RSA-OAEP before modular exponentiation is illustrated in Figure 7 and formally specified in Figures 8 and 9.

RSA-OAEP (and schemes like it) are motivated by the following ideas Firstly, the padding scheme is randomised and has full length. Secondly, since S_1 and S_2 are both outputs of hash functions, the bitstring S looks random and algebraic relationships among the message components m are not preserved. Thirdly, if you are given a decryption Oracle and you send it a random element of $\mathbb{Z}_N^*$ then the decryption process will fail with high probability, so the decryption Oracle essentially becomes useless. A rigorous proof of the security of RSA-OAEP can be given in the random Oracle model. This proof shows how to transform a CCA2 Indistinguishibility (IND) Adversary into an algorithm which solves the RSA Problem.

RSA-OAEP (and similar systems) are usually part of a *hybrid* system. Such a hybrid system uses public key cryptography such as RSA-OAEP to transmit a symmetric key and then uses symmetric cryptography for the bulk encryption of data. Suitable padding schemes and the theoretical security analysis of hybrid encryption schemes are an important part of the cryptography literature. There are also padding schemes for public key signatures, such as full domain hash and RSA-PSS (Provably Secure Signature).

5 An Introduction to Discrete Log Cryptosystems

In this Section, we consider the Discrete Logarithm Problem that underlies the Diffie-Hellman, El Gamal and Digital Signature Standard (DSS) cryptosystems, and some of the algorithms to address the Discrete Log Problem.

Many of the more powerful methods used to address the discrete logarithm problem are directly related to the more powerful integer factorisation methods. This equivalence motivates the highly general statement that factoring an RSA-type integer N and taking discrete logarithms in $\mathbb{F}_p^*$ usually have broadly equivalent complexity when $N \approx p$. In particular, there is a version of the General Number Field Sieve (GNFS) for solving the discrete logarithm problem modulo p with complexity $O(L_p(1/3, 1.923))$, where L is the subexponential function.

5.1 The Discrete logarithm Problem

We first recap the basic notation that we use when discussing the discrete logarithm problem. For a prime p, we denote the *finite field* of p elements by $\mathbb{F}_p$ (equivalent to $\mathbb{Z}_p$), and we let $\mathbb{F}_p^* = \mathbb{F}_p \backslash \{0\}$ denote the *multiplicative group* of non-zero elements, which is a cyclic group. We note that all statements of this Section about the filed $\mathbb{F}_p$ can also be generalised to finite fields of the form $\mathbb{F}_{p^m}$, though we do not discuss the details in this case. A generator for $\mathbb{F}_p^*$ is called a *primitive root*, and there are $\varphi(p-1)$ primitive roots in $\mathbb{F}_p^*$. For $g \in \mathbb{F}_p^*$, the *subgroup generated by* g is the set $\langle g \rangle = \{g^a \bmod p : a \in \mathbb{Z}\}$, and the order of the element g is the same as the number of elements in the group $\langle g \rangle$. Thus if $g \in \mathbb{F}_p^*$ is a primitive root, then $\langle g \rangle = \mathbb{F}_p^*$ and so the order of g is $p-1$.

Figure 10 gives an algorithm to test if g is a primitive root in $\mathbb{F}_p^*$ if the factorisation of $p-1$ is known. However, there is no known algorithm to test whether g is a primitive root modulo p without knowledge of the factorisation of $p-1$. In such circumstances, it seems unlikely that we could prove that FACTOR is Turing reducible to TEST-PRIMITIVE-ROOT.

The obvious algorithm to generate a primitive root of $\mathbb{F}_p^*$ when the factorisation of $p-1$ is known can be obtained by repeatedly generating a random element $g \in \mathbb{F}_p^*$ and testing whether g is a primitive root (using the Figure 10 algorithm) until a primitive root is found. The probability that such a random g is a primitive root of $\mathbb{F}_p^*$ is $\dfrac{\varphi(p-1)}{(p-1)} > \dfrac{1}{6\log_e(\log_e(p-1))}$ for $p > 16$. This probability is typically quite large, and is for example usually larger than 0.2. Thus the complexity of finding a primitive root in

Input: g, p
Output: 'Yes' if g is a primitive root modulo p
1. Compute the set Q of all primes $q \mid (p-1)$
2. for each $q \in Q$ do
3. if $g^{(p-1)/q} = 1 \bmod p$ then
4. return 'No'
5. end if
6. end for
7. return 'Yes'

Figure 10: **Primitive Root Algorithm** to test if g is primitive root of $\mathbb{F}_p^*$.

$\mathbb{F}_p^*$ is polynomial time if the factorisation of $p-1$ is known.

Discrete Logarithm Problem (DLP).
Let p be a prime and let $g, h \in \mathbb{F}_p^*$. Find an integer x (if it exists) such that $h = g^x \bmod p$. This integer x is denoted by $x = \log_g(h)$.

A solution to the exists if and only if $h \in \langle g \rangle$, that is to say h lies in the subgroup generated by g, and in this case the resulting discrete logarithm $x = \log_g h$ is determined modulo $p-1$. In general, the discrete logarithm problem in the finite field $\mathbb{F}_p^*$ is a hard computational problem. Thus we can potentially design cryptosystems motivated by the discrete logarithm problem, in the same way that the design of the RSA and Rabin cryptosystems was motivated by the factoring problem. We also therefore have to consider various algorithms for solving the discrete logarithm problem in order to analyse the security of such discrete logarithm cryptosystems.

5.2 The Silver-Pohlig-Hellman (SPH) Method

In a discrete logarithm problem in $\mathbb{F}_p^*$ for $g, h \in \mathbb{F}_p^*$, we need to compute an integer x such that $h = g^x \bmod p$, where x is defined modulo $p-1$. The idea of the Silver-Pohlig-Hellman method is to compute x modulo the prime powers $p_i^{e_i}$ if the factorisation $p-1 = \prod_{i=1}^{r} q_i^{e_i}$ of $p-1$, is known, and then to recover the full solution for $x \bmod (p-1)$ using the Chinese Remainder Theorem. We note that this method is completely general and can potentially find a discrete logarithm in any abelian group.

The main tool used by the SPH method is the group homomorphism

$$\Phi_{q_i}(z) = z^{(p-1)/q_i^{e_i}} \bmod p$$

from $\mathbb{F}_p^*$ to a cyclic subgroup $\mathbb{C}_{q_i^{e_i}}$ of $\mathbb{F}_p^*$ of order $q_i^{e_i}$. Thus if $h = g^x \bmod p$, then this homomorphism gives

$$\Phi_{q_i}(h) = \Phi_{q_i}(g^x) = \Phi_{q_i}(g)^{x \bmod q_i^{e_i}} \bmod p.$$

as $\mathrm{Im}(\Phi_{q_i}) = \mathbb{C}_{q_i^{e_i}}$ is cyclic of order $q_i^{e_i}$. For each prime q_i, we can use the homomorphism Φ_{q_i} to obtain $\Phi_{q_i}(g)$ and $\Phi_{q_i}(h)$ and hence find the value of $x \bmod q_i^{e_i}$. We can then the Chinese Remainder Theorem to find $x \bmod p-1$ and hence solve this Discrete Logarithm problem.

Example. We consider $\mathbb{F}_p^*$ for $p = 419$ and the subgroup $\langle g \rangle$ for $g = 190$, an element of order $q = \frac{1}{2}(p-1) = 209$ in $\mathbb{F}_p^*$. Suppose we wish to solve $190^x = 254 \bmod 419$ for x, so $h = 254$ in $g^x = h \bmod p$. The factorisation of $q = \frac{1}{2}(p-1)$ is given by $q = 209 = 11 \times 19$, so we take $q_1 = 11$ and $q_2 = 19$ to obtain the two homomorphisms

$$\begin{aligned} \Phi_{11}(z) &= z^{209/11} = z^{19} \bmod 419 \\ \text{and } \Phi_{19}(z) &= z^{209/19} = z^{11} \bmod 419. \end{aligned}$$

However, these two homomorphisms satisfy

$$\Phi_{q_i}(g)^x = g^{(q/q_i)^x} = \Phi_{q_i}(h) = h^{q/q_i} \bmod p$$

so for $g = 190$, $h = 254$ and $p = 419$, and noting that the exponents "work modulo q_i" we obtain

$$\begin{aligned} & (190^{19})^x = 152^{(x \bmod 11)} = 254^{19} = 169 \quad \bmod 419, \\ \text{and} \quad & (190^{11})^x = 7^{(x \bmod 19)} = 254^{11} = 199 \quad \bmod 419. \end{aligned}$$

Thus we have transformed the calculation of a discrete logarithm in a cyclic group of size $p - 1 = 418$ into the calculation of discrete logarithms in the two far smaller cyclic groups of sizes 11 and 19. We can potentially determine these simpler calculations, and in this case an exhaustive search of the possible exponents for these cyclic groups shows that

$$\begin{aligned} 152^{(3 \bmod 11)} &= 169 \quad \mathrm{mod}419 \\ \text{and } 7^{(8 \bmod 19)} &= 199 \quad \mathrm{mod}419. \end{aligned}$$

Thus we obtain the linear congruences

$$x = 3 \bmod 11 \quad \text{and} \quad x = 8 \bmod 19.$$

The Chinese Remainder Theorem shows that the discrete log is

$$x = 179 \bmod 208,$$

	0	1	...	10	11	12	13	14	15
Baby Step g^i	1	190	...	236	7	73	43	209	324
Giant Step hu^j	254	112	...	257	209	333	292	363	5

Figure 11: **Baby-Step-Giant-Step Algorithm** for $190^x = 254 \bmod 419$.

that is to say $190^{179} = 254 \bmod 419$. □

We can further adapt the Silver-Pohlig-Hellman method to particular situations of interest. For example, suppose the g_0 has order q^e with $e > 1$ and $h_0 = g_0^x \bmod p$. We can write $x = x_0 + x_1 q + \cdots + x_{e-1} q^{e-1}$ where $0 \leq x_0, \ldots x_{e-1} < q$. Raising the original equation to the power q^{e-1} gives

$$h_0^{q^{e-1}} = (g_0^{q^{e-1}})^{x_0} \bmod p.$$

We can then potentially compute x_0 by trying all possibilities between 0 and q or by some other method. We can then consider

$$h_1 = h_0 g_0^{-x_0} = g_0^{x_1 q + \cdots + x_{e-1} q^{e-1}} = g_0^{q(x_1 + \cdots + x_{e-1} q^{e-2}} \bmod p$$

to recover x_1 and so on to obtain a full solution for x modulo q^e.

Complexity. The complexity of the SPH method depends on the size of the largest prime factor q_r. Computing the homomorphisms Φ_i and other associated values has polynomial time complexity. Hence the complexity is $\tilde{O}(q_r)$ where q_r is the largest prime divisor of $p-1$. In particular, small prime factors of $p-1$ give no meaningful added security in discrete logarithm systems. Thus we will generally assume that every element g of interest has prime order q dividing $(p-1)$.

5.3 The Baby-Step-Giant-Step (BSGS) Method

The Baby-Step-Giant-Step (BSGS) method is an example of the time and memory tradeoff approach in algorithm design, in which we can speed up a process at the expense of increased storage requirements. In order to to solve $g^x = h \bmod p$ to calculate the discrete logarithm x, we suppose that g has order q and let $m = \lceil \sqrt{q} \rceil$. The BSGS method proceeds by calculating two lists, each with $m \approx q^{\frac{1}{2}}$ elements, and then looking for a match between the two lists. In more detail, we compute one list $g^0, g^1, \ldots, g^{m-1}$ of *baby steps* and another list $hu^0, hu^1, \ldots, hu^{m-1}$ of *giant steps*, where $u = g^{-m}$.

If $h \in \langle g \rangle$, then we can find a match between the two lists of the form $g^i = hu^j = hg^{-mj} \bmod p$ for some $0 \leq i, j < m$ giving $h = g^{mj+i} \bmod p$, and so $x = mj + i$ is the required discrete logarithm.

Example. We consider the previous example with $\mathbb{F}_p^*$ for $p = 419$ and the subgroup $\langle g \rangle$ for $g = 190$, an element of order $q = \frac{1}{2}(p-1) = 209$ in $\mathbb{F}_p^*$. where we wish to solve $190^x = 254 \bmod 419$ for x, so $h = 254$ in $g^x = h \bmod p$. We take $m = \lceil \sqrt{209} \rceil = 15$. Figure 11 gives a partial Table of the baby steps $g^i = 190^i \bmod 419$ and the giant steps $hu^j = 254 \times 172^j \bmod 419$, where $u = g^{-m} = 190^{-15} = 333^{15} = 172 \bmod 419$. We see that there is a match "at 209" for baby step $i = 14$ and giant step $j = 11$, as

$$g^{14} = 190^{14} = hu^{11} = hg^{-11m} = 254 \times 190^{-165} = 209 \bmod 419.$$

Thus $h = 254 = g^{14+165} = g^{179} = 190^x \bmod 419$, and so the required discrete log is $x = 179$. □

Complexity. The BSGS method requires $O(\sqrt{q}(\log p)^2)$ time to compute all of the baby steps and all of the giant steps, and the Tables of baby steps and giant steps require $O(\sqrt{q}\log(p))$ storage. Searching these Table takes time $O(\log(q)\log(p)) = O((\log p)^2)$, and so the second stage has time complexity $O(\sqrt{q}(\log p)^2)$. Thus the total running time is $\tilde{O}(\sqrt{q})$ steps, the storage requirement is $\tilde{O}(\sqrt{q})$, which may be prohibitive. However the BSGS method is one that potentially works in any abelian group.

5.4 The Pollard Rho Method

In order to to solve $g^x = h \bmod p$ to calculate the discrete logarithm x, where we suppose that g has order q, the Pollard Rho method uses a deterministic walk through the elements of of $\langle g \rangle$ that appears pseudorandom. This Pollard Rho method for finding a discrete logarithm has the same time complexity $\tilde{O}(\sqrt{q})$, where q dividing $p-1$ is the order of g, as the Baby-Step-Giant-Step (BSGS) method but overcomes the storage problems. in

The *birthday paradox* shows that such a pseudorandom walk returns to some previously seen element after about $\sqrt{q}$ steps, after which the deterministic walk simply repeats this cycle. Thus this pseudorandom walk through $\langle g \rangle$ can be thought as resembling the Greek letter rho ρ, where the letter tail represents the lead-in period, which is then followed by a repeating cycle. The Pollard Rho method partitions $\mathbb{F}_p^*$ into three (or more) subsets, for example if we identify $\mathbb{F}_p^*$ with integers in the set $\{u | 1 \leq u < p\} = S_1 \cup S_2 \cup S_3$ where these partition subsets S_1, S_2 and S_3 are given by

$$S_1 = \left\{1, \ldots, \left\lfloor \tfrac{1}{3}p \right\rfloor\right\}, S_2 = \left\{\left\lceil \tfrac{1}{3}p \right\rceil, \ldots, \left\lfloor \tfrac{2}{3}p \right\rfloor\right\}, S_3 = \left\{\left\lfloor \tfrac{2}{3}p \right\rfloor, \ldots, p-1\right\}.$$

i	1	2	...	12	13	14	15	...	22	23	24	25	...
x_i	1	190	...	192	411	63	238	...	287	411	63	238	...
a_i	0	1	...	12	24	24	25	...	104	104	104	105	...
b_i	0	0	...	22	44	45	45	...	185	186	187	187	...

Figure 12: **Pollard Rho Algorithm** for $190^x = 254 \bmod 419$.

We can then define a deterministic walk through $\langle g \rangle \leq \mathbb{F}_p^*$ by setting $x_1 = 1$ and then iteratively defining

$$x_{i+1} = f(x_i) = \begin{cases} gx_i \bmod p & \text{if } x_i \in S_1 \\ x_i^2 \bmod p & \text{if } x_i \in S_2 \\ hx_i \bmod p & \text{if } x_i \in S_2. \end{cases}$$

This process yields a pseudorandom sequence $x_1, x_2, \ldots$ of elements in $\langle g \rangle \leq \mathbb{F}_p^*$ giving the "rho-like" behaviour: an initial 'random' section, followed by a segment that repeats. This sequence $x_1, x_2, \ldots$ therefore allows us to find a collision $x_i = x_j$ of the sequence (for $i \neq j$), and furthermore we can write

$$x_i = g^{a_i} h^{b_i} \bmod p \quad \text{and} \quad x_j = g^{a_j} h^{b_j} \bmod p,$$

where the sequences $a_1, a_2, \ldots$ and $b_1, b_2, \ldots$ can be determined by setting $a_1 = b_1 = 0$ and then using the iterative updating

$$\begin{array}{llll} a_{i+1} = a_i + 1 \bmod q & \text{and} & b_{i+1} = b_i & \text{if } x_i \in S_1, \\ a_{i+1} = 2a_i \bmod q & \text{and} & b_{i+1} = 2b_i \bmod q & \text{if } x_i \in S_2, \\ a_{i+1} = a_i \bmod q & \text{and} & b_{i+1} = b_i + 1 \bmod q & \text{if } x_i \in S_3. \end{array}$$

Thus we can determine a_i, b_i, a_j, b_j such that

$$x_i = g^{a_i} h^{b_i} = x_j = g^{a_j} h^{b_j} \bmod p, \quad \text{or equivalently } g^{a_i - a_j} = h^{b_j - b_i} \bmod p.$$

If $b_j - b_i$ is invertible modulo q, then we obtain

$$h = g^{c_{j,i}(a_i - a_j)} \bmod p, \quad \text{where } c_{j,i} = (b_j - b_i)^{-1} \bmod q,$$

in which case the solution to the discrete logarithm problem $h = g^x \bmod p$ is $x = c_{j,i}(a_i - a_j)$. If $b_j - b_i$ is not invertible modulo q, then we have a whole cycle of points giving $x_{i'} = x_{j'}$ and we can just try other values i', j'.

Example. We consider the previous example with $\mathbb{F}_p^*$ for $p = 419$ and the subgroup $\langle g \rangle$ for $g = 190$, an element of order $q = \frac{1}{2}(p-1) = 209$

i	1	2	...	19	20	21	...
x_i	1	190	...	316	235	336	...
a_i	0	1	...	52	52	104	...
b_i	0	0	...	91	92	384	...
x_{2i}	190	389	...	59	235	287	...
a_{2i}	1	3	...	15	16	32	...
b_{2i}	0	1	...	48	49	99	...

Figure 13: **Floyd Cycle Finding Method** for $190^x = 254 \bmod 419$.

in $\mathbb{F}_p^*$. where we wish to solve $190^x = 254 \bmod 419$ for x, so $h = 254$ in $g^x = h \bmod p$. Figure 12 gives parts of the random walk x_i for the Pollard Rho algorithm in this case, together with the corresponding a_i and b_i. It can be seen that a repeating cycle $411, 63, 238, \ldots, 287$ first begins at 13 and then repeats at 23 and so on, so we can take $i = 13$ and $j = 23$ to obtain

$$x_{13} = g^{24}h^{44} = x_{23} = g^{104}h^{186} = 41 \bmod 419$$
$$\text{giving } g^{24-104} = g^{-80} = g^{129} = h^{186-44} = h^{142} \bmod 419.$$

However $142^{-1} = 131 \bmod 418$ and so we obtain

$$g^{129\times 142^{-1}} = g^{129\times 131} = g^{179} = 190^{179} = h = 254 \bmod 419,$$

giving solution $x = 179$ to the discrete log problem $190^x = 254 \bmod 419$. □

The birthday paradox shows that we expect to obtain a matching pair $x_i = x_j$ in the list $x_1, x_2, \ldots$ once this list contains about $\sqrt{q}$ elements, where q is the order of g. However, the method as described above still requires the storage and processing of the list $x_1, x_2, \ldots$ of size about $q^{\frac{1}{2}}$. A method that avoids such storage is *Floyd's cycle finding* method. This method involves iteratively computing the pair (x_i, x_{2i}) and checking whether $x_i = x_{2i}$ at each step. The idea is that x_{2i} travels twice as fast round the "head" of the "rho" ρ as x_i, and so the x_{2i} eventually catches up with the x_i list. For example, suppose we have a rho ρ with a "tail" of length t and a "head" of length h, then it takes t steps before both x_i and x_{2i} are on the "head" and then at most h additional steps before the fast sequence x_{2i} catches up with the slow sequence x_i, so we detect a collision after at most $h + t$ steps.

Example. We consider the previous example with $\mathbb{F}_p^*$ for $p = 419$ and the subgroup $\langle g \rangle$ for $g = 190$, an element of order $q = \frac{1}{2}(p-1) = 209$ in $\mathbb{F}_p^*$. where we wish to solve $190^x = 254 \bmod 419$ for x, so $h = 254$ in

$g^x = h \bmod p$. Figure 13 gives parts of the random walk x_i and x_{2i} and associated values. We see that for $i = 20$ the "slow walk" x_i and the "fast walk" x_{2i} coincide "at 235", so we have

$$\begin{aligned} 235 &= x_{20} = g^{a_{20}} h^{b_{20}} = 190^{52} \times 254^{92} \mod 419 \\ &= x_{40} = g^{a_{40}} h^{b_{40}} = 190^{16} \times 254^{49} \mod 419 \end{aligned}$$

$$\text{giving } 190^{52-16} = 190^{36} = 254^{49-92} = 254^{-43} = 254^{166} \bmod 419.$$

We note that $166^{-1} = 34 \bmod 209$, so $190^{34 \times 36} = 190^{179} = 254 \bmod 419$, giving discrete logarithm $x = 179$ in $g^x = h \bmod p$.

Complexity. The complexity of the Pollard Rho algorithm is fundamentally given by the Birthday paradox, which shows that there is a collision after about $(\frac{1}{2}\pi)^{\frac{1}{2}} q^{\frac{1}{2}}$ steps of the walk. This collision can essentially be detected by the Floyd cycle finding method which can calculate all required quantities $x_i, a_i, b_i, x_{2i}, a_{2i}, b_{2i}$ dynamically, so using minimal storage. Each step of the walk can be performed in $O((\log p)^2)$ bit operations, so the expected running time is $O(q^{\frac{1}{2}} (\log p)^2)$ with small memory requirement. Further enhancements can be made to the Pollard Rho method. For example, the Pollard Kangaroo or Lambda Method is a variant of the Pollard Rho method which is useful if the discrete logarithm is known to lie in a short interval. Furthermore, the Pollard methods can be generalised to give a discrete logarithm method which is easily distributed over the internet.

5.5 The Index Calculus Method

The Index Calculus method is based on the same ideas as those of the quadratic sieve factoring algorithm. We let g be a primitive root modulo p and let $h = g^x \bmod p$, and as usual we are trying to find x. The idea is to choose a *factor base* $\mathcal{B} = \{p_1, \ldots, p_r\}$ consisting of the first r primes. The Index Calculus method first aims to find the discrete logarithms of these primes p_i with respect to g for $1 \leq i \leq r$. The Index Calculus method then attempts to use this information to compute the discrete logarithm of h with respect to g.

The Index Calculus method first computes random powers $w = g^k \bmod p$ where $1 \leq k < q$ and $1 \leq w < p$. If $w = \prod_{i=1}^{r} p_i^{e_i}$ factors over the factor base $\mathcal{B} = \{p_1, \ldots, p_r\}$, then we have a *relation*. We store the exponents $(e_1, \ldots, e_r)$ as a row in a matrix and value k as a corresponding component in a vector, and we note that

$$k = \sum_{i=1}^{r} e_i \log_g(p_i) \bmod (p-1).$$

We eventually obtain a matrix of full rank and can perform Gaussian elimination (over the ring $\mathbb{Z}_{p-1}$) to diagonalise the matrix. We therefore compute the actual discrete logs of each prime p_i, that is to say we find a_i such that $p_i = g^{a_i} \bmod p$. The final stage is to find a relation which includes h. We choose random values k until $hg^k \bmod p$ factors over $\mathcal{B}$ as $\prod_{i=1}^{r} p_i^{e_i}$. It is then trivial to compute the discrete logarithm of h as

$$\log_g(h) = \left(\sum_{i=1}^{r} e_i a_i \right) - k \bmod (p-1).$$

Example. We let $p = 223$, $g = 122$ (which has order $p - 1 = 222$) and $h = 199$, and we wish to find x such that $h = 199 = g^x = 122^x \bmod 223$. We construct the factor base $\mathcal{B} = \{2, 3, 5\}$ and find the following small powers of g with smooth factorisations.

k	$g^k \bmod p$	Factorization
6	120	$2^3 \times 3 \times 5$
12	128	2^7
13	6	2×3

We represent the relations as the matrix and corresponding vector, where the row order in this case is chosen to give a 1 in the top left entry.

$$\left(\begin{array}{ccc|c} 1 & 1 & 0 & 13 \\ 7 & 0 & 0 & 12 \\ 3 & 1 & 1 & 6 \end{array} \right).$$

The columns of the matrix give the appropriate powers of the factor base $\mathcal{B} = \{2, 3, 5\}$. We now perform an appropriate form of row reduction to give

$$\longrightarrow \left(\begin{array}{ccc|c} 1 & 1 & 0 & 13 \\ 0 & -7 & 0 & -79 \\ 0 & -2 & 1 & -33 \end{array} \right) \longrightarrow \left(\begin{array}{ccc|c} 1 & 1 & 0 & 13 \\ 0 & 1 & 0 & 43 \\ 0 & -2 & 1 & -33 \end{array} \right)$$

$$\longrightarrow \left(\begin{array}{ccc|c} 1 & 1 & 0 & 13 \\ 0 & 1 & 0 & 43 \\ 0 & 0 & 1 & 53 \end{array} \right) \longrightarrow \left(\begin{array}{ccc|c} 1 & 0 & 0 & 192 \\ 0 & 1 & 0 & 43 \\ 0 & 0 & 1 & 53 \end{array} \right)$$

We deduce that $2^1 = 2 = g^{192} \bmod 223$, that $3^1 = 3 = g^{43} \bmod p$ and that $5^1 = 5 = g^{53} \bmod 223$. We now look for elements of the form hg^i expressible over the factor base $\mathcal{B}$. We note that $hg = 194 \bmod p$ does not factor over $\mathcal{B}$, but that

$$hg^2 = 30 = 2 \times 3 \times 5 = g^{192+43+53} = g^{288} = g^{66} \bmod 223.$$

This implies that $h = 199 = g^{64} = 122^{64} \bmod 223$, so the required discrete log is $x = 64$ in $122^x = 199 \bmod 223$. □

There are some subtleties with the linear algebra since we are not working over a field. One further trick to improve performance slightly is to include -1 as an element of $\mathcal{B}$. We can then reduce $w = g^k$ to the range $-p/2 < w < p/2$ and try to factor over $\mathcal{B}$. Another trick is the "Waterloo improvement". Given $w = g^k \bmod p$ we can use Euclid's algorithm to find integers b, c such that $bw = c \bmod p$ and $0 < b, c < 2\sqrt{p}$. These integers b and c are more likely to factor over $\mathcal{B}$ than w.

Complexity. The complexity of Index Calculus depends on the size of the factor base. If we choose $\mathcal{B} = \{\text{primes } p_i \leq B\}$, then the size of the factor base $\#\mathcal{B} \approx B/\log_e(B)$. The probability that a random integer $1 \leq w < p$ is *B-smooth* (expressible solely in terms of elements of $\mathcal{B}$) is approximately u^{-u} where $u = \log_e(p)/\log_e(B)$.

In the first stage of the Index Calculus method, we must gather approximately $\#\mathcal{B}$ relations. For each relation we expect to have to try roughly u^u different choices for $w = g^k \bmod p$, and factorising each one over $\mathcal{B}$ requires $O(\#\mathcal{B}(\log p)^2)$ operations. Thus the total complexity of the first stage is $O((\#\mathcal{B})^2 u^u (\log p)^2)$. The storage requirement is $O((\#\mathcal{B})^2(\log p))$ if naive methods are used.

The second stage of Index Calculus requires Gaussian elimination over the ring of integers modulo $p-1$. This stage has complexity $O((\#\mathcal{B})^3(\log p)^2)$ if naive methods are used. The right approach is to try to balance these costs by taking

$$u^u \approx \#\mathcal{B}.$$

In fact, we find that the right choice for B, for an appropriate constant c is

$$\ln(B) = c' \left(\log_e(p) \log_e(\log_e(p))\right)^{\frac{1}{2}}$$

This choice yields a total complexity of $O(L_p(1/2, c))$, where L is the subsexponential function.

6 Further Discrete Log Cryptosystems

This Section discusses the Diffie-Hellman, El Gamal and DSS cryptosystems, which are based on the Discrete Logarithm Problem and related Problems, together with the relationships between these Problems.

6.1 The Diffie-Hellman Key Agreement Protocol

The Diffie–Hellman key agreement Protocol (also called Diffie–Hellman key exchange) can be used when two parties Alice and Bob want to agree on a secret random key K and is described in Figure 14. This allows both Alice and Bob know $K = g^{ab} \bmod p$, where the public information is the prime p, the base element $g \in \mathbb{F}_p^*$ and the two public keys $g^a, g^b \bmod p$, and a, b are the private keys. This shared secret value K can then be used to derive a key for a symmetric encryption scheme for the bulk encryption of data or some other use. However, an eavesdropper knowing only $g^a, g^b \bmod p$ cannot determine K unless they can solve the following computational problem.

Diffie–Hellman Problem (DHP). Given the triple (g, g^a, g^b) of elements of $\mathbb{F}_p^*$, compute $g^{ab} \bmod p$.

We note that the Diffie-Hellman problem (DHP) is sometimes called the Computational Diffie–Hellman problem (CDHP) in order to clearly distinguish it from the decisional version of this problem, which is stated below.

Decisional Diffie–Hellman Problem (DDHP). Given the quadruple (g, g^a, g^b, g^c) of elements of $\mathbb{F}_p^*$, does $g^c = g^{ab} \bmod p$?

For completeness, we also state the clearly related Discrete Logarithm Problem (DLP) discussed earlier in Section 5.1.

Discrete Logarithm Problem (DLP). Let p be a prime and let $g, h \in \mathbb{F}_p^*$. Find an integer x (if it exists) such that $h = g^x \bmod p$. This integer x is denoted by $x = \log_g(h)$.

The Decisional Diffie-Hellman Problem, the (computational) Diffie-Hellman Problem and the Discrete Logarithm Problem are related by the chain of Turing reductions

$$\text{DDHP} \ \leq_T \ \text{DHP} \ \leq_T \ \text{DLP} \ ,$$

where the individual reductions are given by the following approaches.

- DHP $\leq_T$ DLP. Compute x from g^x and y from g^y, and compute g^{xy}.
- DDHP $\leq_T$ DHP Compute g^{xy} from g^x and g^y, and check if $g^c = g^{xy}$.

Set-Up. Prime p and an element $g \in \mathbb{F}_p^*$.
Alice's private and public keys.
Alice chooses a random integer $1 \leq a < p-1$ as her *private key*.
Alice sends $A = g^a \bmod p$ to Bob as her *public key*.
Bob's private and public keys.
Bob chooses a random integer $1 \leq a < p-1$ as his *private key*.
Bob sends $A = g^a \bmod p$ to Alice as his *public key*.
Alice's and Bob's shared secret value.
On receiving B, Alice computes $K = B^a = g^{ab} \bmod p$.
On receiving A, Bob computes $K = A^b = g^{ab} \bmod p$.

Figure 14: Diffie-Hellman Key Agreement Protocol in $\mathbb{F}_p^*$.

Furthermore, we note that the "reverse" Turing reduction from the Discrete Logarithm Problem (DLP) to Diffie-Hellman Problem (DHP) has not been established, though it seems difficult to conceive of a method of solving the DHP without also solving the DLP.

6.2 The El Gamal Public Key Encryption Scheme

We give a formal description of the El Gamal public key encryption scheme, based on the Discrete Logarithm Problem, for a security parameter λ below, and we note that the El Gamal encryption process is randomised. A more informal presentation of El Gamal encryption is given in Figure 15.

- `KeyGen`. Generate a λ-bit prime p and an element $g \in \mathbb{F}_p^*$. Choose a random integer $0 < a < p-1$ and set $h = g^a \bmod p$. The public key is (p, g, h) and the private key is a.
- All users in a system can share the same prime p, when the parameters p and g are fixed for all users, and only the value $h = g^a \bmod p$ changes. The message space is $M_\lambda = \mathbb{F}_p^*$, and following "message doubling" the ciphertext space is $C_\lambda = \mathbb{F}_p^* \times \mathbb{F}_p^*$.
- `Encrypt`. To encrypt a message m with a public key (p, g, h), a random k between 0 and $p-1$ is chosen, and the values $c_1 = g^k \bmod p$ and $c_2 = mh^k \bmod p$. are computed. The ciphertext is then (c_1, c_2).
- `Decrypt`. The ciphertext (c_1, c_2) is decrypted to give the message $m = c_2 c_1^{-a} \bmod p$.

Example. Consider an El Gamal encryption scheme using the prime $p = 223$, base element $g = 79$, private key $a = 133$ and public key $h = g^a = 79^{133} = 85 \bmod p = 223$.

Encryption. To encrypt a message $m = 43$, we choose a random value $k = 203$ and calculate

$$\begin{aligned} c_1 &= g^k = 79^{203} = 71 \mod p = 223 \\ \text{and } c_2 &= mh^k = 43 \times 85^{203} = 114 \mod p = 223. \end{aligned}$$

Thus a ciphertext corresponding to message $m = 43$ is $(c_1, c_2) = (71, 114)$.

Decryption. The decryption of ciphertext $(c_1, c_2) = (71, 114)$ can be calculated as $m = c_2 c_1^{-a} = 114 \times 71^{-133} = 114 \times 151 = 43 \bmod 223$. □

In practice, the "message" m encrypted by the El Gamal `Encrypt` process is usually the output of some padding scheme, as with RSA and Rabin. Similarly, we normally encrypt a bulk message using a symmetric cipher with a symmetric key derived from an El Gamal ciphertext.

The Textbook version of El Gamal encryption given above does not have semantic security under a passive attack. For example, if we consider Legendre symbols, then $\left(\frac{g}{p}\right) = 1$ means that $\left(\frac{m}{p}\right) = \left(\frac{c_2}{p}\right)$. Similarly, if $\left(\frac{g}{p}\right) = -1$ then the parity of k and a are deduced from $\left(\frac{c_1}{p}\right)$ and $\left(\frac{h}{p}\right)$, so we can deduce $\left(\frac{m}{p}\right)$ from $\left(\frac{c_2}{p}\right)$. The issue of whether this Textbook El Gamal encryption process has the one-way encryption (OWE) security property depends on whether the DHP is hard or not.

El Gamal encryption is best understood as a Diffie–Hellman *key transport* protocol. By this we mean that Bob is essentially conducting a Diffie–Hellman key exchange with Alice where he sends g^k and where Alice's component is her public key g^a. The shared key here is g^{ak} which can then be used as a key for any symmetric encryption scheme.

El Gamal encryption and decryption have a polynomial time complexity and so can be considered efficient; an El Gamal encryption requires two exponentiations modulo p, and an El Gamal decryption requires a single modular encryption.

An El Gamal encryption scheme can be similarly specified for a finite field $\mathbb{F}_{p^n}$ with a prime power number of elements. More generally, an El Gamal encryption scheme can also be specified for an arbitrary finite abelian group by replacing modulo multiplication with the appropriate abelian group operation.

Set-Up. Prime p and an element $g \in \mathbb{F}_p^*$.

Alice's private and public keys.

Alice chooses a random integer $1 \leq a < p-1$ as her *private key.*

Alice publishes $h = g^a \bmod p$ as her *public key.*

Encryption of a message $m \in \mathbb{F}_p^*$.

A random integer k such that $1 \leq k < p-1$ is chosen.

The values $c_1 = g^k \bmod p$ and $c_2 = mh^k \bmod p$ are calculated.

Ciphertext corresponding to m is (c_1, c_2).

Decryption of a ciphertext $(c_1, c_2) \in \mathbb{F}_p^* \times \mathbb{F}_p^*$.

The message corresponding to ciphertext (c_1, c_2) is $c_2 c_1^{-a} \bmod p$.

Figure 15: El Gamal Encryption Scheme in $\mathbb{F}_p^*$.

6.3 Parameter Sizes and Performance for El Gamal

We have seen that index calculus methods can solve discrete logarithms in subexponential time $L_p(1/3, 1.923)$, and so we must choose p to be sufficiently large. Current recommendations are that p is at least 2^{1024} and more usually $p \geq 2^{2048}$, so a prime p used in discrete logarithm based cryptography is comparable in size to an RSA modulus. However, Pollard Rho and similar methods can solve discrete logarithms in a subgroup of $\mathbb{F}_p^*$ of prime order q in time $\widetilde{O}(\sqrt{q})$, so current recommendations are that $q > 2^{160}$.

We can therefore work in a subgroup of order q in a finite field $\mathbb{F}_p^*$, where q is much smaller than p, provided both p and q are sufficiently large, by choosing g in the various protocol descriptions to be an element of order q, rather than a random element of $\mathbb{F}_p^*$. We can efficiently find primes p and q of the right size such with q dividing $(p-1)$ by choosing a prime q of the right size and then choosing random values k of the right size until $p = kq + 1$ is a prime.

In the El Gamal encryption system, the public key is a pair of elements $g, h = g^a \in \mathbb{F}_p^*$, where g has prime order q. An element g of order q can be generated by generating a random $u \in \mathbb{F}_p^*$, and letting $g = u^{\frac{p-1}{q}}$, so g has order q unless $g = 1$. The message m is an element of $\mathbb{F}_p^*$, and the ciphertext is (g^k, mh^k) is an element of $\mathbb{F}_p^* \times \mathbb{F}_p^*$, where k is a random integer Thus ciphertext is a pair of group elements which is very large (say 4096 bits) compared with an RSA ciphertext (say 2048 bits), though this disadvantage is less of a problem when hybrid encryption is used. However, if g is chosen

to have order $q \approx 2^{160}$, then the private key a and random value k are relatively short. El Gamal encryption can be performed relatively quickly, though we have to do two exponentiations modulo p, and each of them is much slower than using short public exponents for RSA, whereas El Gamal decryption is roughly twice as fast as RSA decryption using the Chinese Remainder Theorem, as detailed in the Table below.

	Encrypt	Decrypt
RSA	Very fast	Slow
El Gamal	Slow	Fast

In summary, the basic computational operation for a cryptosystem based on the discrete logarithm is modular exponentiation, as for RSA. For RSA, there are various shortcuts to speed up this process, such as small public exponents and the use of the Chinese Remainder Theorem. Working in a subgroup of relatively small order q is an analogous shortcut for discrete logarithm cryptography. Other shortcuts have been proposed for discrete logarithm based systems, such as window methods and choosing low Hamming weight exponents, but we will not consider them here.

6.4 Security of El Gamal Encryption

We now give a precise security reduction for an El Gamal cryptosystem, proving following the result. We then give a number of instances where the El Gamal cryptosystem does not have certain security properties.

Theorem. The computational problem of breaking One-Way Encryption (OWE) security of the El Gamal cryptosystem under a passive attack is polynomial time equivalent to Diffie-Hellman Problem (DHP).

Proof. Let $\mathcal{O}$ be an oracle which solves the DHP in the subgroup of order q in $\mathbb{F}_p^*$. We consider an El Gamal system with private key x and public key $h = g^x \bmod p$, in which a message m is encrypted to give ciphertext $(c_1, c_2) = (g^k, mh^k)$. We submit $(g, h, c_1) = (g, g^x, g^k)$ to the DHP Oracle $\mathcal{O}$, which returns g^{xk} to this Diffie-Hellman problem. We can then retrieve the message m as $c_2 g^{-xk} = mh^k g^{-xk} = mg^{xk} g^{-xk} = m \bmod p$, so

$$\text{OWE Security} \leq_T \text{DHP}.$$

For the other direction, let $\mathcal{O}$ be an oracle which takes an El Gamal public key $(g, h) = (g, g^x)$ for unknown x and an El Gamal ciphertext $(c_1, c_2) = (g^k, mh^k)$, and returns the corresponding message m. Given a DHP input (g, g^x, g^k), we submit public key (g, g^x) and ciphertext $(g^k, 1)$ to the El

Gamal Decryption Oracle $\mathcal{O}$ which returns m satisfying $mh^k = 1$, so providing $m = h^{-k} = g^{-xk} \bmod p$. Thus we can obtain g^{-xk} as the solution to this Diffie-Hellman problem, and so

$$\text{DHP} \leq_T \text{OWE Security}.$$

Thus breaking One-Way Encryption (OWE) security for the El Gamal cryptosystem under a passive attack is polynomial time equivalent to solving the Diffie-Hellman Problem (DHP). □

. El Gamal encryption does not have the OWE security property under an adaptive chosen ciphertext (CCA2) attack. Given an El Gamal ciphertext $C = (c_1, c_2)$, an El Gamal Decryption Oracle $\mathcal{O}$ with input $C' = (c_1, 2c_2)$ provides m' giving the original message as $m = 2^{-1}m'$.

The El Gamal cryptosystem does not provide semantic security under passive attacks, because we can compute the Legendre symbol of the message in polynomial time from the public key and the ciphertext. If $\left(\frac{g}{p}\right) = 1$ then $\left(\frac{m}{p}\right) = \left(\frac{c_2}{p}\right)$. By contrast, if $\left(\frac{g}{p}\right) = -1$ then we can see if a and k are odd or even, and a is odd if and only if $\left(\frac{h}{p}\right) = -1$ and so on, so we can determine $\left(\frac{g^{ak}}{p}\right)$. If $\left(\frac{h}{p}\right) = \left(\frac{c_1}{p}\right) = -1$ then $\left(\frac{m}{p}\right) = -\left(\frac{c_2}{p}\right)$ otherwise $\left(\frac{m}{p}\right) = \left(\frac{c_2}{p}\right)$.

An attack similar to the Legendre symbols attack can be applied for any small prime $\ell \mid (p-1)$, as the Legendre symbol is just a group homomorphism from $\mathbb{F}_p^*$ to the subgroup $\{1, -1\}$ of order 2. As we saw when discussing the Silver-Pohlig-Hellman method, for any $\ell \mid (p-1)$ there is a homomorphism from $\mathbb{F}_p^*$ to the unique subgroup of order ℓ. We can therefore compute $a \bmod \ell$ and $k \bmod \ell$ and thus obtain the value of the projection of m into the subgroup of order ℓ. We therefore generally assume that all public keys and messages lie in a subgroup of $\mathbb{F}_p^*$ of large prime order q.

For Indistinguishability (IND) under passive attacks, we suppose that $C = (c_1, c_2)$ is an El Gamal ciphertext which is the encryption of either m_0 or m_1, which are elements of $\mathbb{F}_p^*$ of large prime order q, so $c_1 = g^k \bmod p$ for some k and $c_2 = m_i h^k \bmod p$ for $i = 0$ or 1. The problem of distinguishing between the two encryptions is equivalent to the problem of determining whether $(g, c_1 = g^k, h = g^a, c_2 m_0^{-1})$ is a valid Diffie-Hellman tuple, that is

to say whether $c_2 = m_0 g^{ak} \bmod p$ or not, so demonstrating the following result.

Theorem The computational problem of breaking the Indistinguishability (IND) security property of El Gamal under passive attacks is polynomial time equivalent to the Decision Diffie-Hellman Problem (DDHP).

The best way to perform an El Gamal encryption is generally by using padding schemes (similar to RSA-OAEP) or hybrid encryption. We can prove IND-CCA2 security in the random oracle model if the the Diffie-Hellman Problem (DHP) is hard. The Cramer-Shoup scheme is a variant of El Gamal with IND-CCA2 security in the standard model.

6.5 El Gamal Signature Scheme

We can also construct digital signatures from schemes based on the difficulty of taking discrete logarithms. However, there is a conceptual difference between an RSA signatures and a discrete logarithm signature. We can think of RSA as a function

$$F(m) = c = m^e \bmod p$$

for which the owner of the private key has an inverse

$$F^{-1}(c) = c^d \bmod p.$$

We encrypt by computing $F(m)$, and we construct a signature by computing $s = F^{-1}(H(m))$. By contrast, El Gamal encryption can be thought of as the function

$$F(k, m) = (g^k, mh^k).$$

However, El Gamal decryption does not give the inverse F^{-1}, as k is not computed because the discrete logarithm problem is hard (even for the owner of the private key).

The construction of a digital signature based on the discrete logarithm problem requires a more general approach in which the signer of a message with public verification key $h = g^a \bmod p$ proves they know the value of the private signing key a. This is the approach taken by the El Gamal signature scheme, which is shown for an appropriate of cryptographic hash function (mapping into $\mathbb{F}_p^*$) in Figure 16. We note that this El Gamal signature scheme can be adapted to a finite field $\mathbb{F}_{p^n}^*$ and also generalised to a finite abelian group.

Set-Up. Prime p and an element $g \in \mathbb{F}_p^*$.

Alice's private and public keys.

Alice chooses a random integer $1 \leq a < p-1$ as her *private key.*

Alice publishes $h = g^a \bmod p$ as her *public key.*

Signature for a message $m \in \mathbb{F}_p^*$.

A random integer k coprime to $p-1$ with $1 \leq k < p-1$ is chosen.

Values $r = g^k \bmod p$, $s = (H(m) - ar)k^{-1} \bmod p-1$ are calculated.

Signature corresponding to m is (r, s).

Verification of a signature (r, s) **for a message** m.

Verify that $0 < r < p$ and $0 < s < p-1$.

Signature is valid if $h^r r^s \bmod p$ equals $g^{H(m)} \bmod p$.

Figure 16: El Gamal Signature Scheme in $\mathbb{F}_p^*$ with Hash Function H.

Example. Consider an El Gamal signature scheme using the prime $p = 223$, base element $g = 79$, private key $a = 133$ and public key $h = g^a = 79^{133} = 85 \bmod p = 223$.

Signature. In order to sign a message m with hash value $H(m) = 117$, we first choose a random value $k = 157$, with $k^{-1} = 181 \bmod (p-1) = 222$. We then calculate

$$\begin{aligned} r &= g^k = 79^{157} = 44 \mod p = 223 \\ s &= (H(m) - ar)k^{-1} = (117 - 133 \times 44)181 = 37 \mod (p-1) = 222. \end{aligned}$$

Thus a signature corresponding to message m with hash value $H(m) = 117$ is $(r, s) = (44, 37)$.

Verification. The verification of the signature $(r, s) = (44, 37)$ for the message m with hash value $H(m) = 117$ is valid as $h^r r^s = 85^{44}44^{37} = 62 \times 40 = 27 \bmod p = 223$ equals $g^{H(m)} = 79^{117} = 27 \bmod p = 223$. □

6.6 The Digital Signature Standard (DSS)

The *Digital Signature Standard* (DSS) was introduced as FIPS-186 (Federal Information Processing Standard) by NIST (National Institute of Standards and Technology) in 1994. The DSS requires the use of an approved hash function, and NIST has released various standardised hash functions over the last 30 years, most recently SHA-3 (also known as Keccak) in 2015. The

DSA Set-Up.

Key length L and Modulus length N with $N < L$, with
$(L, N) \in \{(1024, 160), (2048, 224), (2048, 256), (3072, 256)\}$.

Approved cryptographic hash function H with output at least N bits.

An N-bit prime q and an L-bit prime p such that q divides $p - 1$.

A value h with $1 < h < p - 1$ and $g = h^{\frac{p-1}{q}} \bmod p$ of order q.

Alice's private and public keys.

Alice chooses a random integer $1 \leq x \leq q - 1$ as her *private key*.

Alice publishes $y = g^x \bmod p$ as her *public key*.

Signature for a message $m \in \mathbb{F}_p^*$.

A random integer k with $1 \leq k < q$ is chosen.

Value $r = (g^k \bmod p) \bmod q$ is computed.

Values $s = (H(m) + xr)k^{-1} \bmod q$ is computed.

Signature corresponding to m is (r, s).

Verification of a signature (r, s) **for a message** m.

Verify that $0 < r < p$ and $0 < s < q$.

Compute $u_1 = H(m)s^{-1} \bmod q$ and $u_2 = rs^{-1} \bmod q$.

Signature is valid if $(g^{u_1} y^{u_2} \bmod p) \bmod q$ equals r.

Figure 17: Digital Signature Algorithm (DSA) within DSS

specific asymmetric signing transformation within the DSS is termed the Digital Signature Algorithm (DSA), and the DSA is specified in Figure 17.

The Digital Signature Algorithm (DSA) is a signature scheme based on the Discrete Logarithm Problem and is a development of the El Gamal signature scheme; the DSA can be regarded within the general framework of *Schnor signatures*. The DSA uses arithmetic in $\mathbb{F}_p^*$ for some L-bit prime p, but uses the cyclic subgroup $\langle g \rangle < \mathbb{F}_p^*$ of order q for a N-bit prime q dividing $p - 1$ with $N < L$, so giving a signature (r, s) for a message m to be given as a pair of values from $\{0, \ldots, q\}$. Thus DSA gives a $2N$-bit signature for a message, rather than the longer $2L$-bit of an equivalent El Gamal signature.

Example. We consder a DSA-like signature scheme based $q = 317$ ($L = 9$ bit prime) and $p = 509737$ ($N = 19$ bit prime), so $(p - 1)/q = 1608$. We choose $h = 23$ and set $g = h^{\frac{p-1}{q}} = 23^{1608} = 488659 \bmod p = 509737$. The private key is $x = 287$ and the corresponding public key is $y = g^x = 488659^{287} = 46932 \bmod p$.

Signature. To sign a message m with hash value $H(m) = 147$, we first choose a random value $k = 219$, with $k^{-1} = 262 \bmod q = 317$. We then calculate

$$\begin{aligned} r &= (g^k \bmod p) \bmod q = (488659^{219} \bmod p) \bmod q = 246360 = 51 \bmod q \\ s &= (H(m) + xr)k^{-1} = (145 + 287 \times 51)262 = 302 \bmod q. \end{aligned}$$

Thus we obtain the 18-bit signature $(r, s) = (51, 302)$ for the message hash $H(m) = 147$. We note for comparison that a corresponding El Gamal signature scheme in $\mathbb{F}_p^*$ gives a 38-bit signature.

Verification. For the message hash $H(m)$ with signature $(r, s) = (51, 302)$, we have $s^{-1} = 302^{-1} = 169 \bmod q$, so

$$\begin{aligned} u_1 &= H(m)s^{-1} = 147 \times 169 = 117 \mod q = 317 \\ \text{and } u_2 &= rs^{-1} = 51 \times 169 = 60 \mod q = 317. \end{aligned}$$

Thus the signature $(r, s) = (51, 302)$ verifies as valid because $r = 51$ equals

$$\begin{aligned} (g^{u_1} y^{u_2} \bmod p) \bmod q &= (509737^{117} 46932^{60} \bmod p) \bmod q \\ &= (146344 \times 149944 \bmod p) \bmod q \\ &= 246360 = 51 \mod q = 317. \end{aligned}$$

□

The security of DSA depends on the difficulty finding discrete logarithms in $\mathbb{F}_p^*$, for which the best index calculus methods give a complexity function of $L_p(1/3, 1.932)$. The DSA standard requirement that p is an L-bit prime for some $L \geq 1024$ makes finding a discrete logarithm within the $\mathbb{F}_p^*$ structure using index calculus infeasible.

The DSA can also be regarded as being constructed with the subgroup $\langle g \rangle$ of $\mathbb{F}_p^*$ generated by the base element g of order q. The security of DSA would therefore be compromised if it was possible to solve a discrete logarithm problem in a generic cyclic group (ignoring specific $\mathbb{F}_p^*$ structure) of order q. However, the best discrete logarithm algorithms for a generic cyclic group, such as the Pollard Rho method, have a square root $q^{\frac{1}{2}}$ complexity. The DSA standard requirement that q is an N-bit prime for some $N \geq 160$ means than finding a discrete logarithm within this arbitrary cyclic group of order $q \approx 2^N$ would have a time complexity of about $2^{\frac{1}{2}N} \geq 2^{80}$ operations, which is currently reckoned to be infeasible.

In summary, the DSA can be thought of as variant of an El Gamal signature scheme which produces a shorter signature than an equivalent El Gamal signature for the same perceived security. The standardised parameter sizes of an L-bit prime p for $L \geq 1024$ (and more usually $L \geq 2048$) and N-bit prime q for $N \geq 160$ (with q dividing $p - 1$) are chosen so finding discrete logarithms either in $\mathbb{F}_p^*$ or in a cyclic subgroup of $\mathbb{F}_p^*$ of order q is infeasible.

7 Introduction to Lattice-based Cryptosystems

Lattices play a fundamental role in modern cryptology. Lattice-based cryptosystems seem to be resistant to quantum computing, unlike RSA and basic El Gamal. Furthermore, lattice-based cryptosystems can potentially provide homomorphic encryption, that is to say a cryptosystem that allows for processing of ciphertexts with decrypting them. Lattice reduction is also a fundamental technique in many areas of cryptanalysis. This Section provides the mathematical background for the lattices and lattice problems underlying lattice-based cryptography.

7.1 Lattices, Bases and Determinants

A lattice in cryptography is generally a discrete algebraic substructure on the n-dimensional real vector space $\mathbb{R}^n$, and we write vectors in $\mathbb{R}^n$ as row vectors when considering lattices. We first recap some of the basic definitions for vector spaces. The *span* of a set of vectors $\{v_1, \ldots, v_m\}$ is the set of all linear combinations $\sum_{i=1}^{m} \lambda_i v_i$, where $\lambda_i \in \mathbb{R}$, and that a set of vectors $\{v_1, \ldots, v_m\}$ is *linearly independent* if $\sum_{i=1}^{m} \lambda_i v_i = 0$ has only the solution $\lambda_1 = \cdots = \lambda_m = 0$. An informal definition of a linearly independent set of vectors is that we cannot express any one of them linearly in terms of the others. A set of vectors $\{v_1, \ldots, v_m\}$ is a *basis* for $\mathbb{R}^n$ if the vectors are linearly independent and their span is the whole of $\mathbb{R}^n$. The *inner product* of two vectors $v, w \in \mathbb{R}^n$ is $\langle v, w \rangle = \sum_{i=1}^{n} v_i w_i$, and the *norm* (or length) of a vector $v \in \mathbb{R}^n$ is $\|v\| = \sqrt{\langle v, v \rangle}$.

A lattice is a specialisation of a vector space where only *integer* linear combinations of basis vectors are considered. As for vector spaces, a lattice does not have a unique basis, and there are infinitely many choices for the basis matrix of a lattice.

Definition. The *lattice* $\mathcal{L}$ generated by the linearly independent vectors $b_1, \ldots, b_n \in \mathbb{R}^n$ is

$$\mathcal{L} = \left\{ \sum_{i=1}^{n} \lambda_i b_i \;\middle|\; \lambda_1, \ldots, \lambda_n \in \mathbb{Z} \right\}.$$

A *lattice basis matrix* $B = \begin{pmatrix} \overline{\quad b_1 \quad} \\ \overline{\quad \vdots \quad} \\ \overline{\quad b_n \quad} \end{pmatrix}$ for the lattice $\mathcal{L}$ is the $n \times n$ matrix formed by taking the rows to be the basis vectors $b_1, \ldots, b_n$, when

$$\mathcal{L} = \{zB | z \in \mathbb{Z}^n\}.$$

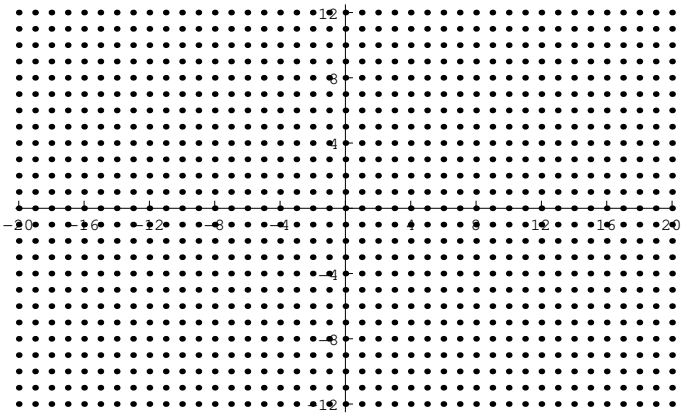

Figure 18: The grid $\mathbb{Z}^2$ generated by the "standard basis" $\{(1,0),(0,1)\}$, and for example other bases $\{(7,11),(5,8)\}$ and $\{(46,31),(43,29)\}$.

Example. The lattice $\mathcal{L}$ in $\mathbb{R}^2$ generated by $\{(0,1),(1,0)\}$ is the integer grid $\mathbb{Z}^2$ with basis matrix $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, and is illustrated in Figure 18. However,

$$(1,0) = (8,-11)\begin{pmatrix} 7 & 11 \\ 5 & 8 \end{pmatrix} \quad \text{and} \quad (0,1) = (-5,7)\begin{pmatrix} 7 & 11 \\ 5 & 8 \end{pmatrix},$$

so an element in $\mathcal{L} \in \mathbb{Z}^2$ can also be expressed as an integer linear combination of $(7,11)$ and $(5,8)$ in the form

$$(z_1,z_2) = (8z_1 - 5z_2)(7,11) + (-11z_1 + 7z_2)(5,8) \quad [z_1,z_2 \in \mathbb{Z}].$$

Thus $\{(7,11),(5,8)\}$ is also a basis for $\mathcal{L} = \mathbb{Z}^2$. Similarly $\{(46,31),(43,29)\}$ is also a basis for $\mathcal{L} = \mathbb{Z}^2$ as

$$(z_1,z_2) = (29z_1 - 43z_2) \times (46,31) + (-31z_1 + 46z_2) \times (43,29) \quad [z_1,z_2 \in \mathbb{Z}]$$

and so on. The matrices $\begin{pmatrix} 7 & 11 \\ 5 & 8 \end{pmatrix}$ and $\begin{pmatrix} 46 & 31 \\ 43 & 29 \end{pmatrix}$ are therefore also basis matrices for $\mathcal{L} = \mathbb{Z}^2$. □

Example. The lattice $\mathcal{L}$ in $\mathbb{R}^2$ generated by the vectors $(47,31)$ and $(43,29)$ is illustrated in Figure 19. We note that the lattice of Figure 18 could be generated by $(46,31)$ and $(43,29)$ so we can obtain very different lattice even though only one component of one basis vector has been altered by 1. We also note that we can write

$$(4,2) = (47,31) - (43,29)(1,-7) = 11 \times (47,31) - 12 \times (43,29)$$

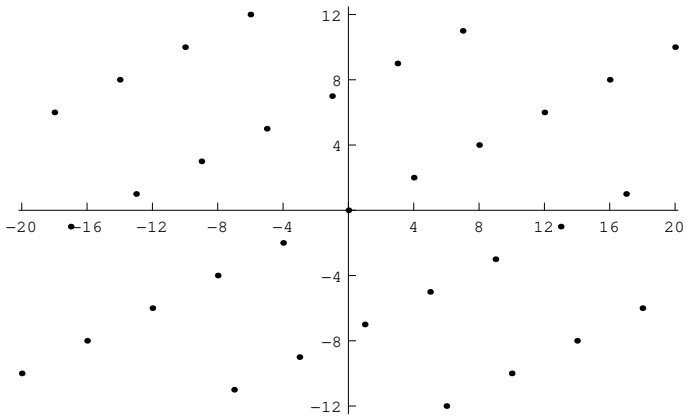

Figure 19: The lattice generated by the basis $\{(47, 31), (43, 29)\}$ and also by the basis $\{(4, 2), (1, -7)\}$.

so $(4, 2)$ and $(1, -7)$ are also in the lattice $\mathcal{L}$. Thus the lattice $\mathcal{L}$ is also generated by the shorter vectors $(4, 2)$ and $(1, -7)$. □

Example. The 3-dimensional lattice generated by the vectors $(-3, -2, 3)$, $(-4, 1, -1)$ and $(-2, 3, 3)$ is illustrated in Figure 20. □

We have seen that a lattice can have many different basis matrices. The Lemma below shows how two such basis matrices are related.

Lemma. Suppose that B and B' are two $n \times n$ basis matrices for the same lattice $\mathcal{L}$, then there exists an $n \times n$ integer matrix U with determinant $\det(U) = \pm 1$ (a *unimodular* matrix) such that $B' = UB$. □

Proof. Every row b'_i of B' is an integer linear combination $b'_i = \sum_{j=1}^{n} u_{ij} b_j$ of the rows of B, where $u_{ij} \in \mathbb{Z}$. Thus we can write $B' = UB$, where $U = (u_{ij})$ is an $n \times n$ integer matrix. Similarly, there exists an $n \times n$ integer matrix U' such that $B = U'B' = (U'U)B$. However, B is invertible since its rows are a basis for $\mathbb{R}^n$, and so $U'U = I$ (the identity matrix). Thus $\det U \cdot \det U' = 1$, but U and U' are integer matrices, so $\det U = \pm 1$ (and $\det U' = \pm 1$). □

Corollary. Any two basis matrices B and B' for the same lattice $\mathcal{L}$ have the same determinant up to sign, that is $|\det B| = |\det B'|$. □

The absolute value of the determinant of any basis matrix of $\mathcal{L}$, known as the lattice volume, is therefore an invariant of the lattice $\mathcal{L}$, giving the definition below. The volume gives an indication of the typical distance between "neighbouring" lattice points in many circumstances.

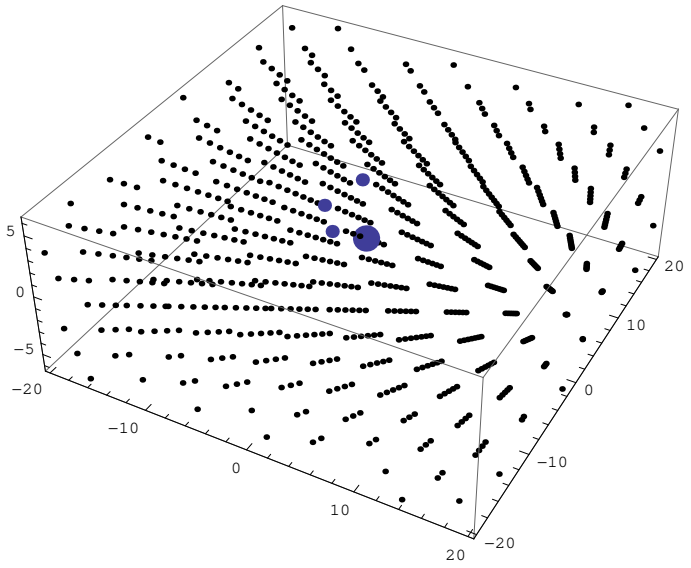

Figure 20: The lattice in $\mathbb{R}^3$ generated by the vectors $(-3,-2,3)$, $(-4,1,-1)$ and $(-2,3,3)$. The big dot is the origin and the three small dots are these three basis vectors.

Definition. The *determinant* $\det(\mathcal{L})$ or *volume* $\mathrm{Vol}(\mathcal{L})$ of a lattice $\mathcal{L}$ with basis matrix B is

$$\det(\mathcal{L}) = \mathrm{Vol}(\mathcal{L}) = |\det(B)|.$$

The *fundamental parallelepided* of the lattice basis matrix B is $([0,1)^n)\,B$ and gives a "repeating pattern" for the lattice, with volume given by the volume $\mathrm{Vol}(\mathcal{L})$ of the lattice $\mathcal{L}$. □

Example. The integer grid $\mathbb{Z}^2$ generated by $\{(1,0),(0,1)\}$ has volume

$$\mathrm{Vol}(\mathbb{Z}^2) = \left|\det\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\right| = \left|\det\begin{pmatrix} 7 & 11 \\ 5 & 8 \end{pmatrix}\right| = \left|\det\begin{pmatrix} 46 & 31 \\ 43 & 29 \end{pmatrix}\right| = 1,$$

and the square $\{(0,0),(1,0),(0,1),(1,1)\}$ as fundamental parallelepiped. □

Example. The 2-dimensional lattice $\mathcal{L}$ generated by $\{(47,31),(43,29)\}$ and $\{(4,2),(1,-7)\}$ of Figure 19 has volume

$$\mathrm{Vol}(\mathcal{L}) = \left|\det\begin{pmatrix} 46 & 31 \\ 43 & 29 \end{pmatrix}\right| = \left|\det\begin{pmatrix} 4 & 2 \\ 1 & -7 \end{pmatrix}\right| = 30,$$

and the parallelogram with vertices $\{(0,0),(4,2),(1,-7),(5,-5)\}$ as fundamental parallelepiped, as illustrated in Figure 21. □

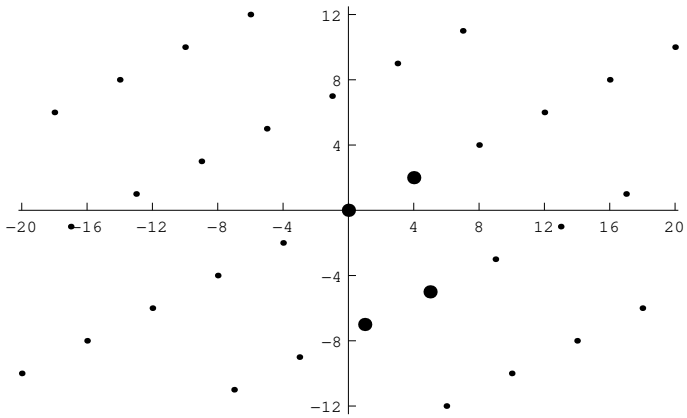

Figure 21: The fundamental parallelepiped of the lattice generated by $\{(4,2),(1,-7)\}$.

Example. The lattice $\mathcal{L}$ generated by the vectors $(-3,-2,3)$, $(-4,1,-1)$ and $(-2,3,3)$ of Figure 20 has $\mathrm{Vol}(\mathcal{L}) = \left|\det \begin{pmatrix} 3 & -2 & 3 \\ -4 & 1 & -1 \\ -2 & 3 & 3 \end{pmatrix}\right| = 76.$ □

Some of the bases for a lattice we have been discussing are generally better for computations about that lattice than other bases. Loosely speaking, a "nice" lattice basis is one consisting of short and near-orthogonal vectors, so $\{(1,0),(0,1)\}$ is a much "nicer" basis for $\mathbb{Z}^2$ than $\{(46,31),(43,29)\}$. More generally, the lattice determinant or volume for a lattice $\mathcal{L}$ with basis vectors basis vectors $b_1,\ldots,b_n$ satisfies

$$\mathrm{Vol}(\mathcal{L}) \leq \prod_{i=1}^{n} \|b_i\|.$$

This bound is an equality if the basis vectors $b_1,\ldots,b_n$ are orthogonal. More generally, this bound is tight if these basis vectors $b_1,\ldots,b_n$ are "close" to orthogonal.

7.2 Computational Problems in Lattices

There are several natural computational problems relating to a lattice $\mathcal{L}$ in $\mathbb{R}^n$, and we now specify the two most important lattice problems.

Shortest Vector Problem (SVP). Given a basis matrix B for a lattice $\mathcal{L}$, compute a non-zero vector $v \in \mathcal{L}$ such that $\|v\|$ is minimal.

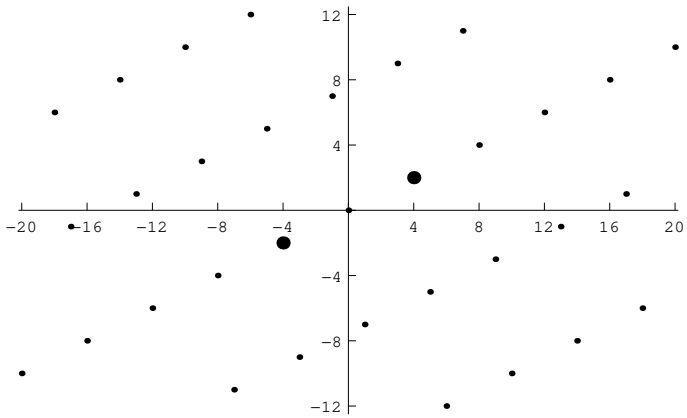

Figure 22: The shortest nonzero vectors $(4, 2)$ and $(-4, -2)$ in the lattice generated by $(47, 31)$ and $(43, 29)$.

Closest Vector Problem (CVP). Given a basis matrix B for $\mathcal{L}$ and a vector $w \in \mathbb{R}^n$, compute $v \in \mathcal{L}$ such that $\|w - v\|$ is minimal.

The solution to a Shortest Vector Problem (SVP) or Closest Vector Problem (CVP) in a lattice is not necessarily unique. However, these computational problems are in general hard, and in particular both CVP and SVP are NP-hard under certain assumptions. Furthermore, these lattice problems do not seem susceptible to Quantum Computer attacks, unlike factoring and discrete logarithm cryptosystems.

Whilst the Shortest Vector Problem (SVP) or Closest Vector Problem (CVP) in a lattice are in general hard problems, the difficulty of SVP and CVP depends on the nature of the basis used to specify the lattice. The following two examples illustrate that SVP and CVP can be starightforward with a "nice" lattice basis and far more difficult with a differenct lattive basis, even in the same lattice.

Example. We consider the lattice $\mathcal{L}$ generated by $v_1 = (47, 31)$ and $v_2 = (43, 29)$, with target vector $(-11, 4)$, illustrated in Figure 23, We can find a shorter "better" basis by subtracting integer multiples of one basis vector from another, which potentially gives shorter lattice vectors. This is a sequence of elementary row operations restricted to integer multiples to give the better basis. In this case, we can define $v_3 = -v_1 + v_2 = (-4, -2)$ and $v_4 = v_2 + 11v_3 = (-1, 7)$ to give the succession of basis matrices

$$\begin{pmatrix} 47 & 31 \\ 43 & 29 \end{pmatrix} \longrightarrow \begin{pmatrix} -4 & -2 \\ 43 & 29 \end{pmatrix} \longrightarrow \begin{pmatrix} -4 & -2 \\ -1 & 7 \end{pmatrix}.$$

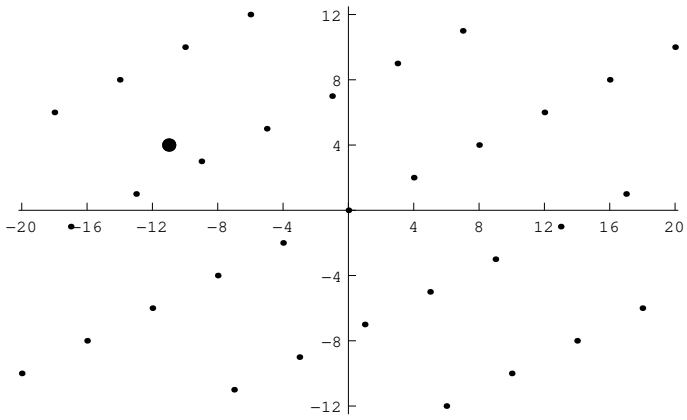

Figure 23: The target vector $(-11, 4)$ and the lattice $\mathcal{L}$ generated by $(47, 31)$ and $(43, 29)$

This gives the better basis $(-4, -2)$ and $(-1, 7)$, as given in a previous example. Thus $(-4, -2)$ and $(4, 2)$ are the shortest vectors in $\mathcal{L}$. Figure 23 shows that $(-11, 4)$ is the closest lattice vector to target vector $(-9, 3)$. □

Example. Let $\mathcal{L}$ be the lattice with basis matrix $B = \begin{pmatrix} 1001 & 0 \\ 0 & 2016 \end{pmatrix}$. Clearly, every lattice vector is of the form $(1001a, 2016b)$ where $a, b \in \mathbb{Z}$. Thus the shortest non-zero vectors are $(1001, 0)$ and $(-1001, 0)$. Similarly, the closest vector to $(5432, 6000)$ is clearly $(5005, 6048)$. This example was easy because the given basis is orthogonal. However, an equivalent basis may be obtained via, for example, by pre-multiplying the basis matrix B by the unimodular matrix $U = \begin{pmatrix} 1847 & 757 \\ 893 & 366 \end{pmatrix}$ to give

$$B' = UB = \begin{pmatrix} 1848847 & 1526112 \\ 893893 & 737856 \end{pmatrix}.$$

This new basis is far from orthogonal, and so it would be more difficult to solve the problems using this basis. □

7.3 Solving the Closest Vector Problem (CVP) by Rounding

The best lattice basis to use for approaching a Shortest Vector Problem (SVP) or Closest Vector Problem (CVP) is an orthogonal basis $\{v_1, \ldots, v_n\}$ for $\mathcal{L}$ if one exists. For CVP, it is easy to decompose a target vector w in

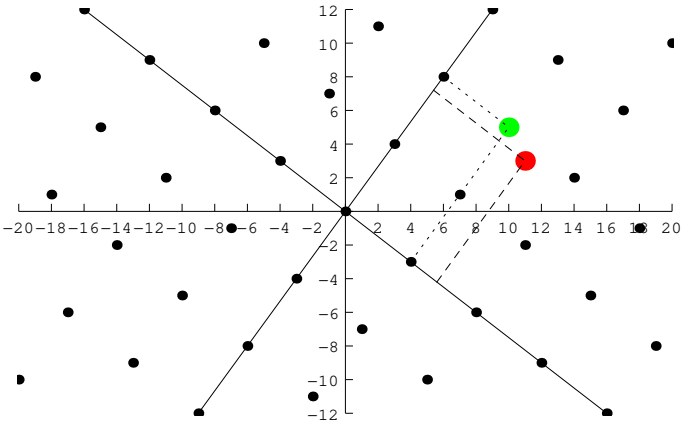

Figure 24: Orthogonal Lattice generated by $(3,4)$ and $(-4,3)$, with target vector $(11,3)$ (red dot) and closest lattice vector $(10,5)$ (green dot)

this case over this orthogonal basis $\{v_1, \ldots, v_n\}$ to obtain $w = \sum_{i=1}^{n} \frac{\langle w, v_i \rangle}{\langle v_i, v_i \rangle} v_i$. Rounding these scalar multiples $\frac{\langle w, v_i \rangle}{\langle v_i, v_i \rangle}$ to the nearest integer $\left[\frac{\langle w, v_i \rangle}{\langle v_i, v_i \rangle} \right]$ to give the closest lattice vector to the target vector w as

$$\sum_{i=1}^{n} \left[\frac{\langle w, v_i \rangle}{\langle v_i, v_i \rangle} \right] v_i.$$

Example. Consider the lattice $\mathcal{L}$ generated by the orthogonal basis vectors $v_1 = (3,4)$ and $v_2 = (-4,3)$ of volume 25, as shown in Figure 24. The target vector $w = (11,3)$ (red dot) can be expressed as

$$\begin{aligned} w = (11,3) &= \frac{\langle w, v_1 \rangle}{\langle v_1, v_1 \rangle} v_1 + \frac{\langle w, v_2 \rangle}{\langle v_2, v_2 \rangle} v_2 \\ &= \frac{\langle (11,3), (3,4) \rangle}{|(3,4)|^2} (3,4) + \frac{\langle (11,3), (-4,3) \rangle}{|(-4,3)|^2} (-4,3) \\ &= \frac{45}{25} (3,4) - \frac{35}{25} (-4,3) = 1.8\, v_1 - 1.4\, v_2 \end{aligned}$$

Thus we can compute the nearest lattice vector $u \in \mathcal{L}$ to the target vector $w(11,3)$ by rounding the scalar multiples of v_1 and v_2 to the nearest integer to obtain

$$u = [1.8]\, v_1 - [1.4]\, v_2 = 2\, v_1 - v_2 = 2\, (3,4) - (-4,3) = (10,5).$$

This rounding process is illustrated by the dashed and dotted lines in Figure 24 to give the nearest lattice vector $u = (10, 5)$, a dstance $\sqrt{5}$ from the target vector $w = (11, 3)$. □

Babai's rounding technique is the generalisation of the method given in the previous from an orthogonal basis for a lattice $\mathcal{L}$ to a general basis. However, Babai's method works best when a lattice $\mathcal{L}$ has a basis consisting of vectors which are orthogonal or close to orthogonal. Given a general lattice basis matrix for a lattice $\mathcal{L}$ with rows $b_1, \ldots, b_n$, we can express any vector $w \in \mathbb{R}^n$ as a unique linear combination

$$w = \sum_{i=1}^{n} \lambda_i b_i, \qquad \text{where } \lambda_1, \ldots \lambda_n \in \mathbb{R},$$

and where we can compute $\lambda_1, \ldots, \lambda_n$ by computing wB^{-1} or by solving a system of linear equations. The rounding technique is simply to set

$$v = \sum_{i=1}^{n} [\lambda_i] b_i,$$

where $[\lambda_i]$ means take the closest integer to the real number λ_i. Thus $v \in \mathcal{L}$ is a lattice vector by construction.

Babai's result is that this method does find the closest vector to w if the basis is sufficiently close to orthogonal and if the vector w is sufficiently close to a lattice point v. More precisely, if the lattice basis is a particularly nice "LLL-reduced" basis (see later), then the method gives a lattice vector within distance $1 + 2n(9/2)^{n/2}$ of the target vector.

Example. We consider the lattice $\mathcal{L}$ shown in Figure 23, and we look at the CVP for the given target vector $(-11, 4)$ with the basis $B_1\{(47, 31), (43, 29)\}$ or the basis $B_2 = \{(4, 2), (1, -7)\}$.

- **Basis $B_1 = \{(47, 31), (43, 29)\}$ for lattice $\mathcal{L}$.**
  Target Vector $(-11, 4) = -16.37(47, 31) + 17.63(43, 29)$.
  Babai rounding gives $-16(47, 31) + 18(43, 29) = (22, 26)$.
  Babai rounding does not solve CVP with $(-11, 4)$ in $\mathcal{L}$ with basis B_1.

- **Basis $\{(4, 2), (1, -7)\}$ for lattice $\mathcal{L}$.**
  Target Vector $(-11, 4) = -2.43(4, 2) - 1.27(1, -7)$.
  Babai rounding gives $-2(4, 2) - 1(1, -7) = (-9, 3)$.
  Babai rounding does solve CVP with $(-11, 4)$ in $\mathcal{L}$ with basis B_2.

Example. Consider the lattice $\mathcal{L}$ in $\mathbb{R}^3$ with basis vectors $b_1 = (7, 0, 1)$, $b_2 = (1, 17, 1)$ and $b_3 = (-3, 0, 10)$, and suppose we wish to find the closest vector in $\mathcal{L}$ to the target vector $w = (100, 205, 305)$. We can write

$$w = (100, 205, 305) \approx 24.09b_1 + 12.06b_2 + 26.89b_3.$$

Babai rounding gives

$$v = [24.09]b_1 + [12.06]b_2 + [26.89]b_3 = 24b_1 + 12b_2 + 27b_3 = (99, 204, 306) \in \mathcal{L}$$

as the nearest lattice vector to the target vector $w = (100, 205, 305)$. The difference between this closest lattice vector v and the target vector w is

$$w - v = (1, 1, -1), \qquad \text{with } |v - w| = \sqrt{3} = 1.7.$$

Thus the lattice vector v very close to the target vector w, For comparison the lattice $\mathcal{L}$ has volume $\text{Vol}(\mathcal{L}) = 1241 = 10.7^3$. □

7.4 Solving the Closest Vector Problem using Embedding

The embedding technique for solving the Closest Vector Problem (CVP) in a lattice $\mathcal{L}$ for target vector w is to "embed" both $\mathcal{L}$ and w in a higher dimensional lattice $\mathcal{L}'$ in such a way that a solution for the Shortest Vector Problem (SVP) in $\mathcal{L}'$ gives a solution to this Closest Vector Problem (CVP) with target vector w in the original lattice $\mathcal{L}$.

Suppose that the original n-dimensional lattice $\mathcal{L}$ has $n \times n$ basis matrix B with rows $b_1, \ldots, b_n$ and that the target vector $w \in \mathbb{R}^n$. A solution to this Closest Vector Problem (CVP) corresponds to finding integers $\lambda_1, \ldots, \lambda_n$ such that

$$w \approx \sum_{i=1}^{n} \lambda_i b_i.$$

In other words, if we define the "error" in this Closest Vector Problem solution to be $e = w - \sum_{i=1}^{n} \lambda_i b_i$, then $\|e\|$ should be small. The embedding technique constructs a higher dimensional lattice $\mathcal{L}'$ which essentially contains this short vector e. However, we note for emphasis that this short vector e is not a lattice vector in the original lattice $\mathcal{L}$.

The new $(n+1)$-dimensional "embedding" lattice $\mathcal{L}'$ has $(n+1) \times (n+1)$ basis matrix B' given in block form as $B' = \left(\begin{array}{c|c} B & 0 \\ \hline w & 1 \end{array}\right)$, so $\mathcal{L}'$ has basis vectors

$$(b_1, 0),\ \cdots,\ (b_n, 0),\ (w, 1).$$

In particular, we can see that

$$(-\lambda_1, \ldots, -\lambda_n, 1)B' = \left(-\sum_{i=1}^{n} \lambda_i b_i + w, 1 \right) = (e, 1) \in \mathcal{L}'$$

is a vector in this new embedding lattice $\mathcal{L}'$. If this lattice vector $(e, 1)$ is shorter than any other lattice vector in $\mathcal{L}'$, then we may be able to find $(e, 1)$ by solving the Shortest Vector Problem (SVP) in $\mathcal{L}'$. We can then solve the original Closest Vector Problem in the original lattice $\mathcal{L}$ by subtracting e from w to obtain $v = w - e \in \mathcal{L}$ as the closest lattice vector in $\mathcal{L}$ to the target vector w.

Example. We return to the example of finding the closest vector to the target vector $(-11, 4)$ in the 2-dimensional lattice $\mathcal{L}$ generated by $(47, 31)$ and $(43, 29)$, which is illustrated in Figure 23. We use the short basis $(4, 2)$ and $(1, -7)$ for the lattice $\mathcal{L}$ to find the closest vector to the target vector $w = (-11, 4)$. The embedding technique uses an embedding lattice $\mathcal{L}'$ with basis matrix

$$\begin{pmatrix} 4 & 2 & 0 \\ 1 & -7 & 0 \\ -11 & 4 & 1 \end{pmatrix}, \quad \text{which reduces to} \quad \begin{pmatrix} -2 & 1 & 1 \\ 2 & 3 & 1 \\ -1 & -2 & 3 \end{pmatrix}$$

using elementary row operations or the more sophisticated techniques. The shortest vector in the embedding lattice $\mathcal{L}'$ is $(-2, 1, 1)$ (with final component 1) giving $e = (-2, 1)$. We can therefore recover

$$v = w - e = (-11, 4) - (-2, 1) = (-9, 3) \in \mathcal{L}$$

as the closest lattice vector to the target vector $w = (-11, 4)$. □

We can show that this embedding method works if the size of the "error" vector e is sufficiently small compared with the vectors in the lattice $\mathcal{L}$. The difference between this embedding method and Babai's rounding method for addressing the Closest Vector Problem is that Babai's method uses a fixed nice basis, whereas the embedding technique involves solving a new Shortest Vector Problem each time. Hence the Babai method is more efficient as lattice reduction needs to be performed only once, but the embedding technique can be useful when there is no very nice basis available.

7.5 Lattice Reduction and the LLL Algorithm

In the Closest Vector Problem example above, the closest vector was found by "reducing" the given basis matrix to a "nice" basis matrix. The process

of lattice reduction is the development and use of an algorithm to transform a given lattice basis into a "nice" lattice basis consisting of vectors which are short and close to orthogonal, and this nice lattice basis can be used for computations about the lattice. The LLL algorithm devised by Lenstra, Lenstra and Lovász in 1982 achieves this, and influenced the development of later algorithms. The LLL algorithm has become one of the most fundamental and important algorithms in mathematics. The LLL algorithm generalises Euclid's algorithm and is closely related to the Gram–Schmidt orthogonalisation method, and we discuss the LLL algorithm in the next Section.

The Gram–Schmidt process is very useful for a vector space, but it is no good for a lattice $\mathcal{L}$ even if an orthogonal basis exists for $\mathcal{L}$ as the Gram-Schmidt coefficients μ_{ij} are not required to be integers. Thus the linear combinations involved are not necessarily elements of the lattice $\mathcal{L}$. The LLL algorithm is an algorithm which uses some ideas of the Gram–Schmidt algorithm, and in fact requires that a Gram–Schmidt basis be maintained throughout the algorithm. The LLL algorithm ensures that the linear combinations used to update the lattice vectors are integer linear combinations of lattice vectors, and so are still in the lattice.

The goal of the LLL algorithm is to produce a basis which is as close to being orthogonal, and for which the vectors are short. Ideally, we would want the first vector b_1 of the basis to be a shortest non-zero vector in the lattice, but SVP does not in general have a polynomial time solution. The method of the LLL algorithm is that it slightly relaxes the requirements on the length of vectors so that it runs in polynomial time, but the LLL algorithm outputs a short vector but not necessarily the shortest vector. The LLL algorithm outputs an LLL-reduced basis as specified below. The first condition in this definition ensures that output basis vectors are not too far from orthogonal, and the second Lovász condition is satisfied if the output basis vectors are close to orthogonal or are roughly ordered by length.

Definition. Let $\{b_1, \ldots, b_n\}$ be a (ordered) basis for a lattice, and let $\{b_1^*, \ldots, b_n^*\}$ be the corresponding Gram–Schmidt orthogonal basis, with Gram–Schmidt coefficients $\mu_{ij} = \dfrac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle}$ $(1 \leq j < i \leq n)$. The (ordered) basis $\{b_1, \ldots, b_n\}$ is *LLL-reduced* if the following two conditions hold.

(i) $|\mu_{ij}| \leq \frac{1}{2}$ for $1 \leq j < i \leq n$.

(ii) (Lovász condition) $\|b_i^*\|^2 \geq \left(\frac{3}{4} - \mu_{i,i-1}^2\right)\|b_{i-1}^*\|^2$ for $2 \leq i \leq n$.

Example. The lattice basis $\{b_1, b_2\}$ with $b_1 = (4, 2)$ and $b_2 = (1, -7)$ illustrated in Figure 19 has corresponding Gram–Schmidt non-lattice orthogonal

basis $\{b_1^*, b_2^*\}$ with $b_1^* = (4, 2)$ and $b_2^* = (3, -6)$, so $\|b_1^*\|^2 = 20$, $\|b_2^*\| = 45$. and $\mu_{2,1} = -\frac{1}{2}$. Thus both conditions are satisfied, and so $\{(4, 2), (1, -7)\}$ is an LLL-reduced lattice basis. □

Example. The lattice basis $\{b_1, b_2\}$ with $b_1 = (47, 31)$ and $b_2 = (43, 29)$ is also a basis for the lattice of the previous Example illustrated in Figure 19. This lattice basis $\{b_1, b_2\}$ has corresponding Gram–Schmidt non-lattice orthogonal basis $\{b_1^*, b_2^*\}$ with $b_1^* = (47, 31)$ and $b_2^* = (-\frac{93}{317}, \frac{141}{317})$. Thus $\mu_{2,1} = \frac{292}{317} > \frac{1}{2}$, and so $\{(47, 31), (43, 29)\}$ is not an LLL-reduced lattice basis. □

The first basis vector b_1 output by the LLL algorithm is guaranteed to satisfy $\|b_1\| \leq 2^{(n-1)/2}\|v\|$, where v is a shortest vector in the lattice. For any 3-dimensional example such as that above, this gives $\|b_1\| \leq 2\|v\|$, that is to say the shortest LLL-reduced basis vector is at most double the length of a shortest nonzero vector. More generally, the LLL algorithm allows us to solve SVP up to an exponential multiplicative constant, which is not enough to solve the SVP in a general lattice. However, the vector b_1 output by the LLL algorithm is in practice much closer to the shortest vector than this bound would suggest, and in most cases will output a shortest vector in the lattice.

We remarked earlier that the LLL algorithm had polynomial time complexity. In more detail, if a lattice $\mathcal{L}$ of dimension n has a basis $b_1, \ldots, b_n$ such that $\|b_i\| \leq M$ for some bound M, then the running time of the LLL algorithm is

$$O\left(n^6 (\log M)^3\right),$$

which is polynomial time. In practice, this complexity estimate is in most cases rather pessimistic, and the LLL algorithm usually runs more quickly.

The LLL algorithm is implemented in Mathematica by `LatticeReduce`.

`B` $= \{\{426, -84, 390\}, \{25, -270, -299\}, \{1225, 284, 1767\}\}$
`LatticeReduce[B]`
$\{\{-3, -4, -1\}, \{175, -70, -249\}, \{273, -288, 339\}\}$
Lattice with basis matrix B has a short vector $(-3, -4, -1)$ found by LLL.

7.A Appendix. LLL Algorithm

Details of the LLL Algorithm are included in these Notes for completeness. However the specific details of the working of the LLL algorithm are not examinable.

7.A1 Gram–Schmidt Orthogonalisation

The Gram–Schmidt algorithm is a method to transform a vector space basis $\{b_1, \ldots, b_n\}$ for $\mathbb{R}^n$ into an orthogonal basis $\{b_1^*, \ldots, b_n^*\}$. The first step is to set $b_1^* = b_1$. We now want to choose b_2^* so that $\{b_1^*, b_2^*\}$ spans the same subspace as $\{b_1, b_2\}$, but such that $\langle b_1^*, b_2^* \rangle = 0$. We can do this by computing

$$\mu_{2,1} = \frac{\langle b_2, b_1^* \rangle}{\langle b_1^*, b_1^* \rangle}, \quad \text{and then setting} \quad b_2^* = b_2 - \mu_{2,1} b_1^*.$$

The remaining process is no more complicated than this. Suppose we have produced an orthogonal basis $\{b_1^*, \ldots, b_{i-1}^*\}$. We first compute

$$\mu_{ij} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle} \quad [1 \leq j < i], \quad \text{and then set} \quad b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^*.$$

We can prove by induction that this process produces an orthogonal basis.

Example. Perform the Gram–Schmidt orthogonalisation process on the vectors

$$b_1 = (1, 0, 1), \quad b_2 = (2, 3, 4) \quad \text{and} \quad b_3 = (1, 4, 0).$$

We first set $b_1^* = b_1 = (1, 0, 1)$. We next compute

$$\mu_{2,1} = \frac{\langle b_2, b_1^* \rangle}{\langle b_1^*, b_1^* \rangle} = \frac{6}{2} = 3, \quad \text{and set} \quad b_2^* = b_2 - 3b_1 = (-1, 3, 1).$$

We next compute $\mu_{3,1} = \dfrac{\langle b_3, b_1^* \rangle}{\langle b_1^*, b_1^* \rangle} = \dfrac{1}{2}$ and $\mu_{3,2} = \dfrac{\langle b_3, b_2^* \rangle}{\langle b_2^*, b_2^* \rangle} = \dfrac{11}{11} = 1$, so

$$b_3^* = b_3 - \tfrac{1}{2} b_1^* - b_2^* = (\tfrac{3}{2}, 1, -\tfrac{3}{2}).$$

Thus the orthoganlised basis is $(1, 0, 1)$, $(-1, 3, 1)$ and $(\frac{3}{2}, 1, -\frac{3}{2})$.

7.A2 LLL Algorithm Details

We first note that an LLL-reduced basis exists for any lattice $\mathcal{L}$ An LLL-reduced basis $\{b_1, \ldots, b_n\}$ for a lattice $\mathcal{L}$ has the following properties.

- If v is a shortest nonzero vector in $\mathcal{L}$, then $\|b_1\| \leq 2^{(n-1)/2}\|v\|$.
- If $j \leq i$ then $\|b_j\| \leq 2^{(i-1)/2}\|b_i^*\|$.
- $\det(L) \leq \prod_{i=1}^{n} \|b_i\| \leq 2^{n(n-1)/4} \det(L)$.
- $\|b_1\| \leq 2^{(n-1)/4} \det(L)^{1/n}$.

We give a sketch proof of first property based on the facts that

$$2^{(1-i)/2}\|b_1\| \leq \|b_i^*\| \quad \text{and} \quad \|xB\| \geq \min_{1 \leq i \leq n} \|b_i^*\|$$

for all non-zero integer vectors $x \in \mathbb{Z}^n \setminus \{0\}$. We define λ_1 to be the length of a shortest vector in the lattice L, so we want to show that if $\{b_1, \ldots, b_n\}$ is LLL-reduced then $\|b_1\| \leq 2^{(n-1)/2}\lambda_1$. The result follows, since $\lambda_1 = \|xB\|$ for some $x \in \mathbb{Z}^n$ and so

$$\lambda_1 \geq \min_{1 \leq i \leq n} \|b_i^*\| \geq \min_{1 \leq i \leq n} 2^{(1-i)/2}\|b_1\| = 2^{(1-n)/2}\|b_1\|.$$

A corollary to the above result is that if $\|b_1\| \leq \|b_i^*\|$ for all $1 \leq i \leq n$ then b_1 is correct solution to the SVP.

The LLL algorithm works rather like the Gram–Schmidt method in that it iteratively generates a basis such that the two desired conditions are satisfied. The first condition is easily met by taking suitable integer linear combinations of existing lattice vectors. If the second Lovász condition is not met, then b_i is not significantly longer than b_{i-1}, and so we can swap b_i and b_{i-1} and backtrack. Upon termination, the LLL algorithm outputs an LLL-reduced lattice basis construction. The LLL algorithm is given in full in Figure 25, and we now give an example of the LLL algorithm in action.

7.A3 LLL Example

We consider the lattice $\mathcal{L}$ generated by

$$b_1 = (1, 0, 0), \quad b_2 = (4, 2, 15) \quad \text{and} \quad b_3 = (0, 0, 3).$$

We now run the LLL algorithm.

Input: Basis $\{b_1, \ldots, b_n\}$ for the lattice $\mathcal{L}$.
Output: LLL-reduced basis $\{b_1, \ldots, b_n\}$ for the lattice $\mathcal{L}$.

1. Start with $k = 2$ and vectors b_1 and b_2.
2. Set $b_1^* = b_1$ and $B_1 = \langle b_1, b_1 \rangle = \|b_1^*\|^2$.
3. If $k > n$ then halt.
4. (Size reduction) Reduce b_k using b_j for $j = (k-1), \ldots, 1$.
   Set $b_k = b_k - [\mu_{kj} b_j$, where $[\cdot]$ denotes rounding.
   Update the values $\mu_{k,j}$ and b_k^* and compute $B_k = \langle b_k^*, b_k^* \rangle = \|b_k^*\|^2$.
   (It is important to start with b_{k-1} rather than b_1.)
5. If $B_k \geq (3/4 - \mu_{k,k-1}^2) B_{k-1}$, then set $k = k + 1$ and go to Step 3.
6. Otherwise, swap b_k with b_{k-1} and set $k = k - 1$ and go to Step 3.
7. Return basis $\{b_1, \ldots, b_k\}$.

Figure 25: The LLL algorithm.

$k = 1$ **Step**.
Set $b_1 = (1, 0, 0)$ and $B_1 = 1$.

$k = 2$ **Step**.
We first compute $\mu_{2,1} = \frac{4}{1} = 4$, so

$$b_2 = b_2 - [\mu_{2,1}] b_1 = b_2 - 4b_1 = (4, 2, 15) - (4, 0, 0) = (0, 2, 15).$$

We now want to check the Lovász condition. We compute $\mu_{2,1} = 0$, and so $b_2^* = b_2 = (0, 2, 15)$, giving $B_1 = 1$ and $B_2 = \langle b_2^*, b_2^* \rangle = 229$. The Lovász condition is satisfied as we clearly have $B_2 > (3/4 - \mu_{2,1}^2) B_1$.

$k = 3$ **Step**.
We compute $\mu_{3,2} = 45/229 \approx 0.19$, so $[\mu_{3,2}] = 0$ there is no reduction to be performed on b_3. We also compute $\mu_{3,1} = 0$ so no size reduction is required. We now compute the corresponding Gram-Schmidt vector $b_3^* = b_3 - \frac{45}{229} b_2^* = (0, -\frac{90}{229}, \frac{12}{229})$. We have $B_2 = 229$ and $B_3 = \langle b_3^*, b_3^* \rangle = \frac{8244}{52441} \approx 0.157$. From this we can check that $B_3 < (3/4 - \mu_{3,2}^2) B_2 \approx 166.1$, so we swap b_2 and b_3 and set $k = 2$.

$k = 2$ **Step (again)**.
At this point we have the vectors $b_1 = (1, 0, 0)$ and $b_2 = (0, 0, 3)$ with $b_1^* = b_1$ and $b_2^* = b_2$. We first note check that $\mu_{2,1} = 0$ and so no size reduction on b_2 is required. Secondly, $B_1 = 1$ and $B_2 = 9$, so $B_3 > (3/4 - \mu_{2,1}^2) B_1 = 0.75$. Thus we set $k = 3$.

$k = 3$ **Step (again)**.
We have $b_3 = (0, 2, 15)$, and so we compute $\mu_{3,2} = \frac{45}{9} = 5$. Thus we can reduce $b_3 = b_3 - 5b_2 = (0, 2, 0)$, and we compute $\mu_{3,1} = 0$, so no size reduction is required. Furthermore, as $\mu_{3,1} = \mu_{3,2} = 0$, we have $b_3^* = b_3$ and $B_3 = 4$. Thus $B_3 < (3/4 - \mu_{3,2}^2)B_2 = \frac{27}{4} = 6.75$ and so we should swap b_2 and b_3 and set $k = 2$.

$k = 2$ **Step (yet again)**.
We can check that the $k = 2$ phase runs without making any changes. We have $B_1 = 1$ and $B_2 = 4$, so we set $k = 3$ again.

$k = 3$ **Step (yet again)**.
We have $\mu_{3,2} = \mu_{3,1} = 0$ and so b_3 remains unchanged. Finally, $B_3 = 9 > (3/4 - \mu_{3,2}^2)B_2 = 3$ and so we set $k = 4$ and halt.

An LLL-reduced basis for the lattice $\mathcal{L}$ is given bu

$$b_1 = (1, 0, 0), \quad b_2 = (0, 2, 0) \quad \text{and} \quad b_3 = (0, 0, 3).$$

8 Further Lattice-based Cryptosystems

In this Section, we consider the specific cryptosystems whose security depends on the difficult of lattice problems.

8.1 The GGH Cryptosystem

The *Goldreich-Goldwasser-Halevi* or GGH cryptosystem is a cryptosystem based on lattices, whose security relies on the fact that the Closest Vector Problem (CVP) can be a hard problem. The GGH cryptosystem is simple and fast to compute. As usual, we consider Textbook GGH and discuss the encryption version of GGH, omitting the signature version.

GGH Set-Up.
The *private key* is a nice basis matrix B for a lattice $\mathcal{L}$ in $\mathbb{R}^n$ and a "random" unimodular matrix U (so $\det U = \pm 1$). The *public key* is the "not so nice" basis matrix $B' = UB$. The message space is the space of row vectors $(\lambda_1, \ldots, \lambda_n)$ of integers in the range $-M \leq \lambda_i \leq M$ for some suitably chosen constant M. We note a unimodular matrix U can be generated as a product $U = U_1 U_2$ of an upper triangular matrix U_1 and a lower triangular matriix U_2, where the diagonal elements of both U_1 and U_2 are ± 1.

GGH Encryption.
To encrypt a message $m = (\lambda_1, \ldots, \lambda_n)$ using a public key basis matrix B', we first compute the lattice element $v = mB' \in \mathcal{L}$, that is to say if B' has rows $b'_1, \ldots, b'_n$ we compute the sum $v = \sum_{i=1}^{n} \lambda_i b'_i$. We then choose a "small" error vector $e = (e_1, \ldots, e_n)$ either in $\mathbb{R}^n$ or $\mathbb{Z}^n$ and compute the ciphertext c (not generally an elemnt of the lattice $\mathcal{L}$) as

$$c = v + e = mB' + e.$$

GGH Decryption.
To decrypt a ciphertext c with the private key matrix B, we use the Babai rounding technique with respect to the nice basis given by B for the lattice. More precisely, we multiply c by B^{-1} to obtain

$$cB^{-1} = (mB' + e)B^{-1} = mUBB^{-1} + eB^{-1} = mU + eB^{-1}.$$

The rounding method considers the expression

$$[cB^{-1}] = [mU + eB^{-1}],$$

where the rounding $[\cdot]$ is component-wise to the nearest integer. This rounding removes the term eB^{-1} as long as it is small enough, in which case we obtain

$$[cB^{-1}] = [mU] = mU$$

as m is an integer vector and U is an integer matrix. We can then multiply the output of this rounding $[cB^{-1}] = mU$ by U^{-1} to obtain the message m as

$$m = (mU)U^{-1} = [cB^{-1}]U^{-1}.$$

Example. *GGH Set-Up.* We consider the GGH cryptosystem based on the lattice $\mathcal{L}$ with basis matrix $B = \begin{pmatrix} 17 & 0 \\ 0 & 19 \end{pmatrix}$. We let $U = \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix}$ be the unimodular matrix, so the private key is (B, U). The public key is the lattice matrix matrix $B' = UB = \begin{pmatrix} 34 & 57 \\ 51 & 95 \end{pmatrix}$.

GGH Encryption. To encrypt the message $m = (2, -5)$, we take $e = (1, -1)$ as the small error vector to give the ciphertext

$$c = mB' + e = (2, -5)\begin{pmatrix} 34 & 57 \\ 51 & 95 \end{pmatrix} + (1, -1) = (-186, -362).$$

GGH Decryption To decrypt the ciphertext $c = (-186, -362)$, we compute

$$cB^{-1} = (-186, -362)\begin{pmatrix} 17 & 0 \\ 0 & 19 \end{pmatrix}^{-1} = \left(-\frac{186}{17}, -\frac{362}{19}\right) = (-10.94, -19.05).$$

We apply rounding to $cB^{-1} = (-10.94, -19.05)$ to obtain $[cB^{-1}] = (-11, -19)$. We note that $U^{-1} = \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix}^{-1} = \begin{pmatrix} 5 & -3 \\ -3 & 2 \end{pmatrix}$, so the message m can be recovered by

$$m = [cB^{-1}]U^{-1} = (-11, -19)\begin{pmatrix} 5 & -3 \\ -3 & 2 \end{pmatrix} = (2, -5).$$

Example. We show how to implement GGH in Mathematica by giving a 3-dimensional example. To set up a GGH public key, we make the following Mathematica commands.

```
B = {{101,0,0},{0,503,0},{0,0,1007}}
U1 = {{1,2,3},{0,1,-4},{0,0,1}}
```

```
U2 = {{1,0,0},{-5,1,0},{2,-3,1}}
U = U1.U2
Det[U]
1
Binv = Inverse[B]
Uinv = Inverse[U]
BB = U.B
{{ -303, -3521, 3021}, {-1313, 6539, -4028},
  {202,-1509, 1007}}
```

To encrypt the message $(2, -5, 10)$ with small random error $(-2, 1, 2)$, we make the following Mathematica commands.

```
m = {2,-5,10}
c = m.BB + {-2,1,2}
{ 7977, -54826, 36254 }
```

To decrypt the ciphertext $(7977, -54826, 36254)$ with small random error $(-2, 1, 2)$, we make the following Mathematica commands.

```
v = c.Binv
{ 78.980, -108.998, 36.002 }
w = Round[v]
{ 79, -109, 36 }
w.Uinv
{2, -5, 10}
```

8.2 Cryptanalysis of the GGH Cryptosystem

There are two natural ways to attack the GGH cryptosystem. Firstly, we can try to obtain the private key matrix B from the public key matrix B'. Secondly, we can try to solve the Closest Vector Problem (CVP) with target vector c with respect to the lattice $\mathcal{L}$ defined by B'. Lattice reduction algorithms have a role to play in both these attacks.

In the first attack we attempt to obtain the private key matrix B from the public key matrix B', and so we simply run the LLL algorithm on the basis given in the public key matrix B'. As $B' = UB$ for some unimodular matrix U, both B and B' are basis matrices for the same lattice $\mathcal{L}$. In appropriate circumstances, the LLL algorithm outputs the good basis B, or at least a basis B'' which can be used for solving a Closest Vector Problem.

In the second attack, we regard the ciphertext vector c as the target vector in a Closest Vector Problem in the lattice $\mathcal{L}$ with the public key matrix B' as a basis matrix. The GGH encryption equation $c = mB' + e$

shows that solving the Closest Vector Problem for target vector c in the lattice generated by B' should return the lattice vector mB' (if e is a small vector), from which we can find the message m. We can therefore use the embedding technique to attempt to solve this Closest Vector Problem by using the LLL algorithm to attempt to solve this Shortest Vector Problem in the higher dimensional embedding lattice.

Example. We consider an instance of the GGH cryptosystem with public key matrix B' and ciphertext c given by

$$B' = \begin{pmatrix} 7 & 69 & -990 \\ 56 & 575 & -8514 \\ -77 & -644 & 8019 \end{pmatrix} \quad \text{and} \quad c = (-274, -2368, 30592).$$

Regarding the public key matrix B' as the basis matrix for a lattice $\mathcal{L}$, we can run the LLL algorithm on this basis given by B' by using the Mathematica command `LatticeReduce` with the following commands to recover a "nicer" lattice basis matrix B for $\mathcal{L}$, which we regard as the private. key matrix B. The corresponding unimodular matrix U is then given by $U = B'B^{-1}$.

```
B1 = { {7, 69, -990}, {56, 575, -8514}, {-77, -644, 8019} }
LatticeReduce[ B1 ]
{{7,0,0},{0,23,0},{0,0,99}}
```

Thus we obtain the private key matrices

$$B = \begin{pmatrix} 7 & 0 & 0 \\ 0 & 23 & 0 \\ 0 & 0 & 99 \end{pmatrix} \quad \text{and} \quad U = B'B^{-1} = \begin{pmatrix} 1 & 3 & -10 \\ 8 & 25 & -86 \\ -11 & -28 & 81 \end{pmatrix}.$$

We therefore compute

$$\begin{aligned} cB^{-1} &= (-274, -2368, 30592) \begin{pmatrix} 7 & 0 & 0 \\ 0 & 23 & 0 \\ 0 & 0 & 99 \end{pmatrix}^{-1} \\ &= \left(-\frac{274}{7}, -\frac{2368}{23}, \frac{30592}{99} \right) = (-39.12, -102.96, 309.01) \end{aligned}$$

We apply rounding to $cB^{-1} = (-39.12, -102.96, 309.01)$ to obtain the integer vector $[cB^{-1}] = (-39, -103, 309)$ We can then complete the decryption to find the message m as

$$m = [cB^{-1}]U^{-1} = (-39, -103, 309) \begin{pmatrix} -383 & 37 & -8 \\ 298 & -29 & 6 \\ 51 & -5 & 1 \end{pmatrix} = (2, -1, 3). \quad \square$$

Example. We consider the GGH cryptosystem of the previous example with the given public key lattice basis matrix B for the lattice $\mathcal{L}$ and ciphertext c given by

$$B' = \begin{pmatrix} 7 & 69 & -990 \\ 56 & 575 & -8514 \\ -77 & -644 & 8019 \end{pmatrix} \quad \text{and} \quad c = (-274, -2368, 30592).$$

We use the embedding technique to construct the 4-dimensional lattice basis matrix

$$B^* = \left(\begin{array}{rrr|r} 7 & 69 & -990 & 0 \\ 56 & 575 & -8514 & 0 \\ -77 & -644 & 8019 & 0 \\ \hline -274 & -2368 & 30592 & 1 \end{array}\right).$$

for a higher dimensional embedding lattice $\mathcal{L}^*$. Applying the Mathematica command `LatticeReduce` to this basis matrix B^* for $\mathcal{L}^*$ gives the LLL-reduced basis matrix

$$B^{**} = \left(\begin{array}{rrr|r} -1 & 1 & 1 & 1 \\ 5 & 2 & 2 & 2 \\ -1 & -15 & 8 & 8 \\ -1 & -2 & 51 & -48 \end{array}\right)$$

for the embedding lattice $\mathcal{L}^*$. We therefore deduce that the shortest vector in this embedding lattice is $(e|1) = (-1, 1, 1|1)$, and so the error vector is $e = (-1, 1, 1)$. Thus the message m can then be recovered as

$$\begin{aligned} m &= (c - e)B'^{-1} = ((-274, -2368, 30592) - (-1, 1, 1))\ B'^{-1} \\ &= (-273, 2369, 30591) \begin{pmatrix} 7 & 69 & -990 \\ 56 & 575 & -8514 \\ -77 & -644 & 8019 \end{pmatrix}^{-1} = (2, -1, 3). \quad \square \end{aligned}$$

In order to use the GGH cryptosystem, we need to be careful to choose parameters so that these attacks are avoided. The first attack is prevented by making the matrices so large that the LLL algorithm is not effective. The second attack is also prevented by large matrices and also by making sure that the error vectors e used in encryption are not too small.

There are challenge keys and ciphertexts for the GGH cryptosystem on the internet. These challenges start with dimension $n = 200$ and go up to dimension $n = 400$. The public key for the $n = 200$ example requires about 660Kb memory, which is far more than an RSA public of about 1Kb.

8.3 The Learning with Errors (LWE) Problem

The Learning with Errors or LWE Problem is a fundamental problem in modern cryptology. For simplicity, we give an informal "modulo q" formulation of the LWE Problem for a prime q.

Learning with Errors (LWE) Set-Up.
(i) Consider a field $\mathbb{F}_q$ and a secret (column) vector $s \in \mathbb{F}_q^n$ of dimension n.
(ii) Let A be an $m \times n$ matrix over $\mathbb{F}_q$ with elements generated randomly and uniformly from $\mathbb{F}_q$, so $A_{ij} \sim \text{Uni}(\mathbb{F}_q)$.
(iii) Let e be a vector of dimension m with components $e_1, \ldots, e_m \sim \chi$ generated independently according to a distribution χ on $\mathbb{F}_q$ centred on 0 with small "variance". (Note χ is usually a discretised Gaussian distribution.)
(iv) Set $b = As + e$ over $\mathbb{F}_q$.

(Computational) Learning with Errors (LWE) Problem.
Given (A, b) from an LWE Set-up ($b = As + e \bmod q$), find the secret s.

Example. Consider an LWE Problem for a prime $q = 991$ with $m = 16$ "noisy functionals", a secret s of dimension $n = 4$ and error distribution χ a discretised Normal $\text{N}(0, 3.0^2)$ distribution with standard deviation 3.0. For $b = As + e \bmod q$, an instance of the LWE Problem with these parameters is to find $s \in \mathbb{F}_{991}^4$ (in fact secret $s = (357, 966, 927, 60)$) given

$$A = \begin{pmatrix} 205 & 618 & 410 & 672 \\ 948 & 394 & 740 & 309 \\ 981 & 508 & 128 & 912 \\ 340 & 569 & 367 & 266 \\ 99 & 555 & 442 & 969 \\ 199 & 790 & 542 & 572 \\ 774 & 636 & 186 & 174 \\ 385 & 469 & 689 & 229 \\ 745 & 96 & 779 & 11 \\ 531 & 128 & 144 & 158 \\ 939 & 21 & 934 & 552 \\ 741 & 708 & 623 & 32 \\ 360 & 118 & 973 & 917 \\ 835 & 669 & 454 & 459 \\ 30 & 86 & 701 & 394 \\ 101 & 658 & 903 & 379 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 465 \\ 487 \\ 531 \\ 527 \\ 779 \\ 386 \\ 307 \\ 229 \\ 311 \\ 324 \\ 835 \\ 773 \\ 391 \\ 392 \\ 224 \\ 406 \end{pmatrix}.$$

If the rows of the matrix A are $a_1^T, \ldots, a_m^T$, then a component of the LWE Set-up $b = As + e$ essentially gives $b_j = a_j^T s + e_j \bmod q$. Thus the Learning with Errors (LWE) Problem asks us to "learn" the secret s given noisy linear functionals $b_1, \ldots, b_m$ (together with $a_1^T, \ldots, a_m^T$) of s, that is to say we have to learn the secret s in the presence of the errors given by e.

The LWE problem can be thought of as a discrete version (over $\mathbb{F}_q$) of the linear model $Y = As + e$ over the real numbers $\mathbb{R}$ considered in statistics. However, unlike the modulo q LWE Problem, this real-valued linear model version generally has a solution, for example given by taking s to be $(A^T A)^{-1} A^T b$ when this matrix inverse exists.

There is also a decisional variant of the LWE Problem (given below), and it can be shown that Computational-LWE and Decisional-LWE are equivalent problems.

Decisional Learning with Errors (LWE) Problem.
Given (A, b) from either an LWE Set-up or uniformly randomly generated (A, b), decide whether (A, b) is from an LWE set-up or uniformly random.

8.4 The Learning with Errors (LWE) Problem and Lattices

The LWE Problem based on $b = As + e$ over $\mathbb{F}_q$ for $m \times n$ matrix A naturally gives rise to the consideration of the image $\mathrm{Im}(A) = \{\, Ax \mid x \in \mathbb{F}_q^n \,\}$ of the $\mathbb{F}_q$-matrix A. In particular, we consider the related lattice

$$\mathcal{L}_{\mathrm{Im}(A)} = \{\, y \in \mathbb{Z}^m \mid y = Az \bmod q \text{ for some } z \in \mathbb{Z}^n \,\}$$

arising from this finite field vector subspace $\mathrm{Im}(A) \leq \mathbb{F}_q^m$ of dimension n. This type of lattice is an example of a *q-ary* lattice. This subspace $\mathrm{Im}(A)$ has q^n points, so this q-ary lattice has volume

$$\mathrm{Vol}\left(\mathcal{L}_{\mathrm{Im}(A)}\right) = q^{m-n}.$$

For $m \geq n$, this q-ary lattice can be thought of as the finite subspace $\mathrm{Im}(A) \leq \mathbb{F}_q^m$ with q^n points "repeating" with period q over the whole of $\mathbb{Z}^m$. Figure 26 illustrates such a q-ary lattice in $\mathbb{F}_{11}^2$ based on the subspace

$$\mathrm{Im}\left(\begin{pmatrix} 2 \\ 5 \end{pmatrix}\right) = \left\{ \begin{matrix} (0,0), & (1,8), & (2,5), & (3,2), & (4,10), & (5,7), \\ (6,4), & (7,1), & (8,9), & (9,6), & (10,3) \end{matrix} \right\} \leq \mathbb{F}_{11}^2$$

repeating periodically to give the q-ary lattice

$$\mathcal{L}_{\mathrm{Im}((2,5)^T)} = \left\{ \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \;\middle|\; \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 5 \end{pmatrix} (z) \bmod 11 \quad [z \in \mathbb{Z}] \right\}.$$

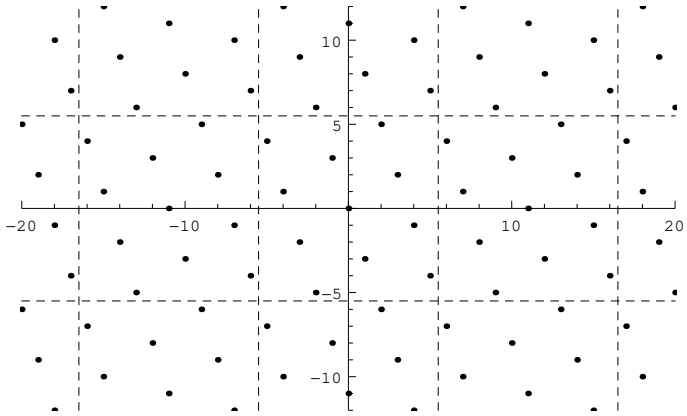

Figure 26: The q-ary lattice $\mathcal{L}_{\mathrm{Im}((2,5)^T)}$ as a "periodic" version of a 1-dimensional subspace $\mathrm{Im}((2,5)^T)$ of 2-dimensional vector space $\mathbb{F}_{11}^2$.

The q-ary lattice $\mathcal{L}_{\mathrm{Im}(A)}$ is generated by the column vectors of the $m \times n$ matrix A modulo q, or equivalently by the row vectors of the $n \times m$ matrix A^T modulo q. Thus we can find an $m \times m$ lattice basis matrix for the q-ary lattice $\mathcal{L}_{\mathrm{Im}(A)}$ by reducing the $(m+n) \times m$ matrix $A_q^T = \begin{pmatrix} A^T \\ \hline qI_m \end{pmatrix}$ to give an $m \times m$ lattice basis matrix $B_{\mathrm{Im}(A)}$ for the q-ary lattice $\mathcal{L}_{\mathrm{Im}(A)}$. Such a reduction of A_q^T to give this q-ary lattice basis matrix $B_{\mathrm{Im}(A)}$ can be found by using the Mathematica command `Lattice Reduce` on the $(m+n) \times m$ matrix A_q^T

Example. The q-ary matrix $\mathcal{L}_{\mathrm{Im}(A)}$ for the 16×4 matrix A modulo $q = 997$ of the Section 8.3 Example has volume $q^{m-n} = 997^{12} \approx 9 \times 10^{35}$. A basis matrix $B_{\mathrm{Im}(A)}$ for this q-ary lattice found by the above method is given by

$$\begin{pmatrix}
69 & -12 & 7 & 108 & 13 & 48 & -74 & 53 & 8 & 76 & 60 & -98 & -7 & -79 & 19 & 14 \\
-6 & 23 & -38 & 23 & -34 & 35 & 43 & -75 & 110 & -31 & 74 & 77 & 28 & 113 & -71 & 18 \\
-66 & 65 & 46 & -39 & 34 & 1 & -29 & 49 & 9 & -8 & 14 & 91 & 38 & -19 & 5 & 152 \\
89 & -77 & -73 & 53 & -105 & -11 & -49 & 33 & -57 & -101 & -74 & -10 & -9 & -51 & -13 & 2 \\
125 & -35 & -47 & -105 & -44 & 39 & -67 & -39 & 57 & -24 & -14 & -48 & 37 & 122 & -29 & 18 \\
33 & 5 & -34 & -51 & -51 & -37 & -45 & 84 & 38 & 44 & 55 & 52 & 23 & 133 & -100 & 75 \\
34 & -60 & 24 & 38 & -95 & -21 & 52 & 24 & 48 & 79 & -27 & -57 & 20 & 21 & -51 & -25 \\
12 & -6 & 15 & -16 & -89 & -84 & 84 & 113 & 60 & -53 & -24 & 16 & -25 & 101 & 100 & -38 \\
41 & 3 & -32 & 15 & 120 & -30 & 51 & -7 & 52 & 43 & -98 & 11 & 23 & -13 & -79 & -53 \\
106 & -144 & -14 & -1 & -7 & -45 & -87 & -43 & -16 & 80 & -117 & -19 & 6 & -38 & 6 & -11 \\
-45 & 66 & 98 & -72 & 27 & 84 & -49 & 74 & 73 & 15 & -115 & -9 & -7 & 66 & 0 & -51 \\
-38 & -4 & 34 & 51 & -112 & -25 & 25 & -11 & 118 & 61 & 130 & 60 & -38 & -14 & 59 & -170 \\
53 & 23 & -9 & 13 & -47 & -24 & 63 & -43 & -39 & 44 & -104 & 49 & -104 & 66 & -34 & 1 \\
78 & 51 & 64 & 5 & 64 & -106 & 73 & 58 & 75 & -65 & 35 & 68 & 93 & 58 & 13 & 18 \\
-124 & 71 & -96 & 56 & -58 & -93 & -84 & 24 & -2 & -34 & 100 & -6 & 39 & -104 & -107 & 23 \\
46 & 49 & -69 & 80 & -75 & 79 & -100 & -42 & 40 & 20 & 13 & 5 & 101 & 17 & 46 & -56
\end{pmatrix}$$

The input (A, b) to an LWE Problem satisfies $b = As + e \bmod q$. Thus we can potentially find the lattice point As in the q-ary lattice $\mathcal{L}_{\mathrm{Im}(A)}$ by solving a Closest Vector Problem with target vector b in this q-ary lattice. Using the embedding technique, this lattice point As can potentially be found by reducing the $(m + n + 1) \times (m + 1)$ matrix

$$\left(\begin{array}{c|c} b^T & 1 \\ \hline -A^T & 0 \\ \hline qI_m & 0 \end{array}\right).$$

as the corresponding embedding lattice should contain the small vector $(e|1) = \big((b - As)^T \big| 1\big)$. Finding such a small vector in this q-ary embedding lattice gives $e = b - As$. Thus we can find the q-ary lattice point $As = b - e = b - (b - As) \bmod q$. Having found the q-ary lattice point As, the secret s can be recovered by routine linear algebra, for example by the Mathematica command `LinearSolve[A,b-e,Modulus->q]`.

Example Constricting the above q-ary embedding lattice basis matrix for the Section 8.3 Example and then reducing this matrix gives the extremely short embedding lattice vector

$$(e|1) = (2, 3, 3, 0, 0, 2, 4, 1, -2, 1, 3, -2, 2, 0, 5, -5|1),$$

from which we obtain the q-ary lattice point $(As)^T = (b - e)$ of

$$(463, 484, 528, 527, 779, 384, 303, 228, 313, 323, 832, 775, 389, 392, 219, 411)^T$$

We can then obtain the secret $s = (357, 966, 927, 60)$. □

It can be seen that solving the LWE Problem depends on solving a lattice problem such as SVP or CVP in a q-ary lattice. Thus the difficulty of solving the LWE Problem depends on the quality of lattice reduction in a q-ary lattice or an associated lattice.

8.5 A Learning with Errors (LWE) Cryptosystem

We now give a public key cryptosystem using the Learning with Errors (LWE) Problem. The public key in this Learning with Errors (LWE) cryptosystem is the input to an LWE Problem, and the private key is the corresponding output of this LWE Problem, so

LWE Cryptosystem Set-up. The system parameters are an odd prime p, with $p' = \frac{1}{2}(p-1)$, a distribution χ on $\mathbb{F}_q$ centred on 0 with small "variance",

and parameters m giving the number of noisy functionals or samples and n giving the dimension of the secret s.

LWE Cryptosystem Key Generation. Alice's *private key* s is a vector of $\mathbb{F}_q^n$ of dimension n chosen uniformly at random. Alice's *public key* (A, b) is an $m \times n$ matrix A with entries $A_{ij} \sim \text{Uni}(\mathbb{F}_q)$ chosen uniformly at random from $\mathbb{F}_q$ and a vector $b = As + e \in \mathbb{F}_q^m$ of dimension m, where the components $e_1, \ldots, e_m \sim \chi$ of e are independent with distribution χ. Thus Alice's public key (A, b) forms the input to an LWE Problem, the solution to which gives the private key s.

LWE Cryptosystem Encryption. To encrypt a single message bit θ with Alice's public key (a, b), Bob first generates a random binary vector x uniformly from $\{0, 1\}^m$. Bob then generates the ciphertext (v, w) for the message bit θ, where

$$v = (x^T A)^T \in \mathbb{F}_q^n \quad \text{and} \quad w = x^T b + \theta q' \in \mathbb{F}_q.$$

LWE Cryptosystem Decryption. Alice decrypts the ciphertext (v, w) with her private key s by calculating

$$w - v^T s = x^T b + \theta q' - x^T As = \theta q' + x^T (b - As) = \theta q' + x^T e \in \mathbb{F}_q.$$

The quantity $x^T e$ is the sum of a selection of $e_1, \ldots, e_m$ (as $x \in \{0, 1\}^m$), so $x^T e$ should be small giving $w - u^T s \approx \theta q'$. Thus Alice decrypts (u, w) as $\theta = 0$ if $w - u^T s \approx 0$ and as $\theta = 1$ if $w - u^T s \approx q' = \frac{1}{2}(q - 1)$.

Example. We consider the LWE set-up of the Example of Section 8.3 based on a prime $q = 991$ with $m = 16$ "noisy functionals", a secret s of dimension $n = 4$ and error distribution χ a discretised Normal $\text{N}(0, 3.0^2)$ distribution with standard deviation 3.0. Alice's public key (A, b) from the LWE equation $b = As + e$ is as stated in this earlier Example.

Encryption. To encrypt a message bit $\theta = 1$, Bob first chooses a random $x = \{0, 1\}^m$, say in this case $x = (0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1)^T$. Bob then encrypts message bit $\theta = 1$ as ciphertext (u, v), where

$$\begin{aligned} v &= x^T A = a_4^T + a_6^T + a_9^T + a_{13}^T + a_{16}^T = (754, 249, 591, 163)^T \\ \text{and} \quad w &= x^T b + \theta q' = b_4 + b_6 + b_9 + b_{13} + b_{16} + q' = 534. \end{aligned}$$

(The encryption of bit $\theta = 0$ gives $v = (754, 249, 591, 163)^T$ and $w = 39$.)

Decryption. To decrypt the ciphertext (v, w) with $v = (71, 74, 964, 206)^T$ and $w = 494$, with private key $s = (357, 966, 927, 60)^T$, Alice calculates

$$w - v^T s = 534 - (754, 249, 591, 163)(357, 966, 927, 60)^T = 492.$$

As $w - v^T s = 492$ is close to $q' = \frac{1}{2}(q-1) = 495$, Alice decrypts ciphertext (v, w) as the message bit $\theta = 1$. (The decryption of $v = (754, 249, 591, 163)^T$ and $w = 39$ gives $w - v^T s = 988 = -3 \bmod 991$ and message bit $\theta = 0$.) $\square$

The LWE cryptosystem as outlined above requires a considerable effort to encrypt a single message bit θ, giving a cryptosystem with a slow throughput. In practice, a variant of the LWE cryptosystem is used in which elements of $\mathbb{F}_q$ are replaced with polynomials over $\mathbb{F}_q$, that is to say with ring elements. This gives a Ring-LWE cryptosystem in which several message bits can be processed in the same encryption calculation. Furthermore, the Ring-LWE cryptosystem naturally gives rise to a Ring-LWE Problem, whose solution is fundamentally the solution to a particular lattice problem

8.6 Brief Summary of Lattices in Cryptography

Lattices have become a fundamental part of modern cryptology. Their first use was to break a *Knapsack cryptosystem*, (an early public key system) in 1982 by framing a "knapsack question" as a (soluble) lattice question. Lattice techniques have since been used as an analytical tool in cryptology in various different areas.

Lattices have attracted much interest in cryptology over the last two decades as Lattice Problems (SVP, CVP etc.) do not seem amenable to Quantum Computing. This contrasts with the situation RSA and El Gamal systems where factoring and discrete log problems can be solved if hidden algebraic structure can be found, a situation tailor-made for Quantum computing. Thus lattice-based cryptosystems seem likely to form an important part of the future of public key cryptography if it is required that asymmetric algorithms should be *quantum-resistant*. Many of the cryptosystem submissions to the recent NIST (National Institute of Standards and Technology) process to begin the standardisation of some *post-quantum* cryptosystems are based on lattices.

Lattice-based cryptosystems can also potentially give new functionalities for cryptosystems. For example, (fully) homomorphic encryption has been an important development over the last decade and generally depend on lattice-based cryptosystems. Homomorphic encryption potentially allows for a ciphertext to be updated directly when a message or plaintext is updated (for example a person's age) without the ciphertext having to be decrypted.

9 Introduction to Elliptic Curve Cryptosystems

Elliptic curves are a major topic in number theory with connections to algebra, galois theory, complex analysis and topology. They have an extremely high profile in mathematics research due to recent applications, such as the proof of Fermat's Last Theorem. From a cryptographic viewpoint, elliptic curves can be used to define a variant of the Discrete Logarithm Problem, and hence variants of Discrete Logarithm cryptosystems.

9.1 Algebraic and Projective Curves

An **algebraic curve** over a field K is the set of solutions to a polynomial equation $f(x, y) \in K[x, y]$.

Example. Let K be the real field $\mathbb{R}$ and consider the curves given by

$$y - x^2, \qquad x^2 + y^2 - 1, \qquad y^2 - x^2.$$

The above curves are called **affine** as their graphs are subsets of the affine plane K^2. A **line** is an equation of the form $ax + by + c = 0$.

We can also consider projective geometry. The projective plane is the set

$$K^3 \setminus \{0\} = \{(X, Y, Z) \colon X, Y, Y \in K, \text{ at least one of } X, Y, Z \neq 0\}$$

with the equivalence relation $\sim$ given by $(X, Y, Y) \sim (\alpha X, \alpha Y, \alpha Z)$ for any $\alpha \in K^*$. Thus this equivalence relation identifies any two nonzero points of K^3 with each other if they lie on the same straight line (through the origin). We identify the affine plane K^2 with the subset $(X, Y, 1)$ of the projective plane. The points $(X, Y, 0)$ of the projective place which do not correspond to points in the affine plane are called **points at infinity**.

9.2 Specification of Elliptic Curves

An elliptic curve can be specified over a field of characteristic 0 (such as the reals) or over a finite field. For cryptographic use, we are primarily interested in elliptic curves over a finite field $\mathbb{F}_p$ for some prime p, where for simplicity we assume that $p > 3$. However, we note that we can define elliptic curves over $\mathbb{F}_2$ or $\mathbb{F}_3$, but the functional form is different.

Elliptic curves originate by considering homogeneous multivariate polynomials of degree 3 (all terms have degree 3) in three variables over a field

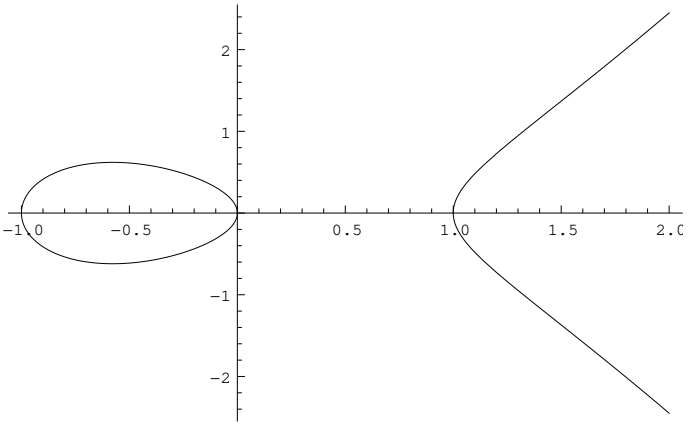

Figure 27: Elliptic curve $y^2 = x^3 - x$ over the real numbers $\mathbb{R}$.

K. As noted above, we generally take $K = \mathbb{F}_p$ for a prime $p > 3$. After translations and rescalings etc., the generic form for such a polynomial is

$$Y^2Z = X^3 + aXZ^2 + bZ^3.$$

for constants $a, b \in K$. For $Z \neq 0$, we can reduce the number of variables in this expression by by dividing by Z^3 and setting $x = \dfrac{X}{Z}$ and $y = \dfrac{Y}{Z}$ (or essentially by taking $Z = 1$) to obtain an elliptic curve E.

Definition. An **Elliptic Curve** over a finite field $\mathbb{F}_p$ (for $p > 3$) or field of characteristic 0 is a curve of the form

$$E : y^2 = x^3 + ax + b \quad \text{for constants } a \text{ and } b.$$

Figure 27 shows the elliptic curve $y^2 = x^3 - x$ ($a = -1$ and $b = 0$) over the real numbers. This elliptic curve has two disconnected parts: an "oval"-shape (for $-1 \leq x \leq 0$) and a part stretching to $\pm\infty$ (for $x \geq 1$). The following Table shows a calculation to give the points on the elliptic curve $y^2 = x^3+2x+6$ mod 7, and Figure 28 shows this elliptic curve. Furthermore, Figure 29 shows the far larger elliptic curve $y^2 = x^3 + 31x + 411$ mod 727 with far more points.

x	0	1	2	3	4	5	6
x^3	0	1	1	6	1	6	6
$2x$	0	2	4	6	1	3	5
6	6	6	6	6	6	6	6
y^2	6	2	4	4	1	1	3
y		3, 4	2, 5	2, 5	1, 6	1, 6	

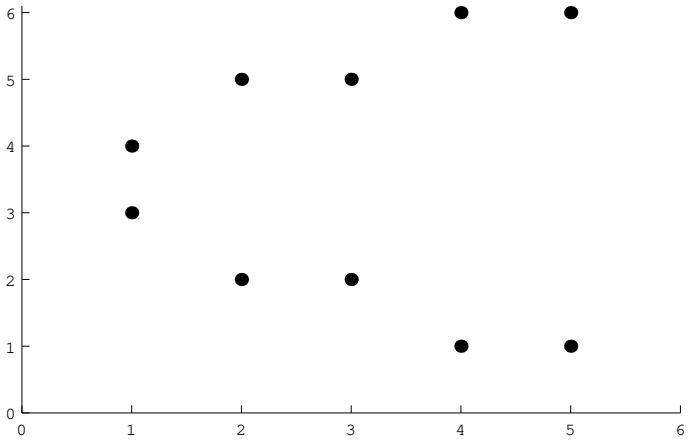

Figure 28: Elliptic Curve $y^2 = x^3 + 2x + 6$ over $\mathbb{F}_7$

Definition. An elliptic curve E given by $y^2 = x^3 + ax + b$ is **singular** if there is a point (x, y) on Esuch that the partial derivatives $2y$ and $3x^2 + a$ both vanish at the same point on E.

The elliptic curve $y^2 = x^3 - 3x + 2$ over the reals (derivatives vanish at $(1, 0)$) and the elliptic curve $y^2 = x^3 - x + 1$ over $\mathbb{F}_{23}$ (derivatives vanish at $(13, 0)$) are singular elliptic curve. A non-singular elliptic curve is one that is "smooth" at all points, and non-singular elliptic curves are required for cryptography. The following result gives a condition for an elliptic curve to be non-singular.

Lemma. An elliptic curve E given by $y^2 = x^3 + ax + b$ over a field of characteristic 0 is non-singular if $4a^3 + 27b^2 = 0$. An elliptic curve E given by $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ $(p > 3)$ is non-singular if $4a^3 + 27b^2 = 0 \bmod p$.

On the associated projective curve for an elliptic curve, there is further **point at infinity**, denoted by $\mathcal{O}$ or ∞. This point at infinity arises from the $Z = 0$ case and cannot be written in the form (x, y). In this case, the set of points on E is

$$E(\mathbb{F}_p) = \left\{ (x, y) \in \mathbb{F}_p^2 \;\middle|\; y^2 \equiv x^3 + ax + b \bmod p \right\} \bigcup \{\infty\}.$$

Example. The (projective) elliptic curve $E(\mathbb{F}_7)$ for $y^2 = x^3 + 2x + 6 \bmod 7$ has 11 points and is given by

$$E(\mathbb{F}_7) = \{\mathcal{O}, (1, 3), (1, 4), (2, 2), (2, 5), (3, 2), (3, 5), (4, 1), (4, 6), (5, 1), (5, 6)\}.$$

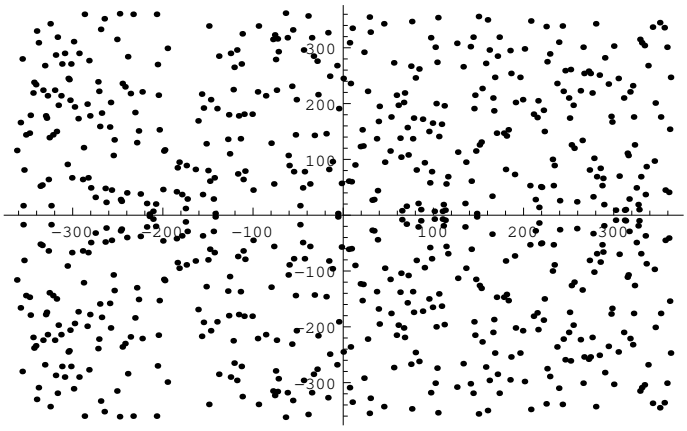

Figure 29: Elliptic curve $y^2 = x^3 + 31x + 411$ over $\mathbb{F}_{727}$

9.3 Elliptic Curve Group

It is a remarkable fact (assumed without proof in these Notes) about an elliptic curves is that there is a group operation on the points of the elliptic curve, arising from the following result. We note that the points $P = (x_P, y_P)$ where the line $y = l(x)$ intersects the curve $f(x, y)$ correspond to the roots x_P of the polynomial $f(x, l(x))$.

Lemma. Suppose ℓ is a line that intersects an elliptic curve E in at least two points (counting multiplicities), then the line ℓ intersects the curve E in exactly three points (counting multiplicities).

The above result means that is possible to define an abelian (commutative) group law for an elliptic curve by satisfying the following requirements.
(a) The identity element is the point at infinity $\mathcal{O}$.
(b) The sum of $(x, y) \in E$ and $(x, -y) \in E$ is the identity element $\mathcal{O}$.
(c) The sum of any three collinear points is this identity element $\mathcal{O}$.

Condition (c) shows that we define addition of two points on an elliptic curve E by the following procedure.

Addition of Elliptic Curve points $P = (x_P, y_P)$ **and** $Q = (x_Q, y_Q)$
(i) Construct the straight line ℓ between P and Q.
(ii) The line ℓ intersects E in a third point $S = (x_S, y_S)$.
(iii) Construct the vertical line through S intersecting E at $R = (x_R, y_R)$.
(iv) The sum of P and Q is R, that is $R = P + Q$

The above construction of $P + Q$ works as the sum of the collinear points

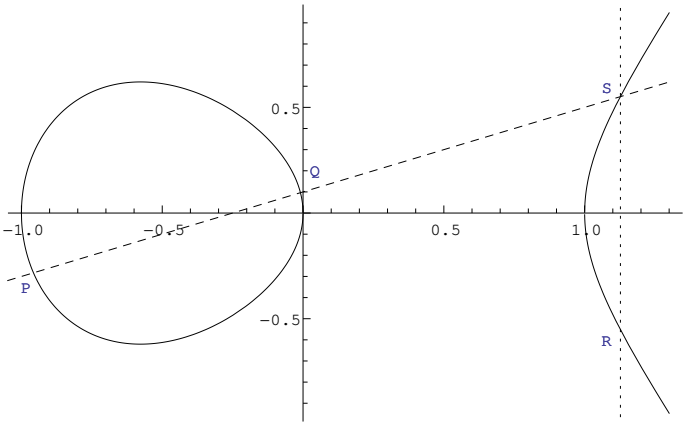

Figure 30: Geometric construction of sum of P and Q on elliptic curve E to give $R = -S = P + Q$

P, Q and R is $P + Q + R = \mathcal{O}$, so $R = -S = P + Q$ as $R + S = \mathcal{O}$. This general geometric construction for the sum of elliptic curve points is illustrated in Figure 30. Addition of a point P to itself is achieved by constructing the tangent line ℓ to E at P and then letting S be the third point (with multiplicity) with E and ℓ intersect and so on.

Example. Consider the addition of $P = (5, 1)$ and $Q = (2, 5)$ on the elliptic curve E given by $y^2 = x^3 + 2x + 6 \bmod 7$ (see earlier Example and Figure 28). The line ℓ between P and Q is $y = x + 3$, and the third point at which E and ℓ intersect is $S = (1, 4)$, so $P + Q + R = \mathcal{O}$. Thus $P + Q = R = -S = (1, -4) = (1, 3)$, as illustrated in Figure 31

The above geometric construction for elliptic curve addition can be translated to coordinate expressions in a simple way to give the following addition process for addition of poitns with coordinates (addition of $\mathcal{O}$ is trivial).

Addition of Elliptic Curve points by Coordinates
Suppose $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$ and $R = P + Q = (x_R, y_R)$
(i) If $x_P = x_Q$ and $y_P = -y_Q$, then $R = \mathcal{O}$.
(ii) Set "gradient" $\lambda = \dfrac{y_Q - y_P}{x_Q - x_P}$ $[P \neq Q]$ and $\lambda = \dfrac{3x_P^2 + a}{2y_P}$ $[P = Q]$.
(iii) Set $x_R = \lambda^2 - x_P - x_Q \bmod p$ and $y_R = -\lambda(x_R - x_P) - y_P$

Example. Consider the addition of elliptic curve points

$$P = (x_P, y_P) = (233, 674) \quad \text{and} \quad Q = (x_Q, y_Q) = (519, 707)$$

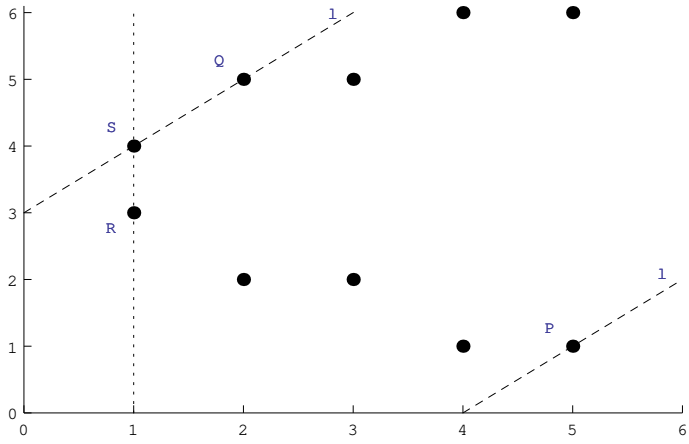

Figure 31: Addition of $P = (5, 1)$ and $Q = (2, 5)$ to give $R = P + Q = (1, 3)$ on elliptic curve given by $y^2 = x^3 + 2x + 6 \bmod 7$

on the elliptic curve E given by $y^2 = x^3 + 31x + 411 \bmod 727$. The gradient $\lambda = (y_Q - y_P)(x_Q - x_P)^{-1} = (707 - 674) \times (519 - 233)^{-1} = 84 \bmod 727$. Thus we have $x_R = \lambda^2 - x_P - x_Q = 84^2 - 233 - 519 = 488 \bmod 727$ and $y_R = -\lambda(x_R - x_P) - y_P = -84(488 - 233) - 674 = 443 \bmod 727$, and so

$$R = (488, 443) = P + Q = (233, 674) + (519, 707) \quad \text{on } E(\mathbb{F}_{727}).$$

The addition rule for elliptic curve groups allows us (in principle) to construct an addition table for any elliptic curve group.

Example. Figure 32 shows such an addition (or Cayley) table for the elliptic curve group $E(\mathbb{F}_7)$ for $y^2 = x^3 + 2x + 6$ of earlier examples. As this group is an abelian group with a prime number (11) of elements, this group is necessarily a cyclic isomorphic to C_{11}. Thus this group is generated by any non-identity element, and any other nonzero element can be expressed as a multiple of this generating element. If we take $(1, 3)$ for the generating element and writing $[k](1, 3) = (1, 3) + \ldots + (1, 3)$ for the k-fold sum of $(1, 3)$ with itself, then $[1](1, 3) = (1, 3)$ and $[11](1, 3) = \mathcal{O}$, so the non-trivial multiples of $(1, 3)$ are given by

$$\begin{array}{cc}
[1](1,3) = (1,3) & [2](1,3) = (1,3) + (1,3) = (2,2) \\
[3](1,3) = (2,2) + (1,3) = (5,1) & [4](1,3) = (5,1) + (1,3) = (3,5) \\
[5](1,3) = (3,5) + (1,3) = (4,1) & [6](1,3) = (4,1) + (1,3) = (4,6) \\
[7](1,3) = (4,6) + (1,3) = (3,2) & [8](1,3) = (3,2) + (1,3) = (5,6) \\
[9](1,3) = (5,6) + (1,3) = (2,5) & [10](1,3) = (2,5) + (1,3) = (1,4).
\end{array}$$

+	$\mathcal{O}$	$(1,3)$	$(2,2)$	$(5,1)$	$(3,5)$	$(4,1)$	$(4,6)$	$(3,2)$	$(5,6)$	$(2,5)$	$(1,4)$
$\mathcal{O}$	$\mathcal{O}$	$(1,3)$	$(2,2)$	$(5,1)$	$(3,5)$	$(4,1)$	$(4,6)$	$(3,2)$	$(5,6)$	$(2,5)$	$(1,4)$
$(1,3)$	$(1,3)$	$(2,2)$	$(5,1)$	$(3,5)$	$(4,1)$	$(4,6)$	$(3,2)$	$(5,6)$	$(2,5)$	$(1,4)$	$\mathcal{O}$
$(2,2)$	$(2,2)$	$(5,1)$	$(3,5)$	$(4,1)$	$(4,6)$	$(3,2)$	$(5,6)$	$(2,5)$	$(1,4)$	$\mathcal{O}$	$(1,3)$
$(5,1)$	$(5,1)$	$(3,5)$	$(4,1)$	$(4,6)$	$(3,2)$	$(5,6)$	$(2,5)$	$(1,4)$	$\mathcal{O}$	$(1,3)$	$(2,2)$
$(3,5)$	$(3,5)$	$(4,1)$	$(4,6)$	$(3,2)$	$(5,6)$	$(2,5)$	$(1,4)$	$\mathcal{O}$	$(1,3)$	$(2,2)$	$(5,1)$
$(4,1)$	$(4,1)$	$(4,6)$	$(3,2)$	$(5,6)$	$(2,5)$	$(1,4)$	$\mathcal{O}$	$(1,3)$	$(2,2)$	$(5,1)$	$(3,5)$
$(4,6)$	$(4,6)$	$(3,2)$	$(5,6)$	$(2,5)$	$(1,4)$	$\mathcal{O}$	$(1,3)$	$(2,2)$	$(5,1)$	$(3,5)$	$(4,1)$
$(3,2)$	$(3,2)$	$(5,6)$	$(2,5)$	$(1,4)$	$\mathcal{O}$	$(1,3)$	$(2,2)$	$(5,1)$	$(3,5)$	$(4,1)$	$(4,6)$
$(5,6)$	$(5,6)$	$(2,5)$	$(1,4)$	$\mathcal{O}$	$(1,3)$	$(2,2)$	$(5,1)$	$(3,5)$	$(4,1)$	$(4,6)$	$(3,2)$
$(2,5)$	$(2,5)$	$(1,4)$	$\mathcal{O}$	$(1,3)$	$(2,2)$	$(5,1)$	$(3,5)$	$(4,1)$	$(4,6)$	$(3,2)$	$(5,6)$
$(1,4)$	$(1,4)$	$\mathcal{O}$	$(1,3)$	$(2,2)$	$(5,1)$	$(3,5)$	$(4,1)$	$(4,6)$	$(3,2)$	$(5,6)$	$(2,5)$

Figure 32: Addition Table for Elliptic Curve Group of $y^2 = x^3+2x+6 \bmod 7$

Furthermore, we can in general calculate a multiple $[n]P$ of an elliptic curve point P efficiently by using a "double and add" approach (an additive version of square and multiply) based on the binary expansion of n.

An elliptic curve group is an abelian (commutative) group, so can be expressed as a direct product of cyclic groups. The result below shows that an elliptic curve group is actually the direct product of at most two abelian groups. Thus such an elliptic curve group has either a cyclic or toroidal stricture.

Lemma. An elliptic curve group over a finite field $\mathbb{F}_p$ is either a cyclic group C_m or the product $C_m \times C_n$ of two cyclic groups with n dividing m.

This result that an elliptic curve group can expressed in terms of cyclic groups allows us to define an elliptic curve discrete logarithm problem. General methods of addressing the elliptic curve discrete logarithm problem are discussed in Section 10.6.

Elliptic Curve Discrete Logarithm Problem (ECDLP) Given points P and Q of order q in an elliptic curve $E(\mathbb{F}_p)$, find an integer n (if it exists) such that $Q = [n]P$.

Example. In the elliptic curve group defined by $y^2 = x^3 + 2x + 6$ over $\mathbb{F}_7$, find n such that $[n](1,3) = (4,1)$. (The previous page shows $n = 4$.)

Example. In the elliptic curve group defined by $y^2 = x^3 + 31x + 411$ over $\mathbb{F}_{727}$, find n such that $[n](233,674) = (491,509)$. (This is a difficult Problem.)

10 Further Elliptic Curve Cryptosystems

In this Section we discuss techniques to determine the number of points on an elliptic curve, the specification of elliptic curve cryptosystems and techniques to address the elliptic curve discrete logarithm problem. This allows us to compare the security of elliptic curve cryptosystems with other forms of public key cryptography.

10.1 The Order of an Elliptic Curve Group

A fundamental problem in elliptic curve cryptography is the computation of the order of a specific elliptic curve group $E(\mathbb{F}_p)$, a process often termed *point counting*. Before looking at point counting techniques for a specific elliptic curve, we make some general comments about the size of an elliptic curve group over $\mathbb{F}_p$.

We first note that the following heuristic argument shows that the number of elliptic curve points $\#E(\mathbb{F}_p) \approx p + 1$. For a given $x \in \mathbb{F}_p$, the value $x^3 + ax + b$ is distributed uniformly at random (approximately) in $\mathbb{F}_p$, so $x^3 + ax + b$ is a square in $\mathbb{F}_p$ with approximate probability $\frac{1}{2}$ and a non-square with approximate probability $\frac{1}{2}$. Thus we find that the elliptic curve $E(\mathbb{F}_p)\colon y^2 = x^3 + ax + b$ has 2 points $(x, \pm y)$ with x as first coordinate with probability $\frac{1}{2}$ and no points with x as first coordinate with probability $\frac{1}{2}$. Counting the approximate number of points in $E(\mathbb{F}_p)$ over all $x \in \mathbb{F}_p$ and adding the point at infinity $\mathcal{O}$, we then obtain the heuristic approximation $\#E(\mathbb{F}_p) \approx p + 1$.

The number of points on an elliptic curve group $E(\mathbb{F}_p)$ is made more precise by the following (rigorous) Theorem due to Hasse (1933).

Theorem. An elliptic curve $E(\mathbb{F}_p)$ satisfies $|p + 1 - \#E(\mathbb{F}_p)| \leq 2\sqrt{p}$.

The quantity $t = p+1-\#E(\mathbb{F}_p)$ is termed the *trace of Frobenius*, and so this Theorem states that $|t| \leq 2\sqrt{p}$. This result shows that the number $\#E(\mathbb{F}_p)$ of points in an elliptic curve group is (relatively) close to $p + 1$ for large p and is generally termed the *Hasse bound*, that is

$$(p + 1) - 2\sqrt{p} \leq \#E(\mathbb{F}_p) \leq (p + 1) + 2\sqrt{p}.$$

A simple naive method of counting the points on a specific elliptic curve $E(\mathbb{F}_p)\colon y^2 = x^3 + ax + b$ is to try all values for $x \in \mathbb{F}_p$ and to count the number of solutions y to $y^2 = x^3 + ax + b$ for each x. We note that the number of solutions to $y^2 = x^3 + ax + b$ is 0, 1 or 2 according to whether the

corresponding Jacobi symbol is -1, 0 or 1, so we can obtain the expression

$$\#E(\mathbb{F}_p) = 1 + \sum_{x=0}^{p-1} \left(\left(\frac{x^3 + ax + b}{p} \right) + 1 \right).$$

Each term in this sum of p terms requires the computation of a Legendre symbol, which can each be computed in $O((\log p)^2)$ time, so the overall time complexity of evaluating this expression is $O(p(\log p)^2)$. However, the input size for this expression is in general $O(\log p)$, so evaluating this expression has exponential complexity.

We can improve the efficiency of point counting on a elliptic curve $E(\mathbb{F}_p)$ by expressing the problem as an elliptic curve discrete logarithm problem. We know that $[\#E(\mathbb{F}_p)]P = \mathcal{O}$ for any elliptic curve point $P \in E(\mathbb{F}_p)$, where the Hasse bound shows that $\#E(\mathbb{F}_p) = (p+1) - t$ with $|t| \leq 2\sqrt{p}$. If we define the elliptic curve point $Q = [p+1]P$, we obtain

$$Q = [p+1]P = [p+1]P - \mathcal{O} = [p+1-\#E(\mathbb{F}_p)]P = [t]P,$$

where t is small. Solving the resulting elliptic curve discrete logarithm problem $Q = [t]P$ gives t modulo the order of P. For a general point P, the order of P is usually much larger than the relatively small t (as $|t| \leq 2\sqrt{p}$), so solving this elliptic curve discrete logarithm problem generally recovers t directly as an integer, and so gives the the order $\#E(\mathbb{F}_p) = (p+1) - t$ of the elliptic curve group $E(\mathbb{F}_p)$. However, this method of point counting is still exponential, even though some improvements to generic discrete logarithm algorithms can be made when the "answer" is known to be "small".

Schoof's algorithm (1985) is a development of the above ideas to give an algorithm with polynomial complexity. The basic idea is to adapt the above method by choosing elliptic curve points P of small orders $2, 3, 5, 7, \ldots$ in constructing the elliptic curve discrete logarithm problem $Q = [t]P$, where $Q = [p+1]P$. Solution of these elliptic curve discrete logarithm problems gives t modulo $2, 3, 5, 7, \ldots$, and hence t by the Chinese Remainder Theorem. The details of how to find such points P are complicated and involve working in an extension field $\mathbb{F}_{p^n}$ of $\mathbb{F}_p$. Schoof's algorithm has subsequently been improved by Elkies and by Atkin (and others) to give a method to determine the number of points $\#E(\mathbb{F}_p)$ on a specific elliptic curve $E(\mathbb{F}_p)$ with time complexity $O((\log p)^6)$ (using naive arithmetic).

10.2 Elliptic Curve Parameter Generation

We later specify various elliptic curve cryptosystems. In order to specify such cryptosystems, we first have to select an appropriate elliptic curve for cryptography. One method is simply to use an appropriate set of elliptic curve parameters from one of the elliptic curve cryptography standards. More generally, the following method with polynomial time complexity generates parameters (p, a, b, q, P) such that p and q are κ-bit primes and base point $P \in E(\mathbb{F}_p)\colon y^2 = x^3 + ax + b$ has prime order q.

Elliptic Curve Parameter Generation

- Generate a κ-bit prime p.
- Choose random a, b such that $4a^3 + 27b^2 \neq 0 \bmod p$.
  - Define the elliptic curve $E(\mathbb{F}_p)\colon y^2 = x^3 + ax + b$.
  - Fine $q = \#E(\mathbb{F}_p)$ by Schoof.
  - If q is not prime (use compositeness test) then repeat this step.
- Choose a random point $P = (x_P, y_P) \in E(\mathbb{F}_p)$ (note P has order q).
- Output $(p, q, a, b, P = (x_P, y_P))$.

10.3 Elliptic Curve Diffie-Hellman Key Agreement

Cryptosystems defined in the multiplicative group $\mathbb{F}_p^*$ of a finite field $\mathbb{F}_p$ can usually be adapted to give a corresponding elliptic curve cryptosystem by replacing elements of $\mathbb{F}_p^*$ with elliptic curve points and multiplications in $\mathbb{F}_p^*$ with the corresponding group operation of point addition in the elliptic curve group. In particular, this means it is straightforward to produce an elliptic curve version of the Diffie-Hellman key agreement protocol, and we do so below. This elliptic curve version of Diffie-Hellman produces a shared secret elliptic curve point, which can then be further processed to give a shared secret symmetric key (for example). However, this process also gives rise to an elliptic curve version of the Diffie Hellman Problem (DHP).

Elliptic Curve Diffie-Hellman Problem (ECDHP). Given the triple $(P, [a]P, [b]P)$ of points in $E(\mathbb{F}_p)$, compute the elliptic curve point $[ab]P$.

Example. Given a triple $((0, 482), (321, 495), (667, 638)) = (P, [a]P, [ab]P)$ of points in the elliptic curve group $E(\mathbb{F}_{727})\colon y^2 = x^3 + 31x + 411$, find $[ab]P$.

We note that ECDHP $\leq_T$ ECDLP, and that a decision version of this Problem can be specified in the obvious way. In order to break the following elliptic cure key agreement scheme, we have to solve ECDHP.

Elliptic Curve Diffie-Hellman Key Agreement
Alice and Bob want to agree on a secret key K

- Alice and Bob choose a large prime p and a random elliptic curve $E(\mathbb{F}_p)\colon y^2 = x^3 + ax + b$ and a point $P \in E(\mathbb{F}_p)$ of prime order q.
- Alice chooses a random integer $0 < z_A < q$ and sends $Q_A = [x_A]P$ to Bob.
- Bob chooses a random integer $0 < z_B < q$ and sends $Q_B = [z_B]P$ to Alice.
- On receiving Q_B, Alice computes $K = [z_A]Q_B = [z_A z_B]P$.
- On receiving Q_A, Bob computes $K = [z_B]Q_A = [z_A z_B]P$.

Example. Alice and Bob choose the system parameters $p = 73$ (prime), the elliptic curve $E(\mathbb{F}_{73})\colon y^2 = x^3 + 50x + 39$ with 79 elements and generating point $P = (2, 1)$ of order $q = 79$.
Alice chooses $z_A = 7$ and sends $Q_A = [7]P = [7](2, 1) = (5, 66)$ to Bob.
Bob chooses $z_B = 67$ and sends $Q_B = [73]P = [73](2, 1) = (70, 9)$ to Alice.
Alice obtains $K = [z_A]Q_B = [7](70, 9) = (58, 41)$.
Bob obtains $K = [z_B]Q_A = [73](5, 66) = (58, 41)$.

10.4 Elliptic Curve El Gamal Encryption

El Gamal encryption is best understood as a static Diffie-Hellman key transport protocol. In the El Gamal elliptic curve setting, this means that Alice contributes a static public key point $[a]P$ and Bob contributes a public key $[k]P$ depending on a randomly generated value k, fresh for each message. The shared secret elliptic curve point $K = [ak]P$ is then used to generate a symmetric encryption scheme.

Elliptic Curve El Gamal Set-Up. Generate an elliptic curve $E(\mathbb{F}_p)\colon y^2 = x^3 + ax + b$ and a point $P \in E(\mathbb{F}_p)$ of order q as described above. Alice chooses a random integer z as private key and sets $Q = [z]P$ as her public key elliptic curve point.

Elliptic Curve El Gamal Encryption. Let $E_{K'}$ be an n-bit symmetric encryption function (such as AES) with key K'. The message space is $\{0, 1\}^n$

and the ciphertext space $E(\mathbb{F}_p) \times \{0,1\}^n$. Bob chooses a random integer k with $0 < k < q$ and derives a symmetric key K' from elliptic curve point $K = [k]Q$. Bob then calculates the ciphertext (C_1, C_2) for an n-bit message m as

$$C_1 = [k]P \quad \text{and } C_2 = E_{K'}(m).$$

Elliptic Curve El Gamal Decryption. Given a ciphertext (C_1, C_2) Alice computes $K = [z]C_1$ and then derives the symmetric key K' for the decryption function $D_{K'}$. Alice can then recover the message as $m = D_{K'}(C_2)$.

An elliptic curve El Gamal encryption requires two point multiplications and decryption requires one, plus the further cost of the symmetric encryption and decryption (assumed to be efficient). Hence elliptic curve El Gamal encryption and decryption are polynomial time and efficient. The ciphertext requirement is the n-bit length of the message block and an elliptic curve point $C_1 = (x, y)$. In fact, an elliptic curve point can be specified with an x-coordinate and a further bit (such as the parity) to indicate which of the two square roots is being used.

10.5 An Elliptic Curve Signature Scheme

The ECDSA is an elliptic curve signature scheme standardised by NIST that is elliptic curve version of DSA. However, we describe a type of elliptic curve signature (similar to ECDSA) known as a Schnorr signature. The set-up for this elliptic curve signature scheme is the same as for the elliptic curve El Gamal encryption scheme.

Elliptic Curve Signature Generation. Let m be a message to be signed and H a cryptographic hash function mapping into $\mathbb{F}_q$. Alice chooses a random integer k with $1 < k < q$ and computes $[k]P$. Alice then signs a message m with the signature (r, s) using her private signing key z, where

$$r = H(m, [k]P) \quad \text{and} \quad s = k + zr \bmod q.$$

Elliptic Curve Signature Verification. Bob first checks that $0 < r < q$ and that $0 \leq s < q$, and then Bob calculates $r' = H(m, [s]P - [r]Q)$ using Alice's public verification key point Q. If r' equals signature component r, then the signature (r, s) for message m is "Valid".

10.6 Elliptic Curve Discrete Logarithm Problem

The security of these elliptic curve cryptosystems depends on a discrete logarithm problem in the elliptic curve group. Many of the earlier ideas for computing discrete logarithms in a generic abelian group can be applied in the elliptic curve setting. We restate this discrete logarithm below for completeness.

Elliptic Curve Discrete Logarithm Problem (ECDLP). Given points P and Q of order q in an elliptic curve $E(\mathbb{F}_p)$, find an integer n (if it exists) such that $Q = [n]P$.

Silver-Pohlig-Hellman Method. Suppose $Q = [\lambda]P$ in an elliptic curve $E(\mathbb{F}_p)$, where P has order $n = \prod_{i=1}^{r} p_i$ (for simplicity) and therefore the order of $Q \in \langle P \rangle$ divides the order of n. We note that the more general case $\prod_{i=1}^{r} p_i^{e_i}$ is similar. To solve the ECDLP, that is to say to find λ, we can map the problem $Q = [\lambda]P$ to subgroups of prime order just like we did in the case of $\mathbb{F}_p^*$.

We set $P_i' = [n/p_i]P$ and $Q_i' = [n/p_i]Q$, so both P_i' and Q_i' have order dividing p_i with $Q_i' = [\lambda]P_i'$. Hence by solving the ECDLP in the subgroup $\langle P_i' \rangle$ of order p_i we determine the value of λ modulo p_i. The Chinese Remainder Theorem can then give the full solution λ (exactly as in $\mathbb{F}_p^*$).

The Silver-Pohlig-Hellman Method shows that the complexity of solving an elliptic curve discrete logarithm problem depends on the size of the largest prime divisor of the group order.

Example. Consider the elliptic curve discrete logarithm problem $Q = [\lambda]P$ in the elliptic curve group $E(\mathbb{F}_{31})$: $y^2 = x^3 + 3x + 2$ of order 35, where the elliptic curve points are $P = (0, 8)$ (of order 35) and $Q = (2, 4)$. Thus we wish to solve $[\lambda](0, 8) = (2, 4)$ in $E(\mathbb{F}_{31})$. This ECDHP and the SPH subgroups are illustrated in Figure 33.

We map to the subgroup of order 5 by multiplying by $[35/5]$ to obtain $P_5' = [35/5]P = [7](0, 8) = (27, 22)$ and $Q_5' = [35/5]P = [7](2, 4) = (10, 28)$. Thus $[\lambda \bmod 5]P_5' = [\lambda \bmod 5](27, 22) = (10, 28)$, and so $\lambda = 2 \bmod 5$.

We map to the subgroup of order 7 by multiplying by $[35/7]$ to obtain $P_7' = [35/7]P = [5](0, 8) = (19, 6)$ and $Q_7' = [35/7]P = [5](2, 4) = (25, 4)$. Thus $[\lambda \bmod 7]P_7' = [\lambda \bmod 7](19, 6) = (25, 4)$, and so $\lambda = 3 \bmod 5$.

These elliptic curve discrete logarithm subgroup searches show

$$\lambda = 2 \bmod 5 \quad \text{and} \quad \lambda = 3 \bmod 7.$$

The Chinese Remainder Theorem then shows that $\lambda = 17 \bmod 30$, and so

$$[17](0, 8) = (2, 4).$$

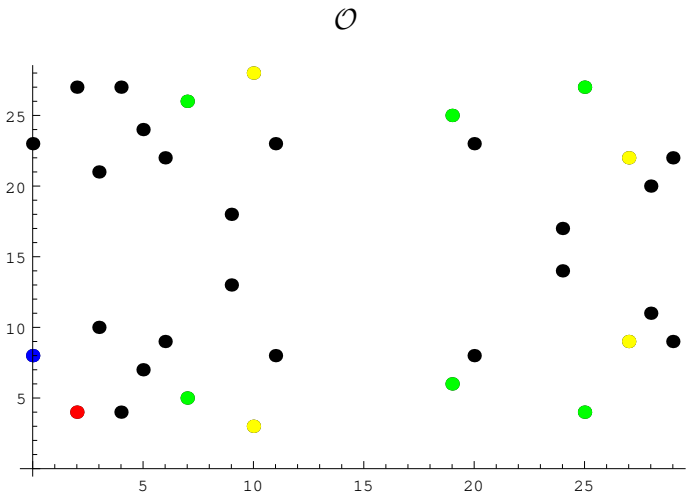

Figure 33: Elliptic curve $E(\mathbb{F}_{31})\colon y^2 = x^3 + 3x + 2$ illustrating ECDLP $[\lambda](0,8) = (2,4)$ with $(0,8)$ (blue dot) and $(2,4)$ (red dot) and showing the subgroups of order 7 (green dots) and order 5 (yellow dots).

The Baby-Step Giant-Step Method The Baby-Step Giant-Step method works exactly as in the $\mathbb{F}_p^*$ case discussed above. Suppose that P have prime order q and that $Q = [\lambda]P$ for some unknown λ. We let $m = \lceil \sqrt{q} \rceil$ and construct a list of baby steps $[0]P, [1]P, \ldots, [m-1]P$. We then construct a list of giant steps $Q + [0]R, Q + [1]R, \ldots, Q + [m-1]P$, where $R = -[m]P$. Constructing and storing these lists has an $O(\sqrt{q})$ complexity. We then look for a match between these two lists, when we have

$$[i]P = Q + [j]R = Q - [jm]P, \quad \text{so } [i + jm]P = Q.$$

Example. Consider the elliptic curve discrete logarithm problem $Q = [\lambda]P$ in the elliptic curve group $E(\mathbb{F}_{727})\colon y^2 = x^3 + 31x + 411$ of order 717, where the elliptic curve points are $P = (0, 482)$ (of order 717) and $Q = (375, 485)$. Thus we wish to solve $[\lambda](0, 482) = (375, 485)$ in $E(\mathbb{F}_{727})$, and this ECDLP is illustrated in Figure 34.

We take $m = \lceil 727 \rceil = 27$ to give the Baby Steps and Giant Steps as calculated below.

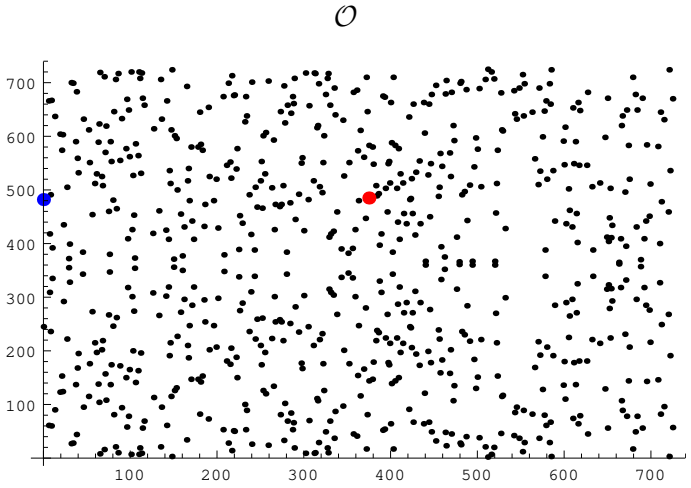

Figure 34: Elliptic curve $E(\mathbb{F}_{727})$: $y^2 = x^3 + 31x + 411$ illustrating ECDLP $[\lambda](0, 408) = (375, 485)$ with $(0, 408)$ (blue dot) and $(375, 485)$ (red dot).

- Baby Steps $P_i = [i]P$ for $i = 1, \ldots, 27$

$(0, 482)$	$(491, 509)$	$(665, 318)$	$(302, 9)$	$(599, 136)$
$(215, 175)$	$(9, 667)$	$(243, 339)$	$(148, 3)$	$(166, 517)$
$(180, 645)$	$(98, 649)$	$(721, 3)$	$(422, 482)$	$(305, 245)$
$(111, 19)$	$(542, 685)$	$(103, 527)$	$(654, 411)$	$(509, 706)$
$(64, 104)$	$(585, 724)$	$(670, 496)$	$(226, 275)$	$(63, 589)$
$(165, 34)$	$(580, 520)$			

- Giant Steps $Q_j = Q - [27j]P$ for $j = 1, \ldots, 27$

$(602, 290)$	$(480, 699)$	$(96, 58)$	$(22, 124)$	$(609, 431)$
$(297, 550)$	$(411, 197)$	$(86, 717)$	$(647, 598)$	$(97, 409)$
$(403, 17)$	$(617, 546)$	$(684, 78)$	$(632, 221)$	$(127, 614)$
$(396, 459)$	$(663, 365)$	$(10, 335)$	$(321, 495)$	$(605, 225)$
$(52, 115)$	$(285, 643)$	$(256, 120)$	$(667, 638)$	$(111, 19)$
$(180, 82)$	$(422, 245)$			

There is a match in the Baby Step and Giant Step Tables at $(111, 19)$ given by

$$(111, 19) = P_{16} = [16]P = Q_{25} = Q - [25][27]P = Q - [675]P.$$

Thus we have $Q = [675]P + [16]P = [691]P$ to obtain the discrete logarithm 691, that is $[691](0, 482) = (375, 485)$.

Pollard Rho Method. The Pollard Rho method can be similarly directly adapted from the $\mathbb{F}_p^*$ setting to give a method with $\sqrt{q}$ complexity and minimal storage.

Index Calculus Method. The most important fact about elliptic curves is that there do not seem to be subexponential time index calculus algorithms for solving the ECDLP. The key idea is that there is no natural equivalent notion of a "small prime" that is used to define the factor base used in regular index calculus in $\mathbb{F}_p^*$. Thus there does not appear to be a method of constructing an elliptic curve "factor base" and hence no notion of "smoothness" for elliptic curve points.

10.7 Elliptic Curve Cryptosystem Key Sizes

The lack of an elliptic curve discrete logarithm algorithm that directly addresses specific elliptic curve structure, apart from a few anomalous elliptic curves (not covered in these Notes), means that we generally only have to concern ourselves with "square root" discrete logarithm algorithms for generic abelian groups. Thus we can work securely with elliptic curves $E(\mathbb{F}_p)$ for a prime p of size about 160 to 170 bits. The work effort required to solve the resulting elliptic curve discrete logarithm problem is then an infeasible 2^{80} operations.

The consequence is that elliptic curve cryptosystems are better in many respects than comparable cryptosystems. Elliptic curve cryptosystems require less bandwidth and are faster for the same perceived security. In particular, elliptic curve cryptosystems can be implemented more efficiently on a device with restricted computational capacity, such as a smart card, and elliptic curve signatures are supported in the latest chip and pin standards.

Appendix: Introductory Group Theory

Definition. A *group* $(G, \circ)$ is set G equipped with a binary relation $\circ$, that is to say a function $G \times G \to G$ given by $(g_1, g_2) \mapsto g_1 \circ g_2 \in G$, satisfying the following conditions

- *Associativity.* $(a \circ b) \circ c = a \circ (b \circ c)$ for $a, b, c \in G$.
- *Identity Element.* There is an element $e \in G$ such that $e \circ g = g \circ e = g$ for all $g \in G$.
- *Inverse Elements.* For any $g \in G$, there is an element $g^{-1} \in G$ such that $g \circ g^{-1} = g^{-1} \circ g = e$.

Note. The group operation $\circ$ is often omitted if clear from the context.

Example. The integers $\{0, 1, 2, 3, n-1\}$ under addition modulo n often written $(\mathbb{Z}_n, +)$ form a group of order n.

Example. The nonzero integers $\{1, 2, 3, p-1\}$ under multiplication modulo a prime p, often written $(\mathbb{Z}_p^*, \cdot)$ or $(\mathbb{F}_p, \cdot)$, form a group of order $p-1$,

Example. The symmetries of an equilateral triangle form a group of order 6, having an identity element, three reflections about each angle bisector and two rotations of $\pm 120°$.

Definition. A *subgroup* $(H, \circ)$ of a group $(G, \circ)$ is a subset $H \subset G$ such that $(H, \circ)$ is a group, and is written $H \leq G$.

Lagrange's Theorem. The order of a subgroup $H \leq G$ divides the order of the group G.

Example. $(\mathbb{Z}_{12}, +)$, $(\{0, 2, 4, 6, 8, 10\}, +)$, $(\{0, 3, 6, 9\}, +)$, $(\{0, 4, 8\}, +)$ and $(\{0, 6\}, +)$ are the subgroups of $(\mathbb{Z}_{12}, +)$
$(\mathbb{Z}_5^*, \cdot)$. and $(\{1, 4\}, \cdot)$ are the subgroups of $(\mathbb{Z}_5^*, \cdot)$.
The identity and the two rotations are one of the subgroups of the symmetries of an equilateral triangle.

Definition. A group *homomorphism* $\phi\colon (G, \circ) \to (H, \cdot)$ (where H may be G and so on) is a mapping from G to H satisying $\phi(g \circ g') = \phi(g) \cdot \phi(g')$. $\mathrm{Im}(\phi) \leq H$ is a subgroup of H and $\mathrm{Ker}(\phi) \leq G$ is a subgroup of G.

Example. $\phi\colon (\mathbb{Z}_{12}, +) \to (\mathbb{Z}_{12}, +)$ defined by $z \mapsto 2z$ is a group homomorphism with $\mathrm{Im}(\phi) = (\{0, 2, 4, 6, 8, 10\}, +)$ and $\mathrm{Ker}(\phi) = (\{0, 6\}, +)$.

Definition. A group *isomorphism* $\phi\colon (G,\circ) \to (H,\cdot)$ is a bijective homomorphism, giving notation $G \cong H$. Isomorphic groups essentially have the same structure.

Example. $\phi\colon (\mathbb{Z}_5^*,\cdot) \to (\mathbb{Z}_4,\cdot)$ defined by $(1,2,3,4) \mapsto (0,1,3,2)$ is an isomorphism, so $(\mathbb{Z}_5^*,\cdot) \cong (\mathbb{Z}_4,\cdot)$ This isomorphism can be thought of as a discrete logarithm isomorphism obtained by mapping an element to "its power of 2".

Definition. A group $(G,\circ)$ is a *commutative* or *abelian* group if the binary operation $\circ$ is a commutative operation, that is $g\circ g' = g'\circ g$ for all $g, g' \in G$.

Example. $(\mathbb{Z}_n,+)$ and $(\mathbb{Z}_p^*,+)$ are abelian groups. The symmetries of an equilateral triangle are not an abelian group.

Note. The groups of interest in Cryptography II, usually arising from commutative number theory, are generally finite abelian groups. Abelian groups are often written with "additive" notation.

Definition. For a group element $g \in (G,\circ)$ of order n, so $g\circ\ldots\circ g = g^n = e$, the finite *cyclic* subgroup generated by g is $\langle g\rangle = \{g, g^2, \ldots, g^n = e\} \leq G$ of size n. A cyclic group of size n is often generically referred to as C_n.

Fundamental Theorem of Finite Abelian Groups. Every finite abelian group G can be expressed as the direct sum of cyclic subgroups of prime-power order.

Example. $C_6 = C_2 \times C_3 = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)\}$, where addition is the first component is $\bmod\ 2$ and the second component $\bmod\ 3$.

Finite Field Theorem. The multiplicative group $\mathbb{F}_{p^n}^*$ of a finite field $\mathbb{F}_{p^n}$ with p^n elements (p is prime) is cyclic, so C_{p^n-1}.
Note. Result shows existence of primitive roots modulo p.