Arithmetic Algorithms And Applications
Contents

Articles

Linear congruence theorem 1
Chinese remainder theorem 2
Discrete logarithm 10
Integer factorization 12
Pollard's rho algorithm 17
RSA (cryptosystem) 19
Diffie–Hellman key exchange 29

References

Article Sources and Contributors 36
Image Sources, Licenses and Contributors 37

Article Licenses

License 38
Linear congruence theorem

In modular arithmetic, the question of when a linear congruence can be solved is answered by the linear congruence theorem. If $a$ and $b$ are any integers and $n$ is a positive integer, then the congruence

$$ax \equiv b \pmod{n}$$

has a solution for $x$ if and only if $b$ is divisible by the greatest common divisor $d$ of $a$ and $n$ (denoted by $\gcd(a,n)$). When this is the case, and $x_0$ is one solution of (1), then the set of all solutions is given by

$$\{x_0 + k \frac{n}{d} \mid k \in \mathbb{Z}\}.$$ 

In particular, there will be exactly $d = \gcd(a,n)$ solutions in the set of residues $\{0,1,2,\ldots,n-1\}$. The result is a simple consequence of Bézout's identity.

Example

For example, examining the equation $ax \equiv 2 \pmod{6}$ with different values of $a$ yields

- $3x \equiv 2 \pmod{6}$
  
  Here $d = \gcd(3,6) = 3$ but since 3 does not divide 2, there is no solution.

- $5x \equiv 2 \pmod{6}$
  
  Here $d = \gcd(5,6) = 1$, which divides any $b$, and so there is just one solution in $\{0,1,2,3,4,5\}$: $x = 4$.

- $4x \equiv 2 \pmod{6}$
  
  Here $d = \gcd(4,6) = 2$, which does divide 2, and so there are exactly two solutions in $\{0,1,2,3,4,5\}$: $x = 2$ and $x = 5$.

Solving a linear congruence

In general solving equations of the form:

$$ax \equiv b \pmod{n}$$

If the greatest common divisor $d = \gcd(a,n)$ divides $b$, then we can find a solution $x$ to the congruence as follows: the extended Euclidean algorithm yields integers $r$ and $s$ such $ra + sn = d$. Then $x = rb/d$ is a solution. The other solutions are the numbers congruent to $x$ modulo $n/d$.

For example, the congruence

$$12x \equiv 20 \pmod{28}$$

has 4 solutions since $\gcd(12, 28) = 4$ divides 20. The extended Euclidean algorithm gives $(-2) \cdot 12 + 1 \cdot 28 = 4$, i.e. $r = -2$ and $s = 1$. Therefore, one solution is $x = -2 \cdot 20/4 = -10$, and $-10 \equiv 4 \pmod{7}$. All other solutions will also be congruent to 4 modulo 7. Since the original equation uses modulo 28, the entire solution set in the range from 0 to 27 is $\{4, 11, 18, 25\}$. 


Linear congruence theorem

System of linear congruences

By repeatedly using the linear congruence theorem, one can also solve systems of linear congruences, as in the following example: find all numbers \( x \) such that

\[
2x \equiv 2 \pmod{6} \\
3x \equiv 2 \pmod{7} \\
2x \equiv 4 \pmod{8}
\]

By solving the first congruence using the method explained above, we find \( x \equiv 1 \pmod{3} \), which can also be written as \( x = 3k + 1 \). Substituting this into the second congruence and simplifying, we get

\[
9k \equiv -1 \pmod{7}.
\]

Solving this congruence yields \( k \equiv 3 \pmod{7} \), or \( k = 7l + 3 \). It then follows that

\[
x = 3(7l + 3) + 1 = 21l + 10.
\]

Substituting this into the third congruence and simplifying, we get

\[
42l \equiv -16 \pmod{8}
\]

which has the solution \( l \equiv 0 \pmod{4} \), or \( l = 4m \). This yields \( x = 21(4m) + 10 = 84m + 10 \), or

\[
x \equiv 10 \pmod{84}
\]

which describes all solutions to the system.

Chinese remainder theorem

The Chinese remainder theorem is a result about congruences in number theory and its generalizations in abstract algebra. It was first published in the 3rd to 5th centuries by Chinese mathematician Sun Tzu.

In its basic form, the Chinese remainder theorem will determine a number \( n \) that when divided by some given divisors leaves given remainders.

For example, what is the lowest number \( n \) that when divided by 3 leaves a remainder of 2, when divided by 5 leaves a remainder of 3, and when divided by 7 leaves a remainder of 2? A common introductory example is a woman who tells a policeman that she lost her basket of eggs, and that if she makes three portions at a time out of it, she was left with 2, if she makes five portions at a time out of it, she was left with 3, and if she makes seven portions at a time out of it, she was left with 2. She then asks the policeman what is the minimum number of eggs she must have had. The answer to both problems is 23.

Theorem statement

The original form of the theorem, contained in the 5th-century book Sunzi's Mathematical Classic (孫子算經) by the Chinese mathematician Sun Tzu and later generalized with a complete solution called Dayanshu (大衍術) in Qin Jiushao's 1247 Mathematical Treatise in Nine Sections (數書九章, Shushu Jiuzhang), is a statement about simultaneous congruences.

Suppose \( n_1, n_2, \ldots, n_k \) are positive integers that are pairwise coprime. Then, for any given sequence of integers \( a_1, a_2, \ldots, a_k \), there exists an integer \( x \) solving the following system of simultaneous congruences.

\[
x \equiv a_1 \pmod{n_1} \\
x \equiv a_2 \pmod{n_2} \\
\vdots \\
x \equiv a_k \pmod{n_k}
\]

Furthermore, all solutions \( x \) of this system are congruent modulo the product, \( N = n_1 n_2 \ldots n_k \).
Hence \( x \equiv y \pmod{n_i} \) for all \( 1 \leq i \leq k \), if and only if \( x \equiv y \pmod{N} \).

Sometimes, the simultaneous congruences can be solved even if the \( n_i \)'s are not pairwise coprime. A solution \( x \) exists if and only if:

\[
a_i \equiv a_j \pmod{\gcd(n_i, n_j)} \quad \text{for all } i \text{ and } j
\]

All solutions \( x \) are then congruent modulo the least common multiple of the \( n_i \).

Sun Tzu's work contains neither a proof nor a full algorithm. What amounts to an algorithm for solving this problem was described by Aryabhata (6th century; see Kak 1986). Special cases of the Chinese remainder theorem were also known to Brahmagupta (7th century), and appear in Fibonacci's Liber Abaci (1202).

A modern restatement of the theorem in algebraic language is that for a positive integer \( n \) with prime factorization \( p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k} \) we have the isomorphism between a ring and the direct product of its prime power parts:

\[
\mathbb{Z}/n\mathbb{Z} \cong \mathbb{Z}/p_1^{a_1}\mathbb{Z} \times \mathbb{Z}/p_2^{a_2}\mathbb{Z} \times \cdots \times \mathbb{Z}/p_k^{a_k}\mathbb{Z}
\]

The theorem can also be restated in the language of combinatorics as the fact that the infinite arithmetic progressions of integers form a Helly family (Duchet 1995).

**Existence and Uniqueness**

The existence and uniqueness of the solution can easily be seen through a non-constructive argument. There are \( N = n_1 n_2 \cdots n_k \) different \( k \)-tuples of remainders. Let us call this set \( R \). And there are also \( N \) different numbers between 1 and \( N \). For each number between 1 and \( N \), there corresponds member of \( R \). Can two numbers \( a, b \), between 1 and \( N \) correspond to the same member of \( R \)? That is, can they have the same set of remainders when divided by \( n_1, n_2, \ldots, n_k \)? If they did then \( a - b \) would be divisible by each \( n \). Since the \( n \)'s are relatively prime, \( a - b \) would be divisible by their product: \( N \). This can't be. So this function from \{1, \ldots, N\} to \( R \) is one-to-one. Since \{1, \ldots, N\} and \( R \) have the same number of elements, this function must also be onto. Thus we have established the existence of a bijection.

Existence can be seen by an explicit construction of \( x \). We will use the notation \([a^{-1}]_b\) to denote the multiplicative inverse of \( a \pmod{b} \) as calculated by the Extended Euclidean algorithm. It is defined exactly when \( a \) and \( b \) are coprime; the following construction explains why the coprimality condition is needed.

**Case of two equations**

Given the system (corresponding to \( k = 2 \))

\[
x \equiv a_1 \pmod{n_1}
\]

\[
x \equiv a_2 \pmod{n_2}
\]

Since \( \gcd(n_1, n_2) = 1 \), we have from Bézout's identity

\[
n_2[n_2^{-1}]_{n_1} + n_1[n_1^{-1}]_{n_2} = 1
\]

This is true because we agreed to use the inverses that came out of the Extended Euclidean algorithm; for any other inverses, it would not necessarily hold true, but only hold true \( \pmod{n_1 n_2} \).

Multiplying both sides by \( x \), we get

\[
x = x n_2[n_2^{-1}]_{n_1} + x n_1[n_1^{-1}]_{n_2}
\]

If we take the congruence modulo \( n_1 \) for the right-hand-side expression, it is readily seen that

\[
x n_2[n_2^{-1}]_{n_1} + x n_1[n_1^{-1}]_{n_2} \equiv x \times 1 + x \times 0 \times [n_1^{-1}]_{n_2} \equiv x \pmod{n_1}
\]

But we know that

\[
x \equiv a_1 \pmod{n_1}
\]

thus this suggests that the coefficient of the first term on the right-hand-side expression can be replaced by \( a_1 \). Similarly, we can show that the coefficient of the second term can be substituted by \( a_2 \).
We can now define the value
\[ x \equiv a_1 n_2 [n_2^{-1}] n_1 + a_2 n_1 [n_1^{-1}] n_2 \]
and it is seen to satisfy both congruences by reducing. For example
\[ a_1 n_2 [n_2^{-1}] n_1 + a_2 n_1 [n_1^{-1}] n_2 \equiv a_1 \times 1 + a_2 \times 0 \times [n_1^{-1}] n_2 \equiv a_1 \pmod{n_1} \]

**General case**

The same type of construction works in the general case of \( k \) congruence equations. Let \( N = n_1n_2\ldots n_k \) be the product of every modulus then define
\[ x := \sum_i a_i \frac{N}{n_i} \left[ \left( \frac{N}{n_i} \right)^{-1} \right] n_i \]
and this is seen to satisfy the system of congruences by a similar calculation as before.

**Finding the solution with basic algebra and modular arithmetic**

For example, consider the problem of finding an integer \( x \) such that
\[ x \equiv 2 \pmod{3} \]
\[ x \equiv 3 \pmod{4} \]
\[ x \equiv 1 \pmod{5} \]
A brute-force approach converts these congruences into sets and writes the elements out to the product of \( 3 \times 4 \times 5 = 60 \) (the solutions modulo 60 for each congruence):
\[ x \in \{2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59, \ldots \} \]
\[ x \in \{3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, \ldots \} \]
\[ x \in \{1, 6, 11, 16, 21, 26, 31, 36, 41, 46, 51, 56, \ldots \} \]
To find an \( x \) that satisfies all three congruences, intersect the three sets to get:
\[ x \in \{11, \ldots \} \]
Which can be expressed as
\[ x \equiv 11 \pmod{60} \]

Another way to find a solution is with basic algebra, modular arithmetic, and stepwise substitution.

We start by translating these congruences into equations for some \( t \), \( s \), and \( u \):

- **Equation 1:** \( x = 2 + 3t \)
- **Equation 2:** \( x = 3 + 4s \)
- **Equation 3:** \( x = 1 + 5u \)

Start by substituting the \( x \) from equation 1 into congruence 2:
\[ 2 + 3t \equiv 3 \pmod{4} \]
\[ 3t \equiv 1 \pmod{4} \]
\[ t \equiv (3)^{-1} \equiv 3 \pmod{4} \]
meaning that \( t = 3 + 4s \) for some integer \( s \).

Substitute \( t \) into equation 1:
\[ x = 2 + 3t = 2 + 3(3 + 4s) = 11 + 12s \]
Substitute this \( x \) into congruence 3:
\[ 11 + 12s \equiv 1 \pmod{5} \]
Casting out fives, we get
$$1 + 2s \equiv 1 \pmod{5}$$

$$2s \equiv 0 \pmod{5}$$

meaning that

$$s = 0 + 5u$$

for some integer $u$.

Finally,

$$x = 11 + 12s = 11 + 12(5u) = 11 + 60u$$

So, we have solutions 11, 71, 131, 191, ...

Notice that 60 = lcm(3,4,5). If the moduli are pairwise coprime (as they are in this example), the solutions will be congruent modulo their product.

### A constructive algorithm to find the solution

The following algorithm only applies if the $n_i$’s are pairwise coprime. (For simultaneous congruences when the moduli are not pairwise coprime, the method of successive substitution can often yield solutions.)

Suppose, as above, that a solution is required for the system of congruences:

$$x \equiv a_i \pmod{n_i} \text{ for } i = 1, \ldots, k$$

Again, to begin, the product $N = n_1 n_2 \ldots n_k$ is defined. Then a solution $x$ can be found as follows.

For each $i$ the integers $n_i$ and $N/n_i$ are coprime. Using the extended Euclidean algorithm we can find integers $r_i$ and $s_i$ such that $r_i n_i + s_i N/n_i = 1$. Then, choosing the label $e_i = s_i N/n_i$, the above expression becomes:

$$r_i n_i + e_i = 1$$

Consider $e_i$. The above equation guarantees that its remainder, when divided by $n_i$, must be 1. On the other hand, since it is formed as $s_i N/n_i$, the presence of $N$ guarantees a remainder of zero when divided by any $n_j$ when $j \neq i$.

$$e_i \equiv 1 \pmod{n_i} \text{ and } e_i \equiv 0 \pmod{n_j} \text{ for } j \neq i$$

Because of this, and the multiplication rules allowed in congruences, one solution to the system of simultaneous congruences is:

$$x = \sum_{i=1}^{k} a_i e_i$$

For example, consider the problem of finding an integer $x$ such that

$$x \equiv 2 \pmod{3}$$

$$x \equiv 3 \pmod{4}$$

$$x \equiv 1 \pmod{5}$$

Using the extended Euclidean algorithm, for $x$ modulo 3 and 20 [4×5], we find $(-13) \times 3 + 2 \times 20 = 1$; i.e., $e_1 = 40$.

For $x$ modulo 4 and 15 [3×5], we get $(-11) \times 4 + 3 \times 15 = 1$, i.e. $e_2 = 45$. Finally, for $x$ modulo 5 and 12 [3×4], we get $5 \times 5 + (-2) \times 12 = 1$, i.e. $e_3 = -24$. A solution $x$ is therefore $2 \times 40 + 3 \times 45 + 1 \times (-24) = 191$. All other solutions are congruent to 191 modulo 60, $[3 \times 4 \times 5 = 60]$, which means they are all congruent to 11 modulo 60.

Note: There are multiple implementations of the extended Euclidean algorithm which will yield different sets of $e_1 = -20$, $e_2 = -15$, and $e_3 = -24$. These sets however will produce the same solution; i.e., $(-20)2 + (-15)3 + (-24)1 = -109 = 11$ modulo 60.
Statement for principal ideal domains

For a principal ideal domain $R$ the Chinese remainder theorem takes the following form: If $u_1, \ldots, u_k$ are elements of $R$ which are pairwise coprime, and $u$ denotes the product $u_1 \cdots u_k$, then the quotient ring $R/uR$ and the product ring $R/u_1R \times \cdots \times R/u_kR$ are isomorphic via the isomorphism

$$f : R/uR \rightarrow R/u_1R \times \cdots \times R/u_kR$$

such that

$$f(x + uR) = (x + u_1R, \ldots, x + u_kR) \quad \text{for every } x \in R$$

This map is well-defined and an isomorphism of rings; the inverse isomorphism can be constructed as follows. For each $i$, the elements $u_i$ and $u/u_i$ are coprime, and therefore there exist elements $r$ and $s$ in $R$ with

$$ru_i + su_i = 1$$

Set $e_i = s u_i/u_i$. Then the inverse of $f$ is the map

$$g : R/u_1R \times \cdots \times R/u_kR \rightarrow R/uR$$

such that

$$g(a_1 + u_1R, \ldots, a_k + u_kR) = \left( \sum_{i=1}^k a_i u_i^{-1} \left( \frac{u}{u_i} \right)^{-1} \right) u$$

for all $a_1, \ldots, a_k \in R$.

This statement is a straightforward generalization of the above theorem about integer congruences: the ring $\mathbb{Z}$ of integers is a principal ideal domain, the surjectivity of the map $f$ shows that every system of congruences of the form

$$x \equiv a_i \pmod{u_i} \quad \text{for } i = 1, \ldots, k$$

can be solved for $x$, and the injectivity of the map $f$ shows that all the solutions $x$ are congruent modulo $u$.

Statement for general rings

The general form of the Chinese remainder theorem, which implies all the statements given above, can be formulated for commutative rings and ideals. If $R$ is a commutative ring and $I_1, \ldots, I_k$ are ideals of $R$ that are pairwise coprime (meaning that $I_i + I_j = R$ for all $i \neq j$), then the product $I$ of these ideals is equal to their intersection, and the quotient ring $R/I$ is isomorphic to the product ring $R/I_1 \times R/I_2 \times \cdots \times R/I_k$ via the isomorphism

$$f : R/I \rightarrow R/I_1 \times \cdots \times R/I_k$$

such that

$$f(x + I) = (x + I_1, \ldots, x + I_k) \quad \text{for all } x \in R$$

Here is a version of the theorem where $R$ is not required to be commutative:

Let $R$ be any ring with 1 (not necessarily commutative) and $I_1, \ldots, I_k$ be pairwise coprime 2-sided ideals. Then the canonical $R$-module homomorphism $R \rightarrow R/I_1 \times \cdots \times R/I_k$ is onto, with kernel $I_1 \cap \cdots \cap I_k$. Hence, $R/(I_1 \cap \cdots \cap I_k) \simeq R/I_1 \times \cdots \times R/I_k$ (as $R$-modules).

Applications

- In the RSA algorithm calculations are made modulo $n$, where $n$ is a product of two large prime numbers $p$ and $q$. 1,024-, 2,048- or 4,096-bit integers $n$ are commonly used, making calculations in $\mathbb{Z}/n\mathbb{Z}$ very time-consuming. By the Chinese remainder theorem, however, these calculations can be done in the isomorphic ring $\mathbb{Z}/p\mathbb{Z} \oplus \mathbb{Z}/q\mathbb{Z}$ instead. Since $p$ and $q$ are normally of about the same size, that is about $\sqrt{n}$, calculations in the latter representation are much faster. Note that RSA algorithm implementations using this isomorphism are more susceptible to fault injection attacks.

- The Chinese remainder theorem may also be used to construct an elegant Gödel numbering for sequences, which is needed to prove Gödel's incompleteness theorems.
The following example shows a connection with the classic polynomial interpolation theory. Let \( r \) complex points ("interpolation nodes") \( \lambda_1, \ldots, \lambda_r \) be given, together with the complex data \( a_{jk} \) for all \( 1 \leq j \leq r \) and \( 0 \leq k < \nu_j \). The general Hermite interpolation problem asks for a polynomial \( P(x) \in \mathbb{C}[x] \) taking the prescribed derivatives in each node \( \lambda_j \):

\[
P^{(k)}(\lambda_j) = a_{jk}, \quad \forall 1 \leq j \leq r, \ 0 \leq k < \nu_j
\]

Introducing the polynomials

\[
A_j(x) := \sum_{k=0}^{\nu_j-1} \frac{a_{jk}}{k!} (x - \lambda_j)^k
\]

the problem may be equivalently reformulated as a system of \( r \) simultaneous congruences:

\[
P(x) \equiv A_j(x) \pmod{(x - \lambda_j)^{\nu_j}}, \quad \forall 1 \leq j \leq r
\]

By the Chinese remainder theorem in the principal ideal domain \( \mathbb{C}[x] \), there is a unique such polynomial \( P(x) \) with degree \( \deg(P) < n := \sum_j \nu_j \). A direct construction, in analogy with the above proof for the integer number case, can be performed as follows. Define the polynomials \( Q := \prod_{j=1}^r (x - \lambda_j)^{\nu_j} \) and \( Q_j := \frac{Q}{(x - \lambda_j)^{\nu_j}} \).

The partial fraction decomposition of \( \frac{1}{Q} \) gives \( r \) polynomials \( S_j \) with degrees \( \deg(S_j) < \nu_j \) such that

\[
\frac{1}{Q} = \sum_{j=1}^r \frac{S_j}{(x - \lambda_j)^{\nu_j}}
\]

so that \( \sum_{j=1}^r S_j Q_j = 1 \). Then a solution of the simultaneous congruence system is given by the polynomial

\[
\sum_{j=1}^r A_j S_j Q_j = A_j + \sum_{i=1}^r (A_i - A_j) S_i Q_i \equiv A_j \pmod{(x - \lambda_j)^{\nu_j}} \quad \forall 1 \leq j \leq r
\]

and the minimal degree solution is this one reduced modulo \( Q \), that is the unique with degree less than \( n \).

The Chinese remainder theorem can also be used in secret sharing, which consists of distributing a set of shares among a group of people who, all together (but no one alone), can recover a certain secret from the given set of shares. Each of the shares is represented in a congruence, and the solution of the system of congruences using the Chinese remainder theorem is the secret to be recovered. Secret Sharing using the Chinese Remainder Theorem uses, along with the Chinese remainder theorem, special sequences of integers that guarantee the impossibility of recovering the secret from a set of shares with less than a certain cardinality.

The Good-Thomas fast Fourier transform algorithm exploits a re-indexing of the data based on the Chinese remainder theorem. The Prime-factor FFT algorithm contains an implementation.

Dedekind’s theorem on the linear independence of characters states (in one of its most general forms) that if \( M \) is a monoid and \( k \) is an integral domain, then any finite family \( \{f_i\}_{i \in I} \) of distinct monoid homomorphisms \( f_i : M \to k \) (where the monoid structure on \( k \) is given by multiplication) is linearly independent; i.e., every family \( (\alpha_i)_{i \in I} \) of elements \( \alpha_i \in k \) satisfying \( \sum_{i \in I} \alpha_i f_i = 0 \) must be equal to the family \( (0)_{i \in I} \).

**Proof using the Chinese Remainder Theorem:** First, assume that \( k \) is a field (otherwise, replace the integral domain \( k \) by its quotient field, and nothing will change). We can linearly extend the monoid homomorphisms \( f_i : M \to k \) to \( k \) algebra homomorphisms \( F_i : k[M] \to k \), where \( k[M] \) is the monoid ring of \( M \) over \( k \). Then, the condition \( \sum_{i \in I} \alpha_i f_i = 0 \) yields \( \sum_{i \in I} \alpha_i F_i = 0 \) by linearity. Now, we notice that if \( i \neq j \) are two elements of the index set \( I \), then the two \( k \)-linear maps \( F_i : k[M] \to k \) and \( F_j : k[M] \to k \) are not proportional to each other (because if they were, then \( f_i \) and \( f_j \) would also be proportional to each other, and thus equal to each other since \( f_i(1) = f_j(1) \) (since \( f_i \) and \( f_j \) are monoid homomorphisms), contradicting the assumption that they be distinct). Hence, their kernels \( \text{Ker} F_i \) and \( \text{Ker} F_j \) are distinct. Now, \( \text{Ker} F_i \) is a maximal ideal of \( k[M] \) for every \( i \in I \) (since \( k[M]/\text{Ker} F_i \cong F_i(k[M]) = k \) is a field), and the ideals \( \text{Ker} F_i \) and \( \text{Ker} F_j \) are coprime whenever \( i \neq j \) (since they are distinct and maximal). The Chinese Remainder Theorem (for general rings) thus yields that the map
\[ \phi : k[M] / K \to \prod_{i \in I} k[M] / \text{Ker} F_i \]
given by
\[ \phi(x + K) = (x + \text{Ker} F_i)_{i \in I} \]
for all \( x \in k[M] \)
is an isomorphism, where \( K = \prod_{i \in I} \text{Ker} F_i = \bigcap_{i \in I} \text{Ker} F_i \). Consequently, the map
\[ \Phi : k[M] \to \prod_{i \in I} k[M] / \text{Ker} F_i \]
given by
\[ \Phi(x) = (x + \text{Ker} F_i)_{i \in I} \]
for all \( x \in k[M] \)
is surjective. Under the isomorphisms \( k[M]/\text{Ker} F_i \to \text{F}_i(k[M]) = k \), this map \( \Phi \) corresponds to the map
\[ \psi : k[M] \to \prod_{i \in I} k \]
given by
\[ x \mapsto [F_i(x)]_{i \in I} \]
for every \( x \in k[M] \).
Now, \( \sum_{i \in I} \alpha_i F_i = 0 \) yields \( \sum_{i \in I} \alpha_i u_i = 0 \) for every vector \( (u_i)_{i \in I} \) in the image of the map \( \psi \). Since \( \psi \) is surjective, this means that \( \sum_{i \in I} \alpha_i u_i = 0 \) for every vector \( (u_i)_{i \in I} \in \prod_{i \in I} k \). Consequently, \( (\alpha_i)_{i \in I} = (0)_{i \in I} \).
QED.

**Non-commutative case: a caveat**

Sometimes in the commutative case, the conclusion of the Chinese Remainder Theorem is stated as
\[ R/(I_1 I_2 \ldots I_k) \cong R/I_1 \times \ldots \times R/I_k \]
This version does not hold in the non-commutative case, since \( (I_1 \cap \ldots \cap I_k) \neq (I_1 I_2 \ldots I_k) \), as can be seen from the following example
Consider the ring \( R \) of non-commutative real polynomials in \( x \) and \( y \). Let \( I \) be the principal two-sided ideal generated by \( x \) and \( J \) the principal two-sided ideal generated by \( xy + 1 \). Then \( I + J = R \) but \( I \cap J \neq IJ \).

**Proof**

Observe that \( I \) is formed by all polynomials with an \( x \) in every term and that every polynomial in \( J \) vanishes under the substitution \( y = -1/x \). Consider the polynomial \( p = (xy + 1)x \). Clearly \( p \in I \cap J \). Define a term in \( R \) as an element of the multiplicative monoid of \( R \) generated by \( x \) and \( y \). Define the degree of a term as the usual degree of the term after the substitution \( y = x \). On the other hand, suppose \( q \subseteq J \). Observe that a term in \( q \) of maximum degree depends on \( y \) otherwise \( q \) under the substitution \( y = -1/x \) can not vanish. The same happens then for an element \( q \in J \). Observe that the last \( y \), from left to right, in a term of maximum degree in an element of \(IJ\) is preceded by more than one \( x \). (We are counting here all the preceding \( xs \). E.g., in \( x^2 y^2 x^2 \) the last \( y \) is preceded by \( 3xs \).) This proves that \((xy + 1)x \notin IJ \) since that last \( y \) in a term of maximum degree \((xyx)\) is preceded by only one \( x \). Hence \( I \cap J \neq IJ \).

On the other hand, it is true in general that \( I + J = R \) implies \( I \cap J = IJ + JI \). To see this, note that \( I \cap J = (I \cap J)(I + J) \subseteq IJ + JI \), while the opposite inclusion is obvious. Also, we have in general that, provided \( I_1, \ldots, I_m \) are pairwise coprime two-sided ideals in \( R \), the natural map
\[ R/(I_1 \cap I_2 \cap \ldots \cap I_m) \to R/I_1 \oplus R/I_2 \oplus \cdots \oplus R/I_m \]
is an isomorphism. Note that \( I_1 \cap I_2 \cap \ldots \cap I_m \) can be replaced by a sum over all orderings of \( I_1, \ldots, I_m \) of their product (or just a sum over enough orderings, using inductively that \( I \cap J = IJ + JI \) for coprime ideals \( I, J \)).
References


External links

- Weisstein, Eric W., "Chinese Remainder Theorem" [6], *MathWorld*.
- C# program and discussion [7] at codeproject
- University of Hawaii System [8] CRT by Lee Lady
- Full text of the Sunzi Suanjing [9] (Chinese) — Chinese Text Project

References

Discrete logarithm

In mathematics, a **discrete logarithm** is an integer \( k \) solving the equation \( b^k = g \), where \( b \) and \( g \) are elements of a group. Discrete logarithms are thus the group-theoretic analogue of ordinary logarithms, which solve the same equation for real numbers \( b \) and \( g \), where \( b \) is the base of the logarithm and \( g \) is the value whose logarithm is being taken.

Computing discrete logarithms is believed to be difficult. No efficient general method for computing discrete logarithms on conventional computers is known, and several important algorithms in public-key cryptography base their security on the assumption that the discrete logarithm problem has no efficient solution.

**Example**

Discrete logarithms are perhaps simplest to understand in the group \((\mathbb{Z}_p)^\times\). This is the group of multiplication modulo the prime \( p \); its elements are congruence classes modulo \( p \), and the group product of two elements may be obtained by ordinary integer multiplication of the elements followed by reduction modulo \( p \).

The \( k \)th power of one of the numbers in this group may be computed by finding its \( k \)th power as an integer and then finding the remainder after division by \( p \). This process is called modular exponentiation. For example, consider \((\mathbb{Z}_{17})^\times\). To compute \( 3^4 \) in this group, compute \( 3^4 = 81 \), and then divide 81 by 17, obtaining a remainder of 13. Thus \( 3^4 = 13 \) in the group \((\mathbb{Z}_{17})^\times\).

The discrete logarithm is just the inverse operation. For example, consider the equation \( 3^k \equiv 13 \pmod{17} \) for \( k \). From the example above, one solution is \( k = 4 \) is a solution, but it is not the only solution. Since \( 3^{16} \equiv 1 \pmod{17} \)— as follows from Fermat’s little theorem—it also follows that if \( n \) is an integer then \( 3^{k+16n} \equiv 3^4 \times (3^{16})^n \equiv 13 \times 1^n \equiv 13 \pmod{17} \). Hence the equation has infinitely many solutions of the form \( 4 + 16n \). Moreover, since 16 is the smallest positive integer \( m \) satisfying \( 3^m \equiv 1 \pmod{17} \), i.e. 16 is the order of 3 in \((\mathbb{Z}_{17})^\times\), these are the only solutions. Equivalently, the set of all possible solutions can be expressed by the constraint that \( k \equiv 4 \pmod{16} \).

**Definition**

In general, let \( G \) be any group, with its group operation denoted by multiplication. Let \( b \) and \( g \) be any elements of \( G \). Then any integer \( k \) that solves \( b^k = g \) is termed a **discrete logarithm** (or simply **logarithm**, in this context) of \( g \) to the base \( b \). We write \( k = \log_b g \). Depending on \( b \) and \( g \), it is possible that no discrete logarithm exists, or that more than one discrete logarithm exists. Let \( H \) be the subgroup of \( G \) generated by \( b \). Then \( H \) is a cyclic group, and \( \log_b g \) exists for all \( g \) in \( H \). If \( H \) is infinite, then \( \log_b g \) is also unique, and the discrete logarithm amounts to a group isomorphism

\[
\log_b : H \to \mathbb{Z}.
\]

On the other hand, if \( H \) is finite of size \( n \), then \( \log_b g \) is unique only up to congruence modulo \( n \), and the discrete logarithm amounts to a group isomorphism

\[
\log_b : H \to \mathbb{Z}_n,
\]

where \( \mathbb{Z}_n \) denotes the ring of integers modulo \( n \). The familiar base change formula for ordinary logarithms remains valid: If \( c \) is another generator of \( H \), then

\[
\log_c(g) = \log_n(b) \cdot \log_b(g).
\]
Algorithms

List of unsolved problems in computer science

| Can the discrete logarithm be computed in polynomial time on a classical computer? |

No efficient classical algorithm for computing general discrete logarithms \( \log_b g \) is known. The naive algorithm is to raise \( b \) to higher and higher powers \( k \) until the desired \( g \) is found; this is sometimes called \textit{trial multiplication}. This algorithm requires running time linear in the size of the group \( G \) and thus exponential in the number of digits in the size of the group. There exists an efficient quantum algorithm due to Peter Shor.

More sophisticated algorithms exist, usually inspired by similar algorithms for integer factorization. These algorithms run faster than the naive algorithm, some of them linear in the \textit{square root} of the size of the group, and thus exponential in half the number of digits in the size of the group. However none of them runs in polynomial time (in the number of digits in the size of the group).

- Baby-step giant-step
- Pollard's rho algorithm for logarithms
- Pollard's kangaroo algorithm (aka Pollard's lambda algorithm)
- Pohlig–Hellman algorithm
- Index calculus algorithm
- Number field sieve
- Function field sieve

Comparison with integer factorization

While computing discrete logarithms and factoring integers are distinct problems, they share some properties:

- both problems are difficult (no efficient algorithms are known for non-quantum computers),
- for both problems efficient algorithms on quantum computers are known,
- algorithms from one problem are often adapted to the other, and
- the difficulty of both problems has been used to construct various cryptographic systems.

Cryptography

There exist groups for which computing discrete logarithms is apparently difficult. In some cases (e.g. large prime order subgroups of groups \( (\mathbb{Z}_p)^* \)) there is not only no efficient algorithm known for the worst case, but the average-case complexity can be shown to be about as hard as the worst case using random self-reducibility.

At the same time, the inverse problem of discrete exponentiation is not difficult (it can be computed efficiently using exponentiation by squaring, for example). This asymmetry is analogous to the one between integer factorization and integer multiplication. Both asymmetries have been exploited in the construction of cryptographic systems.

Popular choices for the group \( G \) in discrete logarithm cryptography are the cyclic groups \( (\mathbb{Z}_p)^* \) (e.g. ElGamal encryption, Diffie–Hellman key exchange, and the Digital Signature Algorithm) and cyclic subgroups of elliptic curves over finite fields (see elliptic curve cryptography).
Discrete logarithm

References

• Richard Crandall; Carl Pomerance. Chapter 5, Prime Numbers: A computational perspective, 2nd ed., Springer.

Integer factorization

In number theory, integer factorization or prime factorization is the decomposition of a composite number into smaller non-trivial divisors, which when multiplied together equal the original integer.

When the numbers are very large, no efficient, non-quantum integer factorization algorithm is known; an effort by several researchers concluded in 2009, factoring a 232-digit number (RSA-768), utilizing hundreds of machines over a span of 2 years. The presumed difficulty of this problem is at the heart of widely used algorithms in cryptography such as RSA. Many areas of mathematics and computer science have been brought to bear on the problem, including elliptic curves, algebraic number theory, and quantum computing.

Not all numbers of a given length are equally hard to factor. The hardest instances of these problems (for currently known techniques) are semiprimes, the product of two prime numbers. When they are both large, for instance more than 2000 bits long, randomly chosen, and about the same size (but not too close, e.g. to avoid efficient factorization by Fermat's factorization method), even the fastest prime factorization algorithms on the fastest computers can take enough time to make the search impractical; that is, as the number of digits of the primes being factored increases, the number of operations required to perform the factorization on any computer increases drastically.

Many cryptographic protocols are based on the difficulty of factoring large composite integers or a related problem - for example, the RSA problem. An algorithm that efficiently factors an arbitrary integer would render RSA-based public-key cryptography insecure.

Prime decomposition

By the fundamental theorem of arithmetic, every positive integer has a unique prime factorization. (A special case for 1 is not needed using an appropriate notion of the empty product.) However, the fundamental theorem of arithmetic gives no insight into how to obtain an integer's prime factorization; it only guarantees its existence.

Given a general algorithm for integer factorization, one can factor any integer down to its constituent prime factors by repeated application of this algorithm. However, this is not the case with a special-purpose factorization algorithm, since it may not apply to the smaller factors that occur during decomposition, or may execute very slowly on these values. For example, if N is the number \((2^{521} - 1) \times (2^{607} - 1)\), then trial division will quickly factor 10N as \(2 \times 5 \times N\), but will not quickly factor N into its factors.
Current state of the art

The most difficult integers to factor in practice using existing algorithms are those that are products of two large primes of similar size, and for this reason these are the integers used in cryptographic applications. The largest such semiprime yet factored was RSA-768, a 768-bit number with 232 decimal digits, on December 12, 2009. This factorization was a collaboration of several research institutions, spanning two years and taking the equivalent of almost 2000 years of computing on a single-core 2.2 GHz AMD Opteron. Like all recent factorization records, this factorization was completed with a highly optimized implementation of the general number field sieve run on hundreds of machines.

Difficulty and complexity

If a large, $b$-bit number is the product of two primes that are roughly the same size, then no algorithm has been published that can factor in polynomial time, i.e., that can factor it in time $O(b^k)$ for some constant $k$. There are published algorithms that are faster than $O((1+\varepsilon)^b)$ for all positive $\varepsilon$, i.e., sub-exponential.

The best published asymptotic running time is for the general number field sieve (GNFS) algorithm, which, for a $b$-bit number $n$, is:

$$O\left(\exp\left(\left(\frac{64}{9}b\right)^{\frac{1}{3}}(\log b)^{\frac{2}{3}}\right)\right).$$

For an ordinary computer, GNFS is the best published algorithm for large $n$ (more than about 100 digits). For a quantum computer, however, Peter Shor discovered an algorithm in 1994 that solves it in polynomial time. This will have significant implications for cryptography if a large quantum computer is ever built. Shor's algorithm takes only $O(b^{\frac{3}{2}})$ time and $O(b)$ space on $b$-bit number inputs. In 2001, the first seven-qubit quantum computer became the first to run Shor’s algorithm. It factored the number 15.

When discussing what complexity classes the integer factorization problem falls into, it's necessary to distinguish two slightly different versions of the problem:

- The function problem version: given an integer $N$, find an integer $d$ with $1 < d < N$ that divides $N$ (or conclude that $N$ is prime). This problem is trivially in FNP and it's not known whether it lies in FP or not. This is the version solved by most practical implementations.

- The decision problem version: given an integer $N$ and an integer $M$ with $1 \leq M \leq N$, does $N$ have a factor $d$ with $1 < d < M$? This version is useful because most well-studied complexity classes are defined as classes of decision problems, not function problems. This is a natural decision version of the problem, analogous to those frequently used for optimization problems, because it can be combined with binary search to solve the function problem version in a logarithmic number of queries.

It is not known exactly which complexity classes contain the decision version of the integer factorization problem. It is known to be in both NP and co-NP. This is because both YES and NO answers can be verified in polynomial time given the prime factors (we can verify their primality using the AKS primality test, and that their product is $N$ by multiplication). The fundamental theorem of arithmetic guarantees that there is only one possible string that will be accepted (providing the factors are required to be listed in order), which shows that the problem is in both UP and co-UP. It is known to be in BQP because of Shor's algorithm. It is suspected to be outside of all three of the complexity classes P, NP-complete, and co-NP-complete. It is therefore a candidate for the NP-intermediate complexity class. If it could be proved that it is in either NP-Complete or co-NP-Complete, that would imply NP = co-NP. That would be a very surprising result, and therefore integer factorization is widely suspected to be outside both of those classes. Many people have tried to find classical polynomial-time algorithms for it and failed, and therefore it is widely suspected to be outside P.

In contrast, the decision problem "is $N$ a composite number?" (or equivalently: “is $N$ a prime number?”) appears to be much easier than the problem of actually finding the factors of $N$. Specifically, the former can be solved in polynomial time (in the number $n$ of digits of $N$) with the AKS primality test. In addition, there are a number of
probabilistic algorithms that can test primality very quickly in practice if one is willing to accept the vanishingly small possibility of error. The ease of primality testing is a crucial part of the RSA algorithm, as it is necessary to find large prime numbers to start with.

Factoring algorithms

Special-purpose

A special-purpose factoring algorithm's running time depends on the properties of the number to be factored or on one of its unknown factors: size, special form, etc. Exactly what the running time depends on varies between algorithms.

An important subclass of special-purpose factoring algorithms is the Category 1 or First Category algorithms, whose running time depends on the size of smallest prime factor. Given an integer of unknown form, these methods are usually applied before general-purpose methods to remove small factors. For example, trial division is a Category 1 algorithm.

- Trial division
- Wheel factorization
- Pollard's rho algorithm
- Algebraic-group factorisation algorithms, among which are Pollard's $p - 1$ algorithm, Williams' $p + 1$ algorithm, and Lenstra elliptic curve factorization
- Fermat's factorization method
- Euler's factorization method
- Special number field sieve

General-purpose

A general-purpose factoring algorithm, also known as a Category 2, Second Category, or Kraitchik family algorithm (after Maurice Kraitchik), has a running time depends solely on the size of the integer to be factored. This is the type of algorithm used to factor RSA numbers. Most general-purpose factoring algorithms are based on the congruence of squares method.

- Dixon's algorithm
- Continued fraction factorization (CFRAC)
- Quadratic sieve
- Rational sieve
- General number field sieve
- Shanks' square forms factorization (SQUFOF)

Other notable algorithms

- Shor's algorithm, for quantum computers

Heuristic running time

In number theory, there are many integer factoring algorithms that heuristically have expected running time

$$L_n \left[ 1/2, 1 + o(1) \right] = e^{(1+o(1))(\log n)^{1/2}(\log \log n)^{1/2}}$$

in $o$ and $L$-notation. Some examples of those algorithms are the elliptic curve method and the quadratic sieve. Another such algorithm is the class group relations method proposed by Schnorr, Seysen, and Lenstra that is proved under of the Generalized Riemann Hypothesis (GRH).
**Integer factorization**

The Schnorr-Seysen-Lenstra probabilistic algorithm has been rigorously proven by Lenstra and Pomerance to have expected running time $L_n[1/2, 1 + o(1)]$ by replacing the GRH assumption with the use of multipliers. The algorithm uses the class group of positive binary quadratic forms of discriminant $\Delta$ denoted by $G_\Delta$. $G_\Delta$ is the set of triples of integers $(a, b, c)$ in which those integers are relative prime.

**Schnorr-Seysen-Lenstra Algorithm**

Given is an integer $n$ that will be factored, where $n$ is an odd positive integer greater than a certain constant. In this factoring algorithm the discriminant $\Delta$ is chosen as a multiple of $n$, $\Delta = -dn$, where $d$ is some positive multiplier. The algorithm expects that for one $d$ there exist enough smooth forms in $G_\Delta$. Lenstra and Pomerance show that the choice of $d$ can be restricted to a small set to guarantee the smoothness result.

Denote by $P_\Delta$ the set of all primes $q$ with Kronecker symbol $\left( \frac{\Delta}{q} \right) = 1$. By constructing a set of generators of $G_\Delta$ and prime forms $f_q$ of $G_\Delta$ with $q$ in $P_\Delta$ a sequence of relations between the set of generators and $f_q$ are produced. The size of $q$ can be bounded by $c_0(\log |\Delta|)^2$ for some constant $c_0$.

The relation that will be used is a relation between the product of powers that is equal to the neutral element of $G_\Delta$. These relations will be used to construct a so-called ambiguous form of $G_\Delta$, which is an element of $G_\Delta$ of order dividing 2. By calculating the corresponding factorization of $\Delta$ and by taking a gcd, this ambiguous form provides the complete prime factorization of $n$. This algorithm has these main steps:

1. Let $n$ be the number to be factored.
2. Let $\Delta$ be a negative integer with $\Delta = -dn$, where $d$ is a multiplier and $\Delta$ is the negative discriminant of some quadratic form.
3. Take the $t$ first primes $p_1 = 2, p_2 = 3, p_3 = 5, \ldots, p_t$, for some $t \in \mathbb{N}$.
4. Let $f_q$ be a random prime form of $G_\Delta$ with $\left( \frac{\Delta}{q} \right) = 1$.
5. Find a generating set $X$ of $G_\Delta$.
6. Collect a sequence of relations between set $X$ and $\{f_q : q \in P_\Delta\}$ satisfying: $\left( \prod_{x \in X} x^{r(x)} \right) \cdot \left( \prod_{q \in P_\Delta} f_q^{t(q)} \right) = 1$.
7. Construct an ambiguous form $(a, b, c)$ that is an element $f \in G_\Delta$ of order dividing 2 to obtain a coprime factorization of the largest odd divisor of $\Delta$ in which $\Delta = -4ac$ or $a(a - 4c)$ or $(b - 2a)(b + 2a)$.
8. If the ambiguous form provides a factorization of $n$ then stop, otherwise find another ambiguous form until the factorization of $n$ is found. In order to prevent useless ambiguous forms from generating, build up the 2-Sylow group $S_2(\Delta)$ of $G(\Delta)$.

To obtain an algorithm for factoring any positive integer, it is necessary to add a few steps to this algorithm such as trial division, and the Jacobi sum test.
Expected running time

The algorithm as stated is a probabilistic algorithm as it makes random choices. Its expected running time is at most $L_n[1/2, 1 + o(1)]$.

Notes

References


External links

• msieve (http://sourceforge.net/projects/msieve/) - SIQS and NFS - has helped complete some of the largest public factorizations known

• Video (http://www.youtube.com/watch?v=5kl28hmhin0) explaining uniqueness of prime factorization using a lock analogy.

• A collection of links to factoring programs (http://www.mersenneforum.org/showthread.php?t=3255)


• (ftp://ftp.computing.dcu.ie/pub/crypto/factor.exe) is a public-domain integer factorization program for Windows. It claims to handle 80-digit numbers. See also the web site for this program MIRACL (http://www.shamus.ie/)

• The RSA Challenge Numbers (http://www.rsasecurity.com/rsalabs/node.asp?id=2093) - a factoring challenge, no longer active.

**Pollard's rho algorithm**

**Pollard's rho algorithm** is a general-purpose integer factorization algorithm. It was invented by John Pollard in 1975. It is particularly effective at splitting composite numbers with small factors.

**Core ideas**

The rho algorithm is based on Floyd's cycle-finding algorithm and on the observation that (as in the birthday problem) two numbers \( x \) and \( y \) are congruent modulo \( p \) with probability 0.5 after \( 1.177 \sqrt{p} \) numbers have been randomly chosen. If \( p \) is a factor of \( n \), the integer we are aiming to factor, then \( p \leq \gcd( x - y, n ) \leq n \) since \( p \) divides both \( x - y \) and \( n \).

The rho algorithm therefore uses a function modulo \( n \) as a generator of a pseudo-random sequence. It runs one sequence twice as "fast" as the other; i.e. for every iteration made by one copy of the sequence, the other copy makes two iterations. Let \( x \) be the current state of one sequence and \( y \) be the current state of the other. The GCD of \( x - y \) and \( n \) is taken at each step. If this GCD ever comes to \( n \), then the algorithm terminates with failure, since this means \( x = y \) and therefore, by Floyd's cycle-finding algorithm, the sequence has cycled and continuing any further would only be repeating previous work.

**Algorithm**

The algorithm takes as its inputs \( n \), the integer to be factored; and \( f \), a function with the property that \( x = y \mod p \) implies \( f(x) = f(y) \mod p \). In the original algorithm, \( f(x) = x^2 - 1 \mod n \). The output is either a non-trivial factor of \( n \), or failure. It performs the following steps:\[1]\]

1. \( x \leftarrow 2, \ y \leftarrow 2; \ d \leftarrow 1 \)
2. While \( d = 1 \):
   1. \( x \leftarrow f(x) \)
   2. \( y \leftarrow f(f(y)) \)
   3. \( d \leftarrow \gcd(x - y, n) \)
3. If \( d = n \), return failure.
4. Else, return \( d \).

Note that this algorithm may not find the factors and will return failure for composite \( n \). In that case, use a different \( f(x) \) and try again. Note, as well, that this algorithm does not work when \( n \) is a prime number, since, in this case, \( d \) will be always 1. The algorithm is so-called because the values of \( f \) enter a period \( \mod d \), resulting in a \( \rho \) shape when diagrammed.

**Variants**

In 1980, Richard Brent published a faster variant of the rho algorithm. He used the same core ideas as Pollard but a different method of cycle detection, replacing Floyd's cycle-finding algorithm with the related Brent's cycle finding method.

A further improvement was made by Pollard and Brent. They observed that if \( \gcd(a, n) > 1 \), then also \( \gcd(ab, n) > 1 \) for any positive integer \( b \). In particular, instead of computing \( \gcd( |x - y|, n ) \) at every step, it suffices to define \( z \) as the product of 100 consecutive \( |x - y| \) terms modulo \( n \), and then compute a single \( \gcd(z, n) \). A major speed up results as 100 \( \gcd \) steps are replaced with 99 multiplications modulo \( n \) and a single \( \gcd \). Occasionally it may cause the algorithm to fail by introducing a repeated factor, for instance when \( n \) is a square. But it then suffices to go back to the previous \( \gcd \) term, where \( \gcd(z, n) = 1 \), and use the regular Rho algorithm from there.
Pollard's rho algorithm

Application

The algorithm is very fast for numbers with small factors, but slower in cases where all factors are large. The rho algorithm's most remarkable success has been the factorization of the eighth Fermat number ($F_8$) by Pollard and Brent. They used Brent's variant of the algorithm, which found a previously unknown prime factor. The complete factorization of $F_8$ took, in total, 2 hours on a UNIVAC 1100/42.

Example factorization

Let $n = 8051$ and $f(x) = (x^2 + 1) \mod 8051$.

| $i$ | $x_i$ | $y_i$ | $\text{GCD}(|x_i - y_i|, 8051)$ |
|-----|-------|-------|---------------------------------|
| 1   | 5     | 26    | 1                               |
| 2   | 26    | 7474  | 1                               |
| 3   | 677   | 871   | 97                              |

97 is a non-trivial factor of 8051. Other values of $c$ may give the cofactor (83) instead of 97.

Complexity

The algorithm offers a trade-off between its running time and the probability that it finds a factor. If the squaring function used in the Pollard rho method were replaced by a random function, it would follow that, for all $n$, running the algorithm for $O(n^{1/4})$ steps would yield a factor with probability at most $1/2$. It is believed that the same analysis applies as well to the actual rho algorithm, but this is a heuristic claim, and rigorous analysis of the algorithm remains open.

References

[1] (this section discusses only Pollard's rho algorithm).

Additional reading

• Katz, Jonathan; Lindell, Yehuda (2007), "Chapter 8", Introduction to Modern Cryptography, CRC Press

External links

• Java Implementation (http://www.cs.princeton.edu/introcs/78crypto/PollardRho.java.html)
RSA (cryptosystem)

<table>
<thead>
<tr>
<th>RSA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
</tr>
<tr>
<td>Designers</td>
</tr>
<tr>
<td>First published</td>
</tr>
<tr>
<td>Certification</td>
</tr>
<tr>
<td><strong>Cipher detail</strong></td>
</tr>
<tr>
<td>Key sizes</td>
</tr>
<tr>
<td>Rounds</td>
</tr>
<tr>
<td>Best public cryptanalysis</td>
</tr>
</tbody>
</table>

RSA is a cryptosystem, which is known as one of the first practicable public-key cryptosystems and is yet widely used for secure data transmission. In such a cryptosystem, the encryption key is public and differs from the decryption key which is kept secret. In RSA, this asymmetry is based on the practical difficulty of factoring the product of two large prime numbers, the factoring problem. RSA stands for Ron Rivest, Adi Shamir and Leonard Adleman, who first publicly described the algorithm in 1977. Clifford Cocks, an English mathematician, had developed an equivalent system in 1973, but it wasn't declassified until 1997.

A user of RSA creates and then publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The prime factors must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime factors can feasibly decode the message. Whether breaking RSA encryption is as hard as factoring is an open question known as the RSA problem.
History

The RSA algorithm was publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT; the letters RSA are the initials of their surnames, listed in the same order as on the paper.[1]

MIT was granted U.S. Patent 4,405,829 [2] for a "Cryptographic communications system and method" that used the algorithm in 1983. The patent would have expired on September 21, 2000 (the term of patent was 17 years at the time), but the algorithm was released to the public domain by RSA Security on September 6, 2000, two weeks earlier.[3] Since a paper describing the algorithm had been published in August 1977, prior to the December 1977 filing date of the patent application, regulations in much of the rest of the world precluded patents elsewhere and only the US patent was granted. Had Cocks' work been publicly known, a patent in the US might not have been possible, either.

From the DWPI's abstract of the patent,

The system includes a communications channel coupled to at least one terminal having an encoding device and to at least one terminal having a decoding device. A message-to-be-transferred is enciphered to ciphertext at the encoding terminal by encoding the message as a number M in a predetermined set. That number is then raised to a first predetermined power (associated with the intended receiver) and finally computed. The remainder or residue, C, is... computed when the exponentiated number is divided by the product of two predetermined prime numbers (associated with the intended receiver).

Clifford Cocks, an English mathematician working for the UK intelligence agency GCHQ, described an equivalent system in an internal document in 1973 but, given the relatively expensive computers needed to implement it at the time, it was mostly considered a curiosity and, as far as is publicly known, was never deployed. His discovery, however, was not revealed until 1998 due to its top-secret classification, and Rivest, Shamir, and Adleman devised RSA independently of Cocks' work.

Operation

The RSA algorithm involves three steps: key generation, encryption and decryption.

Key generation

RSA involves a public key and a private key. The public key can be known by everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted in a reasonable amount of time using the private key. The keys for the RSA algorithm are generated the following way:

1. Choose two distinct prime numbers p and q.
   - For security purposes, the integers p and q should be chosen at random, and should be of similar bit-length. Prime integers can be efficiently found using a primality test.
2. Compute \( n = pq \).
   - \( n \) is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.
3. Compute \( \varphi(n) = \varphi(p)\varphi(q) = (p - 1)(q - 1) \), where \( \varphi \) is Euler's totient function.
4. Choose an integer \( e \) such that \( 1 < e < \phi(n) \) and \( \gcd(e, \phi(n)) = 1 \); i.e. \( e \) and \( \phi(n) \) are coprime.
   
   - \( e \) is released as the public key exponent.
   - \( e \) having a short bit-length and small Hamming weight results in more efficient encryption – most commonly \( 2^{16} + 1 = 65,537 \). However, much smaller values of \( e \) (such as 3) have been shown to be less secure in some settings.

5. Determine \( d \) as \( d^{-1} \equiv e \pmod{\phi(n)} \), i.e., \( d \) is the multiplicative inverse of \( e \) (modulo \( \phi(n) \)).
   
   - This is more clearly stated as: solve for \( d \) given \( d \cdot e \equiv 1 \pmod{\phi(n)} \)
   - This is often computed using the extended Euclidean algorithm.
   - \( d \) is kept as the private key exponent.

The public key consists of the modulus \( n \) and the public (or encryption) exponent \( e \). The private key consists of the modulus \( n \) and the private (or decryption) exponent \( d \), which must be kept secret. \( p \), \( q \), and \( \phi(n) \) must also be kept secret because they can be used to calculate \( d \).

- An alternative, used by PKCS#1, is to choose \( d \) matching \( de \equiv 1 \pmod{\lambda} \) with \( \lambda = \operatorname{lcm}(p - 1, q - 1) \), where \( \operatorname{lcm} \) is the least common multiple. Using \( \lambda \) instead of \( \phi(n) \) allows more choices for \( d \). \( \lambda \) can also be defined using the Carmichael function, \( \lambda(n) \).
- The ANSI X9.31 standard prescribes, IEEE 1363 describes, and PKCS#1 allows, that \( p \) and \( q \) match additional requirements: being strong primes, and being different enough that Fermat factorization fails.

**Encryption**

Alice transmits her public key \((n, e)\) to Bob and keeps the private key secret. Bob then wishes to send message \( M \) to Alice.

He first turns \( M \) into an integer \( m \), such that \( 0 \leq m < n \) by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext \( c \) corresponding to

\[
c \equiv m^e \pmod{n}.
\]

This can be done quickly using the method of exponentiation by squaring. Bob then transmits \( c \) to Alice.

**Decryption**

Alice can recover \( m \) from \( c \) by using her private key exponent \( d \) via computing

\[
m \equiv c^d \pmod{n}.
\]

Given \( m \), she can recover the original message \( M \) by reversing the padding scheme.

(In practice, there are more efficient methods of calculating \( c^d \) using the precomputed values below.)

**A working example**

Here is an example of RSA encryption and decryption. The parameters used here are artificially small, but one can also use OpenSSL to generate and examine a real keypair.

1. Choose two distinct prime numbers, such as \( p = 61 \) and \( q = 53 \).
2. Compute \( n = pq \) giving \( n = 61 \times 53 = 3233 \).
3. Compute the totient of the product as \( \phi(n) = (p - 1)(q - 1) \) giving \( \phi(3233) = (61 - 1)(53 - 1) = 3120 \).
4. Choose any number \( 1 < e < 3120 \) that is coprime to 3120. Choosing a prime number for \( e \) leaves us only to check that \( e \) is not a divisor of 3120.
RSA (cryptosystem)

Let \( e = 17 \).
5. Compute \( d \), the modular multiplicative inverse of \( e \pmod{\phi(n)} \) yielding \( d = 2753 \).

The **public key** is \((\text{p} = 3233, e = 17)\). For a padded plaintext message \( m \), the encryption function is

\[
c(m) = m^{17} \mod 3233.
\]

The **private key** is \((n = 3233, d = 2753)\). For an encrypted ciphertext \( c \), the decryption function is

\[
m(c) = c^{2753} \mod 3233.
\]

For instance, in order to encrypt \( m = 65 \), we calculate

\[
c = 65^{17} \mod 3233 = 2790
\]

To decrypt \( c = 2790 \), we calculate

\[
m = 2790^{2753} \mod 3233 = 65.
\]

Both of these calculations can be computed efficiently using the square-and-multiply algorithm for modular exponentiation. In real-life situations the primes selected would be much larger; in our example it would be trivial to factor \( n \) (obtained from the freely available public key) back to the primes \( p \) and \( q \). Given \( e \), also from the public key, we could then compute \( d \) and so acquire the private key.

Practical implementations use the Chinese remainder theorem to speed up the calculation using modulus of factors \((\mod p q \mod p \mod q)\).

The values \( d_p \) and \( d_q \) and \( q_{\text{inv}} \), which are part of the private key are computed as follows:

- \( d_p = d \mod (p - 1) = 2753 \mod (61 - 1) = 53 \)
- \( d_q = d \mod (q - 1) = 2753 \mod (53 - 1) = 49 \)
- \( q_{\text{inv}} = q^{-1} \mod 61 = 38 \) (Hence: \((q_{\text{inv}} \times q) \mod p = 38 \times 53 \mod 61 = 1 \))

Here is how \( d_p \) and \( d_q \) are used for efficient decryption. (Encryption is efficient by choice of public exponent \( e \))

- \( m_1 = c^{d_p} \mod p = 2790^{53} \mod 61 = 4 \)
- \( m_2 = c^{d_q} \mod q = 2790^{49} \mod 53 = 12 \)
- \( h = (q_{\text{inv}} \times (m_1 - m_2)) \mod p = (38 \times -8) \mod 61 = 1 \)
- \( m = m_2 + h \times q = 12 + 1 \times 53 = 65 \) (same as above but computed more efficiently)

**Signing messages**

Suppose Alice uses Bob's public key to send him an encrypted message. In the message, she can claim to be Alice but Bob has no way of verifying that the message was actually from Alice since anyone can use Bob's public key to send him encrypted messages. In order to verify the origin of a message, RSA can also be used to sign a message.

Suppose Alice wishes to send a signed message to Bob. She can use her own private key to do so. She produces a hash value of the message, raises it to the power of \( d \) (modulo \( n \)) (as she does when decrypting a message), and attaches it as a "signature" to the message. When Bob receives the signed message, he uses the same hash algorithm in conjunction with Alice's public key. He raises the signature to the power of \( e \) (modulo \( n \)) (as he does when encrypting a message), and compares the resulting hash value with the message's actual hash value. If the two agree, he knows that the author of the message was in possession of Alice's private key, and that the message has not been tampered with since.
Proofs of correctness

Proof using Fermat's little theorem

The proof of the correctness of RSA is based on Fermat's little theorem. This theorem states that if \( p \) is prime and \( p \) does not divide an integer \( a \) then

\[
a^{p-1} \equiv 1 \pmod{p}.\]

We want to show that \((m^e)^d \equiv m \pmod{pq}\) for every integer \( m \) when \( p \) and \( q \) are distinct prime numbers and \( e \) and \( d \) are positive integers satisfying

\[
ed \equiv 1 \pmod{(p-1)(q-1)}.\]

We can write

\[
ed - 1 = h(p-1)(q-1).
\]

for some nonnegative integer \( h \).

To check two numbers, like \( m^ed \) and \( m \), are congruent mod \( pq \) it suffices (and in fact is equivalent) to check they are congruent mod \( p \) and mod \( q \) separately. (This is part of the Chinese remainder theorem, although it is not the significant part of that theorem.) To show \( m^ed \equiv m \pmod{p} \), we consider two cases: \( m \equiv 0 \pmod{p} \) and \( m \not\equiv 0 \pmod{p} \). In the first case \( m^ed \) is a multiple of \( p \), so \( m^ed \equiv 0 \equiv m \pmod{p} \). In the second case

\[
m^ed = m^{(ed-1)} = m^{h(p-1)(q-1)} = (m^{p-1})^{h(q-1)}m \equiv 1^{h(q-1)}m \equiv m \pmod{p},
\]

where we used Fermat's little theorem to replace \( m^{p-1} \) mod \( p \) with 1.

The verification that \( m^ed \equiv m \pmod{q} \) proceeds in a similar way, treating separately the cases \( m \equiv 0 \pmod{q} \) and \( m \not\equiv 0 \pmod{q} \), using Fermat's little theorem for modulus \( q \) in the second case.

This completes the proof that, for any integer \( m \),

\[
(m^e)^d \equiv m \pmod{pq}.
\]

Proof using Euler's theorem

Although the original paper of Rivest, Shamir, and Adleman used Fermat's little theorem to explain why RSA works, it is common to find proofs that rely instead on Euler's theorem.

We want to show that \( m^ed \equiv m \pmod{n} \), where \( n = pq \) is a product of two different prime numbers and \( e \) and \( d \) are positive integers satisfying \( ed \equiv 1 \pmod{\phi(n)} \). Since \( e \) and \( d \) are positive, we can write \( ed = 1 + h\phi(n) \) for some nonnegative integer \( h \). Assuming that \( m \) is relatively prime to \( n \), we have

\[
m^ed = m^{1+h\phi(n)} = m^{\phi(n)}^h \equiv m^1 \equiv m \pmod{n},
\]

where the second-last congruence follows from the Euler's theorem.

When \( m \) is not relatively prime to \( n \), the argument just given is invalid. This is highly improbable (only a proportion of \( 1/p + 1/q - 1/(pq) \) numbers have this property), but even in this case the desired congruence is still true. Either \( m \equiv 0 \pmod{p} \) or \( m \equiv 0 \pmod{q} \), and these cases can be treated using the previous proof.
Padding

Attacks against plain RSA

There are a number of attacks against plain RSA as described below.

- When encrypting with low encryption exponents (e.g., \(e = 3\)) and small values of the \(m\), (i.e., \(m < n^{1/e}\)) the result of \(m^e\) is strictly less than the modulus \(n\). In this case, ciphertexts can be easily decrypted by taking the \(e\)th root of the ciphertext over the integers.

- If the same clear text message is sent to \(e\) or more recipients in an encrypted way, and the receivers share the same exponent \(e\), but different \(p, q\), and therefore \(n\), then it is easy to decrypt the original clear text message via the Chinese remainder theorem. Johan Håstad noticed that this attack is possible even if the cleartexts are not equal, but the attacker knows a linear relation between them.\(^{[4]}\) This attack was later improved by Don Coppersmith.\(^{[5]}\)

- Because RSA encryption is a deterministic encryption algorithm (i.e., has no random component) an attacker can successfully launch a chosen plaintext attack against the cryptosystem, by encrypting likely plaintexts under the public key and test if they are equal to the ciphertext. A cryptosystem is called semantically secure if an attacker cannot distinguish two encryptions from each other even if the attacker knows (or has chosen) the corresponding plaintexts. As described above, RSA without padding is not semantically secure.

- RSA has the property that the product of two ciphertexts is equal to the encryption of the product of the respective plaintexts. That is \(m_1^e \cdot m_2^e \equiv (m_1 \cdot m_2)^e \pmod{n}\). Because of this multiplicative property a chosen-ciphertext attack is possible. E.g., an attacker, who wants to know the decryption of a ciphertext \(c \equiv m^e \pmod{n}\) may ask the holder of the private key to decrypt an unsuspicious-looking ciphertext \(c' \equiv cr^e \pmod{n}\) for some value \(r\) chosen by the attacker. Because of the multiplicative property \(c'\) is the encryption of \(mr \pmod{n}\). Hence, if the attacker is successful with the attack, he will learn \(mr \pmod{n}\) from which he can derive the message \(m\) by multiplying \(mr\) with the modular inverse of \(r \pmod{n}\).

Padding schemes

To avoid these problems, practical RSA implementations typically embed some form of structured, randomized padding into the value \(m\) before encrypting it. This padding ensures that \(m\) does not fall into the range of insecure plaintexts, and that a given message, once padded, will encrypt to one of a large number of different possible ciphertexts.

Standards such as PKCS#1 have been carefully designed to securely pad messages prior to RSA encryption. Because these schemes pad the plaintext \(m\) with some number of additional bits, the size of the un-padded message \(M\) must be somewhat smaller. RSA padding schemes must be carefully designed so as to prevent sophisticated attacks which may be facilitated by a predictable message structure. Early versions of the PKCS#1 standard (up to version 1.5) used a construction that appears to make RSA semantically secure. However, at Eurocrypt 2000, Coron et al. showed that for some types of messages, this padding does not provide a high enough level of security. Furthermore, at Crypto 1998, Bleichenbacher showed that this version is vulnerable to a practical adaptive chosen ciphertext attack. Later versions of the standard include Optimal Asymmetric Encryption Padding (OAEP), which prevents these attacks. As such, OAEP should be used in any new application, and PKCS#1 v1.5 padding should be replaced wherever possible. The PKCS#1 standard also incorporates processing schemes designed to provide additional security for RSA signatures (e.g., the Probabilistic Signature Scheme for RSA/RSA-PSS).

Secure padding schemes such as RSA-PSS are as essential for the security of message signing as they are for message encryption. Two US patents on PSS were granted (USPTO 6266771 and USPTO 70360140); however, these patents expired on 24 July 2009 and 25 April 2010, respectively. Use of PSS no longer seems to be encumbered by patents. Note that using different RSA key-pairs for encryption and signing is potentially more secure.\(^{[6]}\)\(^{[7]}\)
Security and practical considerations

Using the Chinese remainder algorithm

For efficiency many popular crypto libraries (like OpenSSL, Java and .NET) use the following optimization for decryption and signing based on the Chinese remainder theorem. The following values are precomputed and stored as part of the private key:

- \( p \) and \( q \): the primes from the key generation,
- \( d_p = d \pmod{p-1} \),
- \( d_Q = d \pmod{q-1} \), and
- \( q_{\text{inv}} = q^{-1} \pmod{p} \).

These values allow the recipient to compute the exponentiation \( m = c^d \pmod{pq} \) more efficiently as follows:

- \( m_1 = c^{d_p} \pmod{p} \)
- \( m_2 = c^{d_Q} \pmod{q} \)
- \( h = q_{\text{inv}}(m_1 - m_2) \pmod{q} \) (if \( m_1 < m_2 \) then some libraries compute \( h \) as \( q_{\text{inv}}(m_1 + p - m_2) \pmod{p} \))
- \( m = m_2 + hq \)

This is more efficient than computing \( m \equiv c^d \pmod{pq} \) even though two modular exponentiations have to be computed. The reason is that these two modular exponentiations both use a smaller exponent and a smaller modulus.

Integer factorization and RSA problem

The security of the RSA cryptosystem is based on two mathematical problems: the problem of factoring large numbers and the RSA problem. Full decryption of an RSA ciphertext is thought to be infeasible on the assumption that both of these problems are hard, i.e., no efficient algorithm exists for solving them. Providing security against partial decryption may require the addition of a secure padding scheme.\[^{[citation needed]}\]

The RSA problem is defined as the task of taking \( e \)th roots modulo a composite \( n \): recovering a value \( m \) such that \( c \equiv m^e \pmod{n} \), where \((n, e)\) is an RSA public key and \( c \) is an RSA ciphertext. Currently the most promising approach to solving the RSA problem is to factor the modulus \( n \). With the ability to recover prime factors, an attacker can compute the secret exponent \( d \) from a public key \((n, e)\), then decrypt \( c \) using the standard procedure. To accomplish this, an attacker factors \( n \) into \( p \) and \( q \), and computes \((p-1)(q-1)\) which allows the determination of \( d \) from \( e \). No polynomial-time method for factoring large integers on a classical computer has yet been found, but it has not been proven that none exists. See integer factorization for a discussion of this problem. Rivest, Shamir and Adleman note that Miller has shown that – assuming the Extended Riemann Hypothesis – finding \( d \) from \( n \) and \( e \) is as hard as factoring \( n \) into \( p \) and \( q \) (up to a polynomial time difference).\[^{[8]}\]

As of 2010[9], the largest (known) number factored by a general-purpose factoring algorithm was 768 bits long (see RSA-768), using a state-of-the-art distributed implementation. RSA keys are typically 1024 to 2048 bits long. Some experts believe that 1024-bit keys may become breakable in the near future (though this is disputed); few see any way that 4096-bit keys could be broken in the foreseeable future. Therefore, it is generally presumed that RSA is secure if \( n \) is sufficiently large. If \( n \) is 300 bits or shorter, it can be factored in a few hours on a personal computer, using software already freely available. Keys of 512 bits have been shown to be practically breakable in 1999 when RSA-155 was factored by using several hundred computers and are now factored in a few weeks using common hardware.\[^{[10]}\]

A theoretical hardware device named TWIRL and described by Shamir and Tromer in 2003 called into question the security of 1024 bit keys. It is currently recommended that \( n \) be at least 2048 bits long.\[^{[12]}\]
In 1994, Peter Shor showed that a quantum computer (if one could ever be practically created for the purpose) would be able to factor in polynomial time, breaking RSA; see Shor's algorithm.

**Faulty key generation**

Finding the large primes $p$ and $q$ is usually done by testing random numbers of the right size with probabilistic primality tests which quickly eliminate virtually all non-primes.

Numbers $p$ and $q$ should not be 'too close', lest the Fermat factorization for $n$ be successful, if $p - q$, for instance is less than $2^{n^{1/4}}$ (which for even small 1024-bit values of $n$ is $3 \times 10^{77}$) solving for $p$ and $q$ is trivial. Furthermore, if either $p - 1$ or $q - 1$ has only small prime factors, $n$ can be factored quickly by Pollard's $p - 1$ algorithm, and these values of $p$ or $q$ should therefore be discarded as well.

It is important that the private key $d$ be large enough. Michael J. Wiener showed that if $p$ is between $q$ and $2q$ (which is quite typical) and $d < n^{1/4}/3$, then $d$ can be computed efficiently from $n$ and $e$.

There is no known attack against small public exponents such as $e = 3$, provided that proper padding is used. However, when no padding is used, or when the padding is improperly implemented, small public exponents have a greater risk of leading to an attack, such as the unpadded plaintext vulnerability listed above. 65537 is a commonly used value for $e$. This value can be regarded as a compromise between avoiding potential small exponent attacks and still allowing efficient encryptions (or signature verification). The NIST Special Publication on Computer Security (SP 800-78 Rev 1 of August 2007) does not allow public exponents $e$ smaller than 65537, but does not state a reason for this restriction.

**Importance of strong random number generation**

A cryptographically strong random number generator, which has been properly seeded with adequate entropy, must be used to generate the primes $p$ and $q$. An analysis comparing millions of public keys gathered from the Internet was carried out in early 2012 by Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung and Christophe Wachter. They were able to factor 0.2% of the keys using only Euclid's algorithm.[13][14]

They exploited a weakness unique to cryptosystems based on integer factorization. If $n = pq$ is one public key and $n' = p'q'$ is another, then if by chance $p = p'$, then a simple computation of $\gcd(n,n') = p$ factors both $n$ and $n'$, totally compromising both keys. Lenstra et al. note that this problem can be minimized by using a strong random seed of bit-length twice the intended security level, or by employing a deterministic function to choose $q$ given $p$, instead of choosing $p$ and $q$ independently.

Nadia Heninger was part of a group that did a similar experiment. They used an idea of Daniel J. Bernstein to compute the GCD of each RSA key $n$ against the product of all the other keys $n'$ they had found (a 729 million digit number), instead of computing each $\gcd(n,n')$ separately, thereby achieving a very significant speedup since after one large division the GCD problem is of normal size.

Heninger says in her blog that the bad keys occurred almost entirely in embedded applications, including "firewalls, routers, VPN devices, remote server administration devices, printers, projectors, and VOIP phones" from over 30 manufactures. Heninger explains that the one-shared-prime problem uncovered by the two groups results from situations where the pseudorandom number generator is poorly seeded initially and then reseeded between the generation of the first and second primes. Using seeds of sufficiently high entropy obtained from key stroke timings or electronic diode noise or atmospheric noise from a radio receiver tuned between stations should solve the problem.[15]

Strong random number generation is important throughout every phase of public key cryptography. For instance, if a weak generator is used for the symmetric keys that are being distributed by RSA, then an eavesdropper could bypass the RSA and guess the symmetric keys directly.
Timing attacks
Kocher described a new attack on RSA in 1995: if the attacker Eve knows Alice's hardware in sufficient detail and is able to measure the decryption times for several known ciphertexts, she can deduce the decryption key $d$ quickly. This attack can also be applied against the RSA signature scheme. In 2003, Boneh and Brumley demonstrated a more practical attack capable of recovering RSA factorizations over a network connection (e.g., from a Secure Socket Layer (SSL)-enabled webserver)\[16\] This attack takes advantage of information leaked by the Chinese remainder theorem optimization used by many RSA implementations.

One way to thwart these attacks is to ensure that the decryption operation takes a constant amount of time for every ciphertext. However, this approach can significantly reduce performance. Instead, most RSA implementations use an alternate technique known as cryptographic blinding. RSA blinding makes use of the multiplicative property of RSA. Instead of computing $c^d \pmod{n}$, Alice first chooses a secret random value $r$ and computes $(r^e c)^d \pmod{n}$. The result of this computation after applying Euler's Theorem is $r^d \pmod{n}$ and so the effect of $r$ can be removed by multiplying by its inverse. A new value of $r$ is chosen for each ciphertext. With blinding applied, the decryption time is no longer correlated to the value of the input ciphertext and so the timing attack fails.

Adaptive chosen ciphertext attacks
In 1998, Daniel Bleichenbacher described the first practical adaptive chosen ciphertext attack, against RSA-encrypted messages using the PKCS #1 v1 padding scheme (a padding scheme randomizes and adds structure to an RSA-encrypted message, so it is possible to determine whether a decrypted message is valid). Due to flaws with the PKCS #1 scheme, Bleichenbacher was able to mount a practical attack against RSA implementations of the Secure Socket Layer protocol, and to recover session keys. As a result of this work, cryptographers now recommend the use of provably secure padding schemes such as Optimal Asymmetric Encryption Padding, and RSA Laboratories has released new versions of PKCS #1 that are not vulnerable to these attacks.

Side-channel analysis attacks
A side-channel attack using branch prediction analysis (BPA) has been described. Many processors use a branch predictor to determine whether a conditional branch in the instruction flow of a program is likely to be taken or not. Often these processors also implement simultaneous multithreading (SMT). Branch prediction analysis attacks use a spy process to discover (statistically) the private key when processed with these processors.

Simple Branch Prediction Analysis (SBPA) claims to improve BPA in a non-statistical way. In their paper, "On the Power of Simple Branch Prediction Analysis",\[17\] the authors of SBPA (Onur Aciicmez and Cetin Kaya Koc) claim to have discovered 508 out of 512 bits of an RSA key in 10 iterations.

A power fault attack on RSA implementations has been described in 2010.\[18\] The authors recovered the key by varying the CPU power voltage outside limits; this caused multiple power faults on the server.

Notes
[8] Gary L. Miller, "Riemann's Hypothesis and Tests for Primality" (http://www.cs.cmu.edu/~glmiller/Publications/Papers/Mi75.pdf)
RSA (cryptosystem)

References


External links

- The PKCS #1 standard "provides recommendations for the implementation of public-key cryptography based on the RSA algorithm, covering the following aspects: cryptographic primitives; encryption schemes; signature schemes with appendix; ASN.1 syntax for representing keys and for identifying the schemes".
- Explanation of RSA using colored lamps (http://www.youtube.com/watch?v=vgTtHV04xRI)
- Prime Number Hide-And-Seek: How the RSA Cipher Works (http://www.muppetlabs.com/~breadbox/txt/rsa.html)
- Example of an RSA implementation with PKCS#1 padding (GPL source code) (http://polarssl.org/source_code)
- Kocher's article about timing attacks (http://www.cryptography.com/resources/whitepapers/TimingAttacks.pdf)
- A spreadsheet implementing RSA (https://docs.google.com/spreadsheet/ccc?key=0AmFN4Z5lIsHdHdFMGxXZkZCd2RnQWZBQnZqSU4UVE#gid=0)
Diffie–Hellman key exchange

Diffie–Hellman key exchange (D–H)\[1\] is a specific method of exchanging cryptographic keys. It is one of the earliest practical examples of key exchange implemented within the field of cryptography. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

The scheme was first published by Whitfield Diffie and Martin Hellman in 1976, although it had been separately invented a few years earlier within GCHQ, the British signals intelligence agency, by James H. Ellis, Clifford Cocks and Malcolm J. Williamson but was kept classified. In 2002, Hellman suggested the algorithm be called Diffie–Hellman–Merkle key exchange in recognition of Ralph Merkle's contribution to the invention of public-key cryptography (Hellman, 2002).

Although Diffie–Hellman key agreement itself is an anonymous (non-authenticated) key-agreement protocol, it provides the basis for a variety of authenticated protocols, and is used to provide perfect forward secrecy in Transport Layer Security's ephemeral modes (referred to as EDH or DHE depending on the cipher suite).

The method was followed shortly afterwards by RSA, an implementation of public key cryptography using asymmetric algorithms.

In 2002, Martin Hellman wrote:

The system...has since become known as Diffie–Hellman key exchange. While that system was first described in a paper by Diffie and me, it is a public key distribution system, a concept developed by Merkle, and hence should be called 'Diffie–Hellman–Merkle key exchange' if names are to be associated with it. I hope this small pulpit might help in that endeavor to recognize Merkle's equal contribution to the invention of public key cryptography.\[2\]

U.S. Patent 4,200,770 \[3\], from 1977 is now expired and describes the algorithm. It credits Hellman, Diffie, and Merkle as inventors.

Description

Diffie–Hellman establishes a shared secret that can be used for secret communications while exchanging data over a public network. The following diagram illustrates the general idea of the key exchange by using colours instead of a very large number. The crucial part of the process is that Alice and Bob exchange their secret colours in a mix only. Finally this generates an identical key that is mathematically difficult (impossible for modern supercomputers to do in a reasonable amount of time) to reverse for another party that might have been listening in on them. Alice and Bob now use this common secret to encrypt and decrypt their sent and received data. Note that the starting color (yellow) is arbitrary, but is agreed on in advance by Alice and Bob. The starting color is assumed to be known to any eavesdropping Opponent. It may even be public.
Here is an explanation which includes the encryption’s mathematics:

The simplest and the original implementation of the protocol uses the multiplicative group of integers modulo $p$, where $p$ is prime and $g$ is primitive root mod $p$. Here is an example of the protocol, with non-secret values in blue, and secret values in red:

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secret</td>
<td>Public</td>
</tr>
<tr>
<td>$a$</td>
<td>$p, g$</td>
</tr>
<tr>
<td>$a$</td>
<td>$p, g, A$</td>
</tr>
<tr>
<td>$a, s$</td>
<td>$p, g, A, B$</td>
</tr>
</tbody>
</table>

1. Alice and Bob agree to use a prime number $p = 23$ and base $g = 5$.
2. Alice chooses a secret integer $a = 6$, then sends Bob $A = g^a \mod p$
   - $A = 5^6 \mod 23$
   - $A = 15,625 \mod 23$
   - $A = 8$
3. Bob chooses a secret integer $b = 15$, then sends Alice $B = g^b \mod p$
   - $B = 5^{15} \mod 23$
   - $B = 30,517,578,125 \mod 23$
   - $B = 19$
4. Alice computes $s = B^a \mod p$
DiffieHellman key exchange

- \( s = 19^6 \text{ mod } 23 \)
- \( s = 47,045,881 \text{ mod } 23 \)
- \( s = 2 \)

5. Bob computes \( s = A^b \text{ mod } p \)
- \( s = 8^{15} \text{ mod } 23 \)
- \( s = 35,184,372,088,832 \text{ mod } 23 \)
- \( s = 2^{6} \)

6. Alice and Bob now share a secret (the number 2) because 6 \times 15 is the same as 15 \times 6.

Both Alice and Bob have arrived at the same value, because \((g^a)^b \text{ and } (g^b)^a\) are equal \text{ mod } p. Note that only \(a\), \(b\), and \((g^{ab} = g^{ba}) \text{ mod } p \) are kept secret. All the other values – \(p\), \(g\), \(g^a \text{ mod } p\), and \(g^b \text{ mod } p\) – are sent in the clear. Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel.

Of course, much larger values of \(a\), \(b\), and \(p\) would be needed to make this example secure, since there are only 23 possible results of \(n \text{ mod } 23\). However, if \(p\) is a prime of at least 300 digits, and \(a\) and \(b\) are at least 100 digits long, then even the fastest modern computers cannot find \(a\) given only \(g\), \(p\), \(g^b \text{ mod } p\) and \(g^a \text{ mod } p\). The problem such a computer needs to solve is called the discrete logarithm problem.

Note that \(g\) need not be large at all, and in practice is usually either 2, 3 or 5.

Here’s a more general description of the protocol:

1. Alice and Bob agree on a finite cyclic group \(G\) and a generating element \(g\) in \(G\). (This is usually done long before the rest of the protocol; \(g\) is assumed to be known by all attackers.) We will write the group \(G\) multiplicatively.
2. Alice picks a random natural number \(a\) and sends \(g^a\) to Bob.
3. Bob picks a random natural number \(b\) and sends \(g^b\) to Alice.
4. Alice computes \((g^b)^a\).
5. Bob computes \((g^a)^b\).

Both Alice and Bob are now in possession of the group element \(g^{ab}\), which can serve as the shared secret key. The values of \((g^b)^a\) and \((g^a)^b\) are the same because groups are power associative. (See also exponentiation.)

In order to decrypt a message \(m\), sent as \(mg^{ab}\), Bob (or Alice) must first compute \((g^{ab})^{-1}\), as follows:

Bob knows \(|G|\), \(b\), and \(g^a\). Lagrange’s theorem in group theory establishes that from the construction of \(G\), \(x^{|G|} = 1\) for all \(x \in G\).

Bob then calculates \((g^a)^{|G|-b} = g^{b|G|-b} = g^{a|G|-ab} = g^{a|G|}g^{-ab} = (g^{a|G|})g^{-ab} = g^{-ab} = \left(g^{ab}\right)^{-1}\).

When Alice sends Bob the encrypted message, \(mg^{ab}\), Bob applies \((g^{ab})^{-1}\) and recovers \(mg^{ab}(g^{ab})^{-1} = m(1) = m\).

Chart

Here is a chart to help simplify who knows what. (Eve is an eavesdropper—she watches what is sent between Alice and Bob, but she does not alter the contents of their communications.)
- Let \(g\) = public (prime) base, known to Alice, Bob, and Eve. \(g = 5\)
- Let \(p\) = public (prime) number, known to Alice, Bob, and Eve. \(p = 23\)
- Let \(a\) = Alice’s private key, known only to Alice. \(a = 6\)
- Let \(b\) = Bob’s private key known only to Bob. \(b = 15\)
- Let \(A\) = Alice’s public key, known to Alice, Bob, and Eve. \(A = g^a \text{ mod } p = 8\)
- Let \(B\) = Bob’s public key, known to Alice, Bob, and Eve. \(B = g^b \text{ mod } p = 19\)
Alice knows, Bob doesn't know

<table>
<thead>
<tr>
<th>$p = 23$</th>
<th>$b = ?$</th>
<th>$p = 23$</th>
<th>$a = ?$</th>
<th>$p = 23$</th>
<th>$a = ?$</th>
</tr>
</thead>
<tbody>
<tr>
<td>base $g = 5$</td>
<td>base $g = 5$</td>
<td>base $g = 5$</td>
<td>base $g = 5$</td>
<td>base $g = 5$</td>
<td></td>
</tr>
<tr>
<td>$a = 6$</td>
<td>$b = 15$</td>
<td>$A = 5^a \mod 23$</td>
<td>$B = 5^b \mod 23$</td>
<td>$A = 5^a \mod 23$</td>
<td>$B = 5^b \mod 23$</td>
</tr>
<tr>
<td>$A = 5^6 \mod 23 = 8$</td>
<td>$B = 5^{15} \mod 23 = 19$</td>
<td>$s = A^b \mod 23 = 19$</td>
<td>$s = B^a \mod 23 = 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s = B^a \mod 23 = 2$</td>
<td>$s = A^b \mod 23 = 19$</td>
<td>$s = 19^8 \mod 23 = 8^a \mod 23$</td>
<td>$s = 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s = 2$</td>
<td>$s = 2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Now $s$ is the shared secret key and it is known to both Alice and Bob, but not to Eve. $s = 2$

Note: It should be difficult for Alice to solve for Bob's private key or for Bob to solve for Alice's private key. If it is not difficult for Alice to solve for Bob's private key (or vice versa), Eve may simply substitute her own private/public key pair, plug Bob's public key into her private key, produce a fake shared secret key, and solve for Bob's private key (and use that to solve for the shared secret key). Eve may attempt to choose a public/private key pair that will make it easy for her to solve for Bob's private key. A demonstration of Diffie-Hellman (using numbers too small for practical use) is given here [4]

**Operation with more than two parties**

Diffie-Hellman key agreement is not limited to negotiating a key shared by only two participants. Any number of users can take part in an agreement by performing iterations of the agreement protocol and exchanging intermediate data (which does not itself need to be kept secret). For example, Alice, Bob, and Carol could participate in a Diffie-Hellman agreement as follows, with all operations taken to be modulo $p$:

1. The parties agree on the algorithm parameters $p$ and $g$.
2. The parties generate their private keys, named $a$, $b$, and $c$.
3. Alice computes $g^a$ and sends it to Bob.
4. Bob computes $(g^a)^b = g^{ab}$ and sends it to Carol.
5. Carol computes $(g^{ab})^c = g^{abc}$ and uses it as her secret.
6. Bob computes $g^b$ and sends it to Carol.
7. Carol computes $(g^b)^c = g^{bc}$ and sends it to Alice.
8. Alice computes $(g^{bc})^a = g^{abc}$ and uses it as her secret.
9. Carol computes $g^c$ and sends it to Alice.
10. Alice computes $(g^c)^a = g^{aca}$ and sends it to Bob.
11. Bob computes $(g^{ca})^b = g^{cab}$ and uses it as his secret.

An eavesdropper has been able to see $g^a$, $g^b$, $g^c$, $g^{ab}$, $g^{ac}$, and $g^{bc}$, but cannot use any combination of these to reproduce $g^{abc}$.

To extend this mechanism to larger groups, two basic principles must be followed:

- Starting with an "empty" key consisting only of $g$, the secret is made by raising the current value to every participant’s private exponent once, in any order (the first such exponentiation yields the participant’s own public
key).

• Any intermediate value (having up to $N - 1$ exponents applied, where $N$ is the number of participants in the group) may be revealed publicly, but the final value (having had all $N$ exponents applied) constitutes the shared secret and hence must never be revealed publicly. Thus, each user must obtain their copy of the secret by applying their own private key last (otherwise there would be no way for the last contributor to communicate the final key to its recipient, as that last contributor would have turned the key into the very secret the group wished to protect).

These principles leave open various options for choosing in which order participants contribute to keys. The simplest and most obvious solution is to arrange the $N$ participants in a circle and have keys rotate around the circle, until eventually every key has been contributed to by all $N$ participants (ending with its owner) and each participant has contributed to $N$ keys (ending with their own). However, this requires that every participant perform $N$ modular exponentiations.

By choosing a more optimal order, and relying on the fact that keys can be duplicated, it is possible to reduce the number of modular exponentiations performed by each participant to $\log_2(N) + 1$ using a divide-and-conquer-style approach, given here for eight participants:

1. Participants A, B, C, and D each perform one exponentiation, yielding $g^{abcd}$; this value is sent to E, F, G, and H. In return, participants A, B, C, and D receive $g^{efgh}$.
2. Participants A and B each perform one exponentiation, yielding $g^{efghab}$, which they send to C and D, while C and D do the same, yielding $g^{efghcd}$, which they send to A and B.
3. Participant A performs an exponentiation, yielding $g^{efgheda}$, which it sends to B; similarly, B sends $g^{efghedb}$ to A. C and D do similarly.
4. Participant A performs one final exponentiation, yielding the secret $g^{efghedba} = g^{abcdgfh}$, while B does the same to get $g^{efghedab} = g^{abcdgfh}$; again, C and D do similarly.
5. Participants E through H simultaneously perform the same operations using $g^{abcd}$ as their starting point.

Once this operation has been completed all participants will possess the secret $g^{abcdgfh}$, but each participant will have performed only four modular exponentiations, rather than the eight implied by a simple circular arrangement.

Security

The protocol is considered secure against eavesdroppers if $G$ and $g$ are chosen properly. The eavesdropper ("Eve") would have to solve the Diffie–Hellman problem to obtain $g^{ab}$. This is currently considered difficult. An efficient algorithm to solve the discrete logarithm problem would make it easy to compute $a$ or $b$ and solve the Diffie–Hellman problem, making this and many other public key cryptosystems insecure.

The order of $G$ should have a large prime factor to prevent use of the Pohlig–Hellman algorithm to obtain $a$ or $b$. For this reason, a Sophie Germain prime $q$ is sometimes used to calculate $p = 2q + 1$, called a safe prime, since the order of $G$ is then only divisible by 2 and $q$. $g$ is then sometimes chosen to generate the order $q$ subgroup of $G$, rather than $G$, so that the Legendre symbol of $g^a$ never reveals the low order bit of $a$.

If Alice and Bob use random number generators whose outputs are not completely random and can be predicted to some extent, then Eve's task is much easier.

In the original description, the Diffie–Hellman exchange by itself does not provide authentication of the communicating parties and is thus vulnerable to a man-in-the-middle attack. Eve may establish two distinct key exchanges, one with Alice and the other with Bob, effectively masquerading as Alice to Bob, and vice versa, allowing her to decrypt, then re-encrypt, the messages passed between them. Note that Eve must continue to be in the middle, transferring messages every time Alice and Bob communicate. If she is ever absent, her previous presence is then revealed to Alice and Bob. They will know that all of their private conversations had been
intercepted and decoded by someone in the channel.

A method to authenticate the communicating parties to each other is generally needed to prevent this type of attack. Variants of Diffie–Hellman, such as STS, may be used instead to avoid these types of attacks.

Other uses

Password-authenticated key agreement

When Alice and Bob share a password, they may use a password-authenticated key agreement (PAKE) form of Diffie–Hellman to prevent man-in-the-middle attacks. One simple scheme is to compare hash of $s$ concatenated with the password calculated independently on both ends of channel. A feature of these schemes is that an attacker can only test one specific password on each iteration with the other party, and so the system provides good security with relatively weak passwords. This approach is described in ITU-T Recommendation X.1035, which is used by the G.hn home networking standard.

Public Key

It is also possible to use Diffie–Hellman as part of a public key infrastructure. Alice's public key is simply $(g^a \mod p, g, p)$. To send her a message Bob chooses a random $b$, and then sends Alice $g^b \mod p$ (un-encrypted) together with the message encrypted with symmetric key $(g^a)^b \mod p$. Only Alice can decrypt the message because only she has $a$ (the private key). A preshared public key also prevents man-in-the-middle attacks. In practice, Diffie–Hellman is not used in this way, with RSA being the dominant public key algorithm. This is largely for historical and commercial reasons, namely that RSA created a Certificate Authority for key signing that became Verisign. Diffie–Hellman cannot be used to sign certificates. However, the ElGamal and DSA signature algorithms are mathematically related to it, as well as MQV, STS and the IKE component of the IPsec protocol suite for securing Internet Protocol communications.

Notes

[1] Synonyms of Diffie–Hellman key exchange include:
- Diffie–Hellman key agreement
- Diffie–Hellman key establishment
- Diffie–Hellman key negotiation
- Exponential key exchange
- Diffie–Hellman protocol
- Diffie–Hellman handshake


References


**External links**

• Diffie-Hellman key exchange explained in 5 minutes (https://www.youtube.com/watch?v=3QnD2c4Xovk)

• Oral history interview with Martin Hellman (http://purl.umn.edu/107353), Charles Babbage Institute, University of Minnesota. Leading cryptography scholar Martin Hellman discusses the circumstances and fundamental insights of his invention of public key cryptography with collaborators Whitfield Diffie and Ralph Merkle at Stanford University in the mid-1970s.


• Diffie–Hellman Key Exchange – A Non-Mathematician’s Explanation (http://docs.google.com/viewer?a=v&pid=sites&srcid=bmV0aXAuY29tIi9hbWVYVzg6NTA2NTM0YmNhZjRhZDyzZQ) by Keith Palmgren

• Crypt::DH (https://metacpan.org/module/Crypt::DH) Perl module from CPAN

• Hands-on Diffie–Hellman demonstration (http://ds9a.nl/tmp/dh.html)

• C implementation using GNU Multiple Precision Arithmetic Library (http://oldpiewiki.yoonkn.com/cgi-bin/moin.cgi/DiffieHellmanKeyExchange)Wikipedia:Link rot

• Diffie Hellman in 2 lines of Perl (http://www.crypterspace.org/adam/rsa/Perl-dh.html) (using dc)

• Smart Account Management (SAcct) (http://code.google.com/p/sacct/) (using DH key exchange to derive session key)

• Talk by Martin Hellman in 2007, Google video (http://video.google.com/videoplay?docid=8991737124862867507) (broken link)
Image Sources, Licenses and Contributors


License

Creative Commons Attribution-Share Alike 3.0
creativecommons.org/licenses/by-sa/3.0/