Cryptographic Algorithms

Alex Feinman
Computer Science Department
Brandeis University
Waltham, MA 02454

Introduction

Cryptography has advanced significantly in the past twenty years; the advent of personal computers and pervasiveness of networks of electronic information have made cryptography both feasible and desirable. Perhaps the most notable advance in recent times was the Rivest-Shamir-Adleman public-key algorithm, introduced in 1977 specifically to aid in security on the internet.

This paper will examine the complexity in both time and space of some of the major algorithms of modern cryptography and cryptanalysis. It will also examine algorithms for solving the one-way problems underlying these forms of cryptography, and a few possible attacks on the most commonly-used forms of cryptography today.

Encryption

Cryptography in the modern age is under attack from powerful computers; no longer are simple character-substitution algorithms sufficient to maintain security. Luckily, these same computers allow the creation and use of much more complex encryption algorithms. The trick is finding some operation that can be performed on data that is easy to do but hard to undo, unless you know some secret. Such a problem is often referred to as a trapdoor one-way problem.

A one-way problem is a problem that is easy to do but whose inverse is hard, in some sense. For example, it is easy to change a list of numbers from sorted to unsorted; a simple swap will do it in O(1). But it is more difficult to sort them again, which is $O(n\lg(n))$. A more stark example of a one-way problem is one in which the forward direction is in P, but whose inverse is in

NP. An example of this can be easily constructed out of almost any of the classic NP problems. Let us use k-clique as an example. To construct a one-way problem, user A generates a fully-connected graph of some large size n. This takes at most time $O(n^2)$ Obviously, such a graph contains a k-clique. User A then adds a number of nodes and edges to the graph in such a way as to ensure no new n-cliques are formed (a simple heuristic limiting the number of connections from new nodes to the original n nodes would be sufficient), bringing the total size of the graph to N.

User A then gives this new graph to user B, and asks her to find whether the graph contains a k-clique. Finding out whether it does will take user B at least time $O(a^n)$, which is clearly much harder than the construction of the graph. In a trapdoor problem, the inverse problem is hard unless some certificate is available. This certificate should then collapse the hard problem to something easily computable. In the above example, a certificate consisting of the nodes in the clique would suffice to reduce the problem to $O(n^2)$ again.

Cryptographic algorithms use this trapdoor property to encode information so that only the desired recipients can decode it. Continuing the above example, it would be possible to construct a (highly inefficient) system which would encode the user's binary message into a list of graphs, with 0 or 1 being encoded as whether or not a graph contained a k-clique. Provided the sender and receiver had agreed on a node in the graphs that would be in the cliques ahead of time, the recipient could decode the message in polynomial time.

The RSA Algorithm

The RSA algorithm uses just such an approach. The basic assumption in RSA is that the prime factorization of large numbers is a one-way problem -- very hard to solve in one direction, but very easy to solve in the other. The brute force approach to factorization of a number, by successively dividing by known primes, is

$$O\left(\pi\left(\sqrt{n}\right)\right) \text{ which is approximately } O\left(\frac{2\sqrt{n}}{\ln(n)}\right) \text{ for large n}$$

Even the best algorithms in use today for factoring large numbers, like the Number Field Sieve, are not terribly fast:

$$T_{NFS} = O\left(e^{c(\lg n)^{1/3}(\lg\lg n)^{2/3}}\right), \text{ where } c \le \left(64/9\right)^{1/3}, \text{ depending on the input}$$

Conversely, the inverse operation of creating a composite number from prime factors is simply multiplication, and takes time proportional to the square of the number of bits in the input, or $O(\lg^2 n)$. In most cases for cryptographic applications we are talking about $n \approx 2^{768}$ or larger. Hence, this is an enormous difference; for such n,

$$T_{NFS}\left(2^{768}\right) \approx \left(e^{1.9(768)^{1/3}(\lg 768)^{2/3}}\right) = e^{1.9 \cdot 9.16 \cdot 2.02} \approx 10^{15} \text{ but } T_{Mul}\left(2^{768}\right) = (768)^2 \approx 10^5;$$

one direction is over ten billion times faster than the other.

The RSA algorithm takes advantage of this disparity in the problem to allow a new wrinkle to encryption, the public key/private key system, which allows secure communication between users with no prearranged code. Specifically, it creates two keys S and P for each user, which have the following property:

P(S(M)) = S(P(M)) = M for any message M

RSA does this in the following manner. Users A and B generate their keys as follows:

1. Create p and q, which are large primes

2. Calculate n=pq

3. Choose e=some small odd integer that is relatively prime to ϕ(n), where ϕ(n) is Euler's phi function, which in this case can be shown to be $\phi(n) = (p-1)(q-1)$

4. Calculate d such that ed=1 modulo (p-1)(q-1), i.e., d is the multiplicative inverse of e modulo ϕ(n)

The two keys are then the pairs P = (e, n) and S = (d, n), and the operations performed on messages are $P(M) = M^b \bmod n$, $S(M) = M^a \bmod n$

To encode a message to B, A encodes it using B's public key, and sends the resultant message to B, who can then encode it with his secret key to retrieve the original message.

Analyzing the running time of this algorithm is somewhat problematic, and is rooted deeply in number theory. A full explanation is beyond the scope of this paper, but a brief discussion follows. Creating p and q is the first hurdle; as we have shown, testing for primality is difficult. The usual method is to randomly test numbers in the general area desired such that n is the size the user desires, and pull out two primes of approximately, but not exactly, the same size. The probability of choosing a prime randomly is approximately the density of primes near that number; this probability can be doubled by ignoring even numbers:

$$P_{prime}(n) = \frac{2 \cdot \pi(n)}{n} \approx \frac{2}{\ln n}$$

If a prime is not found, another nearby random number is chosen (with replacement; the odds of choosing the same number are so small that it is not worth the storage requirements to keep track of what has been tested) until a prime is found. The average number of iterations the algorithm must be repeated before found by a simple counting summation:

$$E(k) = 1 + \sum_{i=0}^{\infty} iP(n,i) = 1 + \sum_{i=0}^{\infty} iP(n)(1 - P(n))^i = P(n)\frac{(1 - P(n))}{(1 - (1 - P(n)))^2}$$

$$E(k) = \frac{2}{\ln n}\frac{\left(1 - \frac{2}{\ln n}\right)}{\left(1 - \left(1 - \frac{2}{\ln n}\right)\right)^2} = \frac{2}{\ln n}\frac{\left(1 - \frac{2}{\ln n}\right)}{\left(\frac{2}{\ln n}\right)^2} = \frac{\ln n}{2} - 1$$

For example, let $n \approx 2^{384}$; $\ln n \approx 265$; then

$$E(k) = \frac{265}{2} - 1 = 131 \text{ iterations}$$

This makes the running time for step 1 of key generation using the Number Field Sieve roughly:

$$T_{S1} = O\left(\ln\left(\sqrt{p}\right) \cdot e^{c(\lg p)^{1/3}(\lg \lg p)^{2/3}}\right)$$

As mentioned above, the multiplication n=pq takes $O\left(\lg^2 p\right) = O\left(\left(\tfrac{1}{2}\lg n\right)^2\right) = O\left(\lg^2 n\right)$, assuming p≈q; hence steps 2-4 take $O(\lg^2 n)$. Obviously, step one dominates the running time.

Now let us examine the running time for the actual encryption and decryption. Since encryption and decryption use identical algorithms, we only need to examine one. Encryption occurs by exponentiating, modulo n, the message by d. Hence we need to examine algorithms for modular exponentiation. The most straightforward of these involves repeatedly squaring and multiplying the original number according to the bit pattern of the power. This algorithm is in $O\left(\lg^2 n \lg k\right)$, where n is the input and k is the power.

Obviously, encrypting an entire message through one exponentiation would be unrealistic; then n would be enormous, and the running time of the encryption algorithm would be huge. For this reason implementations of the RSA algorithm usually break the message into smaller blocks, and then append them together into the actual encrypted message, but this block size must not be too small, or probabilistic techniques could be used to extract the mapping between input and output.

It is important to remember that these are theoretical results. In practice, there are many tricks to reducing the number of steps for each part of this algorithm. For example, in general d is chosen carefully to make encryption faster than decryption, simply because for binary computers exponentiation by certain powers is easier than others. Furthermore, more complicated techniques for multiplication can reduce the running time drastically at large key sizes.

DES and Triple-DES

Before there was RSA, there was DES. This is the Data Encryption Standard of the US Government, though it is about to be superseded by AES, the Advanced Encryption Standard. The form of AES has not yet been decided upon, however, so this paper will stick to analysis of DES. Besides, Schneier et al. have already written a nice analysis of the running times of all the AES candidates (see the references section). The Standard is based around DEA, the

Data Encryption Algorithm, which uses a 56-bit key in a 16-round Feistel cypher to encode data.

DEA works by breaking the message into blocks. Each block is then split in half. Half the block is transformed using a subset of the key; the other half of the block is XORed with the transformed half. The two halves are then swapped, and the process is repeated 15 more times, but with no swap on the last round. The resultant block is added to the encrypted message, and is chained into the encryption of the next block. Decryption follows the same steps, but the subkeys are enumerated in reverse order, and decryption starts at the end of the message and works backwards.

Assuming the transform takes time $O(\lg^2 n)$, similar to multiplication, this algorithm takes $O(\lg^2 n)$ total time; the XORing and concatenation steps are dominated by the transform. For this reason, DES is much faster than RSA. Depending on the hardware configuration and optimization of the algorithms, DEA tends to be 100 to 10000 times faster than RSA. But DEA does not have the advantage of arbitrary secure communication that RSA does. Hence, it is common practice to use RSA to exchange a secret key code for a subsequent communication using DES, or some other fast algorithm. The communique containing the secret key to be used is sometimes called a digital envelope.

As DES is actually a rather old standard, it has fallen to attacks from powerful hardware in recent years. With only 2^{56} possible keys, exhaustive search is a realistic option on a network of powerful machines. Recently the RSA Labs DES Challenge reported breaking DES-56 in 56 hours on a specially-built machine. Hence, it has become customary to increase security by encrypting the message multiple times with the DES algorithm. The most effective method for this consists of first encrypting the message with one key, then decrypting the resultant method with another key, and finally encrypting that result with a third key. This is provably harder to break than merely encrypting the message with one key, or with other variants on the encrypt/decrypt sequence. It increases the brute-force cost of breaking the message to at most 2^{168}, although because of interactions between keys the cost is somewhat reduced. This is much harder to achieve; on that same box mentioned, it would take 10^{35} hours, or about 10^{31} years, to crack.

There are many algorithms based on Feistel cyphers; in fact, the entire class of algorithms is sometimes referred to as DES-like algorithms. Blowfish and IDEA are two such cyphers. As the analysis of the running time of these algorithms tends to be similar to DES, they will not be expanded upon here. Some of these are quite simple; it is worth noting that RC4, a symmetric stream cypher, can be simply expressed as:

```
#!/usr/local/bin/perl -0777-- -export-a-crypto-system-sig -RC4-3-
lines-PERL

@k=unpack('C*',pack('H*',shift));for(@t=@s=0..255){$y=($k[$_%@k]+$s[$
x=$_

]+$y)%256;&S}$x=$y=0;for(unpack('C*',<>)){$x++;$y=($s[$x%=256]+$y)%25
6;

&S;print pack(C,$_^=$s[($s[$x]+$s[$y])%256])}sub
S{@s[$x,$y]=@s[$y,$x]}
```

Note that possession of this algorithm or a reproduction of such by a non-US citizen is prohibited under US law, as is the export of the algorithm; hence, this paper is now a munition, and as such must not be exported from the US or used by non-citizens. A tattoo of this algorithm in machine-readable bar code recently caused quite a stir in the government. Similar tricks can be played with the other cryptographic algorithms; for example, a simple working implementation of the RSA algorithm can be expressed in only two lines of Perl:

```
#!/usr/local/bin/perl -RSA-2-lines-PERL

print pack"C*",split/\D+/,`echo
"16iII*o\U@{$/=$z;[(pop,pop,unpack"H*",<>

)]}\EsMsKsN0[lN*1lK[d2%Sa2/d0<X+d*lMLa^*lN%0]dsXx++lMlN/dsM0<J]dsJxp"
|dc`
```

Unfortunately, this version does not do key generation, and so cannot be considered a complete implementation of RSA.

Authentication

Encryption is not the only application for cryptography. Often, what is required is not the obfuscation of information to prevent eavesdropping, but preventing undetectable alteration of that information, and the positive identification of the sender. The two applications turn out to be closely related. Signing a message is useless if the message can be altered without detection; similarly, knowing that a message is in its original form is less useful if you don't care who sent it, since then another user could just fabricate a similar message and pass it off as the original. Hence the two are usually rolled into one application, authentication. Authentication can be achieved through permutations of the techniques described thus far, or through new algorithms. Both methods of achieving this will be examined in this section of the paper.

The basic idea behind authentication is to create some sort of summary of the message, which then can be encrypted by the sender. Any interested recipient can then generate their own summary of the message, decrypt the summary in the message, and compare the two. If they match, both the content of the message and the identity of the sender are verified, as long it is unlikely that decrypting the encoded digest with a different key than it was encoded with would produce the correct digest for a message.

It is perhaps easiest to see how to use RSA for this purpose; then anyone can decode the digest using the user's public key. However, it is not clear what the digest function should be. At its base, the digest algorithm is a sort of hashing algorithm; it takes a large number of possible inputs, and produces a reproducible but smaller number for each. The trick is creating a digest algorithm that it is difficult to predict the output of; otherwise, it would be easily defeated. As a trivial example, imagine using simply a parity algorithm as the digest function. While simple to compute, and nice and small, it is simplistic to spoof; almost any message can be altered to change its parity to the one indicated in the digest. Likewise, it should be impossible to compute any information about the message given just the digest; such a hash is called a one-way hash.

There are many methods used currently to perform authentication. Here we will examine one of the most popular, MD5. This stands for Message Digest 5, and is a variant on earlier digest algorithms MD2 and the failed MD4. MD5 takes an input of arbitrary length and returns a 128-bit digest, sometimes called a fingerprint, of that input. To do this, MD5 performs the following steps:

1. Pad message to length 512x+448 bits, adding at least 1 bit

2. Append the original length of the message in bits as a 64-bit number

3. Initialize 4 32-bit registers ABCD to a fixed value (0x01 23 45 67 ... EF FE .. 01)

4. Process the message in 16-word (512-bit) blocks. The exact steps taken here is somewhat extended; for our analysis it is sufficient to summarize, and say that there are 4 rounds of 16 transforms, with each transform of the form:

a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s)

where a, b, c, d are each one of the registers A, B, C, D; X[] and T[] are precomputed tables; i, k, s are integers; and F(b,c,d) is a binary function of three inputs.

This step can be seen to be in $O(\lg n)$, as addition and F() are both in $O(\lg n)$, and all other operations are $O(1)$. But n here is the size of the registers ABCD, in this case, 32 bits. Hence the entire step is independent of the input size, and is $O(1)$

5. The digest consists of the final values of A, B, C, D.

Given an input m of size (lg m) bits, the running time is simply $O(\lg m)$. Obviously, using big-O notation is taking away half the story; the refinements to the various digest algorithms over the years have mostly involved reducing the constant time per block. But it is obvious from this analysis that creating a digest for a message is much faster than encryption, even for a rather long message.

Conclusions

The speed of modern computing is cryptography's savior and its curse; the power that allows strong cyphers also allows the breaking of these cyphers. In general, however, it seems that the balance has been restored between cryptographers and cryptanalysists. Because of the availability of great amounts of spare processor time via the Internet, small numbers of problems can be solved relatively quickly. For older encryption algorithms and methods, the favor lies with the crackers. But as problems can be made arbitrarily difficult, in general by picking large key sizes, the balance swings back toward the encoders.

As an example, the current RC5-64 cracking effort, managed by distributed.net, is attempting to crack a message encoded with RC5-64. RC5 is a secret-key encryption algorithm which is an extension of the RC2 created by Ron Rivest. It allows a variable key size; here, the key size is 64 bits. At the time this was written, well over 30,000 machines around the world were devoting at least part of their processor time to a brute-force search of the keyspace. The current sustained rate of key-checking hovers around 45 billion keys per second. However, even at this rate the search space is barely breached, despite the fact that the problem has been worked on for over a year. At the current rate the search space should be exhausted in about 5000 days; however, if the rate of growth of speed is maintained, a solution should be arrived at much faster than 16 years from now. Nonetheless, it is a sobering thought that a simple increase of a bit or two in the key size would effectively put the problem completely out of the reach of brute force algorithms on current hardware.

A much more serious threat to cryptographic security is the possible advent of quantum computers. If theory can be turned into practice, these machines should be able to perform operations that currently require enormous iteration in a single step. Provided the system is set up correctly, a quantum computer should be able to merely collapse from a multi-state distribution of all possible inputs to the single correct output state in linear time. This would reduce most of the algorithms discussed above to polynomial time, and truly threaten the security of all the algorithms discussed. However, these computers are currently highly theoretical, due to the intractable nature of some of the equations that must be solved to understand how to construct such a computer. While a full discussion of quantum computers is off topic, I feel my Physics blood calling to me, and so I might end up writing such a paper later.

References

Cormen, T., Leiserson, C., Rivest, R. Introduction To Algorithms, MIT Press, 1990

Kaliski, B. RFC 1319, "The MD2 Message-Digest Algorithm"
`http://sunsite.doc.ic.ac.uk/computing/internet/rfc/rfc1319.txt`

Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N. "Performance Comparisons of the AES Submissions". Available from
`http://www.counterpane.com/AES-performance.html`

Rivest, R. RFC 1320, "The MD4 Message-Digest Algorithm"
`http://sunsite.doc.ic.ac.uk/computing/internet/rfc/rfc1320.txt`

Rivest, R. RFC 1321, "The MD5 Message-Digest Algorithm"
`http://sunsite.doc.ic.ac.uk/computing/internet/rfc/rfc1321.txt`

RSA Laboratories FAQ 4.0; available from
`http://www.rsa.com/rsalabs/faq/`

Weisstien, E., ed. The CRC Concise Encyclopedia of Mathematics, CRC Press, 1998 Portions of this reference are available from
`http://www.astro.virginia.edu/~eww6n/math/math0.html`

Information on the RSA Labs DES II Challenge cracking effort is available from
`http://www.rsa.com/rsalabs/des2/`

Information on distributed.net's effort to meet the DES II Challenge is available from
`http://www.distributed.net/rc5/`

Ron Rivest has collected just about all the resources available on the WWW for cryptography on his homepage:
`http://theory.lcs.mit.edu/~rivest/crypto-security.html`

If you need to prove to yourself that testing the primality of large numbers is hard, the two most recently-discovered Mersenne primes are available for checking from `http://www.mersenne.org/files/prime.txt and`
`http://www.mersenne.org/files/prime2.txt`