Ruby Block Cipher Mark 5

Inventor & Author:

January 4 1996

Table of Contents

1. SUMMARY

The Ruby Block Cipher consists of an encryption process that can be used to solve problems involving authentication, protection of privacy, protection of proprietary data, and generation of high quality pseudorandom numbers for simulation purposes. The primary advantage of the Ruby Block Cipher over existing solutions is that it is small enough and fast enough to permit its use within the firmware of a data storage device or backup program, while still being more secure than comparable existing solutions.

Ruby Block Function is a trade-off between speed, size, and security for software and firmware applications that is not intended for very high security, but for reasonable protection of privacy at minimal cost. I explain some applications and modes of use of the Ruby Block Cipher in this document, even though these are not really new.

The Ruby Block Function is essentially a cryptographic hash function with too small of a block size for serious use as a cryptographic hash, but a large enough block size (64 bits) to use as a building block for a Message Digest Cipher.

2. BACKGROUND

While contemplating the need for improvement in security for portions of an embedded controller for a tape drive, it occurred to me that (1) it would be very desirable to embed a small encryption function in the tape drive for the purpose of authentication, but (2) the encryption functions that I knew about (lots of them) were either too slow, too big, or too insecure to seriously consider. (The RC-4 and RC-5 algorithms, both made public after I started work on this algorithm, might have worked OK, as well). I also considered the software implementation speed problems of using real encryption to protect the privacy of data backed up on tape, which was a related, but different problem.

I was looking for something that was very fast on a general purpose microprocessor with a 32-bit word size, that was at least as secure as a dead bolt lock on a solidly built wooden door, and that was small enough to fit in an embedded application.

The result of this brainstorm was named the Ruby (because it was a pretty, little gem of an idea) Block Cipher.

This document describes the "Mark 5" modification of the Ruby Block Cipher. The original Ruby Hash Cipher suffered from some statistical imbalances in its output that are corrected, using the new internal structure, as did the Mark 2 version. The Mark 3 version corrected this problem, but was vulnerable to an analytical cryptanalysis technique known as differential cryptanalysis. The Mark 4 version corrects this problem with the reinstatement of data-driven rotations that were used in the original version, but had a problem with slow avalanche of worst-case key and data input. The Mark 5 version has guaranteed faster avalanche for all data patterns and corrects some implementation bugs in the reference implementation.

3. DESCRIPTION OF RUBY BLOCK CIPHER

The Ruby Block Cipher can be used to

- determine if an authorized device is connected to a diagnostic port for access control purposes,
- encrypt and decrypt bulk data, given a 64-bit key for security and privacy,
- create a 64-bit "fingerprint" or hash of a longer series of bytes for better key management (especially involving pass phrases), or
- generate high quality pseudorandom numbers.

All of these uses build on the same Ruby Block function, which is used in different modes.

3.1 RUBY BLOCK CIPHER MODES OF OPERATION

The five normal modes of operation of the Ruby Block Cipher are (1) authentication, (2) encryption, (3) decryption, (4) hash generation, and (5) pseudorandom number generation. Note that the block size of the Ruby Block Function is too short for use in digital signature applications.

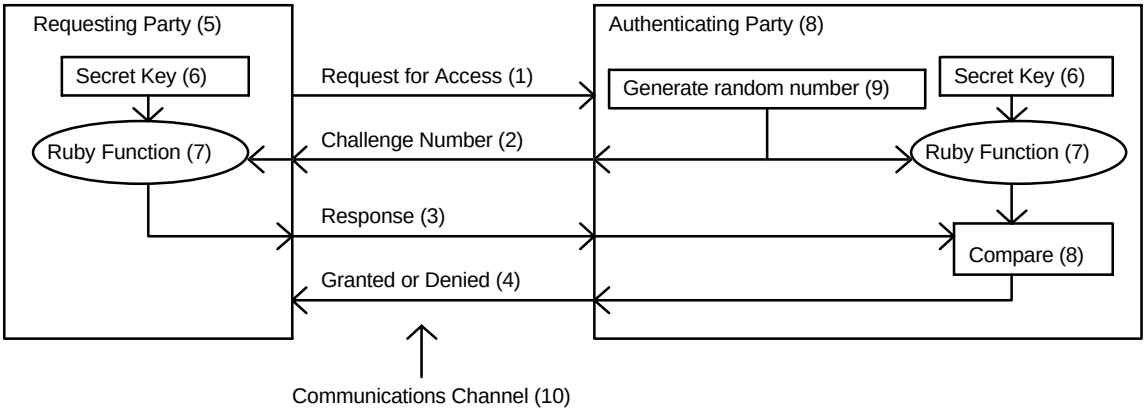
Because the Ruby Block Cipher is not a reversible block cipher (like DES, IDEA, and Diamond2), it must always be used in a chaining mode that uses only the "encrypt" function of a block cipher when used for encryption or decryption.

3.1.1 AUTHENTICATION

Authentication with the Ruby Block Cipher is based on a challenge and reply protocol. In the process of this protocol, the other communicating device proves that it has a copy of the secret key, without revealing what the secret key is. It is assumed the secret key will be protected carefully, and that if it is embedded in a program, that steps will be taken to make reverse engineering the program to get the key reasonably difficult.

Refer to figure 3.1.1. The requesting party (5) sends a request for access (1) to the authenticating party (8) over the communications channel (10). The authenticating party (8) then generates a random number (9) and sends it over the communications channel (10) to the requesting party (5). The requesting party then demonstrates possession of the secret key (6) by returning the output of the Ruby Block of the random number (9) and the secret key (6). The authenticating party does the same calculation, and compares the answers(8). If the answers are the same, the request is granted (4). If they are not, then the request is denied. Observing the messages on the communications channel (10) does not reveal the secret key, because the Ruby Block function is designed to make solving for the key given input/output pairs infeasible.

Figure 3.1.1 Authentication Protocol

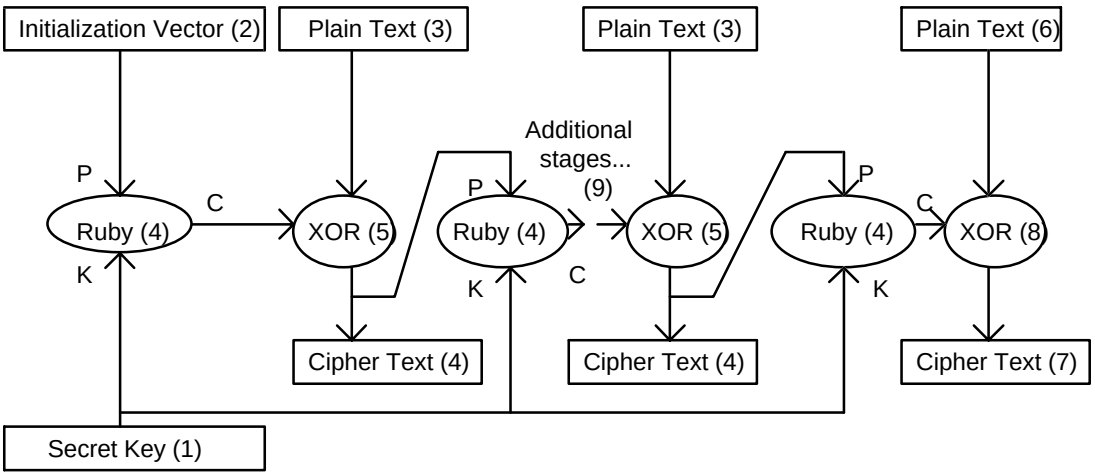


Once the authentication protocol is completed, the authenticating party is reasonably sure that the requesting party has the secret key, even though the key was never transmitted in the clear. This protocol cannot be spoofed by unauthorized requesting party playing back a previous transaction, since the random number will be different each time.

3.1.2 ENCRYPTION

The recommended way to use the Ruby Block Cipher to use standard block chaining with ciphertext feedback. See figure 3.1.2.

Figure 3.1.2 Encryption



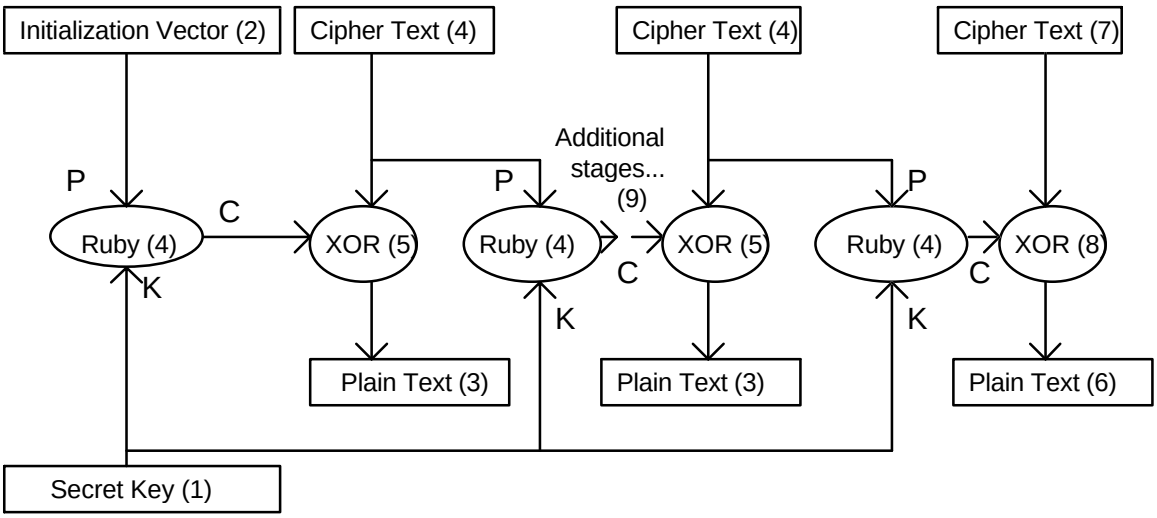
The secret key (1) is combined with an initialization vector (2) using the Ruby Block Function (4) to start the process. The initialization vector need not be kept secret, but should be different for each block of data encrypted. For example, the initialization vector may be the logical block number on a sequential storage device. This value is then combined with the first plain text block (3) using bit-wise addition modulo-2 (also called exclusive-or, XOR for short). This results in the first cipher text block (4). All subsequent plain text blocks are encrypted by combining them with the output of the Ruby Block Function (C) of the previous cipher text block (4) and the secret key (1). All plain text and cipher text blocks are 8 bytes (64 bits) long, except for the last one of each. The last cipher text block (7) will be the same length as the last plain text block (6).

3.1.3 DECRYPTION

Decryption is guaranteed to recover the original plain text, provided the same initialization vector and secret key are used, by the properties of the exclusive-or operation, together with the fact that the Ruby Block Function is deterministic.

See figure 3.1.3. The decryption operation works when the proper secret key (1) and initialization vector (2) are used, since the output of the Ruby Block Function (4) (C) will be the same as it was during encryption. This ensures that the exclusive-or operation (5) will convert the cipher text (4) back to the original plain text (3). The last blocks (7) and (6) may be shorter, but the principle involved is the same. Note that if the wrong secret key is used, the input to each of the exclusive-or operations will be different in an unpredictable way, thus producing garbage for output.

Figure 3.1.3 Decryption



3.1.4 HASH GENERATION

Note that the size of the hash produced by this process is too small for some applications, like digital signatures, which need at least a 128 bit hash (preferably at least 160 bits). This is a good way to reduce a longer pass phrase to 64 bits for use as an encryption key, however.

Hash generation (also called message digest generation) is a four step process:

1. Initialize the system. Set the byte counter and hash state array to zero.

2. Append the length (modulo 2^{32}) of the input to the input as 4 bytes, least significant byte first.

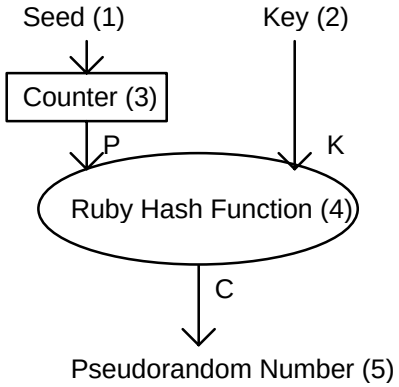
3. Append as many bytes of 0xFF bytes to the input as it takes to make the input an even multiple of 8 bytes.

4. For every 8-byte block of input, update the new hash state array to be the Ruby Block Function of the input block (as the plain text) and the old hash state (as the key). The last output of the Ruby Block function is the hash of the input bytes.

3.1.5 PSEUDORANDOM NUMBER GENERATION

Pseudorandom numbers are numbers arranged in a sequence which, other than the fact that the sequence can be repeated, appear to be perfectly random. Pseudorandom numbers have numerous applications in hardware and software testing, modeling, and in the generation of session keys for cryptographic applications. A pseudorandom number generator of the type listed below is also suitable for creating a stream cipher that can be used for random access to an encrypted file on disk or tape.

Figure 3.1.5 Pseudorandom Number Generation



See figure 3.1.5. To generate pseudorandom numbers, start the sequence by setting the seed (1) to an initial value, which may be zero, or may be set to any desired starting point in the sequence. The seed is loaded into a counter (3), that is incremented every time a new 8-byte pseudorandom number is desired. The output of the counter (3) is combined with a fixed key (2) using the Ruby Block Function (4) to create the pseudorandom number (5). The fixed key (2) need not be kept secret unless the generated pseudorandom numbers are used for key generation or encryption purposes. The output pseudorandom number (5) may then optionally be processed to produce numbers in the desired range and with the desired probability distribution function.

3.1.6 STREAM CIPHER MODES

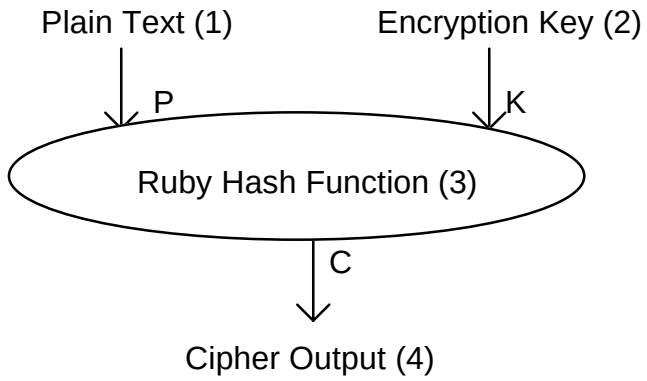
There are several possibilities of ways to convert a block cipher to a stream cipher. One of the simplest methods is to simply use a counter (preloaded

with an initialization vector) as the input to the Ruby Block Function (along with the key), and exclusive-or each bit of the input stream with a bit of the Ruby Block Function output. Note that this is the same as the pseudo-random number generator of figure 3.1.5, except that the output is used as the cipher stream. After each 64-bit output block is used up, the counter is incremented by some constant value (typically 1). Decryption works the same way. As with any stream cipher, a given initialization vector and key combination should only be used once.

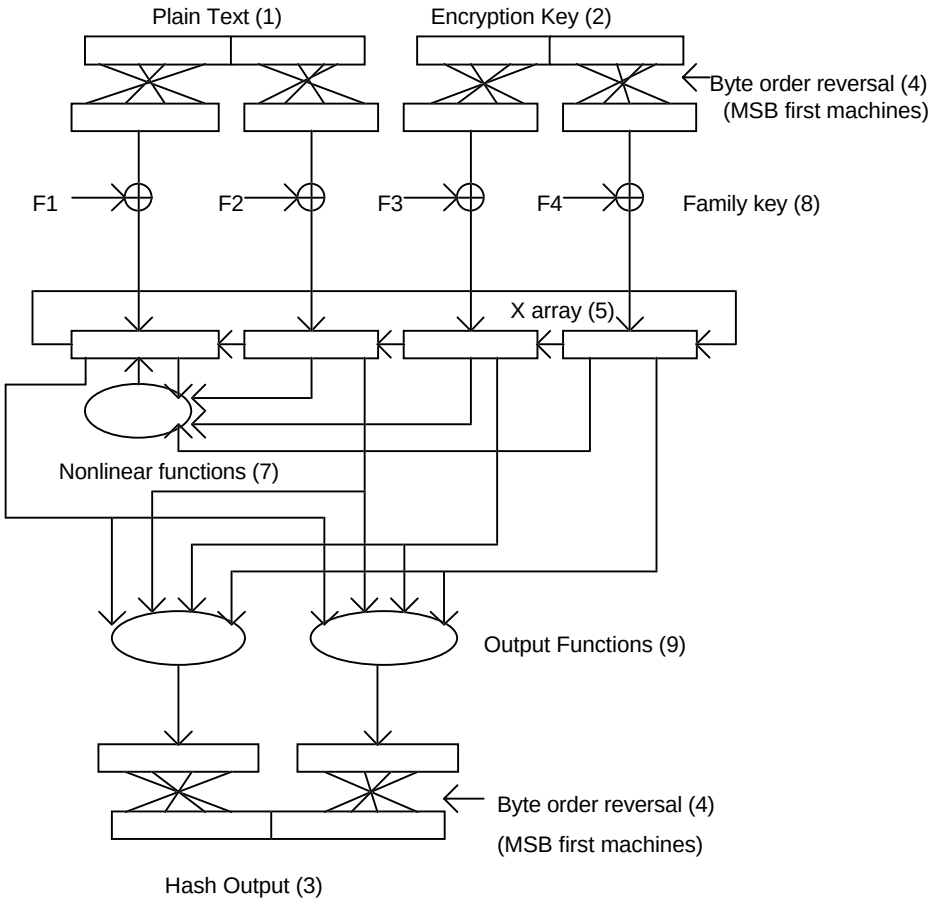
3.2 RUBY BLOCK FUNCTION STRUCTURE

The Ruby Block Function (3) takes two blocks of 64 bits as input, and produces an output of one block of 64 bits. When used as an encryption algorithm, the first input is the plain text (1) and the second input is the encryption key (2). The cipher text is formed by combining the output of the cipher (4) with the input as discussed above. The nature of the process used to determine the output makes it computationally infeasible to derive the key or other plain text blocks, given a pair of corresponding plain text and cipher text blocks.

Figure 3.2 Ruby Hash Function



3.3 INTERNAL STRUCTURE OF RUBY BLOCK CIPHER



The fundamental computational unit within the Ruby Block Cipher is the 32-bit unsigned integer, corresponding to the natural word size of many general purpose computers and microcontrollers. To provide for portability of operation between computer architectures with most significant byte first and least significant byte first ordering of 8-bit bytes within 32-bit words, the order of input bytes within 32-bit words is reversed on computers with most significant byte first architectures after the input stage and before the output stage (4).

A "Family Key" (8) consisting of two randomly chosen (except that the least significant bit should be 1) 32-bit integers are added or multiplied into the intermediate results as part of the nonlinear functions. This Family Key is normally kept constant for any cryptosystem using this cipher, but could be made part of the user key for a larger effective user key size.

The nonlinear feedback functions (7) and output functions (9) are carefully chosen to make cryptanalysis difficult by mixing nonlinearities involving both arithmetic operations, boolean functions, and data-dependent bit rotations, while still operating quickly on a 32-bit computer. When iteratively applied to the X array (5) that is filled from the input values, as shown, with each array rotated by one long word each iteration, the relationship between the output and the four input long words becomes very complex. Normally, there are at least eight iterations, but more may be used where security is more important than speed performance.

For more details (including the definitions of the feedback and output functions), please see the C source code of the prototype implementation of the Ruby Block Cipher, below.

3.4 RUBY SOURCE CODE

Because C source code is so much more compact and precise in describing an encryption algorithm than English, I include the following code fragments. The complete source code for a functional (but simple) encryption program that demonstrates the use of this cipher is available separately in the USA and Canada on the Internet at **ftp://ftp.csn.net/mpj/I_will_not_export/crypto_???????/libraries/ruby_m5.zip**, where the ??????? is revealed in **ftp://ftp.csn.net/mpj/README.**

3.4.1 RUBY.H

```
/* Ruby.h -- Ruby hash cipher prototype

Copyright (C) 1994-1996 Michael Paul Johnson.  All rights reserved.
Inventor and author: Michael Paul Johnson.
Mark 5 modifications: 4 January 1996

The Ruby block cipher is really a small hash function that must be
used in an appropriate chaining method to be reversible.  In other
words, it lacks an electronic codebook mode.  On the other hand, the
electronic codebook mode is one of the least secure modes to use a
block cipher in.  The main advantages of the Ruby block cipher are
(1) speed in software implementations, (2) size, (3) no delay for key
setup (making key feedback modes practical), and (4) reasonable
security for the speed.

The speed of the Ruby cipher derives from the use of 32 bit
operations common to most computers, and from limiting the number of
operations per block encryption.  The operations used are add,
exclusive or, multiply, and logical rotate.

Because there is no separate key setup step to generate internal keys
from an external key, there is no performance penalty for changing
keys often.  This encourages the use of frequently changed session
keys.  It also makes feedback modes that continually change the key
practical.

The security of the Ruby cipher is based on the mixing of both
arithmetic and logical operations, including the very nonlinear
rotation operation, in a finite set of numbers.  The structure used
for this mixing is intended to frustrate computation of the inputs
```

```
from the outputs.  The structure is also designed to mix the effects
of each bit of the input (both the ciphertext and key sections) so
that changing one bit of the input changes an average of about half
of the bits of the output.  This is sometimes called the avalanche
effect.

The Ruby cipher has not yet been fully analyzed for resistance to
cryptanalytic attacks, but was designed with a knowledge of how
several of these attacks work.  The best attack on this cipher might
well be better than a brute force attack on the 64 bit key, but since
some of the more obscure attacks require large quantities of cipher
text encrypted with the same key, I recommend that you use this
cipher in a way that uses a different session key to encrypt each
file or message, even if you use one master key to encrypt the
session keys.

The primary uses intended for the Ruby cipher are authentication and
encryption in embedded applications (such as tape drives) and in
software such as backup programs and secure communications
applications where high speed, small size, and efficient
implementations on 32-bit computer architectures are very important
considerations.

In the following function, P is the Plain text input, K is the Key,
and C is the Cipher text output.

*/

void ruby_crypt(unsigned char *P, unsigned char *K, unsigned char *C);

/*

The following hash functions are NOT intended for use in digital
signature applications, since the block size for Ruby is too short
for that.  They are convenient for reducing a longer passphrase (or
passphrase + salt) to a Ruby key of 64 bits.

*/

void ruby_hash_init(void);
void ruby_hash_update(unsigned char *b, unsigned count = 1);
void ruby_hash_final(unsigned char *hash);
```

3.4.2 RUBY.CPP

```
/* Ruby.cpp -- ruby hash cipher MARK 5 prototype

Copyright (C) 1994-1996 Michael Paul Johnson.  All rights reserved.
Inventor and author: Michael Paul Johnson.
Date invented: 27 July 1994.
Mark 5 modification: 4 January 1996

You may use and distribute this software without payment of royalties
provided that you retain this copyright notice in the source code and
that you do not misrepresent the source of this software.  You are
responsible for compliance with any applicable export regulations.

Note that this is experimental software that implements and describes
an experimental algorithm.  Use it at your own risk.  Neither I nor
Exabyte Corporation will be held responsible for any damages caused by
this program or algorithm.

*/

#ifdef __MSDOS__
#include <mem.h>
#else
#include <memory.h>
#endif

/* #define BIG_ENDIAN /* If machine is MSByte First, for compatibility. */

/*
```

```
The STRENGTH constant is where you get to trade off cryptographic
strength and speed.  The following nonscientific table is presented
as a guideline for selecting a constant.  More study is needed before
the cryptographic strength associated with the value of this constant
can be assigned a more meaningful measure.

4   Minimum       -  speed is of the essence, security secondary.
8   Desk lock     -  reasonable compromise of speed vs security?
16  Dead bolt     -  probably good enough for most things.
20  Portable safe -  security is more important than speed.
32  Anchored safe -  speed isn't much of a concern.
40  Bank vault    -  your pentium has nothing better to do, anyway.
64  Fort Knox     -  If you are willing to wait this long, it would probably
                     make sense to double the block size, too.
*/

#define STRENGTH 8

/*

The family key is a set of two 32-bit integers with a "fairly even"
distribution of ones and zeroes.  These two numbers should not be a
power of 2 (i.e., they should be relatively prime to 0x100000000L).
This means that the effective family key length is a total of 62 bits,
since the two least significant bits are always 1.

This number has two functions.  The most important one is to help
spread the influence of key bits into the "zone" that affects the roll
operations. Another is to give you a convenient way to make your
algorithm "proprietary" to your application.  If interchange with
other Ruby Mark 5 implementations is important, I recommend the use
of 0x456C6091 and 0xAA7110C3.  The real strength is in the user's
key, not the family key, especially if you embed the family key in
the software.

Defining an enlarged user key of 126 bits and using the extra 62 bits
for the family key is an idea I considered, but I really can't
guarantee that this would really increase the security of the
algorithm as much as the key size would seem to indicate. (It
wouldn't be worse than a 64-bit key, but it probably wouldn't nearly
square the strength of the cipher, either).

*/

#define FAMILY_KEY_1 0x456C6091L
#define FAMILY_KEY_2 0xAA7110C3L

#define FASTER_NOT_SMALLER /* if speed is more important than code size.*/

#ifdef FASTER_NOT_SMALLER

/* Does your processor have a logical rotate instruction? If so, you can
   speed this up.
*/

#define roll(X, C) (((X)<<(int)(C))|((X)>>(32-(int)(C))))

#else

/*

Use the roll function instead of the macro if the space consumed by the
expansion of 12 roll macros consumes too much code space.  This might be
significant in some embedded applications.

*/

unsigned long roll(unsigned long X, unsigned long C)
    {
    return (X<<(int)C) | (X>>(32-(int)C));
    }

#endif
```

```
#define f1(A,B,C,D) (((roll(A,D&0x1F)+roll(B,C>>27))^(roll(C,B&0x1F)+roll(D,A>>27))))
#define f2(A,B,C,D) (((roll(A,C&0x1F)^roll(B,D>>27))+(roll(C,A&0x1F)+roll(D,B>>27))))


static unsigned char hash_state[8];
static unsigned char hash_partial[8];
static unsigned long hash_count;
static unsigned hash_position;

void byte_swap(unsigned char *X)
    {
    unsigned char b;

    b=X[0];X[0]=X[3];X[3]=b;
    b=X[1];X[1]=X[2];X[2]=b;
    }

void ruby_crypt(unsigned char *P, unsigned char *K, unsigned char *C)
    {
    int I, J, H, L, M;
    unsigned long X[4];
/*

    It is convenient to call this function with pointers to bytes, but
    the input and output are handled as unsigned longs.
    The key should be selected randomly.
*/

    X[0] = *((unsigned long *)P);
    X[1] = *(((unsigned long *)P)+1);
    X[2] = *((unsigned long *)K);
    X[3] = *(((unsigned long *)K)+1);
#ifdef BIG_ENDIAN
    byte_swap((unsigned char *)(X));
    byte_swap((unsigned char *)(X+1));
    byte_swap((unsigned char *)(X+2));
    byte_swap((unsigned char *)(X+3));
#endif

    for (I=0,J=0,H=1,L=2,M=3; I<(STRENGTH); I++)
        {
        X[J] = f1(X[J],X[H],X[L],X[M]);
        X[J] += FAMILY_KEY_1;
        J = (J+1)&3;
        H = (H+1)&3;
        L = (L+1)&3;
        M = (M+1)&3;
        X[J] = f2(X[J],X[H],X[L],X[M]);
        X[J] *= FAMILY_KEY_2;
        J = (J+1)&3;
        H = (H+1)&3;
        L = (L+1)&3;
        M = (M+1)&3;
        }

    *((unsigned long *)C) = roll(X[0], X[3]&0x1F) + roll(X[1], X[2]>>27);
    *(((unsigned long *)C)+1) = roll(X[2], X[0]>>27) ^ roll(X[3], X[1]&0x1F);

#ifdef BIG_ENDIAN
    byte_swap(C);
    byte_swap(C+4);
#endif
    }

void ruby_hash_init(void)
    {
    hash_position = 0;
    hash_count = 0L;
    memset(hash_state, 0, 8);
    }

void ruby_hash_update(unsigned char *b, unsigned count)
    {
    unsigned u;
```

```
    for (u=0; u<count; u++)
        {
        hash_partial[hash_position++] = *b++;
        if (hash_position >= 8)
            {
            ruby_crypt(hash_partial, hash_state, hash_state);
            hash_position = 0;
            }
        }
    hash_count += count;
    }

void ruby_hash_final(unsigned char *hash)
    {
#ifdef BIG_ENDIAN
    byte_swap((unsigned char *)(&hash_count));
#endif
    ruby_hash_update((unsigned char *) (&hash_count), 4);
    while (hash_position < 8)
        {
        hash_partial[hash_position++] = 0xFF;
        }
    ruby_crypt(hash_partial, hash_state, hash_state);
    memcpy(hash, hash_state, 8);
    }
```

4. SECURITY ANALYSIS

This block cipher should be used with great caution, and only in noncritical applications right now, because it has not be analyzed very well, nor have very many people looked at it. Since the initial publication is only in electronic form and outside of some of the traditional fora for the cryptographic community, it may not get the scrutiny enjoyed by other algorithms. It might, indeed be clever and strong, but someone might also come up with a clever attack that I didn't anticipate. Use this at your own risk.

Security analysis is one of the most difficult subjects in cryptography. With some notable exceptions, such as the classic One Time Pad, there can be no absolute proof of the security of a means of encryption, since that amounts to a proof that a method to break the means does not exist. There are, however, some things that can be done to either increase the confidence in a method or disprove it. In the case of a block cipher, the cipher is called "strong" if there is no better means to solve for the key, given a set of corresponding plain text and cipher text blocks (the known plain text attack) or given that the attacker can have any block encrypted with the unknown key (the chosen plain text attack).

As a general rule, the more people who are skilled in mathematics and cryptanalysis who have tried to break a cipher and failed, the more confidence is gained in that cipher. This is one reason why it is important for the security of the cipher to rest in secrecy of the key, and not in the secrecy of the algorithm. It is also a very good reason to publish an algorithm and subject it to peer review.

Three general forms of attack will be considered here, but only briefly. The first, brute force, can be applied to any block cipher. The second, divide and conquer, is a general technique for substantially reducing the apparent strength of a cipher that must be avoided in cipher design. The third, the analytical attack, takes advantage of the internal structure of the cipher.

4.1 BRUTE FORCE

A brute force attack consists of simply trying all of the possible keys until one works. It is generally assumed that you will know when you have the right key because you know enough about what is encrypted to recognize something that "makes sense." The defense against this attack is to simply make the number of possible keys too big for this to be practical. Since there are 2^{64} possible keys, you would expect to have to try about half of them (2^{63}) before finding the right one. If you spent lots of money on a very fast, special purpose parallel processor that could try a billion (10^9) keys per second, it would take about

$$\frac{(2^{63}\text{keys})(10^{-9}\text{ seconds/key})}{(86{,}400\text{ seconds/day})(365.25\text{ days/year})} = 292\text{ years to break.}$$

This is probably adequate for a large number of applications, especially since such a fast processor would probably cost more than the value of the data being protected.

4.2 DIVIDE AND CONQUER

If there were a way to solve for individual parts of the key, without knowing rest of the key, then it would be possible to do a brute force style attack on the individual pieces, thus taking significantly fewer operations. The Ruby Block Cipher is designed such that the key is essentially a monolithic whole, together with its effect on each input block. Even a small, one bit change in the key results in a drastic change in the output, changing an average of about half of the output bits. This frustrates attempts to see if a trial key is "close" to the real key.

Another division that might be attempted is to solve for the internal state of the cipher for each iteration individually, using a chosen plain text attack. Such attacks get much harder with more iterations (since there are more internal states to solve for). This is a good argument for more rounds in higher security applications. With the minimum of eight iterations, there are twelve unique internal state variables of 32 bits each. Solving for these variables is frustrated by the inherent data loss in hashing 16 bytes down to

8 bytes and the nonlinearities of the functions used (especially the bit rotation and bit selection operations, which are not well behaved things that mathematicians have powerful tools to manipulate).

There is some risk that someone will see a way to solve for parts of the key with selected input/output pairs. Just because I don't know of a way doesn't mean that it is impossible.

4.3 ANALYTIC ATTACK

There is a multitude of analytic attacks possible. If I knew of one that worked on this cipher, I would have redesigned it to be resistant to that attack. Systems of linear equations cannot be used to solve for the key and the internal variables, since the functions used are asymmetrical and nonlinear. A piece-wise linear decomposition of the problem becomes intractable because the data dependent bit rotations effectively break the linear functions in pieces at different points. There is a possibility that an analytic attack exists or may be discovered, but some of the fundamental math problems that exist in this type of solution are similar to those used in the IDEA™ cipher, which has survived a few years of scrutiny by academia. Still, there are some fundamental differences that may make such a solution possible.

5. COMPARISON WITH PRIOR ART

The closest prior art is the IDEA™ Cipher of U. S. Patent #5,214,703, along with some ideas taken from Peter Gutmann's Message Digest Cipher (MDC), documented in the shareware product called "Secure File System." A discussion of cryptographic prior art is contained in Steven T. Degele's verbose U. S. patent #5,297,207, issued March 22, 1994. An excellent survey of the current state of the art is in Bruce Schneier's book, "Applied Cryptography."

Data-dependent bit rotations are also used in Ron Rivest's RC5 cipher, which has a patent application pending. The structure of that cipher is different, since it is a general family of reversible block ciphers using data-dependent bit rotations as the only nonlinear operation.