© 1994 Michael Paul Johnson

The Diamond Encryption Algorithm

by Michael Paul Johnson

Abstract—**Diamond is a royalty-free, symmetric-key block cipher encryption algorithm based on a combination of nonlinear functions. This block cipher may be implemented in hardware or software. Diamond uses a block size of 128 bits and a variable length key. A faster variant of Diamond uses a block size of 64 bits. Diamond is an incremental improvement over MPJ2 and MPJ.**

Index Terms—**Diamond, encryption, cryptography, cryptanalysis, computer security, communications security, MPJ, MPJ2.**

I. INTRODUCTION

General symmetric key block ciphers have numerous applications in computer security, communications security, detection of data tampering, and creation of message digests for authentication purposes. The longer any one such algorithm is used, and the more use it gets, the greater the incentive to break it, and the greater the probability that methods will be devised to break the algorithm. For example Michael J. Wiener has shown that breaking DES is within the capabilities of many nations and corporations [1]. This sort of reduction in the relative security of DES was anticipated several years ago. One proposed solution is the International Data Encryption Algorithm (IDEA™) cipher [2], which was described in [3] and [4] as the Improved Proposed Encryption Standard (IPES). Another one is the MPJ Encryption Algorithm [5], which evolved to the Diamond Encryption Algorithm. In the field of cryptography, it is good to have many good algorithms available.

II. DESIGN OF DIAMOND

Diamond was designed to be strong enough to provide security for the foreseeable future. It was also designed to be easy to generate keys for, and to be practical to implement in hardware, software, or in a hybrid implementation.

A. Strength

Three major factors influence the strength of a block cipher: (1) key length, (2) block size, and (3) resistance of the algorithm to attacks other than brute force (such as differential cryptanalysis) [3] [6]. The key length is variable to allow you to select your own trade-off between security and volume of keying material needed. The block size is chosen to make brute force attacks using precomputed tables require an obviously intractable amount of data storage.

Diamond uses a variable length key. The use of at least a 128 bit key is recommended for long term protection of very sensitive data, as a hedge against the possibility of computing power increasing by several orders of magnitudes in the coming years.

The block size for the Diamond Encryption Algorithm is fixed at 128 bits, because larger block sizes are unlikely to make any practical difference in security, and because this in a convenient binary multiple. Diamond Lite has a block size of 64 bits because this is good enough for some applications, and because it allows a much faster total avalanche effect and greater software speed than the 128 bit block size.

The problem of making sure that there is no known attack that is more efficient than brute force is much more difficult than simply selecting sizes for keys and blocks. This is attempted by creating a composite function of simpler nonlinear functions in such a way that the internal intermediate results cannot be solved for and such that there is a strong dependence of every output

© 1994 Michael Paul Johnson

bit on every input bit and every key bit. An ideal 128 bit block cipher would use a **z** bit key to select one of $2^{\mathbf{z}}$ functions from the set of all one to one and onto functions that map one input block of 128 bits to one output block of 128 bits. Ideally, these $2^{\mathbf{z}}$ functions would be the most nonlinear, difficult to analyze functions out of the $(2^{128})!$ possible functions. In practice, the key selects one of $2^{\mathbf{z}}$ functions from an arbitrary selection of possible functions numbering between $2^{\mathbf{z}}$ and $(2^{128})!$.

The use of purely nonlinear functions makes a large portion of mathematical tools ineffective for cryptanalysis.

B. Ease of Key Generation

Key generation should be as simple as generating a random number by measuring some random physical process. Since there is no complex or secret strong key selection process, distributed key management protocols are practical. Distributed key management is preferable in many applications to centralized key management because there is no single point of failure at which the whole system could be compromised.

C. Practical in Hardware or Software

The prototype of the Diamond Encryption Algorithm is implemented in a program for a personal computer. When properly implemented in hardware, Diamond should not significantly slow down any practical digital data stream. On the other hand, setting up a new key need not be as fast as the encryption and decryption operations, since (1) key change operations are less frequent than encryption and decryption operations, and (2) a slower key setup operation discourages brute force attacks.

III. BASIS OF DESIGN

The thought process that went into the design of Diamond is based on the following ideas:

1. Linear functions and combinations of functions can often be solved analytically in ways that are not obvious to the cipher designer, and should be avoided. This includes standard arithmetic functions, math in finite fields, and Boolean arithmetic.

2. Reversible block ciphers with a block size of **n** bits can be viewed as a simple substitution cipher on an alphabet of $\mathbf{2^n}$ characters, with a key that selects the permutation used.

3. Simple substitution ciphers can be represented with a look-up table or array, but in practice the array required is too big to fit comfortably in a computer's memory.

4. An adequate subset of the oversized look-up table can be simulated by simply interleaving rounds of substitution of sub-blocks with bit permutations that serve to spread functional dependencies across sub-block boundaries.

IV. DESCRIPTION OF ALGORITHM

The Diamond Encryption Algorithm consists of three main parts: (1) key scheduling, (2) substitution steps, and (3) permutation steps. Encryption and decryption both consist of **n** rounds of substitution operations, where **n** is at least 10. Each substitution operation takes each of the 16 input bytes of 8 bits each, and substitutes another byte for it. This done with the contents of the substitution array for that byte position and round number. The key scheduling operation fills the internal substitution arrays based on the key. Between each substitution, a fixed permutation step uses a bit selection process to make each output byte a function of eight different input bytes. Unlike DES, every round alters every byte of the input block (instead of just half of the input

© 1994 Michael Paul Johnson

block). After 5 rounds, every bit of the output block is a nonlinear function of every bit of the input block and every bit of the key. The additional rounds after the fifth round serve to ensure that solving for the contents of the individual substitution arrays is more work than a brute force attack on the cipher. They also serve to increase the number of possible functional relationships that the key selects from, thus making this algorithm closer to the ideal block cipher, and making cryptanalysis more difficult.

A. Key Scheduling

There is one substitution array for each of the 16 bytes of the encryption block for each round. For a ten round implementation of Diamond, 160 substitution arrays are to be filled. Each of the 160 arrays contains 256 elements of one byte each. It is convenient to look at the set of substitution arrays as one three dimensional array, indexed by round, byte position within the 16 byte encryption block, and input byte value. A similarly indexed inverse substitution array is used during decryption. For the substitution to be reversible, each of the 256 possible values of an 8 bit byte must occur exactly once in the array. The process used to make this happen consists of five processes: (1) array filling, (2) element placement, (3) pseudorandom key expansion, (4) pseudorandom number normalization, and (5) array inversion. Although key scheduling can be done more quickly in a dedicated hardware implementation, a more economical hybrid design would do the key scheduling in firmware and the actual encryption or decryption in hardware.

Array filling is simply a nested loop where all 160 substitution arrays are filled. It is concisely expressed in this pseudo code:

```
For rounds := 1 to n
   For byte position := 1 to 16
      For element value := 255 down to 0
         Place this element.
```

Element placement is done by placing the current element in one of the unfilled positions in the current array. The unfilled positions of the current array are numbered from 0 to the value of the element being placed. A number in this same range is then selected by generating a pseudorandom number normalized to this much smaller range. This offset is used to place the current element and mark that location as having been filled. In the trivial case where there is only one more unfilled element, no pseudorandom number is generated.

Pseudorandom key expansion uses a simple method to provide key dependent bits as needed to place array elements. A pointer is set to the first 8-bit byte of the key. A 32 bit CRC accumulator is set to all ones (FFFFFFFF hexadecimal). This initial value is used rather than all zeros so that an all zero external key would not be weak. Every time a pseudorandom number is requested, the CRC is updated using the CCITT CRC-32 [7] using the key byte pointed to by the pointer. The pointer is then moved to the next key byte. After the pointer is moved beyond the end of the last key byte, the CRC is updated with the least significant byte of the size of the key (in bytes), then with the next to least significant byte of the size of the key (in bytes), then the pointer is moved back to the first byte of the key. If the actual key size used is not a multiple of 8 bits, then the unused bits of the last key byte are set to 1, with the used bits occupying the least significant bits of the byte.

Although no upper limit is explicitly given for key size, increasing the key size provides no significant increase in security if more than approximately $28\,672 \cdot \mathbf{n}$ bits are used, where **n** is the number of rounds used. This upper limit is large enough that even fictional computers [8] would have difficulty with a brute force attack.

To normalize the 32 bit accumulator value to the desired number range from 0 to **n**, first perform a logical “and” operation on the accumulator with the value $2^{\mathbf{m}}-1$, where **m** is the smallest integer value such that $2^{\mathbf{m}}-1 \geq \mathbf{n}$. This will select the minimum number of bits required to cover the range needed. If the resulting value is less than or equal to **n**, use it. If it is not, then

© 1994 Michael Paul Johnson

repeat the above process with a new pseudorandom number. If, after 97 attempts the value is still not in range (a very low probability condition), simply subtract **n** from the value and use it.

If the decryption mode of Diamond is to be used, calculate the inverse substitution arrays directly from the encryption substitution arrays as follows:

```
For rounds := 1 to n
   For byte position := 1 to 16
      For k := 0 to 255 do
         inverse array[array[k]] := k
```

B. Substitution

In each substitution round, each byte of the input block is replaced with the contents of the substitution array for that round, byte position, and byte value. For decryption, the same operation is performed with the inverse substitution array. In a hardware implementation, this is can be done quickly by simply addressing static RAM. Note that the substitution arrays used in the Diamond Encryption Algorithm are different from the S-Boxes used in ciphers like DES, in that (1) they are much larger, (2) there are more of them, and (3) they are not used in conjunction with a simpler operation with a key that could be solved for with differential cryptanalysis.

C. Permutation

Between each substitution round, a fixed permutation is performed. The purpose of this permutation step is to increase the effective block size of the cipher by making each output byte a function of 8 input bytes by simply selecting one bit from each of 8 input bytes. Every bit of the input block is used exactly once in the output block. In a hardware, this can be done with literal wire crossings. In software, efficiency is gained by ensuring that every bit ends up in the same position relative to a byte boundary as where it started.

The specific permutation used for encryption takes the least significant bit of each byte from the input byte in the same position. The next most significant bit is taken from the input byte indexed as one byte higher (mod 16). The next most significant bit is taken from the input byte indexed as two higher (mod 16), and so on. For decryption, the inverse of this operation is the same, except the byte positions used are one byte lower (mod 16) instead of higher.

After 2 rounds, every output byte is a function of 8 input bytes and all key bytes (if the key is less than 4080 bytes, which is likely). After 3 rounds, every output byte is a function of 15 input bytes and the key. After 4 rounds, every output byte is a function of every input byte and the key. The minimum of 6 additional rounds are intended to make cryptanalysis more difficult.

V. CRYPTANALYSIS OF DIAMOND

In the design of Diamond, several types of cryptanalytic attacks were considered. The reasons why I currently consider each of them to be computationally infeasible are listed below. If anyone finds an attack on Diamond that is better than a brute force attack on the key, please let me know. The following consists of rough order of magnitude estimations and hand waving, but they are of value anyway. To construct more exact proofs, actual construction of all of the cryptanalytic attacks that your opponent might try is required. It is conjectured that actual construction of the attacks mentioned would be much more complex than the following estimates indicate.

A. Brute Force

Exhaustive key search can be made intractable (beyond the reach of any likely enemy) by choosing a key length of around 120 bits. A loose lower bound of the cost of exhaustive key search can be placed with the following generous assumptions. Assuming a massively parallel

machine can perform a trillion decryptions per second (with different keys) on each of a billion processors, then an exhaustive key search would take an average of about 42 million years. The user may wish to use smaller key sizes in some applications to save in key management costs, while still maintaining adequate protection for the value of the specific data. Larger keys than 128 bits probably do not contribute significantly to the over-all security of a system of data protection, because of some other attacks on data security that are possible. If you do want to use a much larger key, increasing the number of rounds to greater than 10 is recommended.

Another form of brute force attack that is available with block ciphers is the precomputed dictionary attack. The idea here is to create a database of one very probable plain text block encrypted under all possible keys. Sort the resulting cipher text/key pairs by cipher text value and store it in a table. Then to attack a piece of cipher text, look up the possible key in the table and try it on the rest of the message. This may take several iterations, but would be likely to succeed if you could store so much data. The problems here are, of course (1) time to generate the table, and (2) sorting and storing the table.

Note that small key sizes do not necessarily prevent a brute force attack.

B. Analytical Attack with Chosen Plain Text

An analytical attack involves solving for the contents of at least one of the substitution arrays. If one array could be isolated by selecting carefully chosen inputs and outputs, then its contents could be solved for. Once this was done, this knowledge could be used to solve for additional substitution arrays. The problem with this attack is that because every output byte is a function of every input byte and at least 112 substitution arrays, this decomposition is difficult. I conjecture that a loose lower bound on the complexity of this kind of attack is that it would take more operations than $256 \cdot 2^x$, where x is the number of arrays in the dependency chain for each byte. For a 10 round Diamond implementation, this would be an approximate total of $256 \cdot 16(2^{112} + 2^{96} + 2^{80} + 2^{64} + 2^{48} + 2^{33} + 2^{18} + 2^{10} + 2^2) \approx 2 \cdot 10^{37}$ operations. This is about as hard as solving for a 124 bit key with a precomputed plain text attack, but even less practical.

C. Differential Cryptanalysis

Attacking Diamond with a form of differential cryptanalysis as described in [6] does not work directly. A similar approach using differences in known plain text values would have some value in reduced Diamond variants with 3 or fewer rounds, but would be no better than the analytical attack discussed above for 10 or more rounds.

D. Solving for the Key from an Array

If you could solve for one substitution array, and knew (or guessed) the length of the key, a method might be constructed to directly solve for the key from the contents of one substitution array. This reduces the strength lower bound estimate for an analytical attack on a 10 round Diamond system to about $256 \cdot 2^{112} \approx 10^{36}$, or about as strong as a 120 bit key.

E. Bypassing the Algorithm

In any security system, care must be taken to avoid the possibility that the hardware and/or software doing the encryption and decryption is not tampered with or replaced. For very high security applications, a hardware device that is implemented on a tamper resistant chip in a tamper resistant enclosure is preferable to a pure software implementation. Care must also be taken to ensure that the sensitive data is physically secure when in plain text form. Key management,

although it is beyond the scope of this paper, is also a critical concern.

VI. DIAMOND LITE

Where software speed and table space are critical, a variant of Diamond that has a block size of 8 bytes (64 bits) and a minimum of four rounds (8 recommended) is a reasonable compromise. This variant has the advantage that every bit of the output is a function of every bit of the input and every bit of the key after only two rounds. At least four rounds are needed, however, to ensure that the algorithm is strong enough to justify keys of about 64 bits. This only requires 8192 bytes of table space and offers faster speed in software than the full Diamond with a 16 bit block size.

It is conjectured that Diamond Lite with 8 rounds and a key length of 128 bits is at least equivalent in security to the IDEA™ cipher, and more secure than the aging DES algorithm.

VII. LEGAL ISSUES

The Diamond and Diamond Lite Encryption Algorithms may be used for any legal purpose without payment of royalties to the inventor or his employer, however the names "Diamond Encryption Algorithm" and "Diamond Lite Encryption Algorithm" are Trade Marks owned by the inventor, and may not be used in connection with any algorithm that does not comply with the specifications given herein and in the sample programs written by the inventor. The Diamond Encryption Algorithm is the same as the MPJ and MPJ2 Encryption Algorithms, with the exception of the key expansion algorithm. Some governments may restrict the use, publication, or export of strong encryption technology.

VIII. CONCLUSION

Diamond and Diamond Lite are two of several alternatives to the aging and now relatively insecure DES algorithm. Source code for a software implementation of Diamond in C is available from the author for a minimal shipping and handling charge or for free electronic transfer where allowed by law. Contact the author at one of the addresses below for details. Comments, questions, and reports of possible weaknesses should be sent to the author at:

Mike Johnson
PO BOX 1151
LONGMONT CO 80502-1151
USA

BBS: 303-938-9654
Internet mail: mpj@csn.org
Internet ftp: csn.org:/mpj/
CompuServe: 71331,2332

IX. REFERENCES

[1] Michael J. Wiener, "Efficient DES Key Search," Bell-Northern Research, PO Box 3511 Station C, Ottawa, Ontario, K1Y 4H7, Canada, 20 August 1993.

[2] Theodor Brüggemann and Hoger Bürk, "Der Verschlüsselungsalgorithmus IDEA™," Ascom Tech AG, Fachbereich Kryptologie, Ziegelmattstrasse 1, CH - 4503 Solothurn, Switzerland.

[3] Xuejia Lai and James L. Massey, "Markov Ciphers and Differential Cryptanalysis," in *Advances in Cryptology – EUROCRYPT '91*, Springer-Verlag, pp 17-38.

[4] Xuejia Lai "Detailed Description and a Software Implementation of the IPES Cipher," Institut für Signal- und Informationsverarbeitung, ETH Zürich.

[5] Michael Paul Johnson, "Beyond DES: Data Compression and the MPJ Encryption Algorithm," Master's Thesis at the University of Colorado at Colorado Springs, 1989. Available by anonymous ftp to csn.org

in /mpj or on the author's BBS at 303-938-9654.

[6] Eli Biham and Adi Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. New York: Springer-Verlag, 1993.

[7] *C Programmers Guide to NetBIOS*, Howard W. Sams & Co., Inc.

[8] Rick Sternbach and Michael Okuda, *Star Trek the Next Generation Technical Manual*, New York: Pocket Books, 1991.

[9] Xuejia Lai and James L. Massey, "A Proposal for a New Block Encryption Standard," in *Advances in Cryptology – EUROCRYPT '90*, Springer-Verlag, pp 389-404., 1990.