Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)

B. Schneier

Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994, pp. 191-204.

ABSTRACT:

Blowfish, a new secret-key block cipher, is proposed. It is a Feistel network, iterating a simple encryption function 16 times. The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex initialization phase required before any encryption can take place, the actual encryption of data is very efficient on large microprocessors.

The cryptographic community needs to provide the world with a new encryption standard. DES [16], the workhorse encryption algorithm for the past fifteen years, is nearing the end of its useful life. Its 56-bit key size is vulnerable to a brute-force attack [22], and recent advances in differential cryptanalysis [1] and linear cryptanalysis [10] indicate that DES is vulnerable to other attacks as well.

Many of the other unbroken algorithms in the literature--Khufu [11,12], REDOC II [2,23, 20], and IDEA [7,8,9]--are protected by patents. RC2 and RC4, approved for export with a small key size, are proprietary [18]. GOST [6], a Soviet government algorithm, is specified without the S-boxes. The U.S. government is moving towards secret algorithms, such as the Skipjack algorithm in the Clipper and Capstone chips [17].

If the world is to have a secure, unpatented, and freely- available encryption algorithm by the turn of the century, we need to develop several candidate encryption algorithms now. These algorithms can then be subjected to years of public scrutiny and cryptanalysis. Then, the hope is that one or more candidate algorithms will survive this process, and can eventually become a new standard.

This paper discusses the requirements for a standard encryption algorithm. While it may not be possible to satisfy all requirements with a single algorithm, it may be possible to satisfy them with a family of algorithms based on the same cryptographic principles.

AREAS OF APPLICATION

A standard encryption algorithm must be suitable for many different applications:

Bulk encryption. The algorithm should be efficient in encrypting data files or a continuous data stream.

Random bit generation. The algorithm should be efficient in producing single random bits.

Packet encryption. The algorithm should be efficient in encrypting packet-sized data. (An ATM packet has a 48- byte data field.) It should implementable in an application where successive packets may be encrypted or decrypted with different keys.

Hashing. The algorithm should be efficient in being converted to a one-way hash function.

PLATFORMS

A standard encryption algorithm must be implementable on a variety of different platforms, each with their own requirements. These include:

Special hardware. The algorithm should be efficiently implementable in custom VLSI hardware.

Large processors. While dedicated hardware will always be used for the fastest applications, software implementations are more common. The algorithm should be efficient on 32-bit microprocessors with 4 kbyte program and data caches.

Medium-size processors. The algorithm should run on microcontrollers and other medium-size processors, such as the 68HC11.

Small processors. It should be possible to implement the algorithm on smart cards, even inefficiently.

The requirements for small processors are the most difficult. RAM and ROM limitations are severe for this platform. Also, efficiency is more important on these small machines. Workstations double their capacity almost annually. Small embedded systems are the same year after year, and there is little capacity to spare. If there is a choice, the extra computation burden should be on large processors rather than small processors.

ADDITIONAL REQUIREMENTS

These additional requirements should, if possible, be levied on a standard encryption algorithm.

The algorithm should be simple to code. Experiences with DES [19] show that programmers will often make implementation mistakes if the algorithm is complicated. If possible, the algorithm should be robust against these mistakes.

The algorithm should have a flat keyspace, allowing any random bit string of the required length to be a possible key. There should be no weak keys.

The algorithm should facilitate easy key-management for software implementations. Software implementations of DES generally use poor key management techniques. In particular, the password that the user types in becomes the key. This means that although DES has a theoretical keyspace of 256, the actual keyspace is limited to keys constructed with the 95 characters of printable ASCII. Additionally, keys corresponding to words and near words are much more likely.

The algorithm should be easily modifiable for different levels of security, both minimum and maximum requirements.

All operations should manipulate data in byte-sized blocks. Where possible, operations should manipulate data in 32-bit blocks.

DESIGN DECISIONS

Based on the above parameters, we have made these design decisions. The algorithm should:

Manipulate data in large blocks, preferably 32 bits in size (and not in single bits, such as DES).

Have either a 64-bit or a 128-bit block size.

Have a scalable key, from 32 bits to at least 256 bits.

Use simple operations that are efficient on microprocessors: e.g., exclusive-or, addition, table lookup, modular- multiplication. It should not use variable-length shifts or bit-wise permutations, or conditional jumps.

Be implementable on an 8-bit processor with a minimum of 24 bytes of RAM (in addition to the RAM required to store the key) and 1 kilobyte of ROM.

Employ precomputable subkeys. On large-memory systems, these subkeys can be precomputed for faster operation. Not precomputing the subkeys will result in slower operation, but it should still be possible to encrypt data without any precomputations.

Consist of a variable number of iterations. For applications with a small key size, the trade-off between the complexity of a brute-force attack and a differential attack make a large number of iterations superfluous. Hence, it should be possible to reduce the number of iterations with no loss of security (beyond that of the reduced key size).

If possible, have no weak keys. If not possible, the proportion of weak keys should be small enough to make it unlikely to choose one at random. Also, any weak keys should be explicitly known so they can be weeded out during the key generation process.

Use subkeys that are a one-way hash of the key. This would allow the use of long passphrases for the key without compromising security.

Have no linear structures (e.g., the complementation property of DES) that reduce the complexity of exhaustive search [4].

Use a design that is simple to understand. This will facilitate analysis and increase the confidence in the algorithm. In practice, this means that the algorithm will be a Feistel iterated block cipher [21].

Most of these design decisions are not new. Almost all block ciphers since Lucifer [5,21] are Feistel ciphers, and all have a flat keyspace (with the possible exception of a few weak keys). FEAL [13,14,15] and Khufu [11] use a variable number of iterations. Khufu [11] has a large number of subkeys that are a one-way function of the key. RC2 [18] has a variable-length key. GOST [6] uses a 32-bit word length and a 64-bit block size. MMB [2] uses a 32-bit word length and a 128-bit block size.

BUILDING BLOCKS

There are a number of building blocks that have been demonstrated to produce strong ciphers. Many of these can be efficiently implemented on 32-bit microprocessors.

Large S-boxes. Larger S-boxes are more resistant to differential cryptanalysis. An algorithm with a 32-bit word length can use 32-bit S-boxes. Khufu and REDOC III both use a 256-entry, 32-bit wide S-box [11,20].

Key-dependent S-boxes. While fixed S-boxes must be designed to be resistant to differential and linear cryptanalysis, key-dependent S-boxes are much more resistant to these attacks. They are used in the Khufu algorithm [11]. Variable S-boxes, which could possibly be key dependent, are used in GOST [6].

Combining operations from different algebraic groups. The IDEA cipher introduced this concept, combining XOR mod 2^{16}, addition mod 2^{16}, and multiplication mod $2^{16}+1$ [7]. The MMB cipher uses a 32-bit word, and combines XOR mod 2^{32} with multiplication mod $2^{32}-1$ [2].

Key-dependent permutations. The fixed initial and final permutations of DES have been long regarded as cryptographically worthless. Khufu XORs the text block with key material at the beginning and the end of the algorithm [11].

BLOWFISH

Blowfish is a variable-length key block cipher. It does not meet all the requirements for a new cryptographic standard discussed above: it is only suitable for applications where the key does not change often, like a communications link or an automatic file encryptor. It is significantly faster than DES when implemented on 32-bit microprocessors with large data caches, such as the Pentium and the PowerPC.

DESCRIPTION OF THE ALGORITHM

Blowfish is a variable-length key, 64-bit block cipher. The algorithm consists of two parts: a key-expansion part and a data- encryption part. Key expansion converts a key of at most 448 bits into several subkey arrays totaling 4168 bytes.

Data encryption occurs via a 16-round Feistel network. Each round consists of a key-dependent permutation, and a key- and data-dependent substitution. All operations are XORs and additions on 32-bit words. The only additional operations are four indexed array data lookups per round.

Subkeys:

Blowfish uses a large number of subkeys. These keys must be precomputed before any data encryption or decryption.

1. The P-array consists of 18 32-bit subkeys:
P1, P2,..., P18.

2. There are four 32-bit S-boxes with 256 entries each:
S1,0, S1,1,..., S1,255;
S2,0, S2,1,..,, S2,255;
S3,0, S3,1,..., S3,255;
S4,0, S4,1,..,, S4,255.

The exact method used to calculate these subkeys will be described later.

Encryption:

Blowfish is a Feistel network consisting of 16 rounds (see Figure 1). The input is a 64-bit data element, x.

```
Divide x into two 32-bit halves: xL, xR
For i = 1 to 16:
xL = xL XOR Pi
xR = F(xL) XOR xR
Swap xL and xR
Next i
Swap xL and xR (Undo the last swap.)
xR = xR XOR P17
xL = xL XOR P18
Recombine xL and xR
Function F (see Figure 2):
Divide xL into four eight-bit quarters: a, b, c, and d
F(xL) = ((S1,a + S2,b mod $2^{32}$) XOR S3,c) + S4,d mod $2^{32}$
```

Decryption is exactly the same as encryption, except that P1, P2,..., P18 are used in the reverse order.

Implementations of Blowfish that require the fastest speeds should unroll the loop and ensure that all subkeys are stored in cache.

Generating the Subkeys:

The subkeys are calculated using the Blowfish algorithm. The exact method is as follows:

1. Initialize first the P-array and then the four S-boxes, in order, with a fixed string. This string consists of the hexadecimal digits of pi (less the initial 3). For example:

```
P1 = 0x243f6a88
P2 = 0x85a308d3
P3 = 0x13198a2e
P4 = 0x03707344
```

2. XOR P1 with the first 32 bits of the key, XOR P2 with the second 32-bits of the key, and so on for all bits of the key (possibly up to P14). Repeatedly cycle through the key bits until the entire P-array has been XORed with key bits. (For every short key, there is at least one equivalent longer key; for example, if A is a 64-bit key, then AA, AAA, etc., are equivalent keys.)

3. Encrypt the all-zero string with the Blowfish algorithm, using the subkeys described in steps (1) and (2).

4. Replace P1 and P2 with the output of step (3).

5. Encrypt the output of step (3) using the Blowfish algorithm with the modified subkeys.

6. Replace P3 and P4 with the output of step (5).

7. Continue the process, replacing all entries of the P- array, and then all four S-boxes in order, with the output of the continuously-changing Blowfish algorithm.

In total, 521 iterations are required to generate all required subkeys. Applications can store the subkeys rather than execute this derivation process multiple times.

MINI-BLOWFISH

The following mini versions of Blowfish are defined solely for cryptanalysis. They are not suggested for actual implementation. Blowfish-32 has a 32-bit block size and subkey arrays of 16-bit entries (each S-box has 16 entries). Blowfish-16 has a 16-bit block size and subkey arrays of 8-bit entries (each S-box has 4 entries).

DESIGN DECISIONS

The underlying philosophy behind Blowfish is that simplicity of design yields an algorithm that is both easier to understand and easier to implement. Through the use of a streamlined Feistel network--a simple S-box substitution and a simple P-box substitution--I hope that the design will not contain any flaws.

A 64-bit block size yields a 32-bit word size, and maintains block-size compatibility with existing algorithms. Blowfish is easy to scale up to a 128-bit block, and down to smaller block sizes. Cryptanalysis of the mini-Blowfish variants may be significantly easier than cryptanalysis of the full version.

The fundamental operations were chosen with speed in mind. XOR, ADD, and MOV from a cache are efficient on both Intel and Motorola architectures. All subkeys fit in the cache of a 80486, 68040, Pentium, and PowerPC.

The Feistel network that makes up the body of Blowfish is designed to be as simple as possible, while still retaining the desirable cryptographic properties of the structure. Figure 3 is round i of a general Feistel network: Rn,i are reversible functions of text and key, and Ni is a non-reversible function of text and key. For speed and simplicity, I chose XOR as my reversible function. This let me collapse the four XORs into a single XOR, since:

R--1,i+1 = R1,i+1 XOR R2,i-1 XOR R3,i XOR R4,i

This is the P-array substitution in Blowfish. The XOR can also be considered to be part of the non-reversible function, Ni, occurring at the end of the function. (Although equivalent, I chose not to illustrate them in this way because it simplifies description of the subkey-generation process.) There are two XORs that remain after this reduction: R1 in the first round and R2 in the last round. I chose not to eliminate these in order to hide the input to the first non-reversible function.

I considered a more complicated reversible function, one with modular multiplications and rotations. However, these operations would greatly increase the algorithm's execution time. Since function F is the primary source of the algorithm's security, I decided to save time-consuming complications for that function.

Function F, the non-reversible function, gives Blowfish the best possible avalanche effect for a Feistel network: every text bit on the left half of the round affects every text bit on the right half. Additionally, since every subkey bit is affected by every key bit, the function also has a perfect avalanche effect between the key and the right half of the text after every round. Hence, the algorithm exhibits a perfect avalanche effect after three rounds and again every two rounds after that.

I considered adding a reversible mixing function, more complicated than XOR, before the first and after the last round. This would further confuse the entry values into the Feistel network and ensure a complete avalanche effect after the first two rounds. I eventually discarded the addition as a time- consuming complication with no clear cryptographic benefits.

The non-reversible function is designed for strength, speed, and simplicity. Ideally, I wanted a single S-box with 2^{32} 32-bit words, but that was impractical. My eventual choice of 256-entry S-boxes was a compromise between my three design goals. The small-number of bits to large-number of bits may have weaknesses with respect to linear cryptanalysis, but these weaknesses are hidden both by combining the output of four S-boxes and making them dependent on the key.

I used four different S-boxes instead of one S-box primarily to avoid symmetries when different bytes of the input are equal, or when the 32-bit input to function F is a bytewise permutation of another 32-bit input. I could have used one S-box and made each of the four different outputs a non-trivial permutation of the single output, but the four S-box design is faster, easier to program, and seems more secure.

The function that combines the four S-box outputs is as fast as possible. A simpler function would be to XOR the four values, but mixing addition mod 2^{32} and XOR combines two different algebraic groups with no additional instructions. The alternation of addition and XOR ends with an addition operation because an XOR combines the final result with xR.

If the four indexes chose values out of the same S-box, a more complex combining function would be required to eliminate symmetries. I considered using a more complex combining function in Blowfish (using modular multiplications, rotations, etc.), but chose not to because the added complication seemed unnecessary.

The key-dependent S-boxes protect against differential and linear cryptanalysis. Since the structure of the S-boxes is completely hidden from the cryptanalyst, these attacks have a more difficult time exploiting that structure. While it would be possible to replace these variable S-boxes with four fixed S-boxes that were designed to be resistant to these attacks, key-dependent S-boxes are easier to implement and less susceptible to arguments of "hidden" properties. Additionally, these S-boxes can be created on demand, reducing the need for large data structures stored with the algorithm.

Each bit of xL is only used as the input to one S-box. In DES many bits are used as inputs to two S-boxes, which strengthens the algorithm considerably against differential attacks. I feel that this added complication is not as necessary with key- dependent S-boxes. Additionally, larger S-boxes would take up considerably more memory space.

Function F does not depend on the iteration. I considered adding this dependency, but did not feel that it had any cryptographic merit. The P-array substitution can be considered to be part of this function, and that is already iteration-dependent.

The number of rounds is set at 16 primarily out of desire to be conservative. However, this number affects the size of the P- array and therefore the subkey-generation process;

16 iterations permits key lengths up to 448 bits. I expect to be able to reduce this number, and greatly speed up the algorithm in the process, as I accumulate more cryptanalysis data.

In algorithm design, there are two basic ways to ensure that the key is long enough to ensure a particular security level. One is to carefully design the algorithm so that the entire entropy of the key is preserved, so there is no better way to cryptanalyze the algorithm other than brute force. The other is to design the algorithm with so many key bits that attacks that reduce the effective key length by several bits are irrelevant. Since Blowfish is designed for large microprocessors with large amounts of memory, I chose the latter.

The subkey generation process is designed to preserve the entire entropy of the key and to distribute that entropy uniformly throughout the subkeys. It is also designed to distribute the set of allowed subkeys randomly throughout the domain of possible subkeys. I chose the digits of pi as the initial subkey table for two reasons: because it is a random sequence not related to the algorithm, and because it could either be stored as part of the algorithm or derived when needed. There is nothing sacred about pi; any string of random bits--digits of e, RAND tables, output of a random number generator--will suffice. However, if the initial string is non-random in any way (for example, ASCII text with the high bit of every byte a 0), this non-randomness will propagate throughout the algorithm.

In the subkey generation process, the subkeys change slightly with every pair of subkeys generated. This is primarily to protect against any attacked of the subkey generation process that exploit the fixed and known subkeys. It also reduces storage requirements. The 448 limit on the key size ensures that the every bit of every subkey depends on every bit of the key. (Note that every bit of P15, P16, P17, and P18 does not affect every bit of the ciphertext, and that any S-box entry only has a .06 probability of affecting any single ciphertext block.)

The key bits are repeatedly XORed with the digits of pi in the initial P-array to prevent the following potential attack: Assume that the key bits are not repeated, but instead padded with zeros to extend it to the length of the P-array. An attacker might find two keys that differ only in the 64-bit value XORed with P1 and P2 that, using the initial known subkeys, produce the same encrypted value. If so, he can find two keys that produce all the same subkeys. This is a highly tempting attack for a malicious key generator.

To prevent this same type of attack, I fixed the initial plaintext value in the subkey-generation process. There is nothing special about the all-zeros string, but it is important that this value be fixed.

The subkey-generation algorithm does not assume that the key bits are random. Even highly correlated key bits, such as an alphanumeric ASCII string with the bit of every byte set to 0, will produce random subkeys. However, to produce subkeys with the same entropy, a longer alphanumeric key is required.

The time-consuming subkey-generation process adds considerable complexity for a brute-force attack. The subkeys are too long to be stored on a massive tape, so they would have to be generated by a brute-force cracking machine as required. A total of 522 iterations of the encryption algorithm are required to test a single key, effectively adding 29 steps to any brute-force attack.

POSSIBLE SIMPLIFICATIONS

I am exploring several possible simplifications, aimed at decreasing memory requirements and execution time. These are outlined below:

Fewer and smaller S-boxes. It may be possible to reduce the number of S-boxes from four to one. Additionally, it may be possible to overlap entries in a single S-box: entry 0 would consist of bytes 0 through 3, entry 1 would consist of bytes 1 through 4, etc. The former simplification would reduce the memory requirements for the four S-boxes from 4096 bytes to 1024 bytes, the latter would reduce the requirements for a single S-box from 1024 bytes to 259 bytes. Additional steps may be required to eliminate the symmetries that these simplifications would introduce. Additionally, four different 10- or 12-bit indexes into a single large S-box could be used instead of the current series of S-boxes.

Fewer iterations. It is probably safe to reduce the number of iterations from 16 to 8 without compromising security. The number of iterations required for security may be dependent on the length of the key. Note that with the current subkey generation procedure, an 8-iteration algorithm cannot accept a key longer than 192 bits.

On-the-fly subkey calculation. The current method of subkey calculation requires all subkeys to be calculated advance of any data encryption. In fact, it is impossible to calculate the last subkey of the last S-box without calculating every subkey that comes before. An alternate method of subkey calculation would be preferable: one where every subkey can be calculated independently of any other. High-end implementations could still precompute the subkeys for increased speed, but low-end applications could only compute the required subkeys when needed.

CONCLUSIONS

I conjecture that the most efficient way to break Blowfish is through exhaustive search of the keyspace. I encourage all cryptanalytic attacks, modifications, and improvements to the algorithm. Attacks on mini versions of Blowfish, those with a 32- or even a 16-bit block size, are also encouraged. Source code in C and test data can be provided to anyone wishing to implement the algorithm, in accordance with U.S. export laws.

The software magazine *Dr. Dobb's Journal* is sponsoring $1000 contest for the best cryptanalysis of Blowfish received before April 1995. Please contact me for details.

Blowfish is unpatented, and will remain so in all countries. The algorithm is hereby placed in the public domain, and can be freely used by anyone.

ACKNOWLEDGEMENTS

Much of the motivation for this algorithm, as well as the design criteria, was developed with Niels Fergusen. I would also like to thank Eli Biham, Agnes Chan, Peter Gutmann, Angel Johnston, Lars Kundsen, and Matt Robshaw for their helpful suggestions.

REFERENCES

1. E. Biham and A. Shamir, Differential Cryptanalysis of the Data Encryption Standard, Springer-Verlag, 1993.

2. T.W. Cusick and M.C. Wood, "The REDOC-II Cryptosystem," Advances in Cryptology--CRYPTO '90 Proceedings, Springer- Verlag, 1991, pp. 545-563.

3. J. Deamen, R. Govaerts, and J. Vandewalle, "Block Ciphers Based on Modular Arithmetic," Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography, Rome, Italy, 15-16 Feb 1993, pp. 80-89.

4. J.-H. Evertse, "Linear Structures in Blockciphers," Advances in Cryptology--EUROCRPYT '87, Springer-Verlag, 1988, pp. 249- 266.

5. H. Feistel, "Cryptography and Computer Privacy," Scientific American, v. 228, n. 5, May 73, pp. 15-23.

6. GOST 28147-89, "Cryptographic Protection for Data Processing Systems," "Cryptographic Transformation Algorithm," Government Standard of the U.S.S.R., Inv. No. 3583, UDC 681.325.6:006.354. (in Russian)

7. X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," Advances in Cryptology--EUROCRYPT '91 Proceedings, Springer-Verlag, 1991, pp. 17-38.

8. J.L. Massey and X. Lai, "Device for Converting a Digital Block and the Use Thereof," International Patent PCT/CH91/00117, 16 May 1991.

9. J.L. Massey and X. Lai, "Device for the Conversion of a Digital Block and Use of Same," U.S. Patent 5,214,703, 25 May 1993.

10. M. Matsui, "Linear Cryptanalysis Method for DES Cipher," Advances in Cryptology--CRYPTO '93 Proceedings, Springer- Verlag, 1994, in preparation.

11. R.C. Merkle, "Fast Software Encryption Functions," Advances in Cryptology--CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 476-501.

12. R.C. Merkle, "Method and Apparatus for Data Encryption," U.S. Patent 5,003,597, 26 Mar 1991.

13. S. Miyaguchi, "The FEAL-8 Cryptosystem and Call for Attack," Advances in Cryptology--CRYPTO '89 Proceedings, Springer- Verlag, 1990, pp. 624-627.

14. S. Miyaguchi, "Expansion of the FEAL Cipher," NTT Review, v. 2, n. 6, Nov 1990.

15. S. Miyaguchi, "The FEAL Cipher Family," Advances in Cryptology--CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 627-638.

16. National Bureau of Standards, Data Encryption Standard, U.S. Department of Commerce, FIPS Publication 46, Jan 1977.

17. National Institute of Standards and Technology, "Clipper Chip Technology," 30 Apr 1993.

18. RSA Laboratories, Answers to Frequently Asked Questions About Today's Cryptography, Revision 2.0, RSA Data Security Inc., 5 Oct 1993.

19. B. Schneier, "Data Guardians," MacWorld, Feb 1993, 145-151.

20. B. Schneier, Applied Cryptography, John Wiley & Sons, New York, 1994.

21. J.L Smith, The Design of Lucifer, A Cryptographic Device for Data Communication, RC 3326, White Plains: IBM Research.

22. M.J. Weiner, "Efficient DES Key Search," Advances in Cryptology--CRYPTO '93 Proceedings, Springer-Verlag, in preparation.

23. M.C. Wood, "Method of Cryptographically Transforming Electronic Digital Data from One Form to Another," U.S. Patent 5,003,596, 26 Mar 1991.