# FastFlex *(efflex)*
## (A New Fast and Flexible Cryptographic Function)

**Ashish Sharma,**
Army Institute of Technology, Pune, India 411015
**Email:** ashish.sharma.ait@gmail.com
**Web: http://fastflex.sourceforge.net**

**Abstract** - *FastFlex (pronounced efflex) is a New Cryptographic Function which can be used to construct Stream Ciphers, Hash Functions, Message Authentication Codes and Block Ciphers which perform considerably well in both hardware and software environments on Processors of varying word lengths. FastFlex uses a modified version of the Salsa20 [1] Hash Function as its core, along with a 256:256 bit mapping generated using a 256*256 virtual s-box. FastFlex only uses 4 primitive operations, Word Additions, Word Rotations, Word Multiplications and Word XORs, which gives it speed and flexibility. Furthermore Word Multiplications are used only in moderation where the advantages of using them outweigh the slight speed penalty induced by using multiplications over additions and xor. Finally, the design of FastFlex allows significant tradeoffs in Speed, Size and Security in both Hardware and Software.*

**Keywords:** Stream Cipher, Deterministic Random Number Generator, Hash Function, Keyed Hash, 192-bit key security.

## 1 Introduction

Presented in this paper is FastFlex, a New Fast and Flexible Cryptographic Function. FastFlex uses a Weak Hash Function combined with an irreversible 256:256 bit mapping to produce a strong cryptographic function which can then be used for designing other Stream Ciphers, Hash Functions and Block Ciphers. The advantage of using such an approach is that if the security properties of the underlying cryptographic function are well established, it is much easier to establish the security of the constructions using the function. Furthermore, it allows a large part of the development effort to be shared and reused between various different constructions serving different purposes and thus reduces the overall development and implementation cost of the final construction.

## 2 FastFlex Design Goals

The primary design goals of FastFlex were as follows:

- The cryptographic function must be considerably faster than existing functions offering the same level of security
- It must be possible to implement the cryptographic function on different platforms without any significant performance penalties
- It must be possible to implement the cryptographic function on low end smart cards and other constrained environments
- The cryptographic function must accept a key size of 192 bits.
- The cryptographic function must be suitable for use in Stream Ciphers, Hash Functions and Block Ciphers
- The cryptographic function must have small key setup times
- The cryptographic function should not use any operations which make it inefficient on 32-bit microprocessors
- It should be possible to implement the cryptographic function on 8-bit microcontrollers with only 128 bytes of RAM
- The cryptographic function should have variants with different number of rounds to offer tradeoffs between security and performance

Now that the primary design goals have been specified, the structure of the cryptographic function can be described.

## 3 FastFlex Building Blocks

The FastFlex Cryptographic Function is made up of two subroutines:

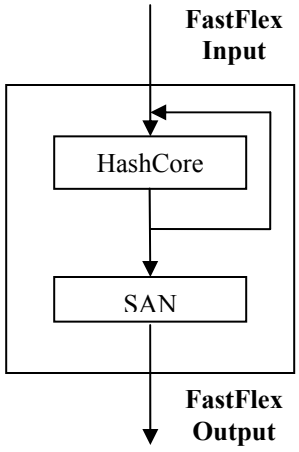



Fig. 1. Structure of FastFlex

The HashCore which iterates a specified number of times during FastFlex depending on the security requirements, and a Substitution Addition Network (SAN) which is a highly nonlinear 256:256 bit mapping executing only once during FastFlex, irrespective of the security requirements.
The input to FastFlex is the input to the HashCore. The output of the HashCore is the input to the SAN. The output of the SAN is the output of FastFlex.

## 3.1 HashCore

The HashCore is central to the design of FastFlex. The requirements for the HashCore are as follows:

- **The HashCore should be very fast** and should only use operations easily implementable on all target Hardware and Software platforms. To this end only Word Additions, Word XOR, Word Rotations and Word Multiplications are used in the HashCore. Furthermore Word Multiplications are used only in moderation where Word Additions and Word XOR operations would either require more time for the same level of diffusion or would not provide the desired diffusion at all.
- The HashCore should not have a very large state. Nor should it have large memory requirements for storage of temporary data.
- The HashCore should have high collision resistance.
- The HashCore need not be cryptographically secure or one-way as the irreversible 256:256 bit mapping SAN prevents direct attacks on it.

Now that the requirements of the HashCore have been specified, it can be defined.

The HashCore has a state size of 8 Words arranged as a 2*4 array.
Let the Initial state of the HashCore be

$$\begin{Bmatrix} W_0 & W_1 & W_2 & W_3 \\ W_4 & W_5 & W_6 & W_7 \end{Bmatrix}$$

The transformation which works on words in the same row can then be defined as

$$\begin{Bmatrix} Y_1 = W_1 \oplus ((W_0 + W_3) <<< 7), \\ Y_2 = W_2 \oplus ((Y_1 + W_0) <<< 13), \\ Y_3 = W_3 \oplus ((Y_2 + Y_1) <<< 17), \\ Y_0 = W_0 \oplus ((Y_3 + Y_2) <<< 23) \end{Bmatrix}$$

Where <<< denotes Left Rotation
And $\oplus$ denotes bitwise XOR

which can be written as
$\{Y_0,Y_1,Y_2,Y_3\} = \mathbf{Row}(W_0,W_1,W_2,W_3)$

The transformation working on the second row is
$\{Y_6,Y_7,Y_4,Y_5\} = \mathbf{Row}(W_6,W_7,W_4,W_5)$

The transformation which works on the words of the same column is defined as

$$\begin{Bmatrix} Y_I = (Y_I \otimes Y_{I+4})_L \\ Y_{I+4} = (Y_I \otimes Y_{I+4})_H \oplus 8b9a7465_{16} \end{Bmatrix}$$

which can be written as
$\{Y_I,Y_{I+4}\} = \mathbf{Col}(Y_I,Y_{I+4})$

where $\otimes$ denotes 32-bit Word Multiplication,
H and L denote the higher and lower words of the product.
The result of the row transformation, followed by the column transformation is the final state of one round of HashCore. The final state is thus given by

$$\begin{Bmatrix} Y_0 & Y_1 & Y_2 & Y_3 \\ Y_4 & Y_5 & Y_6 & Y_7 \end{Bmatrix}$$

The output of the present round becomes the input to the next round till there are no more rounds of HashCore remaining.
**Unless otherwise stated, the HashCore iterates 8 times during one round of FastFlex.**

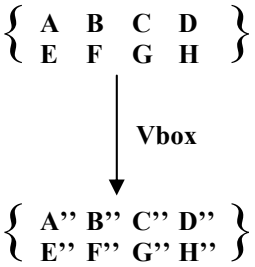
## 3.2 Substitution - Addition Network (SAN)

The Substitution-Addition Network (SAN) is a one way substitution which serves to hide the internal state of the FastFlex function from an attacker. It must be noted that even though the HashCore is quite difficult to reverse, it is the SAN which guarantees the claimed security properties of FastFlex.
The SAN is constructed using a 8*32 bit S-Box, which is in turn derived from the Skipjack F-table. Designed by the Information Security Research Centre (ISRC) at the Queensland University of Technology, Australia, the 8*32 S-box (hereafter referred to as **Qbox**) is the same as is used in the Stream Cipher Proposal NLS (Non-Linear Sober) **[2]**, submitted to the '**estream –Stream Cipher Project**' **[3]**. The specification for the S-box can be found in **[2]**.

The SAN function uses a virtual S-Box constructed using the s-box. The Virtual S-box, Vbox, provides even higher resistance against differential and linear cryptanalysis.
It is made up of fundamental operations which do not commute and should thus provide resistance to a wide spectrum of attacks.

Let us define the Vbox.

- The Vbox is a 256:256, one way bit mapping
- The Vbox uses only fundamental operations which do not commute
- The Vbox is designed for maximum parallelization while maintaining a high level of security




The Vbox consists of 4 layers of operations and is essentially a one way mapping.

Layer 1:
A' = [A + S(B) + F]
B' = [B + S(C) + G]
C' = [C + S(D) + H]
D' = [D + S(A) + E]

Layer 2:
E' = [E + S(A')]
F' = [F + S(B')]
G' = [G + S(C')]
H' = [H + S(D')]

Layer 3:
A'' = [A' ⊕ S(G')]
B'' = [B' ⊕ S(H')]
C'' = [C' ⊕ S(E')]
D'' = [D' ⊕ S(F')]

Layer 4:
E'' = [E' + A''] ⊕ F'
F'' = [F' + B''] ⊕ G'
G'' = [G' + C''] ⊕ H'
H'' = [H' +D''] ⊕ E'

Where S( ) is defined as
$S(X) = [Qbox(X_H) \oplus X]$
and $X_H$ is the most significant byte of X.

Unless otherwise stated, **SAN executes only once during FastFlex.**

Now that both the HashCore and the SAN have been specified, let us summarize FastFlex:

- FastFlex takes 8 words as input and modifies them in place to produce 8 words as output.
- A round of FastFlex translates to 8 rounds of HashCore followed by 1 round of SAN.
- The HashCore takes 8 words as input and produces 8 words as output.
- The SAN acts as a virtual S-Box It takes 8 words as inputs and produces 8 words as output.
- We denote the FastFlex function as
  {A'',B'',C'',D'',E'',F'',G'',H''} =
  $FastFlex(W_0,W_2,W_5,W_7,W_1,W_3,W_4,W_6)$

# 4 FastFlex Design Justifications

- **The Use of Primitive Operations:** Only primitive operations such as Word Addition, Word XOR, Word Rotations and Word Multiplications are used in FastFlex. A long chain of simple operations can approximate any complex operation **[4]**. On the other hand, implementing complex functions in constrained environments is difficult. Most constrained environments, more so than not, already have architectures for implementing simple operations such as Addition, Xoring, Rotation and Multiplication The chain of simple operations as used in FastFlex can thus reach the same desired security level as of any other more complex operation, and at the same time be implementable in constrained environments.
- **The use of Word Multiplications:** Integer multiplications, it can be argued are slow and difficult to implement in Hardware, although on most modern general purpose processors, they are quite fast. Some new hardware chips such as the Xilinx Spartan 3 **[5]** and Altera Cyclone have efficient multiplier architectures pre-fabricated on chip, and this is a trend which seems to be continuing. Thus it is a reasonable assumption that most future hardware chips will also boast such efficient multiplication architectures and thus the performance penalty mentioned above will soon disappear. At present, the advantage of using word multiplications for diffusion on general purpose processors and the fact that future hardware chips will implement word multiplications efficiently leaves no reason not to use them.
- **The use of Word Rotations**: It may be argued that some microprocessors do not have efficient Word Rotations or do not have Word Rotations at all. To overcome this, rotations may be achieved using bit-shifts on such processors.
  X<<<Y = [(X<<Y) | (X>>32-Y)]
- **Rotation Distances:** The exact rotation distances do not matter much. Rotation by any different prime amounts should suffice.
- **The use of S-boxes:** An S-box provides thorough diffusion and confusion in a single operation. S-boxes however take up considerable memory which may increase the cache pressure on some CPUs. Furthermore, constrained environments may not have enough space to implement large S-boxes. A balance is thus needed between the number of S-boxes, the size of S-boxes and the frequency of use of S-boxes in code. The frequency of use of S-boxes is a fact often neglected in many implementations. S-boxes are implemented as array lookups and using an S-box is equivalent to fetching an element from memory. If S-boxes are not frequently used, and they are frequently overwritten in cache, then there can be significant performance degradation as they will have to be fetched from memory again. FastFlex overcomes this

problem by using only a single 8*32 S-box which requires only 1KB of storage, and using the S-box thoroughly when it is brought into cache.

- **Size of the Internal State of FastFlex:** FastFlex has an internal state of 256 bits. It must be noted that a state size of 256 bits is more than overwhelming considering the amount of processing power available today and the present state of cryptanalysis **[6]**. Only a major breakthrough in electronics technology or cryptanalytic techniques, if not both, will change this fact. It seems more reasonable to use better design principles than to increase the state size as of now.
- **The Number of Rounds:** 8 rounds of HashCore and 1 round of SAN seems to be quite conservative and as peer confidence in FastFlex increases, the number of rounds can be decreased. Meanwhile, decreasing the number of rounds is not advisable.
- **The SAN:** SAN hides the internal state of the HashCore. This makes direct guess and determine attacks very difficult against FastFlex.
- **Why not more S-boxes:** A single S-box cannot provide the same security as provided by two or more S-boxes. However, using additional S-boxes would have nearly doubled the memory requirements of the FastFlex object code. Furthermore, S-box lookups are vulnerable to timing attacks on most platforms and there is no efficient workaround to this **[7].** S-box lookups must thus be kept to a minimum whenever possible.

# 5 FastFlex Applications

**In this section are presented a few constructions using the FastFlex Cryptographic Function.**

The following are constructed using the FastFlex cryptographic function:

- **FastStream192:** The construction of a synchronous stream cipher providing 192 bit key security.
- **FastHash256:** A hash function taking input in 256 bit blocks and producing a 256 bit message digest.

## 5.1 FastStream192

**FastStream192** is a synchronous keystream generator. In a synchronous keystream generator, the sender and receiver generate the same keystream using identical copies of the generator and using the same key. The requirements of a good keystream generator are as follows:

- The generated keystream should be unique for every different key-nonce pair.
- The generated keystream should pass all known randomness tests and should thus be indistinguishable from a random message source.
- It should not be practical to recover the secret key by observing the keystream corresponding to that key.

It must be noted that in theory it is not possible for any generator to output a truly random message stream using any fixed series of operations, no matter how complex each operation is taken to be, if the input to the generator is not random. We suffice with what is sufficient randomness, or pseudo randomness, as required by the application.
Let us assume such a keystream generator G generating keystream $G_T$.

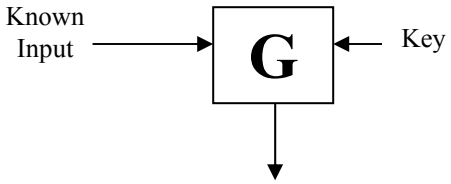



Fig. 2. Pseudorandom Keystream Generator

Encryption then takes place at the transmitter by XORing the message to be encrypted with the corresponding Keystream Word $G_T$.
$\mathbf{C_T = M_T \oplus G_T}$
where $C_T$ is the transmitted ciphertext corresponding to the Message $M_T$
Decryption is done by XORing the ciphertext $C_T$ with a copy of the corresponding keystream $G'_T$, generated locally by the receiver using the same key.

$\mathbf{M'_T = G'_T \oplus C_T}$
If $G'_T$ and $G_T$ are identical, then it can be seen that $M'_T$ and $M_T$ are also identical.

Now that synchronous stream ciphers have been introduced, FastStream192 can be described.

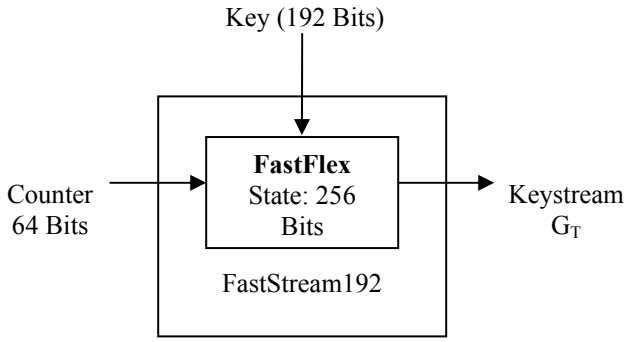



Fig. 3. FastStream192 Keystream Generator

**At the Transmitter (Encryption):**
$\mathbf{C_T = G_T \oplus M_T}$

**At the Receiver (Decryption):**
$\mathbf{M_T = G_T \oplus C_T}$

Synchronized copies of the keystream generator run at both the transmitter and the receiver.

### 5.1.A Keystream Generation in FastStream192

Keystream generation in FastStream192 is done by one round of FastFlex. The keystream generator takes two inputs, a 192 bit secret key K $\{K_0, K_1, K_2, K_3, K_4, K_5\}$ and a 64 bit nonce N $\{N_0, N_1\}$, where $K_0$, $N_0$ are the least significant words of K and N respectively.

The process of keystream generation is as follows:

- The inner state of the FastFlex is loaded with K and N as shown below

$$\begin{Bmatrix} N_0 & K_0 & K_1 & K_2 \\ K_3 & K_4 & K_5 & N_1 \end{Bmatrix}$$

- FastFlex is allowed to run four times with the output of the present round becoming the inner state of FastFlex for the next round.

$$\begin{Bmatrix} N_0 & K_0 & K_1 & K_2 \\ K_3 & K_4 & K_5 & N_1 \end{Bmatrix}$$

$\downarrow$ 4 Rounds of FastFlex

$$\begin{Bmatrix} A & B & C & D \\ E & F & G & H \end{Bmatrix}$$

- {A,B,C,D,E,F,G,H} becomes the internal state of FastStream192. {H,A} form a 64 bit counter, with a {H} as the higher word and {A} as the lower word of the counter.
- The keystream generator is now ready to produce keystream.
  A single round of FastFlex produces 8 words of keystream.

$$\begin{Bmatrix} A & B & C & D \\ E & F & G & H \end{Bmatrix}$$

$\downarrow$ 1 Round of FastFlex

$$\begin{Bmatrix} G_0 & G_1 & G_2 & G_3 \\ G_4 & G_5 & G_6 & G_7 \end{Bmatrix}$$

or in other words
{G} = FastFlex(ABCDEFGH)

- The counter is updated for generating the next 8 words of keystream by incrementing HA by 1. The value of the counter to be loaded for the next round of encryption thus becomes HA + 1.
- On generating every 8 words of keystream, the counter HA is again incremented by 1 before generating the next words of the keystream.

### 5.1.B FastStream192 Design Justifications

- **Key Size of 192 bits:** Although it is possible to implement a larger key size by increasing the state size of FastFlex (FastFlex scales well to increase in size of its state), a 192 bit key seems to be quite secure for quite some time to come. It is still impossible to brute force even a 96 bit key being used in a secure implementation. What is required is more resistance to cryptanalytic tools.
- **Stream Independent of Plaintext and Ciphertext:** The success of many cryptanalytic attacks depends on the ability of the cryptanalyst to induce/toggle some bits in the internal state of the cipher. A synchronous cipher using a counter is thus more secure against such attacks as the cryptanalyst cannot induce bits into its internal state. Furthermore, making the stream dependent on ciphertext forces serialization of the cipher. A serial cipher cannot enjoy the benefits of parallelization.
- **Synchronous Cipher:** For a synchronous cipher to decrypt data correctly, perfect synchronization is required between the transmitter and the receiver. Whenever there is any insertion in the stream, synchronization is lost. Thus synchronous ciphers are immune to Replay Attacks and most Man in the Middle Attacks **[9]**. Any errors in transmission when using synchronous stream ciphers are restricted to individual bits or words and are not propagated. This is an added advantage in some scenarios where bit errors prevail.

### 5.1.C Period of Keystream for FastStream192

FastStream192(K,N) denotes the FastStream192 stream cipher function acting on the 24 Byte key K and the 8 Byte counter sequence N, N + 1,…N + $(2^{64} - 1)$.
FastStream192(192,64) can thus have a sequence length of up to $(2^{64+8} = 2^{72})$ bits, as the sequence N, N +1,… N + $(2^{64} - 1)$ has a length of $2^{64}$ and each round generates 8 words of keystream. Thus it is possible to generate sequences of period $2^{72}$.
**Following a conservative design approach, the keystream for any given Key-Nonce Pair is limited to $2^{64}$ bits**, which is same as the keystream period for AES for a given 128 bit key in counter mode.

## 5.2 FastHash256

FastHash256 is an Unkeyed Hash Function. An unkeyed hash function maps a message M of arbitrary but finite length, into a message digest of fixed length N. M >> N in most cases, a one to one mapping is not possible and two or more inputs are bound to produce the same hash N. When this happens a collision has occurred. That is the hash of two or more different messages has collided.

It is theoretically impossible to avoid collisions. Nonetheless a few properties are expected of practical hash functions **[10]**:

- Given any M, it must be easy to compute N = Hash(M).
- **Preimage Resistance:** Given N, it must be infeasible to find any message M such that Hash(M) = N.
- **2nd Preimage Resistance:** Given a Message M, it must be infeasible to find a Message M' such that Hash(M) = Hash(M').
- **Collision Resistance:** It must be infeasible to find two distinct messages M and M' such that Hash(M) = Hash(M').
- **Non-Correlation:** The correlation between the input and output bits of the hash function should be very low.
- **Avalanche:** A hash function should have good avalanche properties such that every bit of input affects evry bit of output.
- **Partial Preimage Resistance:** It should be difficult to reconstruct any part of the message from the Hash even if a part of the input message is known.

Now that the desirable properties of Unkeyed Hash Functions have been listed, the specification for FastHash256 can be given.

FastHash256 takes input messages in blocks of 256 bits in the form of 8 words and produces as output a Hash of 256 bits in the form of 8 words. Messages are appended with a right justified binary representation of the length of the original message in bits prior to hashing (Merkle-Damgârd Strengthening **[11]**).

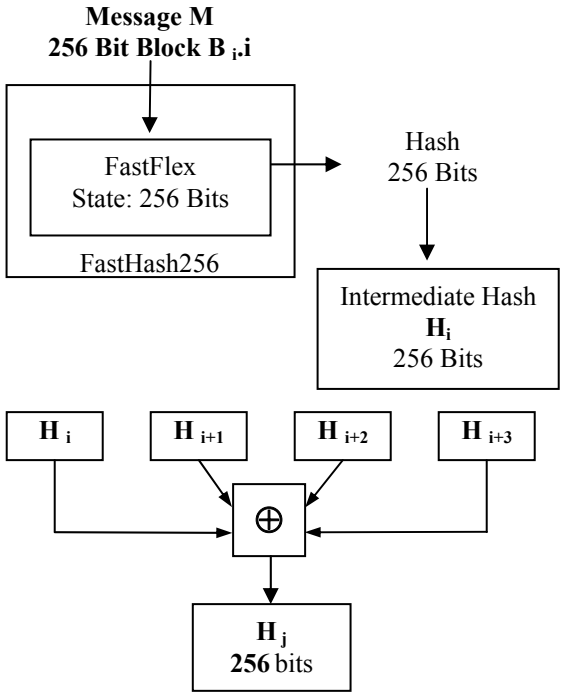



Fig. 4. FastHash256

FastHash256 uses a single iteration of FastFlex. A 4-way 256-bit xor tree is used. The intermediate hashes $H_i$.i, $H_{i+1}$.i+1, $H_{i+2}$.i+2 and $H_{i+3}$.i+3 are xored to from the another intermediate hash $H_j$. $H_j$.j, $H_{j+1}$.j+1, $H_{j+2}$.j+2, and $H_{j+3}$.j+3 are then hashed again and xored to form another intermediate hash $H_k$. The final hash is then hash of $H_k$.k ⊕ $H_{k+1}$.k+1 ⊕ $H_{k+2}$.k+2 ... to obtain a final 256-bit hash.

**Hash = FastHash256(M')**
where M' is the Message M after MD-strengthening.

# 6 Comparisons With AES

AES **[15]** has been the mainstay encryption algorithm for about 3 years now. It has been thoroughly analyzed and even more thoroughly optimized for high performance on a wide variety of platforms, including the AMD Athlon and a Pentium IV. The fastest AES Stream Cipher Implementation in software is reported to take 18 cycles/byte **[16].** It however refers to unpublished software which provides no protection against timing attacks. An implementation providing protection against timing leaks should take considerably longer. Our in-house tests suggest encryption at around 24 cycles/byte on an AMD Athlon 2400+ XP Processor Model 8 and at around 23 cycles/byte on an Intel Pentium IV 2.4 GHz. However, it must be noted that the said AES implementation provides only 128-bit key security whereas FastStream192 provides 192-bit key security.

FastStream192 takes 10 cycles/byte for the optimized assembly implementation and 12 cycles/byte for the reference C implementation when encrypting a 32 byte block with a 24 byte key on an AMD Athlon 2400+ XP Processor Model 8 **[17]**. No key expansion is necessary as in AES. No lookup tables need to be pre-calculated and only a single, fixed 1KB s-box is used.

On the Pentium IV 2.4 GHz **[18]**, at about 15 cycles/byte for the optimized assembly implementation and 16.8 cycles/byte for the Reference C implementation, FastStream192 takes considerably longer than on an AMD Athlon. This is due to the fact that the Pentium IV is somewhat sluggish when it comes to integer multiplication.

FastHash256 performs considerably faster than AES Hash on all platforms, at around 11.4 cycles/byte for our reference C implementation on an AMD Athlon 2400 + XP Processor Model 8 and at around 16.2 cycles/byte on an Intel Pentium IV. AES Hash takes 30 cycles/byte and 27 cycles/byte respectively on the above platforms [19].

The performance results for constructions using FastFlex for the reference C implementation can be summed up as follows. FastStream192 and FastHash256 also have a considerably smaller memory footprint than AES Stream and AES Hash with 1.4 KB as compared to over 4 KB for the reference AES Implementation.

| Function | Block Size | Key Size | Cycles/Byte A32, A64, PIV |
|---|---|---|---|
| AESHash | 128 b | | 30, -, 27 |
| AES Stream | 128 b | 128 b | 24, -, 23 |
| FastHash256 | 256 b | | 11.4, 10.5, 16.2 |
| FastStream 192 | 256 b | 192 b | 12, 11.1, 16.8 |

Table 1. Comparison with AES

QuickStream192, a reduced round variant was also developed for applications requiring very high speed encryption. We can formally define FastFlex as **(HashCore)$^8$.(SAN)**. That is 8 iterations of the HashCore followed by a single iteration of the SAN. QuickFlex is then **(HashCore)$^4$.(SAN)**, that is 4 iterations of the HashCore followed by a single iteration of the SAN. It should be noted that as of now both FastFlex and QuickFlex are secure by some margin. Our efforts at cryptanalysis have not been able to break more than 2 iterations of HashCore.

The performance results for QuickStream192 using QuickFlex for the reference C implementation are quite impressive at around 7.3 cycles/byte on an AMD Athlon 2400+ XP Processor Model 8 and at around 10.8 cycles/byte on an Intel Pentium IV 2.4 GHz. In other words, the reference C implementation of QuickStream192 can encrypt a stream at about 3.18 Gbps on an Athlon.

# 7 Cryptanalysis of FastFlex

In this section relevant cryptanalytic techniques are discussed with their possible application to FastFlex (if any).

## 7.1 Linear Cryptanalysis

Linear cryptanalysis **[21]** is a known plaintext attack that uses a linear relation or relations between inputs and outputs of the encryption algorithm that hold with a certain probability. This approximation can be used to assign probabilities to the possible keys and locate the most probable one. There however do not seem exist any simple linear relations between the inputs and the outputs of FastFlex.

## 7.2 Differential Cryptanalysis

Differential cryptanalysis **[22]** involves finding some pairs of plaintext {W,W'} with a small difference such that they produce a small difference in the corresponding output ciphertext {Y,Y'}. FastFlex as used in FastStream192 accepts a 2 word input (Only two words of the counter change) and produces 8 words as output. It seems unreasonable to find any input pair {W,W'} such that the corresponding output {Y,Y'} also has a small difference. This is because while there are only ($2^{64}$) choices for the input, there are ($2^{256}$) choices for the output differences. The probability of finding a low difference in both input and corresponding output is equivalent to finding a low difference input pair in $2^{64}$ which has a corresponding low difference in $2^{256}$. Now if we assume that FastHash is a one to one mapping (or a near one to one mapping with a small number of collissions), the $2^{64}$ inputs will map to only the corresponding $2^{64}$ outputs spread across $2^{256}$ possible outputs of FastFlex. The possibility of a low difference being carried over from an input pair to the corresponding output pair is thus very low.

## 7.3 Slide Attacks

Slide Attacks **[23]** compare the present state of the cipher to the next state. Fortunately, the two states of the 2*4 matrix (denoted M) in the HashCore do not have the property that

$$\mathbf{Col(Row(M)) = Col(Col(M^T)) = Row(Row(M^T))}$$

Slide attacks are thus not possible as there is no simple relationship between the present and the next state of FastFlex.

# 8 Conclusions

A New Fast and Flexible Cryptographic Function, FastFlex has been presented for peer review in this paper.

It has been used to construct a stream cipher and a hash function.

The stream cipher FastStream192 constructed using FastFlex is significantly faster than AES in counter mode by about 12 cycles/byte on an AMD Athlon and 2 cyles/byte on an Intel Pentium IV. The performance degradation on the Pentium IV may be attributed to the fact Integer Multiplication is considerably slower on Pentium IV as compared to AMD Athlon **[20]**.

The keystream generated by FastStream192 has been analysed thoroughly for any known statistical weaknesses using the NIST Statistical Test Suite (NIST PUB 800-022) and no weaknesses have been found in the keystream. Furthermore, the generated keystream has on average better if not the same statistical randomness as the output of a DRBG using SHA512 (DRBG-SHA512), provided in the NIST test suite itself.

**It is hereby declared that the author has not deliberately inserted any trapdoors in FastFlex, FastStream192 and FastHash256.**

# 9 References


**[1]** Dr. Daniel J Bernstein
The Salsa 20 Specification,
cr.yp.to, ecrypt.eu.org/stream

**[2]** Hawkes, Paddon, Rose, de Vries
Primitive Specification for NLS
ecrypt.eu.org/stream

**[3]** E-Stream Project,
Call for Stream Cipher Proposals,
ecrypt.eu.org/stream

**[4]** D.J. Wheeler, R.J. Needham
The Tiny Encryption Algorithm,
FSE '94

**[5]** The Xilinx Spartan III E
Tech Report,
xilinx.com

**[6]** UCL Crypto Group
Cryptanalysis of Block Ciphers : A Survey
dice.ucl.ac.be/crypto/

**[7]** Dr. Daniel J Bernstein
Cache Timing Attacks on AES,
cr.yp.to

**[8]** Kahn on Codes
Macmillan Publishing Co, 1967

**[9]** Bruce Schneier
Applied Cryptography,
John Wiley & Sons, 1996

**[10]** R.C. Merkle
A Fast Software One-Way Hash Function,
Crypto '90

**[11]** R.C. Merkle
A Digital Signature Based on a Conventional Encryption Function,
Crypto '87

**[12]** MD5 Message Digest
RFC 1321

**[13]** B.S. Kaliski
A Survey of Encryption Standards,
IEEE Micro vol. 13, Dec. 1993

**[14]** NIST
Skipjack and KEA Algorithm Specifications,
csrc.nist.gov/encryption/skipjack-kea.htm

**[15]** NIST
Announcing Development of a Federal Information Standard for Advanced Encryption Standard,
Federal Register, v. 62, n. 1, 2 Jan 1997

**[16]** AES Speed
E-Stream Performance Benchmarks
www.ecrypt.eu.org/stream/perf/#results

**[17]** AMD Processors x86 Code Optimization Guide
Vol. 1, Vol. 2 & Vol. 3,
amd.com

**[18]** Intel Architecture, Software Development Manual
Vol. 2,
intel.com

**[19]** E-Stream Performance Benchmarks
www.ecrypt.eu.org/stream/perf/#results

**[20]** The New AMD A64 Processor
cpuid.com

**[21]** Linear Cryptanalysis Method for DES Cipher
M. Matsui,
Eurocrypt '93

**[22]** Differential Cryptanalysis of DES-Like Cryptosystems
E. Biham, A. Shamir,
Crypto '90

**[23]** Alex Biryukov, David Wagner
Slide Attacks,
FSE '99

**[24] NIST**
PUB800-22, NIST Statistical Test Suite for Testing of Random & Pseudo Random Number Generators

**[25]** NIST
Federal Information Standard, Advanced Encryption Standard,
nist.gov

**[26, 27, 28]** [FIPS140-1, FIPS 180-1, FIPS 186] Federal Information Processing Standards

# 10 Appendices

## 10.1 The Qbox

The Qbox[ ] used in the SAN (Section 3.2) is as follows:

```
0xa3aa1887, 0xd65e435c, 0x0b65c042, 0x800e6ef4,
0xfc57ee20, 0x4d84fed3, 0xf066c502, 0xf354e8ae,
0xbb2ee9d9, 0x281f38d4, 0x1f829b5d, 0x735cdf3c,
0x95864249, 0xbc2e3963, 0xa1f4429f, 0xf6432c35,
0xf7f40325, 0x3cc0dd70, 0x5f973ded, 0x9902dc5e,
0xda175b42, 0x590012bf, 0xdc94d78c, 0x39aab26b,
0x4ac11b9a, 0x8c168146, 0xc3ea8ec5, 0x058ac28f,
0x52ed5c0f, 0x25b4101c, 0x5a2db082, 0x370929e1,
0x2a1843de, 0xfe8299fc, 0x202fbc4b, 0x833915dd,
0x33a803fa, 0xd446b2de, 0x46233342, 0x4fcee7c3,
0x3ad607ef, 0x9e97ebab, 0x507f859b, 0xe81f2e2f,
0xc55b71da, 0xd7e2269a, 0x1339c3d1, 0x7ca56b36,
0xa6c9def2, 0xb5c9fc5f, 0x5927b3a3, 0x89a56ddf,
0xc625b510, 0x560f85a7, 0xace82e71, 0x2ecb8816,
0x44951e2a, 0x97f5f6af, 0xdfcbc2b3, 0xce4ff55d,
0xcb6b6214, 0x2b0b83e3, 0x549ea6f5, 0x9de041af,
0x792f1f17, 0xf73b99ee, 0x39a65ec0, 0x4c7016c6,
0x857709a4, 0xd6326e01, 0xc7b280d9, 0x5cfb1418,
0xa6aff227, 0xfd548203, 0x506b9d96, 0xa117a8c0,
0x9cd5bf6e, 0xdcee7888, 0x61fcfe64, 0xf7a193cd,
0x050d0184, 0xe8ae4930, 0x88014f36, 0xd6a87088,
0x6bad6c2a, 0x1422c678, 0xe9204de7, 0xb7c2e759,
0x0200248e, 0x013b446b, 0xda0d9fc2, 0x0414a895,
0x3a6cc3a1, 0x56fef170, 0x86c19155, 0xcf7b8a66,
0x551b5e69, 0xb4a8623e, 0xa2bdfa35, 0xc4f068cc,
0x573a6acd, 0x6355e936, 0x03602db9, 0x0edf13c1,
0x2d0bb16d, 0x6980b83c, 0xfeb23763, 0x3dd8a911,
0x01b6bc13, 0xf55579d7, 0xf55c2fa8, 0x19f4196e,
0xe7db5476, 0x8d64a866, 0xc06e16ad, 0xb17fc515,
0xc46feb3c, 0x8bc8a306, 0xad6799d9, 0x571a9133,
0x992466dd, 0x92eb5dcd, 0xac118f50, 0x9fafb226,
0xa1b9cef3, 0x3ab36189, 0x347a19b1, 0x62c73084,
0xc27ded5c, 0x6c8bc58f, 0x1cdde421, 0xed1e47fb,
0xcdcc715e, 0xb9c0ff99, 0x4b122f0f, 0xc4d25184,
0xaf7a5e6c, 0x5bbf18bc, 0x8dd7c6e0, 0x5fb7e420,
0x521f523f, 0x4ad9b8a2, 0xe9da1a6b, 0x97888c02,
0x19d1e354, 0x5aba7d79, 0xa2cc7753, 0x8c2d9655,
0x19829da1, 0x531590a7, 0x19c1c149, 0x3d537f1c,
0x50779b69, 0xed71f2b7, 0x463c58fa, 0x52dc4418,
0xc18c8c76, 0xc120d9f0, 0xafa80d4d, 0x3b74c473,
0xd09410e9, 0x290e4211, 0xc3c8082b, 0x8f6b334a,
0x3bf68ed2, 0xa843cc1b, 0x8d3c0ff3, 0x20e564a0,
0xf8f55a4f, 0x2b40f8e7, 0xfea7f15f, 0xcf00fe21,
0x8a6d37d6, 0xd0d506f1, 0xade00973, 0xefbbde36,
0x84670fa8, 0xfa31ab9e, 0xaedab618, 0xc01f52f5,
0x6558eb4f, 0x71b9e343, 0x4b8d77dd, 0x8cb93da6,
0x740fd52d, 0x425412f8, 0xc5a63360, 0x10e53ad0,
0x5a700f1c, 0x8324ed0b, 0xe53dc1ec, 0x1a366795,
0x6d549d15, 0xc5ce46d7, 0xe17abe76, 0x5f48e0a0,
0xd0f07c02, 0x941249b7, 0xe49ed6ba, 0x37a47f78,
0xe1cfffbd, 0xb007ca84, 0xbb65f4da, 0xb59f35da,
0x33d2aa44, 0x417452ac, 0xc0d674a7, 0x2d61a46a,
0xdc63152a, 0x3e12b7aa, 0x6e615927, 0xa14fb118,
0xa151758d, 0xba81687b, 0xe152f0b3, 0x764254ed,
0x34c77271, 0x0a31acab, 0x54f94aec, 0xb9e994cd,
0x574d9e81, 0x5b623730, 0xce8a21e8, 0x37917f0b,
0xe8a9b5d6, 0x9697adf8, 0xf3d30431, 0x5dcac921,
0x76b35d46, 0xaa430a36, 0xc2194022, 0x22bca65e,
0xdaec70ba, 0xdfaea8cc, 0x777bae8b, 0x242924d5,
0x1f098a5a, 0x4b396b81, 0x55de2522, 0x435c1cb8,
0xaeb8fe1d, 0x9db3c697, 0x5b164f83, 0xe0c16376,
0xa319224c, 0xd0203b35, 0x433ac0fe, 0x1466a19a,
0x45f0b24f, 0x51fda998, 0xc0d52d71, 0xfa0896a8,
0xf9e6053f, 0xa4b0d300, 0xd499cbcc, 0xb95e3d40
```

## 10.2 WeakFlex8

WeakFlex8 is an intentionally weakened version of FastFlex for analysis in academic circles. WeakFlex8 uses a key size of 48 bits, manipulates 8 bit bytes instead of 32 bit words.
The two differences in the structure of WeakFlex8 and FastFlex are:

- Rotation distances in the Row() function of HashCore and in the SAN.
- Use of an 8*8 S-box derived from Skipjack instead of a 8*32 S-box as building block of the Virtual S-box.

It should be noted that the structure of FastFlex has not been changed. The numbers of rounds of the HashCore remain the same in WeakFlex8. However, the 48 bit key used in the stream cipher constructed using WeakFlex8 should make cryptanalysis considerably easier and lend more insight into the security properties of FastFlex.

### 10.2.1 WeakFlex8

Being already familiar with the design of FastFlex, we now describe WeakFlex8.

- The HashCore of WeakFlex8 takes as input eight 8-bit bytes and as output produces eight 8-bit bytes.
- The Row() function of HashCore in WeakFlex8 is:

$$\{Y_0,Y_1,Y_2,Y_3\} = \mathbf{Row}(W_0,W_1,W_2,W_3)$$

$$\left\{\begin{array}{l} Y_1 = W_1 \oplus (W_0 + W_3) \lll 1 \\ Y_2 = W_2 \oplus (Y_1 + W_0) \lll 3 \\ Y_3 = W_3 \oplus (Y_2 + Y_1) \lll 5 \\ Y_0 = W_0 \oplus (Y_3 + Y_2) \lll 8 \end{array}\right\}$$

**where**
+ denotes addition Mod $2^8$,
⊕ denotes bitwise XOR
**x<<< y** denotes left rotation of byte x by y positions.

- The transformations on elements in the same row are
  $\{Y_0,Y_1,Y_2,Y_3\} = \mathbf{Row}(W_0,W_1,W_2,W_3)$
  $\{Y_6,Y_7,Y_4,Y_5\} = \mathbf{Row}(W_6,W_7,W_4,W_5)$
- This is followed by transformations on elements in the same column,
  $\{Y_I,Y_{I+4}\} = \mathbf{Col}(Y_I,Y_{I+4})$

  where

$$\left\{\begin{array}{l} Y_I = (Y_I \otimes Y_{I+4}) \;\&\; 0x00FF \\ Y_{I+4} = S(Y_I \otimes Y_{I+4}) \end{array}\right\}$$

- The HashCore iterates 8 times during a round of WeakFlex8.
- The SAN for WeakFlex8 is defined as a 64:64 bit mapping.
- The SAN executes only once during a round of WeakFlex8.
- The present 8 byte output of HashCore becomes the input for the next round, till no more blocks remain

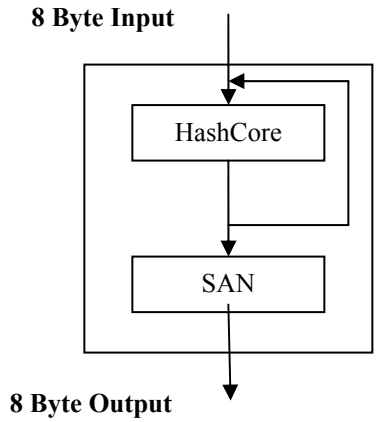



Fig. 5. WeakFlex8

The SAN uses a 8*8 S-box derived from Skipjack (F-Table) **[14]**. The key features of the S-box are:

- High Nonlinearity
- 8:8, one to one bit mapping
- No known algebraic structure

**The S-box[ ] is as follows:**

```
0XA3, 0XD7,  0X09,  0X83,  0XF8,
0X48, 0XF6,  0XF4,  0XB3,  0X21,
0X15, 0X78,  0X99,  0XB1,  0XAF,
0XF9, 0XE7,  0X2D,  0X4D,  0X8A,
0XCE, 0X4C,  0XCA,  0X2E,  0X52,
0X95, 0XD9,  0X1E,  0X4E,  0X38,
0X44, 0X28,  0X0A,  0XDF,  0X02,
0XA0, 0X17,  0XF1,  0X60,  0X68,
0X12, 0XB7,  0X7A,  0XC3,  0XE9,
0XFA, 0X3D,  0X53,  0X96,  0X84,
0X6B, 0XBA,  0XF2,  0X63,  0X9A,
0X19, 0X7C,  0XAE,  0XE5,  0XF5,
0XF7, 0X16,  0X6A,  0XA2,  0X39,
0XB6, 0X7B,  0X0F,  0XC1,  0X93,
0X81, 0X1B,  0XEE,  0XB4,  0X1A,
0XEA, 0XD0,  0X91,  0X2F,  0XB8,
0X55, 0XB9,  0XDA,  0X85,  0X3F,
0X41, 0XBF,  0XE0,  0X5A,  0X58,
0X80, 0X5F,  0X66,  0X0B,  0XD8,
0X90, 0X35,  0XD5,  0XC0,  0XA7,
0X33, 0X06,  0X65,  0X69,  0X45,
0X00, 0X94,  0X56,  0X6D,  0X98,
0X9B, 0X76,  0X97,  0XFC,  0XB2,
0XC2, 0XB0,  0XFE,  0XDB,  0X20,
0XE1, 0XEB,  0XD6,  0XE4,  0XDD,
0X47, 0X4A,  0X1D,  0X42,  0XED,
0X9E, 0X6E,  0X49,  0X3C,  0XCD,
0X43, 0X27,  0XD2,  0X07,  0XD4,
0XDE, 0XC7,  0X67,  0X18,  0X89,
0XCB, 0X30,  0X1F,  0X8D,  0XC6,
0X8F, 0XAA,  0XC8,  0X74,  0XDC,
0XC9, 0X5D,  0X5C,  0X31,  0XA4,
0X70, 0X88,  0X61,  0X2C,  0X9F,
0X0D, 0X2B,  0X87,  0X50,  0X82,
0X54, 0X64,  0X26,  0X7D,  0X03,
0X40, 0X34,  0X4B,  0X1C,  0X73,
0XD1, 0XC4,  0XFD,  0X3B,  0XCC,
0XFB, 0X7F,  0XAB,  0XE6,  0X3E,
0X5B, 0XA5,  0XAD,  0X04,  0X23,
0X9C, 0X14,  0X51,  0X22,  0XF0,
0X29, 0X79,  0X71,  0X7E,  0XFF,
0X8C, 0X0E,  0XE2,  0X0C,  0XEF,
0XBC, 0X72,  0X75,  0X6F,  0X37,
0XA1, 0XEC,  0XD3,  0X8E,  0X62,
0X8B, 0X86,  0X10,  0XE8,  0X08,
0X77, 0X11,  0XBE,  0X92,  0X4F,
0X24, 0XC5,  0X32,  0X36,  0X9D,
0XCF, 0XF3,  0XA6,  0XBB,  0XAC,
0X5E, 0X6C,  0XA9,  0X13,  0X57,
0X25, 0XB5,  0XE3,  0XBD,  0XA8,
0X3A, 0X01,  0X05,  0X59,  0X2A,
0X46
```

The SAN is a one way 64:64 bit mapping. It consists of four layers of operations which do not commute. The structure of the SAN is as follows:

**Layer 1:**
A’ = [A + S-box(B) + F]
B’ = [B + S-box(C) + G]
C’ = [C + S-box(D) + H]
D’ = [D + S-box(A) + E]

**Layer 2:**
E’ = [E + S-box(A’)]
F’ = [F + S-box(B’)]
G’ = [G + S-box(C’)]
H’ = [H + S-box(D’)]

**Layer 3:**
A’’ = [A’ ⊕ S-box(G’)]  B’’ = [B’ ⊕ S-box(H’)]
C’’ = [C’ ⊕ S-box(E’)]  D’’ = [D’ ⊕ S-box(F’)]

**Layer 4:**
E’’ = [E’ + A’’] ⊕ F’
F’’ = [F’ + B’’] ⊕ G’
G’’ = [G’ + C’’] ⊕ H’
H’’ = [H’ +D’’] ⊕ E’

where
S-box(N) is the to the $N^{th}$ entry in S-box[ ],
+ denotes addition Mod $2^8$ and
⊕ denotes bitwise XOR

Let us summarize the design of WeakFlex8:

- WeakFlex8 operates on eight 8-bit bytes.
- WeakFlex8 accepts 8 bytes as input and produces 8 bytes as output for every round.
- A round of WeakFlex8 corresponds to 8 rounds of HashCore and 1 round of SAN.

We now construct a stream cipher providing 48 bit security using WeakFlex8.

### 10.2.2 WeakStream48

WeakStream48 is a deliberately weakened synchronous stream cipher structurally similar to FastStream192. Keystream generation in WeakStream48 is done by one round of WeakFlex8. The keystream generator takes two inputs, a 48 bit secret key **K** $\{K_0, K_1, K_2, K_3 K_4, K_5\}$ and a 16 bit nonce **N** $\{N_0, N_1\}$, where $K_0$, $N_0$ are the least significant bytes of K and N respectively.
It must be noted that FastStream192 and WeakStream48 are identical in structure. The differences between them are as follows:

- WeakStream48 accepts a 48 bit key and a 16 bit nonce whereas FastStream192 accepts a 192 bit key and a 64 bit nonce.
- WeakStream48 produces keystream 8 bytes at a time, whereas FastStream 192 produces keystream 8 words at a time.
- For a given key-nonce pair, WeakStream48 can be used to generate $2^{22}$ bits of keystream, whereas for FastStream192 $2^{64}$ bits of keystream can be generated uing a given key-nonce pair. A longer keystream is generated so as to allow thorough study of its properties.

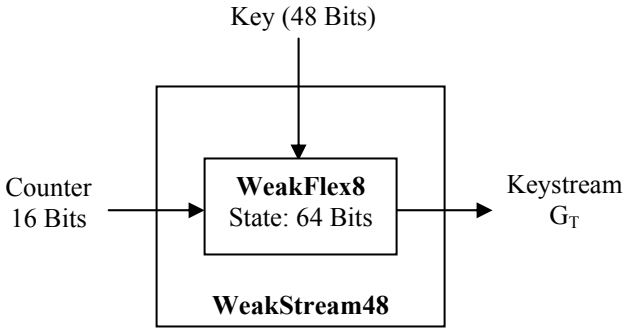



Fig. 6. WeakStream48

**At the Transmitter (Encryption):**
$\mathbf{C_T = G_T \oplus M_T}$

**At the Receiver (Decryption):**
$\mathbf{M_T = G_T \oplus C_T}$

where $M_T$ denotes the message stream,
$G_T$ denotes the keystream
and $C_T$ denotes the corresponding ciphertext stream.

### 10.2.3 Keystream Generation in WeakStream48

Keystream is generated in WeakStream48 following the same approach as used in FastStream192. Each round of WeakStream48 uses 1 round of WeakFlex8. The keystream generator takes two inputs, a 48 bit secret key **K** $\{K_0, K_1, K_2, K_3, K_4, K_5\}$ and a 16 bit nonce **N** $\{N_0, N_1\}$, where $K_0$, $N_0$ are the least significant bytes of K and N respectively.

The process of keystream generation is as follows:

- The inner state of the WeakFlex8 is loaded with **K** and **N** as shown below

$$\begin{Bmatrix} N_0 & K_0 & K_1 & K_2 \\ K_3 & K_4 & K_5 & N_1 \end{Bmatrix}$$

- WeakFlex8 is allowed to run four times with the output of the present round becoming the inner state of WeakFlex8 for the next round.

$$\begin{Bmatrix} N_0 & K_0 & K_1 & K_2 \\ K_3 & K_4 & K_5 & N_1 \end{Bmatrix}$$

$\downarrow$ 4 Rounds of WeakFlex8

$$\begin{Bmatrix} A & B & C & D \\ E & F & G & H \end{Bmatrix}$$

The keystream generator is now ready to produce keystream.

- A single round of WeakFlex8 is then called which produces 8 bytes of keystream.

$$\begin{Bmatrix} A & B & C & D \\ E & F & G & H \end{Bmatrix}$$

$\downarrow$ 1 Round of WeakFlex8

$$\begin{Bmatrix} G_0 & G_1 & G_2 & G_3 \\ G_4 & G_5 & G_6 & G_7 \end{Bmatrix}$$

which can be written as
**{G} = WeakFlex8(A,B,C,D,E,F,G,H)**

- The counter is updated for generating the next 8 bytes of keystream by incrementing HA by 1. The value of the counter to be loaded for the next round of encryption thus becomes HA + 1.
- On generating every 8 bytes of keystream, the counter is again incremented by 1 before generating the next bytes of the keystream.

### 10.2.4 Period of Keystream for WeakStream48

**WeakStream48(K,N)** denotes the WeakStream48 stream cipher function acting on the 6 Byte key K and the 2 Byte sequence N, N + 1,…N + $(2^{16} - 1)$.
**WeakStream48 thus has a period of $2^{22}$,** as the counter rolls over to its initial value after $2^{16}$ increments and for every counter value, 8 bytes of keystream are generated.