A New Version of the Stream Cipher Polar Bear

Johan Håstad[1], John Mattsson[2] and Mats Näslund[3]*

[1] CSC, Royal Inst. of Technology, SE-10044 Stockholm, Sweden
[2] john.mattsson@gmail.com
[3] Communications Security Lab, Ericsson Research, SE-16480 Stockholm, Sweden

June 30, 2006

Abstract

In April 2005, the stream cipher Polar Bear was proposed as a response to the eSTREAM Call for Primitives. Since then, a few attacks, [7] and [5], have indicated certain weaknesses in the design. In this paper we propose a new version of Polar Bear, called Polar Bear 2.0. The main difference is a key-dependent premixing of the D_8 table in conjunction with the key schedule. The new version does not only appear to be more secure, the performance on all packet sizes is improved.

1 Introduction

There is a big industry demand for secure and efficient stream ciphers, the mobile/wireless communication sector being one of the foremost 'consumers' of such ciphers. The fact that stream ciphers do not expand messages, are tolerant to bit-errors[1], etc, are desirable properties for use with narrow-bandwidth wireless links and voice coders designed to perform well in the presence of a few bit errors. From scientific point of view, there is also an interest to get a better understanding for how to design stream ciphers, since many of the proposed schemes in the past have been more or less severely attacked.

As a response to the lack of sufficiently secure and efficient ciphers, the IST 6FP ECRYPT Network of Excellence manages and coordinates a multi-year effort called eSTREAM to identify new stream ciphers suitable for widespread adoption. The initial call for Primitives resulted in 34 candidates. One of the proposals was our stream cipher Polar Bear, which was submitted to both profile I (software) and profile II (hardware). During the first phase of eSTREAM, two attacks on Polar Bear have been found. In [7], a guess-and-determine that recovers the state with a expected complexity of 2^{79} is

*Work described here has in part been supported by the Commission of the European Communities through the IST program under contract IST-2002-507932. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

[1]Of course, the error tolerance also open up for attacks on the integrity.

presented. By using the same attack principle, but with a more careful analysis and selection of guessed values, the complexity was lowered to 2^{57} in [5].

The first phase of eSTREAM has now ended, and an initial classification of the algorithms have been made. Of the 34 proposals, 25 have advanced to the second phase. Polar Bear has advanced in the software profile as well as in the hardware profile. Before the second evaluating phase, tweaks to the algorithms are accepted. In this paper, such a tweak of Polar Bear is presented. The main difference is a key-dependent premixing of the D_8 table in conjunction with the key schedule. The new version does not only appear to be more secure, the performance on all package sizes is improved.

2 Notation

Whenever we say 'AES' or 'Rijndael', we refer to the Rijndael book [4], and not the AES specification [1], since we need the support for 256-bit blocks.

We use $||$ to denote concatenation, $\oplus$ to denote bit-wise modulo-2 sum, and $+_m$ to denote addition modulo 2^m. The swap operation, $x \leftrightarrow y$, interchanges x and y.

We use byte and word to denote 8- and 16-bit quantities, respectively. Hexadecimal quantities are written in typewriter typeface and indexed with '16', i.e. $156 = \mathtt{9C}_{16}$. If $a, b, c, \ldots$ are bytes, a is the most significant byte in $a||b||c||\ldots$ etc. Let S_R be the Rijndael S-box, i.e. $S_R[0] = \mathtt{63}_{16}$, $S_R[1] = \mathtt{7C}_{16}$, $\ldots$, $S_R[256] = \mathtt{16}_{16}$.

We shall make use of arithmetic in $\mathbb{F}_{2^{16}}$, which we throughout represent as

$$\mathbb{F}_2[y]/(y^{16} + y^8 + y^7 + y^5 + 1).$$

Elements of this field shall sometimes be represented by integers, where the integer $a = \sum_{i=0}^{15} a_i 2^i$ represents the field element $a(y) = \sum_{i=0}^{15} a_i y^i$ in the standard polynomial basis representation.

3 Weaknesses in the first version of Polar Bear

In this section we describe the weaknesses found in the original construction [6]. In October 2005, Mattsson described a guess-and-determine attack on Polar Bear. The attack recovers the initial state using the the first 24 bytes of keystream and a process complexity of 2^{79} [7]. By assuming that the α-values are all different, the permutation array D_8 can be treated as a known constant, and by assuming a specific stepping of the registers, the relations between the stages are known. Under this assumption it suffices to guess 4 LFSR stages (64 bits) to recover the state.

Hasanzadeh et al. have lowered the time complexity in a similar attack [5]. By using the same attack principle, but with a more careful analysis and selection of steppings and 'guessed' values, they reach an overall attack complexity of $O(2^{57.4})$. In this way, they only need to guess the values of two LFSR stages. They also notice that, because of the known stepping, only 2^{31} of the 2^{32} values are possible.

The main weakness exploited in the above attacks is the initially known permutation array D_8. We propose that the security is enhanced by adding a key-dependent premixing of the D_8 table in conjunction with the key schedule. We also propose some simplifications that lead to better performance.

4 Specification of Polar Bear 2.0

4.1 Informative Description

We first give an informal description of the cipher's operation. The cipher uses one 7-word (112-bit) LFSR R^0 and one 9-word (144-bit) LFSR R^1. These are viewed as acting over $\mathbb{F}_{2^{16}}$. Besides these registers, the internal state of the cipher also depends on a dynamic permutation of bytes, D_8.

The cipher is primarily designed for a key length of 128 bits, but key lengths in the range 10–16 bytes are accepted. The IV can be any number of bytes, up to a maximum of 31. The IV size may vary from message to message for a given key, if so desired.

The key if first padded to 512 bits. The key expansion is then identical to the Whirlpool key expansion with 12 rounds. The expanded key is used to permute the Rijndael S-box, and the last 192 bytes are used as round keys in the Rijndael encryption.

On each message to be processed, the cipher is initialized by taking the key (more precisely, the expanded key), interpreting the IV as a cleartext block, and applying a (slightly modified) four round Rijndael encryption with block length 256. The resulting cipher text block is loaded into R^0 and R^1. Finally, D_8 is initialized to equal the permuted byte array.

Output is produced 4 bytes at a time. To this end, the two LFSRs are first clocked twice. Bytes are selected from R^0 and R^1, and run through the permutation D_8 to produce the 4 output bytes. Selected entries in D_8 are swapped. Finally, R^0 are modified in preparation for the next output cycle. Entries in R^1 are not modified apart from the LFSR stepping.

We now turn to the normative cipher specification.

4.2 Initialization (Normative)

We use key expansion from Whirlpool with the parameter $r = 12$, and the round function from Rijndael with the parameters $Nb = 8$, $Nr = 4$, and $Nk = 4$, and applying the `MixColumns` transformation in each round. The descriptions in the following two subsections is (intended to be) identical to those in [2] and [4], and are only reproduced for self-containment.

4.2.1 Key Schedule

If the key consists of m bytes $(k_i)_{i=0}^{m-1}$, pad the key by setting $k_i \triangleq S_R[k_{i-m}]$ for $m \leq i \leq 63$. The key expansion is now identical to the Whirlpool key expansion with $r = 12$ and $K^0 \triangleq k_0||k_1||\ldots||k_{63}$. The key schedule expands the 512-bit padded key K^0 to twelve 512-bit round keys K^1, K^2 $\ldots$, K^{12} according to

$$K^r \triangleq \rho[c^r](K^{r-1}),\ r > 0,$$

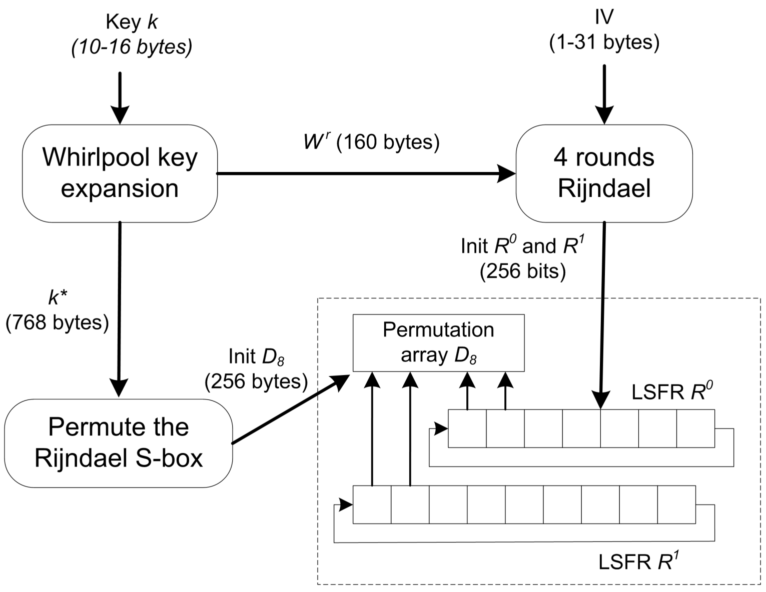

Figure 1: *Key and IV schedule*

where $\rho[\kappa]$ is the Whirlpool round function and c^r is the round constant for the rth round. The round constants are matrices defined as

$$\begin{aligned} c^r_{0j} &\triangleq S_W[8(r-1)+j], && 0 \le j \le 7, \\ c^r_{ij} &\triangleq 0, && 1 \le i \le 7,\ 0 \le j \le 7, \end{aligned}$$

where S_W is the Whirlpool S-box. The Whirlpool round keys are now treated as 768 bytes $(k^*_i)_{i=0}^{767}$, where $K^r = k^*_{64r}||\dots||k^*_{64r+63}$. The expanded key is now used to permute the Rijndael S-box S_R according to

$$\begin{aligned} &\text{For } i = 0 \dots 767 \\ &\quad S_R[i \bmod 256] \leftrightarrow S_R[k^*_i] \end{aligned}$$

We call this table, permuted with the key k, for S^k_R. The last 160 bytes of the expanded key k^*_i are used to form the five round keys used in the Rijndael encryption during the IV schedule. Each W^r is interpreted as a 4 × 8 array of bytes.

$$W^r_{i,j} \triangleq k^*_{608+32r+4j+i}, \quad 0 \le r \le 4,\ 0 \le j \le 7,\ 0 \le i \le 3$$

The key schedule may be done once per key, saving the permuted table S^k_R and the five round keys W^r, $r = 0, \dots, 4$.

4.2.2 IV Schedule

Let the initiation vector contain n bytes, $(IV_i)_{i=0}^{n-1}$ where $n \le 31$. Set $IV_n \triangleq 80_{16}$ and $IV_i \triangleq 00_{16},\ n+1 \le i \le 31$. Load the IV into an 4 × 8 array by setting

$$A_{i,j} \leftarrow IV_{4j+i} \quad 0 \le i \le 3,\ 0 \le j \le 7.$$

Perform a four round encryption using Rijndael. First, for $r = 0, \ldots, 3$, repeat the following four steps:

1. `AddRoundKey` — Add the round key W^r to the state,
$$A_{i,j} \leftarrow W^r_{i,j} \oplus A_{i,j} \quad 0 \le i \le 3,\ 0 \le j \le 7.$$
2. `SubBytes` — Apply the substitution table to each byte, $A_{i,j} \leftarrow S_R[A_{i,j}]$.
3. `ShiftRows` — The bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The second row is shifted one step right, the third row is shifted two steps, and fourth row is shifted three steps. The first row is not shifted.
4. `MixColumns` — Treat each column as a degree 3 polynomial over $\mathbb{F}_{2^8}$ and multiply it by $a(x) = (1+y)x^3 + x^2 + x + y$ modulo $x^4 + 1$.

Then, apply a last `AddRoundKey` transformation with $r = 4$. Use the output to initialize the registers R^0 and R^1

$$\begin{aligned} y_{4j+i} &\triangleq A_{i,j}, && 0 \le i \le 3,\ 0 \le j \le 7 \\ R^0_n &\leftarrow y_{2n+1} || y_{2n}, && 0 \le n \le 6 \\ R^1_n &\leftarrow y_{2n+15} || y_{2n+14}, && 0 \le n \le 8 \end{aligned}$$

The table D_8 is initialized to agree with the permuted Rijndael S-box, $D_8[j] \leftarrow S^k_R[j]$, $0 \le j \le 255$. This completes cipher initialization.

4.3 The output cycle (Normative)

After each update of the cipher's internal state, four bytes are output. Before the first output byte, and between consecutive output pairs of bytes, a state update function is performed as specified below.

4.3.1 Next state function

Let $\ell_0 = 7$ and $\ell_1 = 9$ be the lengths of the LFSRs, R^0, R^1. Register R^i is stepped two steps with a sparse feedback, where each step consists of

- set $f^i \leftarrow \theta^i R^i_{j^i} + \mu^i R^i_0$ for constants θ^i, j^i, and μ^i,
- set $R^i_j \leftarrow R^i_{j+1}$ for $j = 0, 1, \ldots, \ell_i - 2$, and
- feedback $R^i_{\ell_i - 1} \leftarrow f^i$.

Specifically, R^0 and R^1 are defined by the primitive polynomials

$$p^0(x) \triangleq \texttt{5CEB}_{16} \cdot x^7 + \texttt{8B5A}_{16} \cdot x^6 + 1$$

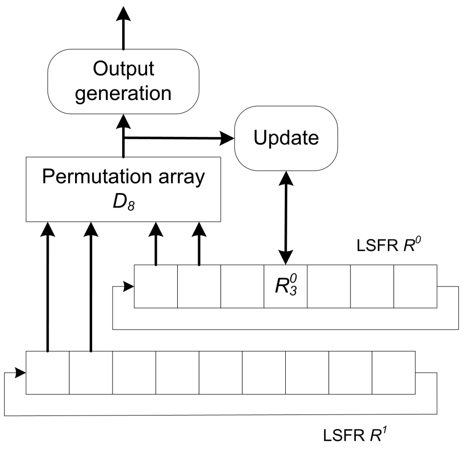

Figure 2: *The output cycle*

and

$$p^1(x) \triangleq \texttt{2C62}_{16} \cdot x^9 + \texttt{689A}_{16} \cdot x^4 + 1,$$

so that $j^0 = 1$, $j^1 = 5$, $\mu^0 = \texttt{5CEB}_{16}$, $\mu^1 = \texttt{2C62}_{16}$, $\theta^0 = \texttt{8B5A}_{16}$, and $\theta^1 = \texttt{689A}_{16}$.

After stepping both R^0 and R^1 above, do the following five steps, first for $i = 0$, then repeat them for $i = 1$:

1. Write $(R^i_{\ell_i-1}, R^i_{\ell_i-2})$ as four bytes $\alpha^i_0||\alpha^i_1||\alpha^i_2||\alpha^i_3$.
2. Let $\beta^i_0 = D_8[\alpha^i_0]$ and $\beta^i_2 = D_8[\alpha^i_2]$.
3. Swap elements in D_8 by $D_8[\alpha^i_0] \leftrightarrow D_8[\alpha^i_2]$.
4. Let $\beta^i_1 = D_8[\alpha^i_1]$ and $\beta^i_3 = D_8[\alpha^i_3]$.
5. Swap elements in D_8 by $D_8[\alpha^i_1] \leftrightarrow D_8[\alpha^i_3]$.

Next, update R^0:

- Update R^0 according to $R^0_3 \leftarrow R^0_3 +_{16} \beta^1_1||\beta^1_2$.

At this point, the internal state is updated, and the output is formed from the above β-values as described next.

4.3.2 Output generation

Form four output bytes $b_0||b_1||b_2||b_3$ where

$$b_0 \triangleq \beta^0_i \oplus \beta^1_{(i+1) \bmod 4}, \qquad i = 0, \ldots, 3.$$

If more output bytes are required, the output cycle is repeated.

5 Rationale

The core of Polar Bear 1.0, the IV/key mixing based on Rijndael and the two LFSRs updating a dynamically changing permutation has been kept. We therfore think it is safe to say that it is indeed a tweak of the same cipher, and not a new design.

As discussed in [7, 5] it appears that initially mixing the table D_8 solves the known security problems of the first version of Polar Bear. Though other changes could have also removed the security problems, the nice feature with adding a pre-mixing of D_8 is that it does not effect the encryption performance, only the key set-up. The choice of mixing the table cyclicly for three "rounds" is motivated by studies of the mixing speed of e.g. RC4. To perform this mixing, more key material is needed. We here chose to use the Whirlpool key schedule rather than that of Rijndael, as it appears that the key schedule of Whirlpool is a better design for this purpose.

Adding the premixing of D_8 indeed seems to strengthen the cipher considerably. We therefore also decided to make some simplifications to increase performance. First, the irregular stepping of the LFSRs has been removed (which by the way did not help against the previous attacks). Thus, the quantity S could also be removed. The number of Rijndael rounds can be reduced from five to four. Finally, the output formation (the order in which the bytes are formed) have been changed slightly as it improves the performance.

To the best of our knowledge, the set of introduced changes strengthen the cipher. In particular, we certify that we have not intentionally introduced any 'back-doors' in Polar Bear 2.0.

6 Performance

Because of the small tweak that changes how the permutation of the table D_8 is done, Polar Bear 2.0 is faster than Polar Bear 1.0 on long sequences as can be seen in the performance figures for the Pentium M 1.6 GHz. This makes Polar Bear 2.0 faster than AES-CTR (AES in counter mode). We believe that performance on long streams can be further optimized with 1–2 cycles/byte. If this is the case, we will post the results on the eSTREAM forum. We will also post the missing performance figures for Polar Bear 2.0.

All the performance tests in Table 1 were done with the eSTREAM testing framework. The data for Intel Pentium M 1.7 GHz is taken from the framework's homepage [3]. The performance figures for the Pentium M 1.6 were created by us with the testing framework Live CD. The Live CD includes Ubuntu Linux, Intel C++ Compiler 8.1, Microsoft Visual C++ Toolkit 2003, and several different version of GCC. The source code is compiled with all three compilers with a large number of compiler options, and the one with the fastest stream performance is chosen.

The stream performance is probably the most important criteria as it is here that stream ciphers has the biggest potential advantage over block ciphers. The key setup is probably the least critical as the time for key setup is typically negligible to the work

Table 1: *Performance figures*

CPU	Name	Stream	40 bytes	Agility	Key Setup	IV Setup
Intel Pentium M 1.7 GHz	SNOW-2.0	4.61	29.82	5.98	63.81	801.73
	RC4*	7.52	335.37	19.52	112.41	13005.38
	AES-CTR	21.78	28.79	24.59	217.74	43.01
Intel Pentium M 1.6 GHz	Polar Bear 2.0	15.68	.	.	.	.
	Polar Bear 1.0	22.69	45.37	26.06	281.81	906.70

Stream – Asymptotic encryption rate (cycles/byte). 40 bytes – Packet encryption rate (cycles/byte). Agility – Parallel encryption rate (cycles/byte). Key and IV setup – The efficiency of the key setup and IV setup (cycles)
* The key size and IV size was 128 bit for AES-CTR, Polar Bear, and SNOW 2.0. As RC4 do not include support for initialization vectors, the benchmarks are for RC4 with a 256 bit key.

needed to generate and exchange the key [3].

As the values in the table above were created with the compiler and options that gave the best performance on long streams, the other benchmarks, such as the numbers of cycles for IV schedule, are not comparable. All variants of the Polar Bear code are fastest with Intel's compiler, SNOW 2.0 and RC4 are fastest with GCC, whereas Microsoft Visual C++ creates the fastest code for AES-CTR.

References

[1] Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.
`http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[2] Paulo S.L.M. Barreto and Vincent Rijmen. The Whirlpool hashing function. ISO/IEC 10118-3:2004, 2003.
`http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`

[3] Christophe De Cannire. eSTREAM testing framework.
`http://www.ecrypt.eu.org/stream/perf/`

[4] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3-540-42580-2. 238 pp.

[5] Mahdi Hasanzadeh, Elham Shakour, and Shahram Khazaei. Improved Cryptanalysis of Polar Bear. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/084, 2005.
`http://www.ecrypt.eu.org/stream`

[6] Johan Håstad and Mats Näslund. The Stream Cipher Polar Bear. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/021, 2005.
`http://www.ecrypt.eu.org/stream`

[7] John Mattsson. A Guess-and-Determine Attack on the Stream Cipher Polar Bear. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/017, 2006.
`http://www.ecrypt.eu.org/stream`

A Test Vector

All values in hexadecimal

A.1 Key schedule

Key k (128 bits):

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Expanded key K^0 (512 bits):

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63
fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f
```

Round keys $W^0, \ldots, W^4$ (256 bits each):

```
7c 13 b4 69 d5 d6 65 3b ed 9b 7e 76 d5 6c 40 da
c8 5d 06 b3 cc cf c8 55 b1 87 00 74 55 a4 77 74

00 fb a2 25 f8 23 0d 94 fb 62 ba 27 8e 94 6a 76
a9 c8 14 b7 dd d4 fc 14 d8 4e 3e 8d af 3a 57 41

15 23 93 d7 28 78 b2 3d 70 0a b3 08 9b ee 6e 15
e4 12 fd ed 13 39 16 10 d9 de 43 86 b9 f5 ef 68

a7 81 4d 15 71 db eb 9d bd 6d ed 8e 6b 22 2f 0a
2d 10 c8 e9 90 e2 32 51 3a 17 c0 9b aa 62 b8 4b

bd 96 bf 0c 5c 22 ad ea 1f fa 7c d4 81 8b 02 8f
95 5c 8c 9b 36 e2 f5 3c 94 d4 e1 36 47 06 5c 75
```

Permuted byte array S_R^k (256 bytes):

```
99 61 7a df 95 47 68 82 ee 55 0e 25 5c b3 cc 5a
9d a4 72 d2 36 85 c3 52 05 8b 1e d3 c4 97 c8 e1
56 31 20 d1 71 3c a5 2b 1d 0a b5 35 c2 e0 69 48
cb 10 46 dd 04 9f 29 4c 7d ca b4 63 f5 80 5e 73
ad 12 60 f8 c0 7c 38 93 8e 2a 8f 5f a8 f4 27 a1
9a da db ce dc 0c 02 91 b2 e9 aa 77 fa 44 2f 6b
d8 6d d0 3a d5 ba e7 fb 01 0d 2c 41 1f 00 cf 09
59 51 c7 4f 53 86 8d f6 f3 64 4b ea 5d 6f 18 28
6a 24 bd a7 21 1b 90 3e 92 8a 30 ef 88 d6 07 a0
14 a2 f1 eb 3f ff 45 4d c5 57 bb f7 c6 19 06 f2
37 9b d4 5b 7b 4e 40 a9 a3 7f 32 98 c1 42 af bc
2e 50 16 fc 3d e5 43 0b 79 e4 65 6e d9 ab 17 9e
```

```
b1 39 84 22 13 2d b0 15 e6 ae 49 26 0f ed 54 fe
ec 08 3b 33 d7 34 b7 bf cd 83 03 66 1c 58 7e 6c
de b8 c9 11 e2 67 be 75 f0 fd b6 8c b9 74 76 78
4a 1a 96 ac 89 f9 81 e3 70 e8 94 87 9c 23 62 a6
```

A.2 IV schedule

Initiation vector IV (128 bits):

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Expanded IV:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Initial value of $R^0_6, \ldots, R^0_0$:

```
8a2b de66 05d5 9e31 4576 a196 1ca9
```

Initial value of $R^1_8, \ldots, R^1_0$:

```
1ce9 798b 846b e758 bca5 30d1 b54f d28d 12ab
```

A.3 Output

Keystream output (The first 16 bytes):

```
ff 58 be 74 44 13 d4 05 62 cf 69 8c 2a 48 d2 63
```