Hermes8F : A Low-Complexity Low-Power Stream Cipher

Ulrich Kaiser

Texas Instruments Deutschland GmbH, 85350 Freising, Germany
d-kaiser@ti.com

Abstract. Since stream ciphers have the reputation to be inefficient in software applications the new stream cipher Hermes8F has been developed. It is based on a 8-bit-architecture and an algorithm with low complexity. The two versions presented here are Hermes8F-80 with 17 byte state and 10 byte key and furthermore Hermes8F-128 with 17 byte state and 16 byte key. Both are suited to run efficiently on 8-bit micro computers and dedicated hardware (e.g. for embedded systems). The estimated performance is up to one encrypted byte per 76 CPU cycles and eight encrypted bytes per 44 cycles in hardware. The clarity and low complexity of the design supports cryptanalytic methods. The 8x8 sized S-BOX provides the non-linear function needed for proper confusion. Hermes8F uses the well-established AES S-BOX, but works also excellent with well-designed random S-BOXes. Hermes8F withstands so far several 'attacks' by means of statistical tests, e.g. the Strict Avalanche Criterion, Berlekamp-Massey-Test, Diehard tests, and FIPS 140-2 tests are met successfully.

1 Introduction

Stream ciphers of today have the reputation to be very efficient in hardware, but slow and costly in software. Often Linear Feedback Shift Registers (LFSRs) are taken as building blocks, because their hardware efficiency and their statistical properties are well known [1,2,3]. - The cryptographic community is well served by a variety of efficient and trusted block ciphers. However, the same doesn't seem to hold for stream ciphers.

In 2004 the ECRYPT Network of Excellence (NoE) initiated a multi-year effort to identify new stream ciphers suitable for widespread adoption. Algorithm designers were invited to submit new stream cipher proposals (http://www.ecrypt.eu.org/stream).

Following public discussions at the State of the Art of Stream Ciphers (SASC) Workshop in Bruegge (October 2004) the ECRYPT NoE proposed to develop new stream ciphers with respect to two profiles :

Profile-1: Stream ciphers for software applications with high throughput needs.
Profile-2: Stream ciphers for hardware applications with restricted resources.

Main criteria are long-term security, efficiency (performance), flexibility, and market requirements. Hermes8F has been designed with respect to 8-bit wide hardware to serve both profiles and these main criteria, concentrating on clarity of design, efficiency, flexibility, and security.

In 2005 the call for new stream cipher proposals resulted in 34 different stream ciphers submissions:

ABC, Achterbahn, Crypt-MT, Decim, Dicing, Dragon, Edon80, F-FCSR, Frogbit, Grain, HC-256, Hermes8, LEX, MAG, MICKEY, Mir-1, Mosquito, NLS, Phelix, Polar Bear, Pomaranch, Py, Rabbit, Salsa20, Sfinks, Sosemanuk, SSS, Trbdk3yaea, Trivium, TSC-3, VEST, WG, Yamb, and ZK-Crypt.

Some of those are restricted by patents, some were released and known to the public already before the competition was announced (e.g. Rabbit). Two conferences followed: the SKEW 2005 in Aarhus, Denmark, and the SASC 2006 in Leuven, Belgium. At SKEW 2005 many stream cipher submissions were presented, while SASC 2006 showed cryptanalysis work and already some design tweaks. In March 2006 the selection process followed and resulted in a) 'focus' and b) secondary algorithms. All others were archived.

Software

a) Dragon-128,HC-256, LEX, Phelix, Py, Salsa20, Sosemanuk

b) ABC, CryptMT, Dicing, NLS, Polar Bear, Rabbit, Yamb

Hardware

a) Grain, Mickey-128, Phelix, Trivium

b) Achterbahn, Decim, Edon80, F-FCSR, Hermes8, LEX, Mickey, Mosquito, NLS, Polar Bear, Pomaranch, Rabbit, Salsa20, Sfinks, TSC-3, VEST, WG, Yamb, Zk-Crypt

So far – until July 2006 – no weakness of Hermes8 [21, 27] was published. However, in the performance tests (http://www.ecrypt.eu.org/stream/perf/) it appeared to be not fast enough. Therefore, the algorithm is redesigned – named Hermes8F - in order to obtain more encrypted bytes per cycle and provide a shorter initialization phase.

The next chapter and its sub-chapters describe the specification of Hermes8F, the algorithm, security properties, strength and advantages, design choices, computational efficiency in software and hardware, implementation items to avoid weaknesses, and early hardware evaluations. After the conclusions, also an outlook is given.

2 Specification of Hermes8F

2.1 Description

Hermes8F is based on the Substitution-Permutation-Network (SPN) principle [1,2,3,10]. The substitution (confusion) is performed by means of an S-BOX. The permutation and diffusion is performed by means of addressing the different state bytes, the different key bytes, and most importantly by the chaining with help of the Accu (Figure 1).

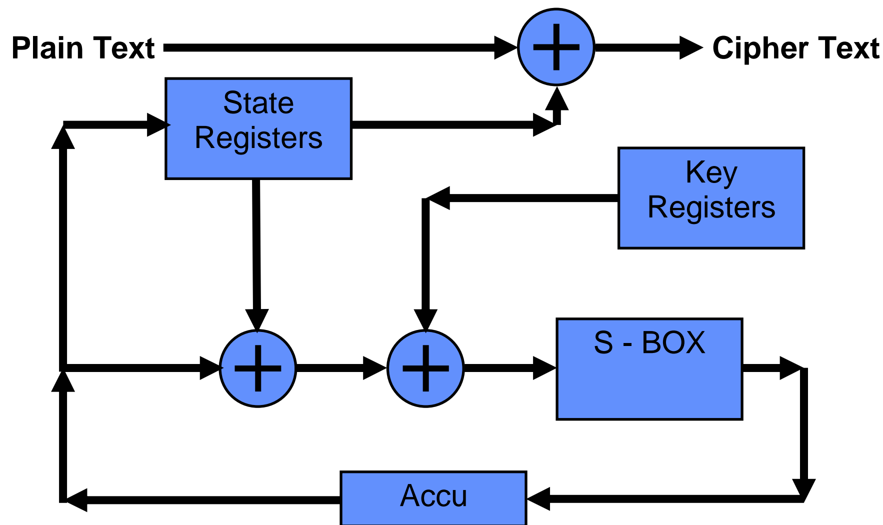

Figure 1. Principle of Hermes8F core operation round

Hermes8F-80 is based on 10 key bytes and 17 state bytes, whereas the slightly larger Hermes8F-128 contains 16 key bytes and also 17 state bytes. There are two pointers involved: p1 addresses one of the state bytes, p2 addresses one of the key bytes (Figure 2). The pointers obey modulo addition operation in order to assure that they always address valid register space.

The core operation (sub-round) consists of

1. Select a certain state byte and EXOR it with Accu,
2. Select a certain key byte and EXOR it with the previous result,
3. Take the previous result and apply the S-BOX function,
4. Store the previous result in Accu,
5. Copy Accu into the same state byte selected in step1.

The S-BOX is 8-bit wide in order to provide the non-linear Boolean function needed for substitution, i.e. confusion [8,9]. One choice is the known SBOX of AES [4,5] which is strong w.r.t. Differential Cryptanalysis (DC). – But random number based S-BOXes are also suitable, if their differential distribution table (ddt) demonstrates good quality [15,

23] with respect to DC attacks. Such random S-BOXes are especially interesting when algebraic attacks are successfully applied to AES in the future [24, 30].

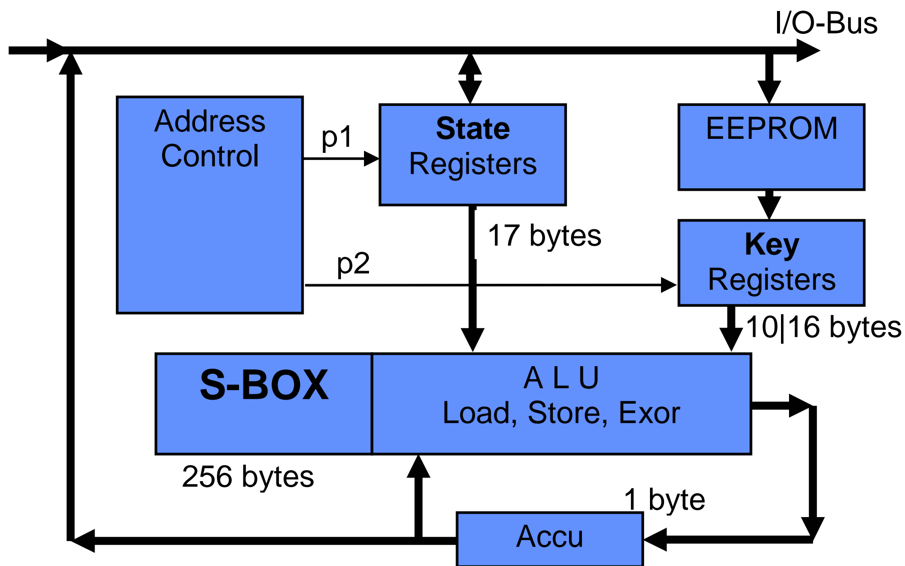

Figure 2. Byte-Architecture of Hermes8F with registers, ALU, and S-BOX

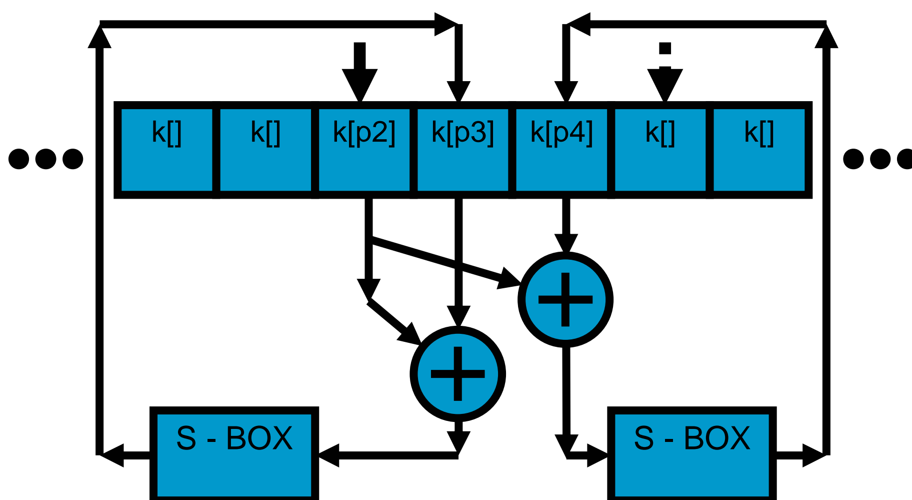

Figure 3. Key Modification and Scheduling Method of Hermes8F

In order to prevent backtracking attacks [22], the key bytes are modified every KEY_STEP3, i.e. every seven steps, during the sub-round loops depending on the position of p2. The details are shown in Figure 3 : Two temporary pointers p3 and p4

are addressing the key bytes that are following the byte addresses selected by p2. The byte k[p2] is not modified because it has to be used in the following sub-round. But the bytes k[p3] and k[p4] are 'rather old' and are therefore candidates for modification; they are replaced by SBOX[k[p3] exor k[p2]] and SBOX[k[p4] exor k[p2]] respectively. The exor'ing with k[p2] is advantageous over the direct application of the SBOX, because the inverse function of the SBOX does exist. The dashed pointer in Figure 3 represents the next p2 position (because KEY_STEP1=3) when addressing the next key byte needed for the next sub-round.

Figure 4 describes how the output bytes for the key stream ks[] are derived from the state bytes state[]. Since the pointer p1 has been incremented after the last sub-round, it points to the 'oldest' available state byte. This is the first byte to be packed into the key stream block of eight bytes for Hermes8F-80 (and eight bytes for Hermes8F-128 as well). Then further bytes follow by means of output pointer po, that is incremented by two in order to separate consecutive sub-round results from each other. This method intents to hamper state compromise attacks [22].

Since a new output block of key stream bytes does not follow earlier than the next STREAM_ROUNDS=2 are completed, the state byte contents corresponding to the same address are separated by 2 x 17 sub-rounds.

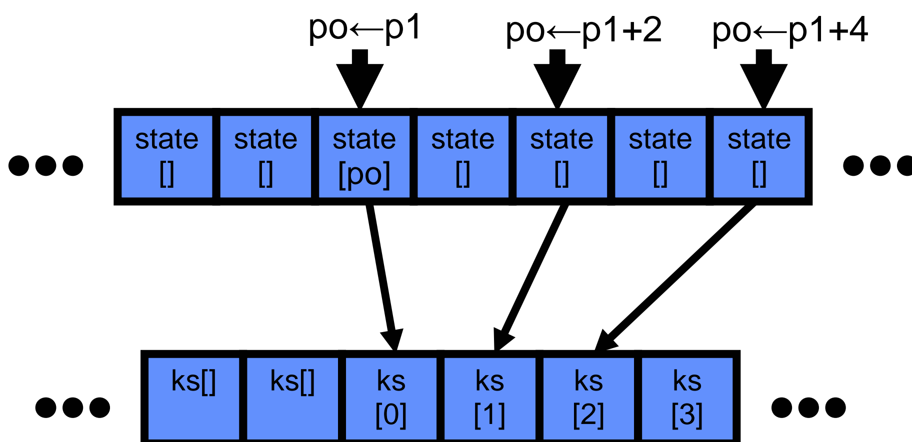

Figure 4. Output Function of Hermes8

During the 34 Hermes8F-80 sub-rounds there are nearly five occurrences of key modification, i.e. about 10 key bytes are modified per output block in relation to ten key byte registers. --- Similar holds for Hermes8F-128 sub-rounds with nearly ten key byte modifications in relation to 16 key byte registers.

A related mechanical model consists of two wheels. One has 17 teeth and needs 17 steps per round, the second one has only ten teeth, but rotates with a three-fold speed. When the first one has performed two rounds with 34 steps, the smaller one has rotated for 102 steps, i.e. about 10 turns.

2.2 Pseudo-Code of Hermes8F-80

```
1       nx ← 17
2       nk ← 10
3       OUTPUTBYTES ← 8
4
5       INIT_ROUNDS ← 5
6       STREAM_ROUNDS ← 2
7       KEY_STEP1 ← 3
8       KEY_STEP2 ← 5
9       KEY_STEP3 ← 7
10
11      k[]     ← load( nk key bytes)
12      state[] ← load( nx IV bytes )
13
14      p1     ← 0
15      p2     ← 0
16      accu   ← 0
17      src    ← 0
18      round  ← 0
19
20      for  INIT_ROUNDS  do
21      begin
22              round ← round + 1
23              /* begin of core */
24              for nx subrounds do
25              begin
26                      accu ← accu exor state[p1] exor k[p2]
27                      accu ← SBOX[ accu ]
28                      state[p1] ← accu
29                      p1 ← ( p1 + 1 ) mod nx
30                      p2 ← ( p2 + KEY_STEP1 ) mod nk
31                      src ← src + 1
32                      if( src ≥ KEY_STEP3 )
33                      then
34                        begin /* two key modifications */
35                              src ← src – KEY_STEP3
36                              p3 ← ( p2 + 1 ) mod nk
37                              p4 ← ( p3 + 1 ) mod nk
38                              k[p3] ← SBOX[ k[p3] exor k[p2] ]
39                              k[p4] ← SBOX[ k[p4] exor k[p2] ]
40                        endif
41              endfor
42              if ( round mod KEY_STEP2 equal 0) then p2 ← ( p2 + 1 ) mod nk
43              /* end of core */
44
45
46      endfor
```

```
47        /* initialization completed */
48
49        pc ← 0
50        for  MAX_ROUNDS  do
51        begin
52                 for  STREAM_ROUNDS  do
53                    round ← round + 1
54                    /* begin of core */
55                    for  nx  subrounds  do
56                    begin
57                          accu ← accu  exor  state[p1]  exor  k[p2]
58                          accu ← SBOX[ accu ]
59                          state[p1] ← accu
60                          p1 ← ( p1 + 1 ) mod nx
61                          p2 ← ( p2 + KEY_STEP1 ) mod nk
62                          src ← src + 1
63                          if( src ≥ KEY_STEP3 )
64                          then
65                            begin  /* two key modifications */
66                                  src ← src – KEY_STEP3
67                                  p3 ← ( p2 + 1 ) mod nk
68                                  p4 ← ( p3 + 1 ) mod nk
69                                  k[p3] ← SBOX[ k[p3] exor k[p2] ]
70                                  k[p4] ← SBOX[ k[p4] exor k[p2] ]
71                            endif
72                    endfor
73                    if ( round  mod  KEY_STEP2  equal 0) then  p2 ← (p2+1) mod nk
74                    /* end of core */
75                 endfor
76                 /* key stream round completed */
77
78                 po ← p1
79                 for  1  to  OUTPUTBYTES  do
80                 begin
81                          ciphertext[pc] ←  plaintext[pc]  exor  state[po]  /* encrypt */
82                          pc ←  pc + 1
83                          po ←  (po + 2)  mod nx
84                 endfor
85        endfor
```

For Hermes8F-128 only one line, i.e. line 2, is changed to nk ← 16.

Lines 14 to 47 show the initialization phase assuming the IV has already been loaded into the state registers. The cyclic pointer p2 to the key registers is incremented in steps larger than 1 in order to assign a certain key byte to every state byte over time. Additionally, the pointer p2 is also incremented after every 5th round (line 42, KEY_STEP2); this shifts the key assignment pattern, too. After every 7 sub-rounds (KEY_STEP3) two key bytes are modified by means of the S-BOX (lines 31-40).

MAX_ROUNDS (line 50) specifies how many multiples of OUTPUTBYTES bytes shall be encrypted. It is assumed that the plaintext is also a multiple of OUTPUTBYTES bytes, i.e. has been padded accordingly.

The encryption by means of the key stream bytes in the state register is shown in lines 52-84. -- During 'key streaming' the inner core of the algorithm (54-74) is the same as described for the initialization phase (23-43). The number of rounds between the output of two blocks of key-stream bytes is defined by STREAM_ROUNDS.

The complete C-code of Hermes8F is listed in appendix A of this document. -- The code of the predecessor Hermes8 and some test environment C-code for SAC tests and FIPS 140-2 tests can be found in [21].

3 Security properties, security levels, attacks

3.1 Strict Avalanche Criterion

The initialization phase has been evaluated with respect to the Strict Avalanche Criterion (SAC) [1, 10]. This has been done not only for the key sensitivity but also for the IV sensitivity. Only two rounds are needed to get very close to the 50% goal (see appendix B for the related SAC plots). If five rounds are performed during the initialization, the security level is assumed to be so high, that only exhaustive search can find the correct key or IV value from known plaintext / cipher text pairs.

Additionally, the same experiments have been performed for two other s-BOXes. One is the S-BOX called sboxAF; it is not as perfect as the AES S-BOX with respect to the differential distribution table numbers [15, 23], but delivers nearly the same SAC results (Appendix C). – The second S-BOX is a bad one, intentionally: Called sboxBAD, it contains 16 rows each with 16 ordered bytes (see appendix D). By means of this linear Boolean function it can be demonstrated that the diffusion process of the Hermes8F algorithm is of high quality; the Strict Avalanche Criterion is fulfilled after about five rounds.

3.2 Differential and Linear Cryptanalysis

The algorithm has been tested for DC and LC weakness (sensitivity, affinity, correlation) with respect to the initialization phase of five rounds. No problems were found.

Additionally, several different output streams of 512 bits each were applied to the Berlekamp-Massey algorithm. For the 2 x 4000 experiments performed, there was no exponential found below X^{242}. See appendix E for the histograms related to these 2 x 4000 experiments.

3.3 Random Number Quality tests

The algorithm has been tested for FIPS 140-2; no problems were found for 2 x 16 x 5000 experiments (see appendix F). – The algorithm was also tested by means of the Diehard test suite; no problems could be discovered for 4 x 16 experiments with either key variation or IV variation (see appendix G for p-value summaries and p-value histograms).

3.4 Some Attack Scenarios

In [22] some attacks on pseudo-random number generators (PRNG) are described: a) direct cryptanalytic attack, b) input-based attacks, c) state compromise extension attack. Since PRNGs are very similar to stream ciphers, the same attacks shall be considered here.

3.4.1 Direct Cryptanalytic Attack

Since the SAC is fulfilled quite well after only three rounds, a direct attack on five round initialization followed by the first two stream rounds (i.e. seven rounds total) seems to be unfeasible with respect to exhaustive search. - However the key stream generation is based on shorter rounds, i.e. only two. But only 8 of 17 state bytes can be directly seen in the output block pattern; the other 9 are hidden.

3.4.2 Input-Based Attacks

An adversary might use the initialization phase and the IV value for known-input, replayed-input or chosen-input attacks. However, there is a stream cipher application rule that the first IV has to be chosen as a good random number; sub-sequent IVs might be derived from that, and no (IV, key)-pair must be used twice. – In Hermes8F the IV is not used to derive any initial pointer value or similar variable. -- Since the SAC properties are strong, it is assumed that input-based attacks are not more efficient than exhaustive search.

3.4.3 State Compromise Extension Attacks

The key stream consists of consecutive blocks of 8 bytes. Two consecutive blocks are separated by 34 sub-rounds. And during these 34 steps the 10 key bytes are modified. This leads to a certain number of unknown bits, i.e. a certain complexity:

	bytes				bytes		
Version	nx	nk	state output	state distance	state unknown	key unknown	bits unknown
Hermes8F-80	17	10	8	34	26	10	288
Hermes8F-128	17	16	8	34	26	10	288

The number of unknown bytes is 36, i.e. more than twice in relation to the 16 key bytes and the TMTO requirements defined for the competition [19].

3.5 Weak Keys

Due to the method of the key scheduling all keys with equal byte pattern are weaker than randomly generated keys.

Example: If the initial key is all zero we obtain for Hermes8F-80 after the 5 initial rounds:

```
Key: 0x  5e 98 08 c5 2f 7a b7 3c 2a 78
```

and for Hermes8F-128 the related result is

```
Key: 0x fb 63 00 63 fb fb fb 63 fb 63 fb fb 63 fb 63 fb
```

The first modified key register with 10 bytes looks quite good. This is because the key register is 10 bytes and the step of pointer p2 is 3 resulting in fast overlapping (see appendix H1).
The modified key for Hermes8F-128 looks worse, because of 16 key bytes resulting in a later overlapping (see appendix H2); however, 15 of 16 key bytes are modified, 9 even twice. – Of course, one could change KEY_STEP3 from 5 to 1 for the initialization phase only, but generally the key bytes have to be produced by means of a good random number generator.

4 Design Choices, Strength and Advantages

The strength and advantages listed below are the result of the following design choices, options, and alternatives:
- The state size is more than twice as the key size, in order to prevent time-memory trade-off attacks [19];
- Substitution Permutation Network (SPN),
- Clarity of design, low complexity [20],
- Use of only registers, three pointers, EXORs, one S-BOX [8,9], small control logic,
- Constants KEY_STEP1, 2, and 3 are chosen as primes not being factors of nx or nk,
- Prevention against related key attacks [4] due to key modification/scheduling,
- Prevention against backtracking attacks [22] due to special key modification/scheduling,
- No bit-shifting, no LFSRs in order to avoid slowdown of software implementations,
- No additions, subtractions, multiplications, divisions in the core data flow,
- No constraint on IV length, beside nx as maximum,
- Low-power architecture [16],
- Scalable architecture concept (StateSize > 17 bytes, KeySize > 16 bytes).

Strength:
- one 8x8 S-BOX (e.g. AES S-BOX),
- the S-BOX is used in every sub-round [10],
- the S-BOX is used for a specialized key scheduling,
- every sub-round involves one state-byte and one key-byte,
- no key leakage, i.e. no conditional branch is dependent directly on key content,
- learned from AES [4,5,11,12].

Advantages:
- number crunching of bytes (=> fast on 8-bit micros),
- no bit-shifting ! (=> high efficiency in software),
- low complexity [20].

5 Computational efficiency

5.1 Computational efficiency in software

The following estimations are based on an 8-bit microcomputer with a two-operand instruction set and a RISC architecture. The S-BOX access is assumed to be a one cycle operation, i.e. table look-up. The **mod** operation is performed by means of conditional subtraction; this is an important software speed-up compared to full modular division.

The Key setup takes 1 cycle per byte. The setup of the primitive including the loading of the IV is described in details in appendix **J1** and results in equation (1), i.e. N1, the number of cycles for the setup, is dependent on the state size and the number of initial rounds.

$$N1 = nx + 5 + INIT_ROUNDS \cdot (3 + nx \cdot \mathbf{14} + 1/7 \cdot nx \cdot 13 + 2) \qquad \mathbf{(1)}$$

The streaming part (see appendix **J2**) results in the number N2 of cycles needed to produce one block of key stream bytes and the related block of cipher text output bytes. Equation (2) shows the dependence of N2 on the state size and OUTPUTBYTES.

$$N2 = 2 \cdot (3 + 2 + nx \cdot \mathbf{14} + 1/7 \cdot nx \cdot 13 + 2) + OUTPUTBYTES \cdot \mathbf{7} \qquad \mathbf{(2)}$$

The graph below shows the asymptotic efficiency curves (limit = 77 for $n \to \infty$); the efficiency for large amounts of data depends therefore as expected on the streaming loop performance. Some savings can be obtained by means of loop un-rolling, e.g. reducing the cycle count by OUTPUTBYTES• **2** for the encryption loop.

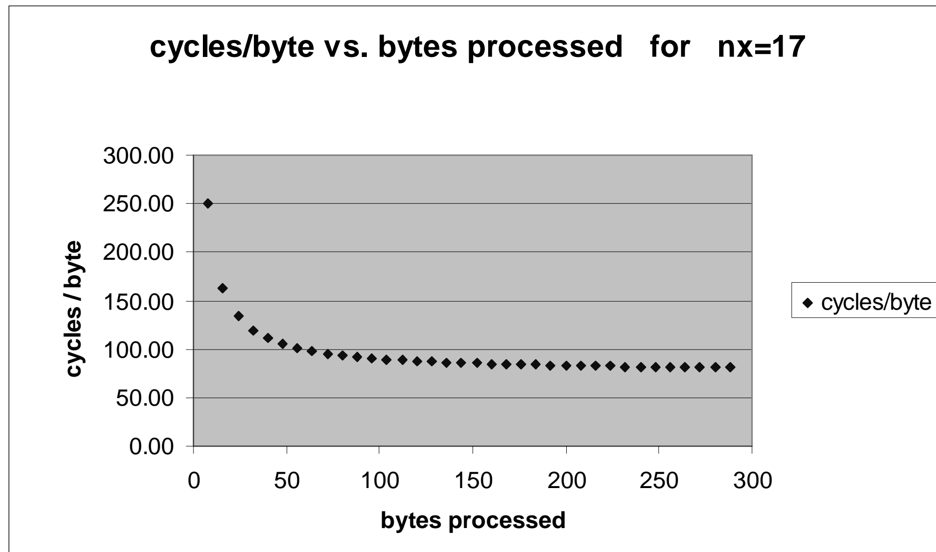

Figure 5. Cycles/byte versus bytes processed for Hermes8F

5.2 Computational efficiency in hardware

As described in the previous chapter, the key stream generation loop and the encryption loop are dominating the efficiency. In hardware, therefore, it is important to perform as many operations in parallel as possible. Since the ROM containing the S-BOX table is pre-charged with clock CLK=1 and read out with the falling edge of CLK, other operations are executed with the rising edge of CLK, e.g. update of registers and round counter. The related control logic (finite state machine, FSM) has the responsibility for the correct timing of the operations, especially the conditional modification of the key byte replacement by means of S-BOX application (line 63-71). This is described in detail in appendix **K**.

The resulting efficiency depends on the degree of parallelism reached and the amount of pipeline registers that are spent additionally. A performance of 8 bytes per 44 clock cycles seems reasonable, therefore (based on equation 3 and nx=17).

$$N3 = 2 \cdot (nx + 1/7 \cdot nx \cdot 2) \quad \text{for} \quad \text{OUTPUTBYTES} \qquad \textbf{(3)}$$

6 Implementation items to avoid weaknesses

Compared to other ciphers, the literature about side-channel attacks on stream ciphers is rare; an overview is given in [18]. For Hermes8F-80 and Hermes8F-128 the following countermeasures are proposed:

a) When the key is loaded from non-volatile memory into the key byte array, the related bus should have bus-scrambling, or 2x8 wire differential drivers, or similar DPA [13,14] protection.

b) The S-BOX should be implemented as ROM with pre-charge technique. This is favorable over the algebraic S-BOX [11,12] in GF(16) with three internal multipliers that are sensitive to products of zero.

c) The Accu should be built with 16 DFFs, so that the inverted output of the S-BOX is stored as well and DPA attacks are hampered.

d) All DFFs in the registers and Accu should be built in CSEM style [16] in order to avoid hazards and minimize DPA susceptibility.

e) The first IV must be generated by means of a TRNG, later IVs can be built by e.g. continuously incrementing the first IV [19].

7 Hardware Evaluations

An electrical Spice3 simulation was performed in an early design stage. The following hardware parts were connected for the simulation schematic:

- SBOX ROM 8 x 8 with pre-charged N-channel MOS transistor array
- Accu (8 D-FlipFlops (DFFs))
- State: one S-Register (8 DFFs)
- Eight capacitors (as replacement for the other nx-1 state registers)
- Key: one K-Register (instead of 8 multiplexers with nk inputs)
- 16 EXOR gates
- One clock driver

Based on the models of a 0.35 CMOS DLP TLM process, a current consumption of only 5uA was obtained when simulating with f=500 kHz, VCC=2V, models=typical, temperature=27°C. – However, the technology allows decreasing the VCC to the sum of one N-channel transistor threshold voltage and one P-channel transistor threshold voltage. This is especially advantageous because the power dissipation is proportional to the supply voltage squared, but only proportional to the clock frequency.

The area estimation (gate count) regarding the CMOS process mentioned above and the method of estimation in [17] is depicted below:

	0.35 CMOS	process in [17]	
Hermes8F-80	1505	3450	gates
Hermes8F-128	1712	4026	gates

The higher numbers regarding [17] are caused by the much higher gate count for the DFF compared to the 0.35 µm CSEM DFF [16], i.e. 12 instead of 4.3 !

The hardware evaluation of Hermes8 in [28] demonstrated that the algorithm is well suited for FPGA platforms supporting the byte-architecture. However, discussions [29] lead to the decision to change the initialization of the pointers, counters, and accu from

```
p1   = ( k[0] ^ k[1] ^ k[2] )  % nx ;
p2   = ( k[3] ^ k[4] ^ k[5] )  % nk ;
accu =   k[6] ^ k[7] ^ k[8]  ;
#define KEY_STEP3  7
src  = ( k[9] ^ k[0] ^ k[3] )  %  KEY_STEP3;
```

to

```
p1   = 0;
p2   = 0;
accu = 0;
src  = 0;
```

in order to avoid complicated hardware efforts with respect to the modulo function. This decision for reduction of complexity on one hand reduces the security slightly; on the other hand such a few hidden bits are targets for timing attacks and, therefore, the 'loss' seems to be negligible.

8 Conclusions

A new Stream Cipher module, Hermes8F, is presented. Following the eSTREAM competition profile rules it comes in two designs: An 80 bit key version, and a 128 bit key version. Both versions fulfill the main criteria of security, efficiency, flexibility and clarity of design. The Hermes8F design is based on a byte-architecture of low complexity and serves low-power applications such as RFID and other embedded systems. Therefore, it is suited to run efficiently on 8-bit micro computers and dedicated hardware. However, a fair comparison with other 32-bit algorithms seems to be difficult.

9 Outlook on Hermes16 and Hermes32

The algorithm principle is not only extendable w.r.t. the number of bytes for state and key, but also w.r.t. the word length of the registers. For example, an architecture with 16 bit words and two S-BOXes (resp. S-BOX calls) could be build with the same property of low complexity [15]. Especially interesting is the low-power processor MSP430 [25] in this case. – The same holds for an architecture with four S-BOXes (resp. S-BOX calls) on a 32-bit digital signal processor (DSP) such as the TMS320C2xxx or the TMS320C5xxx [26] where circular addressing is well supported. – A dedicated hardware can lead to a nearly four-fold throughput, then.

10 Acknowledgments

The author wants to thank for the kind support received from John Gordon, Vincent Rijmen, Christof Paar, Axel Poschmann, Sean Murphy, and Matt Robshaw. Special thanks go to the three anonymous reviewers of the first Hermes8 version, the organizers of the SKEW 2005 workshop, Joan Daemen for the encouraging talk about Simplistic Stream Cipher Design [20], Christophe De Canniere for great C-code support, Tim Good for fruitful discussions about hardware implementations, Matt Henricksen for pointing to an error in the pseudo-code, and Gerhard Trippen [31] for support regarding the Berlekamp-Massey algorithm. -- Also special thanks go to the ECRYPT NoE for providing this interesting challenge.

References

[1] A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997

[2] D. Stinson, Cryptography - Theory and Practice, CRC Press, 1995

[3] B. Schneier, Applied Cryptography, Wiley, 1994

[4] J. Daemen, V. Rijmen, AES Proposal: Rijndael, Version 2, 03/09/99, 45 pages and related Reference Code in C http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaelref.zip

[5] NIST FIPS 197, Advanced Encryption Standard (AES), Nov. 26, 2001, 47 pages

[6] NIST FIPS 140-2, Security Requirements for Cryptographic Modules, May 25, 2001, http://csrc.nist.gov/cryptval, http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

[7] National Institute of Standards and Technology, FIPS PUB 140-2 Annex A: Approved Security Functions, www.nist.gov/cmvp.

[8] J. Seberry, X. Zhang, Y. Zheng, Pitfalls in Designing Substitution Boxes, Crypto'94, Aug. 1994, pp 383ff

[9] J. Gordon, A. Retkin, Are Big S-Boxes Best ?, IEEE Workshop on Communication Security, Santa Barbara, Cal. 1981, pp. 1-6

[10] H. Heys, S. Tavares, Substitution-Permutation Network Resistant to Differential and Linear Cryptanalysis, Journal of Cryptology, Vol. 9, No. 1, pp.1-19, 1996

[11] J. Rejeb, V. Ramaswamy, K. Ghadiri, Hardware Implementation of the Rijndael Algorithm for High-Speed Networks, ISPC 2003, March 2003, Dallas, 6 pages

[12] H. Kuo, I. Verbauwhede, Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm, CHES 2001, LNCS 2162, pp. 51-64, Springer 2001

[13] Kocher, Jaffe, Jun, Differential Power Analysis, Advances in Cryptology, CRYPTO'99, LNCS 1666, Springer 1999, 10 pages

[14] Kocher, Evaluating Cryptosystems, 31 slides, Cryptography Research 2002, http://www.cryptography.com/resources/whitepapers/HackingCryptosystems.pdf

[15] U. Kaiser, Universal Immobilizer Crypto Engine, “UICE, the little brother of AES”, http://www.aes4.org/english/events/aes4/downloads/AES_UICE_slides.pdf

[16] C. Piguet, Design of Low-Power Libraries, ICECS 1998

[17] L. Batina, J. Lano, N. Mentens, S. B. Oers, B. Preenel, I. Verbauwhede, Energy, performance, area versus security trade-offs for stream ciphers, ECRYPT workshop, SASC – The State of the Art of Stream Ciphers, Bruegge, 14.Oct.2004, 9 pages

[18] S. Kumar, K. Lemke, C. Paar, Some Thoughts about Implementation Properties of Stream Ciphers, ECRYPT Workshop, SASC – The State of the Art of Stream Ciphers, Bruegge, 14.Oct.2004, 9 pages

[19] C. DeCanniere, J. Lano, B. Preneel, Comments on the Rediscovery of the Time Memory Data Tradeoffs, KUL, April 2005, 5 pages, http://www.ecrypt.eu.org/stream/TMD.pdf

[20] J. Daemen, Simplistic Stream Cipher Design, Workshop on Symmetric Key Encryption, SKEW 2005, 26.+27.May.2005, Aarhus, Denmark

[21] U. Kaiser, Hermes8, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/012, 2005, http://www.ecrypt.eu.org/stream

[22] J. Kelsey et al., Cryptanalytic Attacks on Pseudorandom Number Generators, Fast Software Encryption, FSE 1998, March 1998, pp.168-188

[23] U. Kaiser, UICE: A Low-Power High-Speed Cryptographic Module for RFID and Embedded Systems, Proceedings of European Conference on Circuit Theory and Design, ECCTD'05, Cork, Ireland, Aug.29-Sep.02, 2005
[24] N. Courtois, General principles of Algebraic Attacks and new Design Criteria for Cipher Components, Proceedings of AES 4, Bonn, Germany, May 2004, LNCS 3373
[25] MSP430 Data Sheets, http://www.ti.com -> Microcontrollers -> MSP430
[26] TMS320C5x User's Guide, Digital Signal Processing Products, Texas Instruments, 1993
[27] U. Kaiser, Hermes8: A Low-Complexity Low-Power Stream Cipher, http://ePrint.iacr.org/2006/019
[28] T. Good et. al, Review of stream cipher candidates from a low resource hardware perspective, Proceedings of SASC 2006, Leuven, Belgium, -- ECRYPT Stream Cipher Project, Report 2006/016, 2006, http://www.ecrypt.eu.org/stream
[29] T. Good, private communication
[30] N. Courtois, How Fast can be Algebraic Attacks on Block Ciphers ? http://ePrint.iacr.org/2006/168
[31] G. Trippen et al, COMP 685A project report Berlekamp-Massey Algorithm, http://ihome.ust.hk/~trippen/Cryptography/BM/report.pdf and the animated JavaScript program http://ihome.ust.hk/~trippen/Cryptography/BM/frameset.html

Appendix

A The C-Code of Hermes8F-80

```
/* ----------------------------------------------------------------------
 *
 * Hermes8F Algorithm  (c) 2006  Dr.-Ing. Ulrich Kaiser, Freising, Germany
 *
 * UKA 08.Nov.2005  extract for API only
 * UKA 09.Apr.2006  speedup and hardware reduction:
 *                  state nx=17      instead of 23
 *                  stream-rounds=2  instead of 3
 *                  init-rounds=5    instead of 10
 *                  initialize pointers and counters to zero
 *
 * ----------------------------------------------------------------------
 */

#include "ecrypt-sync.h"

#define K_LENGTH       10
#define X_LENGTH       17
#define O_LENGTH        8

#define INIT_ROUNDS     5
#define STREAM_ROUNDS   2

#define SBOX            S

#define KEY_STEP1       3
#define KEY_STEP2       5
#define KEY_STEP3       7

#define K_MOD(p) ((p) < K_LENGTH ? (p) : (p) - K_LENGTH)
#define X_MOD(p) ((p) < X_LENGTH ? (p) : (p) - X_LENGTH)

/* ---------- AES encryption SBOX ----------- */

static const u8 S[256] = {
  0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5,
  0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
  0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0,
  0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
  0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC,
  0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
  0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A,
  0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
  0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0,
  0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
  0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B,
  0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
  0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85,
  0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
  0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5,
  0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
  0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17,
  0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
  0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88,
  0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
```

```
  0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C,
  0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
  0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9,
  0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
  0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6,
  0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
  0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E,
  0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
  0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94,
  0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
  0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68,
  0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16};

/*
 * DESCRIPTION OF ALGORITHM
 *
 * accu is one byte
 * s-box is a table with 256 bytes, i.e. 8 bit input and 8 bit output
 *
 * A state byte and a key byte and the previous result (accu)
 * are EXORed. Then the s-box function is applied. The output is
 * fed into accu and into the oldest state byte.
 * Number of initial rounds:  5
 * Number of sub-rounds    : 17
 *
 * Confusion by:   S-Box, non-linear boolean function
 * Diffusion by:   Accu overwriting state byte
 *
 * Hermes8F-80
 *   State register x consists of  17  bytes x[16]..x[0].
 *   Key   register k consists of  10  bytes  k[9]..k[0].
 */

/* ------------------------------------------------------------------ */
/* ------------------------------------------------------------------ */
/* ------------------------------  CORE  ----------------------------- */
/* ------------------------------------------------------------------ */
/* ------------------------------------------------------------------ */

#define CORE(ROUNDS)                                                    \
  do {                                                                  \
    int m;                                                              \
                                                                        \
    for( m = 1; m <= ROUNDS; ++m, ++round )                             \
    {                                                                   \
       int p1;                                                          \
                                                                        \
       for( p1 = 0; p1 < X_LENGTH; ++p1 )                               \
         {                                                              \
           accu ^= ctx->x[p1] ^ ctx->k[p2];  /* linear operation  */    \
           accu = SBOX[accu];              /* NON-LINEAR OPERATION */   \
           ctx->x[p1] = accu;           /* overwrite state byte ! */    \
                                                                        \
           /* update the pointers */                                    \
           p2 = K_MOD(p2 + KEY_STEP1);                                  \
                                                                        \
           /* update two key bytes */                                   \
           if (++src >= KEY_STEP3)                                      \
             {                                                          \
               const int p3 = K_MOD(p2 + 1);  /* scratch */             \
               const int p4 = K_MOD(p3 + 1);  /* scratch */             \
```

```
                                                                     \
              u8 tmp;                         /* scratch */          \
                                                                     \
              tmp = ctx->k[p3] ^ ctx->k[p2];                         \
              ctx->k[p3] = SBOX[tmp];                                \
              tmp = ctx->k[p4] ^ ctx->k[p2];                         \
              ctx->k[p4] = SBOX[tmp];                                \
                                                                     \
              src -= KEY_STEP3;                                      \
            }                                                        \
        } /* for j */                                                \
                                                                     \
      /* key scheduling so that x[] sees different k[] */            \
      if(round % KEY_STEP2 == 0)                                     \
        p2 = K_MOD(p2 + 1);                                          \
                                                                     \
    } /* for m */                                                    \
  } while (0)

/* ------------------------------------------------------------------ */
/* ------------------------------------------------------------------ */
/* ---------------------------  ECRYPT  APIs  ----------------------- */
/* ------------------------------------------------------------------ */
/* ------------------------------------------------------------------ */


/**
   Empty function, must be provided for the API...
*/
void ECRYPT_init()
{
}


/**
 loads key[] into ctx->key[]
 stores key size value in record
 stores iv size value in record
 @param  ctx   ptr to record
 @param  key   ptr to array with crypto key
 @param  keysize  in bits
 @param  ivsize   in bits
*/
void ECRYPT_keysetup(
  ECRYPT_ctx* ctx,
  const u8* key,
  u32 keysize,               /* Key size in bits. */
  u32 ivsize )               /* IV size in bits. */
{
  int j;

  for (j = 0; j < K_LENGTH; ++j)
    ctx->key[j] = key[j];

  ctx->ni = ivsize / 8;

} /* ECRYPT_keysetup ---------------------------------------------------- */


/**
 loads iv[] into x[],
```

```
 runs initial rounds in order to hide IV and KEY from
 first key stream output.
 @param  ctx   ptr to record
 @param  iv    ptr to array with initial value IV
*/
void ECRYPT_ivsetup(
  ECRYPT_ctx* ctx,
  const u8* iv )
{
  const int ni = ctx->ni;  /* IV size in bytes. */

  u8 accu;
  int p1;                  /* pointer to actual state byte */
  int p2;                  /* pointer to actual key byte */
  int src;                 /* sub-round counter */
  int round = 1;           /* round counter */

  int j;

  /* load key  into  key state register */
  for (j = 0; j < K_LENGTH; ++j)
    ctx->k[j] = ctx->key[j];

/*  p1   = (ctx->k[0] ^ ctx->k[1] ^ ctx->k[2]) % X_LENGTH;
    p2   = (ctx->k[3] ^ ctx->k[4] ^ ctx->k[5]) % K_LENGTH;
    accu = (ctx->k[6] ^ ctx->k[7] ^ ctx->k[8]);
    src  = (ctx->k[9] ^ ctx->k[0] ^ ctx->k[3]) % KEY_STEP3;
 */

  p1   = 0;
  p2   = 0;
  src  = 0;
  accu = 0;

  /* fill IV into state[] and pad */
  for (j = 0; j < X_LENGTH; ++j, p1 = X_MOD(p1 + 1))
    ctx->x[j] = (p1 < ni ? iv[p1] : 0);

  /* ----------------- start of algorithm ----------------- */

  CORE(INIT_ROUNDS);

  ctx->accu    = accu;
  ctx->p2      = p2;
  ctx->src     = src;
  ctx->counter = round;

} /* ECRYPT_ivsetup -------------------------------------------------- */



/**
 Encrypts a certain number of bytes of the plaintext.
 @param  in  ctx         ptr to record
 @param  in  plaintext   ptr to array with plaintext
 @param  out ciphertext  ptr to array with ciphertext
 @param  in  msglen      number of bytes to process
*/
void ECRYPT_process_bytes(
  int action,
  ECRYPT_ctx* ctx,
```

```
  const u8* input,
  u8* output,
  u32 msglen)         /* Message length in bytes. */
{
  /* ----------------- start of algorithm ----------------- */

  u8  accu  = ctx->accu;
  int p2    = ctx->p2;     /* pointer to actual key byte */
  int src   = ctx->src;    /* sub-round counter */
  int round = ctx->counter;

  while ((int)(msglen -= O_LENGTH) >= 0)
    {
      int po;              /* output pointer */
      int j;

      CORE(STREAM_ROUNDS);

      for (j = po = 0; j < O_LENGTH; ++j, po = X_MOD(po + 2))
        output[j] = input[j] ^ ctx->x[po];

      output += O_LENGTH;
      input += O_LENGTH;
    }

  /* if msglen was not a multile of O_LENGTH, we need one more block */
  if ((msglen += O_LENGTH) > 0)
    {
      int po;              /* output pointer */
      int j;

      CORE(STREAM_ROUNDS);

      for (j = po = 0; j < msglen; ++j, po = X_MOD(po + 2))
        output[j] = input[j] ^ ctx->x[po];
    }
  else
    {
      ctx->accu    = accu;
      ctx->p2      = p2;
      ctx->src     = src;
      ctx->counter = round;
    }

} /* ECRYPT_process_bytes -------------------------------------------- */

/* ------------------------------------ end --------------------------- */
```

Excerpt from ecrypt-sync.h

```
#define ECRYPT_NAME "Hermes8F-80"    /* [edit] */
#define ECRYPT_PROFILE "HW"

#define ECRYPT_MAXKEYSIZE 80                  /* [edit] */
#define ECRYPT_KEYSIZE(i) (80 + (i)*8)        /* [edit] */

#define ECRYPT_MAXIVSIZE 136                  /* [edit]    23*8 -> 17*8 */
#define ECRYPT_IVSIZE(i) (64 + (i)*8)         /* [edit] */
```

```
/*
 * ECRYPT_ctx is the structure containing the representation of the
 * internal state of your cipher.
 */

typedef struct
{
    u8 key[10];                       /* key          */

    u8 k[10];                         /* key space, key state */
    u8 x[17];                         /* cipher state   23->17*/
    u8 ni, accu, p2, src;             /* parameters   */
   u32 counter;                       /* round counter */

} ECRYPT_ctx;
```

Testvector

```
Hermes8F evaluation
Hermes8F  plaintext setup completed.

key [0.. 9]=  0  0  0  0  0  0  0  0  0 80

iv  [0..16]=  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

Hermes8F key setup ....
Hermes8F key setup completed.

Hermes8F IV setup ....
Hermes8F IV setup completed.
accu=af  p2= 6  src= 1  counter=6
x  = 9d 49 d5 65 a0 9c 6a e1 73 b2 17 48 94 e2 b9 c1 af
k  = ef e7 ec 6d ac 7a b7 3c 2a 84

Hermes8F encryption ....
Hermes8F encryption completed.
f3  9 99 29 19  7 14 31 42 4f 93 16  c dd 94 76
f5 e7 5f  e e9 d4 a5 2f 46  8 36 47 92 47 27  7
72  1 a7 5d d9 69 b6 7a f6 c2 a5 a3 65 9d df 66
4e 67 61 cd 54 91 c0 42 47 12 fa 49 98 28  4 2e
e9 48 14 50 4d  e  6  4 77 80 8a 83 3f e5 e3 e7
47 b7 a3 59 73 ad e7 1c 74 2b  9 56 5b 7c 4a eb
50 bf  9 a1 dd e7 9e b4 87 48 a1 31 ca be 9b 5c
bf 30 33 f0 65 cb ef 19 e5 fd bf 5f 43 bc 6f 79
68 59 3c ce 66 99 e5 10 d2 c6 19 39 b8 bc f8 b8
b8 c1 92 a6 f2 33 e0 88  d ed b2 d2 b9 3c f7 7b
a1 d2 9a c8 93 ee bd 28 4b f4 fe 4f 6f 94 91 16
b1 f4 4a d7 dc 2e f1 91 46 ae dc dc af 23 c3 7c
68 7d ca  e 3c 4f d5 97 36 4b 3d d5 aa 44 c5 2a
70 94 c8 81 39 e4 94 fc b7 ee 4b d4 50  a ea 3e
fe 94 de ea 48 5a 36 50 39 31 1b d3 73 7b 89  a
40  b 8a 28 6d 86  b 6d 5a 6c 3a 95 d3 e8 b9 38
a4 8b e0 a0 2c 74 1c 47 b4 de  a 61 1a 6f 3b 9e
35 f0 65 cb 5e 3e  1 6c  3 ba c0 d7 55 95 bf 94
78 a4 f7 b9 bd 40 be fe 86 3f 71 18 be b8 1e 6d
71 56 af a9 b9 8a f1 63 f8 a1 f0  8 42 c9 3f dd
```

B Strict Avalanche Criterion (SAC) Plots with Min-Mean-Max

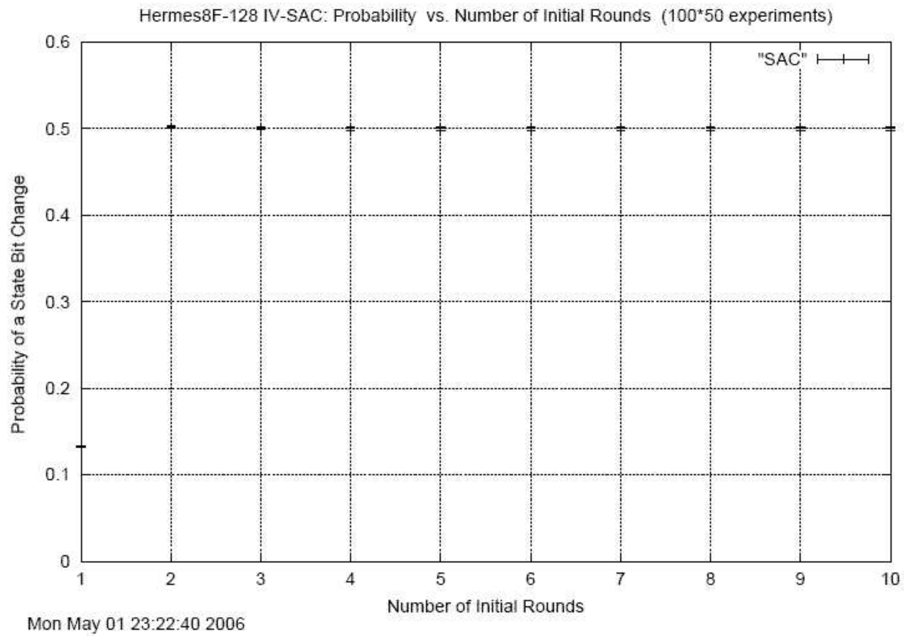

Figure B1. Strict Avalanche Criterion Test regarding IV variation for Hermes8F-128

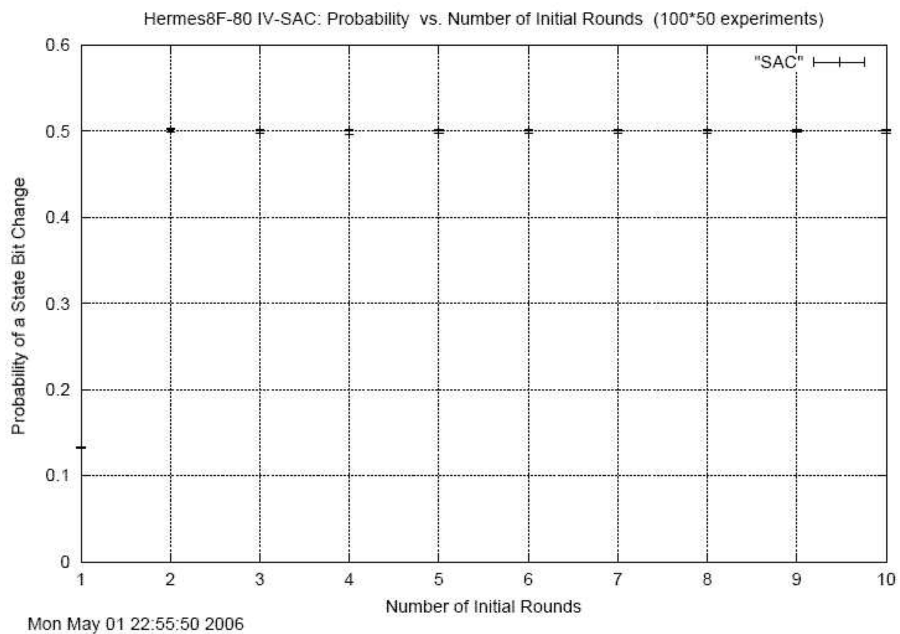

Figure B2. Strict Avalanche Criterion Test regarding IV variation for Hermes8F-80

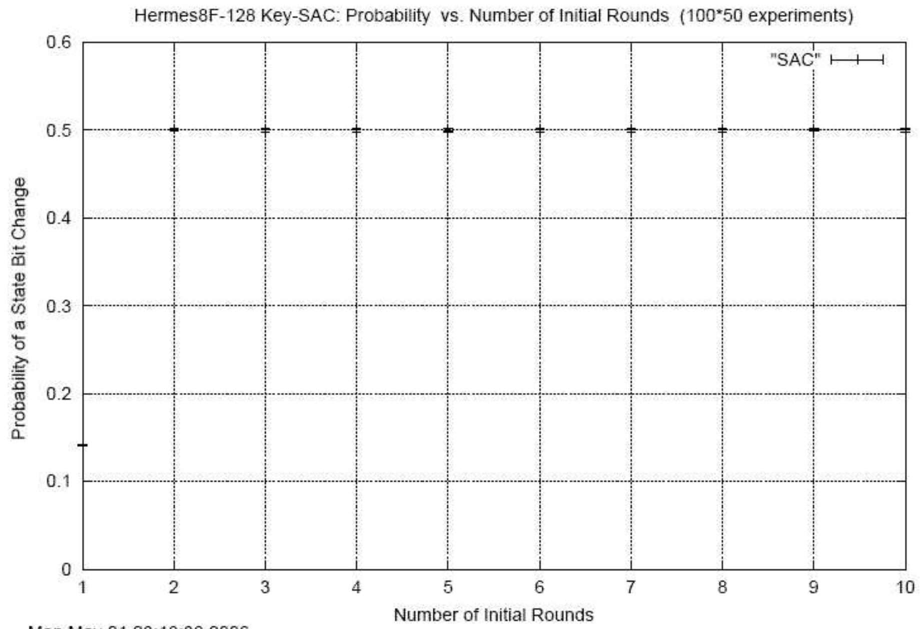

Figure B3. Strict Avalanche Criterion Test regarding key variation for Hermes8F-128

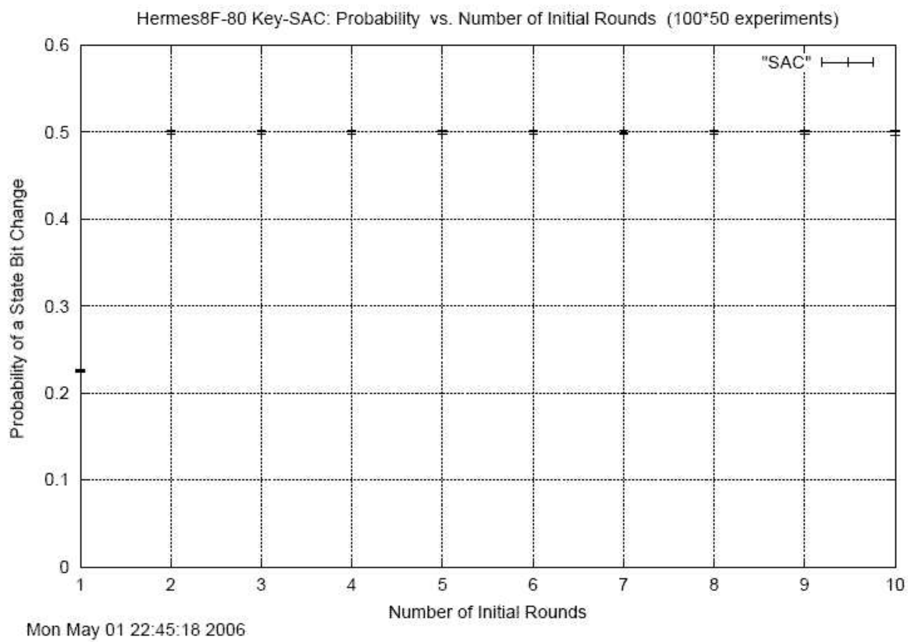

Figure B4. Strict Avalanche Criterion Test regarding key variation for Hermes8F-80

C Strict Avalanche Criterion (SAC) Plots with Min-Mean-Max Hermes8F when using a random S-BOX (sboxAF)

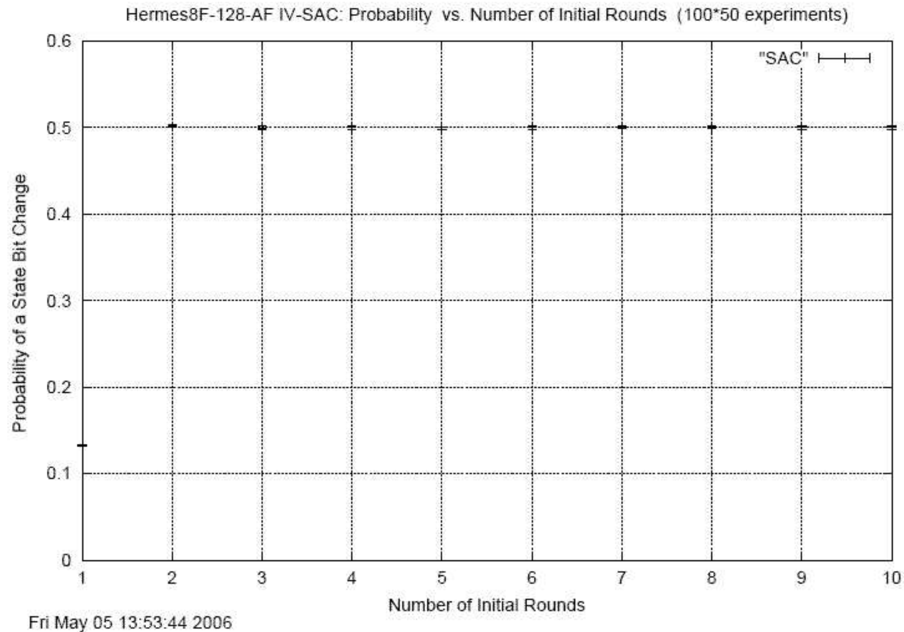

Figure C1. Strict Avalanche Criterion Test regarding IV variation for Hermes8F-128

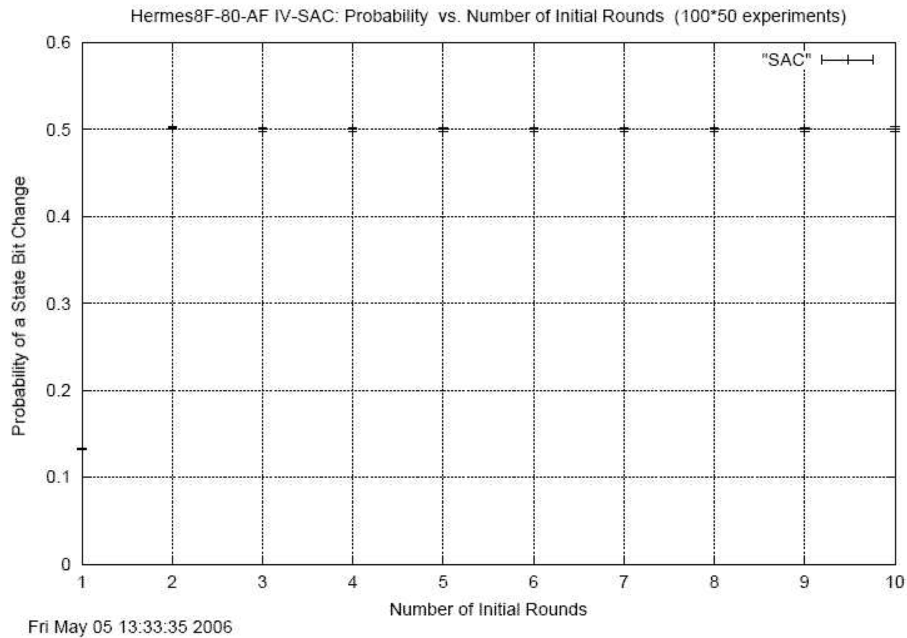

Figure C2. Strict Avalanche Criterion Test regarding IV variation for Hermes8F-80

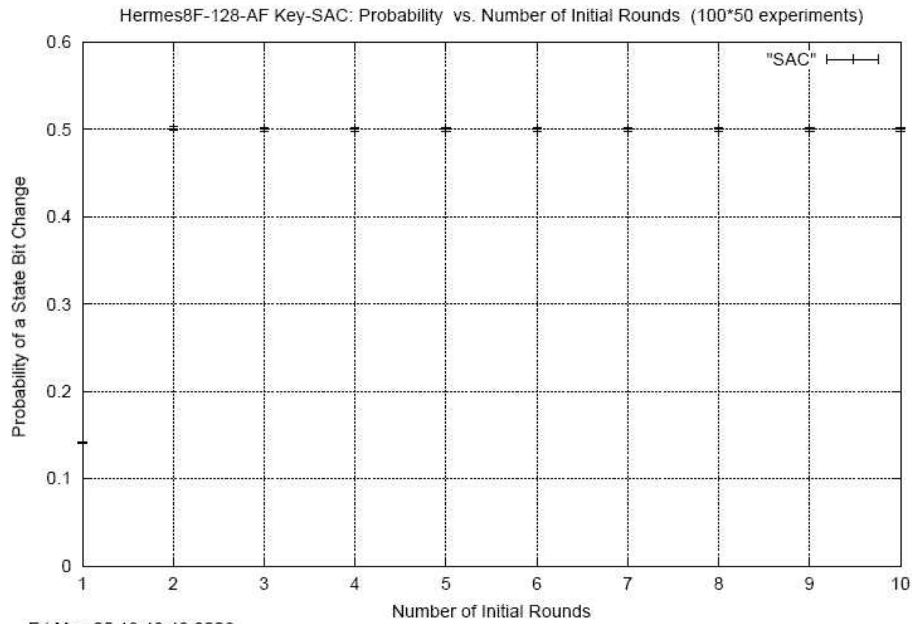

Figure C3. Strict Avalanche Criterion Test regarding key variation for Hermes8F-128

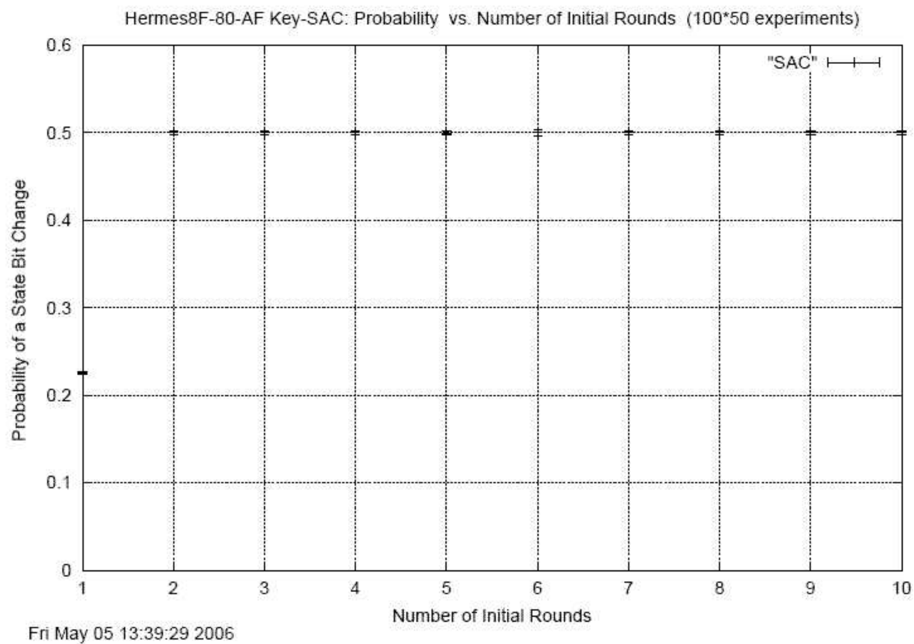

Figure C4. Strict Avalanche Criterion Test regarding key variation for Hermes8F-80

```
unsigned char sboxAF[256] =
{0xAF, 0x1B, 0xDD, 0xBC, 0x30, 0xEB, 0xF0, 0x56,
 0xC1, 0x08, 0x93, 0x36, 0x03, 0xCB, 0x81, 0x80,
 0x43, 0x2F, 0xDF, 0x2D, 0x26, 0x05, 0x0A, 0xF8,
 0x7D, 0x21, 0xE0, 0xC4, 0x06, 0xD5, 0xA6, 0xE8,
 0x8E, 0x70, 0xDC, 0xA4, 0x6D, 0x23, 0xAC, 0x18,
 0x40, 0x00, 0x64, 0x0E, 0xF6, 0x79, 0xB5, 0x1F,
 0x5D, 0x9A, 0x3B, 0xFA, 0x48, 0x5F, 0x74, 0xA1,
 0x8D, 0xD3, 0x5C, 0x4E, 0x9E, 0x14, 0x25, 0xEF,
 0xD6, 0xC9, 0x3F, 0xC5, 0xA0, 0x10, 0x50, 0xFB,
 0x31, 0xF4, 0x17, 0x88, 0xAB, 0x32, 0x76, 0x3E,
 0x15, 0x2A, 0x3D, 0xA9, 0x52, 0x20, 0xC3, 0xFC,
 0x7B, 0x49, 0x3A, 0x6E, 0xB7, 0x1D, 0xAD, 0xAA,
 0x5A, 0x0D, 0x35, 0x38, 0xC8, 0xF5, 0xF3, 0xB3,
 0x8F, 0xE6, 0x13, 0x55, 0x33, 0x8A, 0xC0, 0x67,
 0x2E, 0xE7, 0x82, 0x8C, 0x09, 0xCF, 0x1E, 0x97,
 0x28, 0x07, 0xBA, 0x4D, 0x42, 0x04, 0x73, 0x41,
 0x5B, 0xB1, 0xF9, 0xE5, 0xF7, 0x6C, 0xD8, 0x12,
 0x8B, 0x84, 0xCC, 0xB0, 0x69, 0x37, 0xAE, 0x6F,
 0xE2, 0xDB, 0x0C, 0x86, 0x29, 0x78, 0x34, 0x7C,
 0x1A, 0x85, 0x27, 0xA3, 0x9B, 0x92, 0xE3, 0xBD,
 0x59, 0x63, 0x66, 0x19, 0xCA, 0x5E, 0xFF, 0x75,
 0x72, 0x24, 0x4F, 0x47, 0x61, 0x11, 0x0B, 0xBE,
 0xA7, 0x16, 0x3C, 0xB2, 0xFD, 0x7F, 0x44, 0x99,
 0x6B, 0x98, 0x22, 0x46, 0x4A, 0x1C, 0x02, 0x6A,
 0x51, 0x39, 0x60, 0x4B, 0x57, 0x01, 0x2C, 0xE1,
 0xEE, 0x83, 0x89, 0xDA, 0x58, 0x0F, 0xBB, 0x2B,
 0xD2, 0xD4, 0x62, 0x9F, 0x90, 0x7E, 0xDE, 0xB8,
 0x4C, 0xCD, 0x68, 0xA8, 0xF2, 0x54, 0xE9, 0xE4,
 0xF1, 0xEA, 0xD7, 0x77, 0x9D, 0x96, 0xEC, 0xFE,
 0xB9, 0x91, 0xBF, 0xD1, 0xD9, 0xA2, 0x95, 0xED,
 0xA5, 0xC2, 0x7A, 0xC6, 0xC7, 0x45, 0x94, 0xB6,
 0x65, 0xD0, 0xCE, 0x9C, 0x71, 0x87, 0xB4, 0x53
};/* ddt: 19763 4917 854 106 9 2 0 */
```

D Strict Avalanche Criterion (SAC) Plots with Min-Mean-Max Hermes8F when using a bad S-BOX (sboxBad)

```
unsigned char  sboxBAD[256] = {
160, 159, 158, 157, 156, 155, 154, 153, 152, 151, 150, 149, 148, 147, 146, 145,
  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,  14,  15,  16,
129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144,
  0, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241,
 32,  31,  30,  29,  28,  27,  26,  25,  24,  23,  22,  21,  20,  19,  18,  17,
225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240,
 33,  34,  35,  36,  37,  38,  39,  40,  41,  42,  43,  44,  45,  46,  47,  48,
224, 223, 222, 221, 220, 219, 218, 217, 216, 215, 214, 213, 212, 211, 210, 209,
 64,  63,  62,  61,  60,  59,  58,  57,  56,  55,  54,  53,  52,  51,  50,  49,
193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208,
 65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,  79,  80,
192, 191, 190, 189, 188, 187, 186, 185, 184, 183, 182, 181, 180, 179, 178, 177,
 96,  95,  94,  93,  92,  91,  90,  89,  88,  87,  86,  85,  84,  83,  82,  81,
161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176,
 97,  98,  99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112,
128, 127, 126, 125, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 114, 113,
};
```

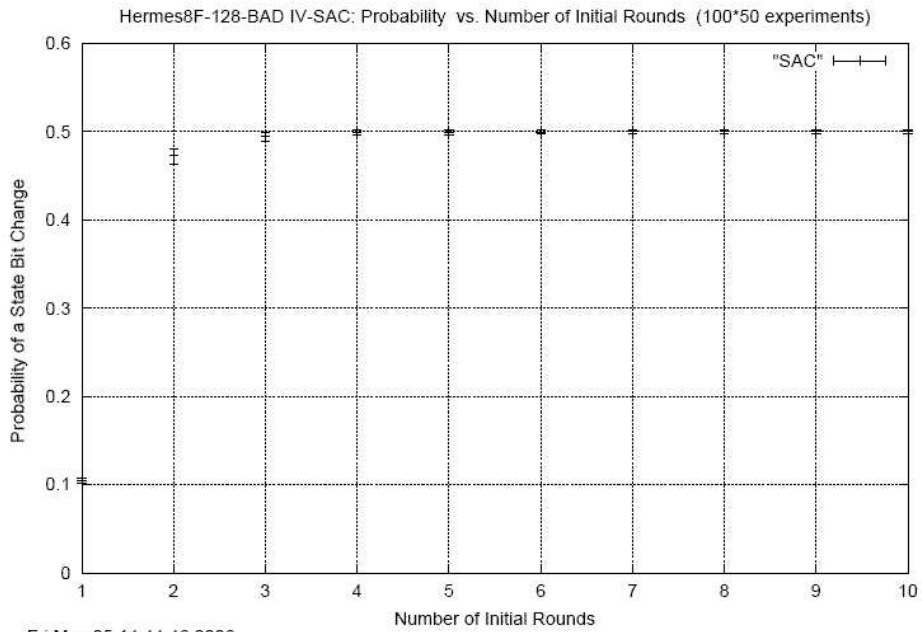

Figure D1. Strict Avalanche Criterion Test regarding IV variation for Hermes8F-128

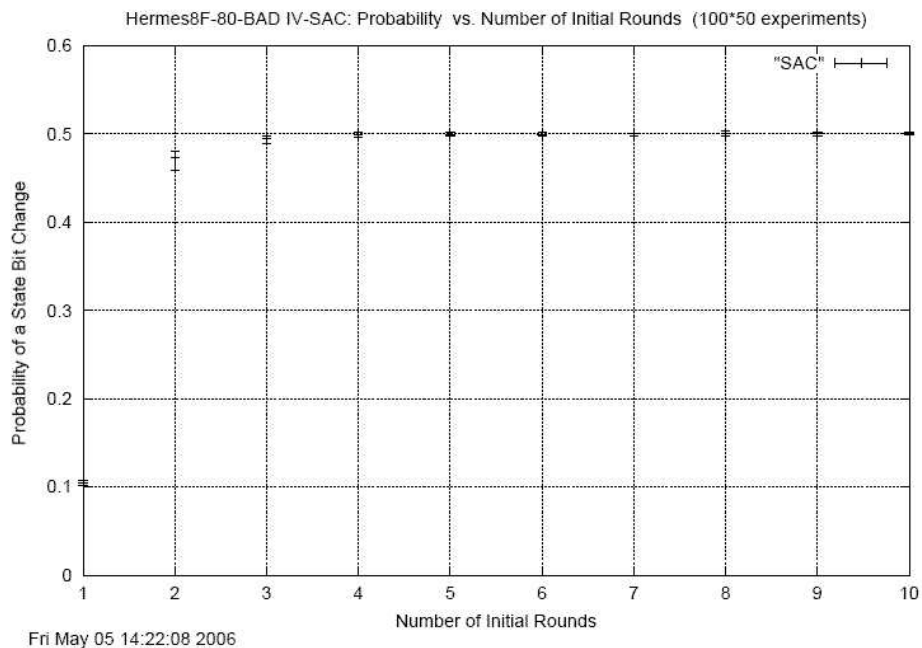

Figure D2. Strict Avalanche Criterion Test regarding IV variation for Hermes8F-80

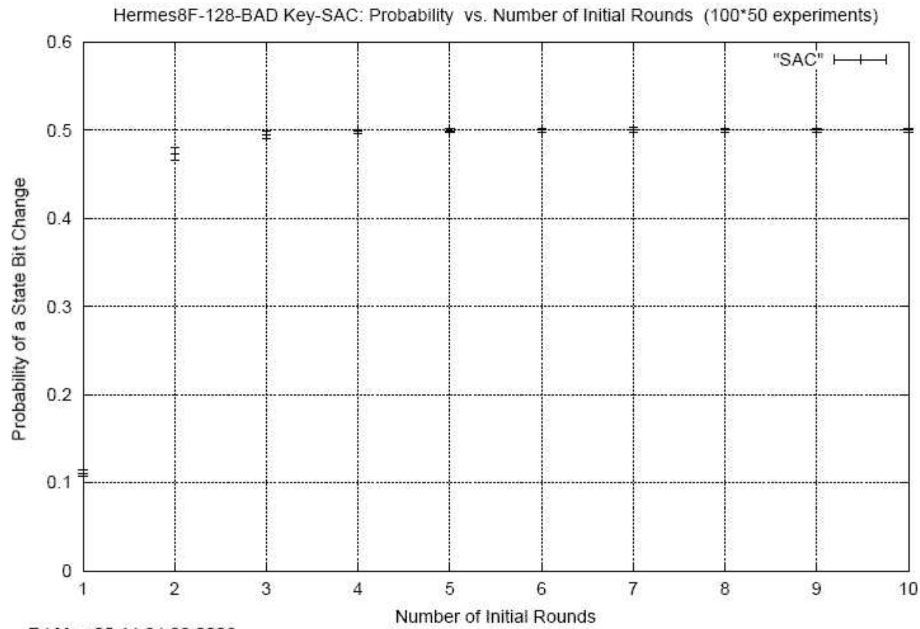

Figure D3. Strict Avalanche Criterion Test regarding key variation for Hermes8F-128

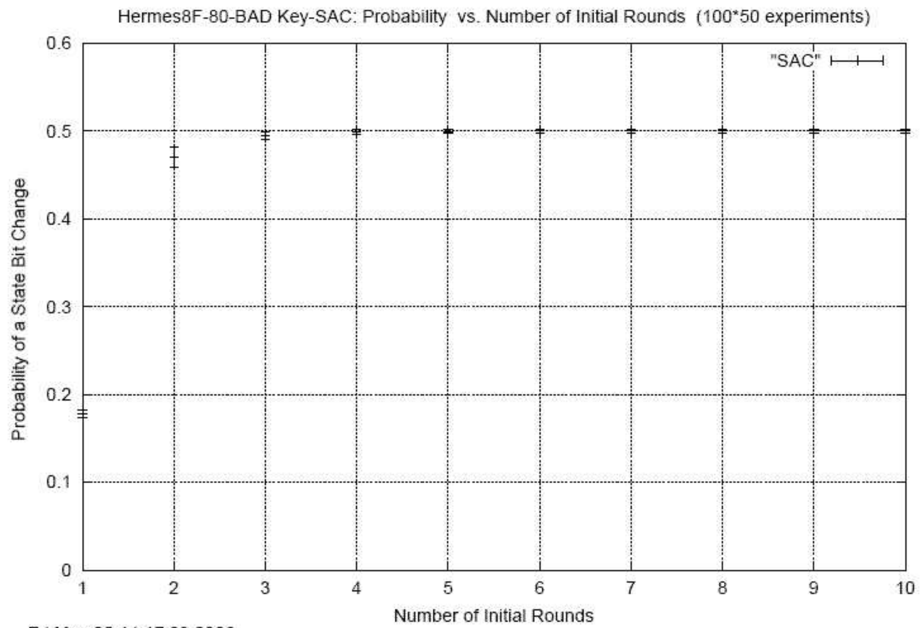

Figure D4. Strict Avalanche Criterion Test regarding key variation for Hermes8F-80

E Berlekamp-Massey Test Results

E1: Results of 16 x 250 tests with the Berlekamp-Massey algorithm over 512 key stream bits.

As expected the highest probability has a result with x^256 as highest exponential in the polynomial.

```
Hermes8F_80
===========

208 lines processed.
4000 numbers processed.

238    0
239    0
240    0
241    0
242    1
243    0
244    1
245    1
246    1
247    3
248    7
249    8
250   22  *
251   43  **
252  116  *****
253  204  **********
254  384  *******************
255  794  ***************************************
256 1476  *************************************************************************
257  711  ***********************************
258  170  ********
259   42  **
260   11
261    3
262    1
263    1
264    0
265    0
266    0
267    0
268    0
```

E2: Results of 16 x 250 tests with the Berlekamp-Massey algorithm over 512 key stream bits.

As expected the highest probability has a result with x^256 as highest exponential in the polynomial.

```
Hermes8F_128
============

208 lines processed.
4000 numbers processed.

238    0
239    0
240    0
241    0
242    1
243    0
244    0
245    2
246    2
247    5
248    6
249    6
250   23  *
251   40  *
252  102  *****
253  191  *********
254  415  ********************
255  707  ***********************************
256 1504  ***************************************************************************
257  762  **************************************
258  192  *********
259   30  *
260   10
261    2
262    0
263    0
264    0
265    0
266    0
267    0
268    0
```

F Results of FIPS 140-2 Tests

```
01.May.2006 16 x 5000 runs of FIPS140-2 test of Hermes8F-80
============================================================

 Number of FAILs : 6   of  total test 5000
 Number of FAILs : 4   of  total test 5000
 Number of FAILs : 5   of  total test 5000
 Number of FAILs : 4   of  total test 5000
 Number of FAILs : 2   of  total test 5000
 Number of FAILs : 7   of  total test 5000
 Number of FAILs : 7   of  total test 5000
 Number of FAILs : 10   of  total test 5000
 Number of FAILs : 5   of  total test 5000
 Number of FAILs : 4   of  total test 5000
 Number of FAILs : 4   of  total test 5000
 Number of FAILs : 0   of  total test 5000
 Number of FAILs : 2   of  total test 5000
```

```
 Number of FAILs : 2   of  total test 5000
 Number of FAILs : 3   of  total test 5000
 Number of FAILs : 3   of  total test 5000

summary:

long runs test FAILED   31
runs test FAILED        24
poker test FAILED        7
monobit test FAILED      7



02.May.2006   16 x 5000 fips140-2 test runs of Hermes8F-128
============================================================

 Number of FAILs : 3   of  total test 5000
 Number of FAILs : 4   of  total test 5000
 Number of FAILs : 8   of  total test 5000
 Number of FAILs : 0   of  total test 5000
 Number of FAILs : 2   of  total test 5000
 Number of FAILs : 2   of  total test 5000
 Number of FAILs : 4   of  total test 5000
 Number of FAILs : 7   of  total test 5000
 Number of FAILs : 7   of  total test 5000
 Number of FAILs : 5   of  total test 5000
 Number of FAILs : 5   of  total test 5000
 Number of FAILs : 6   of  total test 5000
 Number of FAILs : 2   of  total test 5000
 Number of FAILs : 0   of  total test 5000
 Number of FAILs : 3   of  total test 5000
 Number of FAILs : 3   of  total test 5000

summary:

long runs test FAILED   21
runs test FAILED        23
poker test FAILED        8
monobit test FAILED      9
```

G Results of DIEHARD Tests

This random number test evaluates the key streams with 268MB size. Each Hermes version is tested 16 times with key varied and fixed IV first, then with varied IV and fixed key. This results in 64 tests total.

```
diehard_hermes8F_80
===================
X=17(0x00), K=10(0x00), ..... K=10(0xFF)

uka001@linux:~/Cryptography/diehard/diehard> ./evaluate
diehard_hermes8F_80
Overall p-value after applying KStest on 236 p-values = 0.000413
Overall p-value after applying KStest on 236 p-values = 0.873721
Overall p-value after applying KStest on 236 p-values = 0.441866
Overall p-value after applying KStest on 236 p-values = 0.240540
Overall p-value after applying KStest on 236 p-values = 0.819852
Overall p-value after applying KStest on 236 p-values = 0.579611
Overall p-value after applying KStest on 236 p-values = 0.225407
Overall p-value after applying KStest on 236 p-values = 0.224848
Overall p-value after applying KStest on 236 p-values = 0.128240
Overall p-value after applying KStest on 236 p-values = 0.531327
Overall p-value after applying KStest on 236 p-values = 0.014833
Overall p-value after applying KStest on 236 p-values = 0.236369
Overall p-value after applying KStest on 236 p-values = 0.534825
Overall p-value after applying KStest on 236 p-values = 0.986252
Overall p-value after applying KStest on 236 p-values = 0.101074
Overall p-value after applying KStest on 236 p-values = 0.886045

  0 102
  1 160
  2 151
  3 157
  4 140
  5 145
  6 138
  7 161
  8 170
  9 162
 10 123
 11 130
 12 150
 13 162
 14 154
 15 163
 16 163
 17 148
 18 137
 19 165
 20 141
 21 146
 22 144
 23 143
 24 139
numbers processed: 3776
numbers processed per DIEHARD: 236
done.
Sun Jun 25 23:28:09 CEST 2006
```

```
diehard_hermes8F_80x
====================
X=17(0x00), .... 17(0xFF);    K=10(0x00)

uka001@linux:~/Cryptography/diehard/diehard> ./evaluate
diehard_hermes8F_80x
Overall p-value after applying KStest on 236 p-values = 0.246815
Overall p-value after applying KStest on 236 p-values = 0.589293
Overall p-value after applying KStest on 236 p-values = 0.788972
Overall p-value after applying KStest on 236 p-values = 0.567660
Overall p-value after applying KStest on 236 p-values = 0.894311
Overall p-value after applying KStest on 236 p-values = 0.103009
Overall p-value after applying KStest on 236 p-values = 0.930104
Overall p-value after applying KStest on 236 p-values = 0.336186
Overall p-value after applying KStest on 236 p-values = 0.614726
Overall p-value after applying KStest on 236 p-values = 0.613600
Overall p-value after applying KStest on 236 p-values = 0.705532
Overall p-value after applying KStest on 236 p-values = 0.571806
Overall p-value after applying KStest on 236 p-values = 0.317440
Overall p-value after applying KStest on 236 p-values = 0.131707
Overall p-value after applying KStest on 236 p-values = 0.049888
Overall p-value after applying KStest on 236 p-values = 0.886045

  0  94
  1 151
  2 173
  3 183
  4 138
  5 153
  6 144
  7 150
  8 139
  9 151
 10 123
 11 159
 12 157
 13 167
 14 127
 15 140
 16 136
 17 134
 18 165
 19 158
 20 147
 21 119
 22 164
 23 163
 24 163
numbers processed: 3776
numbers processed per DIEHARD: 236
done.
Sun Jun 25 23:30:16 CEST 2006



diehard_hermes8F_128
====================
X=17(0x00), K=16(0x00), ..... K=16(0xFF)

uka001@linux:~/Cryptography/diehard/diehard> ./evaluate
```

```
diehard_hermes8F_128
Overall p-value after applying KStest on 236 p-values = 0.355829
Overall p-value after applying KStest on 236 p-values = 0.351261
Overall p-value after applying KStest on 236 p-values = 0.662788
Overall p-value after applying KStest on 236 p-values = 0.881873
Overall p-value after applying KStest on 236 p-values = 0.390687
Overall p-value after applying KStest on 236 p-values = 0.694212
Overall p-value after applying KStest on 236 p-values = 0.968508
Overall p-value after applying KStest on 236 p-values = 0.254588
Overall p-value after applying KStest on 236 p-values = 0.395196
Overall p-value after applying KStest on 236 p-values = 0.028024
Overall p-value after applying KStest on 236 p-values = 0.039845
Overall p-value after applying KStest on 236 p-values = 0.529176
Overall p-value after applying KStest on 236 p-values = 0.612554
Overall p-value after applying KStest on 236 p-values = 0.836791
Overall p-value after applying KStest on 236 p-values = 0.012220
Overall p-value after applying KStest on 236 p-values = 0.133250

  0 100
  1 146
  2 167
  3 147
  4 146
  5 157
  6 139
  7 153
  8 148
  9 155
 10 146
 11 151
 12 164
 13 149
 14 164
 15 135
 16 170
 17 133
 18 133
 19 158
 20 163
 21 151
 22 143
 23 137
 24 146
numbers processed: 3776
numbers processed per DIEHARD: 236
done.
Sun Jun 25 23:30:44 CEST 2006
```

```
diehard_hermes8F_128x
=====================
X=17(0x00), .... 17(0xFF);    K=16(0x00)

uka001@linux:~/Cryptography/diehard/diehard> ./evaluate
diehard_hermes8F_128x
Overall p-value after applying KStest on 236 p-values = 0.912513
Overall p-value after applying KStest on 236 p-values = 0.603459
Overall p-value after applying KStest on 236 p-values = 0.336733
Overall p-value after applying KStest on 236 p-values = 0.989924
```

```
Overall p-value after applying KStest on 236 p-values = 0.343231
Overall p-value after applying KStest on 236 p-values = 0.041618
Overall p-value after applying KStest on 236 p-values = 0.832506
Overall p-value after applying KStest on 236 p-values = 0.096922
Overall p-value after applying KStest on 236 p-values = 0.239141
Overall p-value after applying KStest on 236 p-values = 0.207043
Overall p-value after applying KStest on 236 p-values = 0.292541
Overall p-value after applying KStest on 236 p-values = 0.951282
Overall p-value after applying KStest on 236 p-values = 0.815580
Overall p-value after applying KStest on 236 p-values = 0.225669
Overall p-value after applying KStest on 236 p-values = 0.182395
Overall p-value after applying KStest on 236 p-values = 0.133250

  0  86
  1 145
  2 162
  3 150
  4 146
  5 142
  6 148
  7 148
  8 132
  9 173
 10 157
 11 156
 12 156
 13 141
 14 144
 15 145
 16 156
 17 159
 18 152
 19 131
 20 163
 21 154
 22 164
 23 148
 24 145
numbers processed: 3776
numbers processed per DIEHARD: 236
done.
Sun Jun 25 23:31:26 CEST 2006
```

H Weak Keys

H1 Evaluation of Hermes8F-80 with Key=IV=0

The initialization phase is started with IV= 17 x 0x00 and key = 10 x 0x00. Then the ‘crunch’ command runs the five rounds of 17 sub-rounds each. It is no surprise that the first accu result is 0x63, because this is the S-BOX output S(0x00). It is also no surprise that the first key modification results in two times 0x63.

After seven sub-rounds the next key modification is performed. Since k[2] and k[3] are equal, the k[3] is updated to 0x63 again. The key byte k[4] changes from 0x00 to 0xfb because k[2] is 0x63.

After twelve key modification steps the resulting key registers looks quite random.

```
Command chosen: c
```

```
CORE!  accu=0x63 p1=1 p2=3 src=1  round=1
CORE!  accu=0xfb p1=2 p2=6 src=2  round=1
CORE!  accu=0x f p1=3 p2=9 src=3  round=1
CORE!  accu=0x76 p1=4 p2=2 src=4  round=1
CORE!  accu=0x38 p1=5 p2=5 src=5  round=1
CORE!  accu=0x 7 p1=6 p2=8 src=6  round=1
CORE-K   k[ 2]= 0x63
CORE-K   k[ 3]= 0x63
CORE!  accu=0xc5 p1=7 p2=1 src=0  round=1
CORE!  accu=0xa6 p1=8 p2=4 src=1  round=1
CORE!  accu=0x24 p1=9 p2=7 src=2  round=1
CORE!  accu=0x36 p1=10 p2=0 src=3  round=1
CORE!  accu=0x 5 p1=11 p2=3 src=4  round=1
CORE!  accu=0x33 p1=12 p2=6 src=5  round=1
CORE!  accu=0xc3 p1=13 p2=9 src=6  round=1
CORE-K   k[ 3]= 0x63
CORE-K   k[ 4]= 0xfb
CORE!  accu=0x2e p1=14 p2=2 src=0  round=1
CORE!  accu=0xe3 p1=15 p2=5 src=1  round=1
CORE!  accu=0x11 p1=16 p2=8 src=2  round=1
CORE!  accu=0x82 p1=0 p2=1 src=3  round=1
CORE!  accu=0xf8 p1=1 p2=4 src=4  round=2
CORE!  accu=0x41 p1=2 p2=7 src=5  round=2
CORE!  accu=0x2f p1=3 p2=0 src=6  round=2
CORE-K   k[ 4]= 0x46
CORE-K   k[ 5]= 0xfb
CORE!  accu=0xcb p1=4 p2=3 src=0  round=2
CORE!  accu=0x60 p1=5 p2=6 src=1  round=2
CORE!  accu=0x85 p1=6 p2=9 src=2  round=2
CORE!  accu=0x 9 p1=7 p2=2 src=3  round=2
CORE!  accu=0x4b p1=8 p2=5 src=4  round=2
CORE!  accu=0x22 p1=9 p2=8 src=5  round=2
CORE!  accu=0xfa p1=10 p2=1 src=6  round=2
CORE-K   k[ 5]= 0x7a
CORE-K   k[ 6]= 0x5a
CORE!  accu=0x16 p1=11 p2=4 src=0  round=2
CORE!  accu=0xfb p1=12 p2=7 src=1  round=2
CORE!  accu=0x 7 p1=13 p2=0 src=2  round=2
CORE!  accu=0xa5 p1=14 p2=3 src=3  round=2
CORE!  accu=0x3f p1=15 p2=6 src=4  round=2
CORE!  accu=0x92 p1=16 p2=9 src=5  round=2
CORE!  accu=0xca p1=0 p2=2 src=6  round=2
CORE-K   k[ 6]= 0xb7
CORE-K   k[ 7]= 0xda
CORE!  accu=0xd1 p1=1 p2=5 src=0  round=3
CORE!  accu=0x87 p1=2 p2=8 src=1  round=3
CORE!  accu=0xc2 p1=3 p2=1 src=2  round=3
CORE!  accu=0x 1 p1=4 p2=4 src=3  round=3
CORE!  accu=0xcc p1=5 p2=7 src=4  round=3
CORE!  accu=0xdc p1=6 p2=0 src=5  round=3
CORE!  accu=0x 3 p1=7 p2=3 src=6  round=3
CORE-K   k[ 7]= 0x3c
CORE-K   k[ 8]= 0xa9
CORE!  accu=0xf1 p1=8 p2=6 src=0  round=3
CORE!  accu=0x43 p1=9 p2=9 src=1  round=3
CORE!  accu=0x56 p1=10 p2=2 src=2  round=3
CORE!  accu=0x26 p1=11 p2=5 src=3  round=3
CORE!  accu=0x5c p1=12 p2=8 src=4  round=3
CORE!  accu=0x89 p1=13 p2=1 src=5  round=3
CORE!  accu=0x71 p1=14 p2=4 src=6  round=3
CORE-K   k[ 8]= 0x2a
```

```
CORE-K   k[ 9]= 0xeb
CORE!  accu=0x30 p1=15 p2=7 src=0  round=3
CORE!  accu=0x b p1=16 p2=0 src=1  round=3
CORE!  accu=0x78 p1=0 p2=3 src=2  round=3
CORE!  accu=0x74 p1=1 p2=6 src=3  round=4
CORE!  accu=0x1b p1=2 p2=9 src=4  round=4
CORE!  accu=0x23 p1=3 p2=2 src=5  round=4
CORE!  accu=0x83 p1=4 p2=5 src=6  round=4
CORE-K   k[ 9]= 0x78
CORE-K   k[ 0]= 0xe5
CORE!  accu=0x96 p1=5 p2=8 src=0  round=4
CORE!  accu=0xd0 p1=6 p2=1 src=1  round=4
CORE!  accu=0x66 p1=7 p2=4 src=2  round=4
CORE!  accu=0x3e p1=8 p2=7 src=3  round=4
CORE!  accu=0x83 p1=9 p2=0 src=4  round=4
CORE!  accu=0x 4 p1=10 p2=3 src=5  round=4
CORE!  accu=0x83 p1=11 p2=6 src=6  round=4
CORE-K   k[ 0]= 0x5e
CORE-K   k[ 1]= 0xbc
CORE!  accu=0x45 p1=12 p2=9 src=0  round=4
CORE!  accu=0x8d p1=13 p2=2 src=1  round=4
CORE!  accu=0xdb p1=14 p2=5 src=2  round=4
CORE!  accu=0x81 p1=15 p2=8 src=3  round=4
CORE!  accu=0xe0 p1=16 p2=1 src=4  round=4
CORE!  accu=0x36 p1=0 p2=4 src=5  round=4
CORE!  accu=0xf2 p1=1 p2=7 src=6  round=5
CORE-K   k[ 1]= 0x98
CORE-K   k[ 2]= 0x27
CORE!  accu=0x 3 p1=2 p2=0 src=0  round=5
CORE!  accu=0xf3 p1=3 p2=3 src=1  round=5
CORE!  accu=0x7d p1=4 p2=6 src=2  round=5
CORE!  accu=0x4a p1=5 p2=9 src=3  round=5
CORE!  accu=0x98 p1=6 p2=2 src=4  round=5
CORE!  accu=0x35 p1=7 p2=5 src=5  round=5
CORE!  accu=0xa3 p1=8 p2=8 src=6  round=5
CORE-K   k[ 2]= 0x 8
CORE-K   k[ 3]= 0x f
CORE!  accu=0x67 p1=9 p2=1 src=0  round=5
CORE!  accu=0x f p1=10 p2=4 src=1  round=5
CORE!  accu=0x74 p1=11 p2=7 src=2  round=5
CORE!  accu=0xd7 p1=12 p2=0 src=3  round=5
CORE!  accu=0xf2 p1=13 p2=3 src=4  round=5
CORE!  accu=0xf7 p1=14 p2=6 src=5  round=5
CORE!  accu=0x78 p1=15 p2=9 src=6  round=5
CORE-K   k[ 3]= 0xc5
CORE-K   k[ 4]= 0x2f
CORE!  accu=0xe1 p1=16 p2=2 src=0  round=5
CORE!  accu=0x9e p1=0 p2=5 src=1  round=5

Key: 0x
 5e 98 08 c5 2f 7a b7 3c 2a 78

State: 0x
 f2 03 f3 7d 4a 98 35 a3 67 0f 74 d7 f2 f7 78 e1 9e
```

H2 Evaluation of Hermes8F-128 with Key=IV=0

The initialization phase is started with IV= 17 x 0x00 and key = 16 x 0x00. Then the ‘crunch’ command runs the five rounds of 17 sub-rounds each. It is no surprise that the

first accu result is 0x63, because this is the S-BOX output S(0x00).
It is also no surprise that the first key modifications result each in two times 0x63. This is because the key register is 16 bytes (and not 10 bytes).

```
Command chosen: c
CORE!  accu=0x63 p1=1 p2=3 src=1  round=1

CORE!  accu=0xfb p1=2 p2=6 src=2  round=1
CORE!  accu=0x f p1=3 p2=9 src=3  round=1
CORE!  accu=0x76 p1=4 p2=12 src=4  round=1
CORE!  accu=0x38 p1=5 p2=15 src=5  round=1
CORE!  accu=0x 7 p1=6 p2=2 src=6  round=1
CORE-K   k[ 6]= 0x63
CORE-K   k[ 7]= 0x63
CORE!  accu=0xc5 p1=7 p2=5 src=0  round=1
CORE!  accu=0xa6 p1=8 p2=8 src=1  round=1
CORE!  accu=0x24 p1=9 p2=11 src=2  round=1
CORE!  accu=0x36 p1=10 p2=14 src=3  round=1
CORE!  accu=0x 5 p1=11 p2=1 src=4  round=1
CORE!  accu=0x6b p1=12 p2=4 src=5  round=1
CORE!  accu=0x7f p1=13 p2=7 src=6  round=1
CORE-K   k[11]= 0x63
CORE-K   k[12]= 0x63
CORE!  accu=0x9c p1=14 p2=10 src=0  round=1
CORE!  accu=0xde p1=15 p2=13 src=1  round=1
CORE!  accu=0x1d p1=16 p2=0 src=2  round=1
CORE!  accu=0xa4 p1=0 p2=3 src=3  round=1
CORE!  accu=0xc6 p1=1 p2=6 src=4  round=2
CORE!  accu=0x58 p1=2 p2=9 src=5  round=2
CORE!  accu=0x5b p1=3 p2=12 src=6  round=2
CORE-K   k[ 0]= 0x63
CORE-K   k[ 1]= 0x63
CORE!  accu=0x2f p1=4 p2=15 src=0  round=2
CORE!  accu=0xf0 p1=5 p2=2 src=1  round=2
CORE!  accu=0x68 p1=6 p2=5 src=2  round=2
CORE!  accu=0x95 p1=7 p2=8 src=3  round=2
CORE!  accu=0xc3 p1=8 p2=11 src=4  round=2
CORE!  accu=0x5f p1=9 p2=14 src=5  round=2
CORE!  accu=0xf9 p1=10 p2=1 src=6  round=2
CORE-K   k[ 5]= 0x63
CORE-K   k[ 6]= 0xfb
CORE!  accu=0xdb p1=11 p2=4 src=0  round=2
CORE!  accu=0xe7 p1=12 p2=7 src=1  round=2
CORE!  accu=0x f p1=13 p2=10 src=2  round=2
CORE!  accu=0xdc p1=14 p2=13 src=3  round=2
CORE!  accu=0x77 p1=15 p2=0 src=4  round=2
CORE!  accu=0x 1 p1=16 p2=3 src=5  round=2
CORE!  accu=0x 6 p1=0 p2=6 src=6  round=2
CORE-K   k[10]= 0x63
CORE-K   k[11]= 0xfb
CORE!  accu=0xe2 p1=1 p2=9 src=0  round=3
CORE!  accu=0xf4 p1=2 p2=12 src=1  round=3
CORE!  accu=0x4b p1=3 p2=15 src=2  round=3
CORE!  accu=0x43 p1=4 p2=2 src=3  round=3
CORE!  accu=0x6d p1=5 p2=5 src=4  round=3
CORE!  accu=0x33 p1=6 p2=8 src=5  round=3
CORE!  accu=0x24 p1=7 p2=11 src=6  round=3
CORE-K   k[15]= 0x63
CORE-K   k[ 0]= 0xfb
CORE!  accu=0x9c p1=8 p2=14 src=0  round=3
```

```
CORE!  accu=0x2e p1=9 p2=1 src=1  round=3
CORE!  accu=0x8d p1=10 p2=4 src=2  round=3
CORE!  accu=0xb1 p1=11 p2=7 src=3  round=3
CORE!  accu=0x96 p1=12 p2=10 src=4  round=3
CORE!  accu=0x2d p1=13 p2=13 src=5  round=3
CORE!  accu=0xa1 p1=14 p2=0 src=6  round=3
CORE-K   k[ 4]= 0x63
CORE-K   k[ 5]= 0xfb
CORE!  accu=0xd8 p1=15 p2=3 src=0  round=3
CORE!  accu=0x35 p1=16 p2=6 src=1  round=3
CORE!  accu=0xe8 p1=0 p2=9 src=2  round=3
CORE!  accu=0x67 p1=1 p2=12 src=3  round=4
CORE!  accu=0x8c p1=2 p2=15 src=4  round=4
CORE!  accu=0x49 p1=3 p2=2 src=5  round=4
CORE!  accu=0x67 p1=4 p2=5 src=6  round=4
CORE-K   k[ 9]= 0x63
CORE-K   k[10]= 0xfb
CORE!  accu=0xa1 p1=5 p2=8 src=0  round=4
CORE!  accu=0x4f p1=6 p2=11 src=1  round=4
CORE!  accu=0x60 p1=7 p2=14 src=2  round=4
CORE!  accu=0xb0 p1=8 p2=1 src=3  round=4
CORE!  accu=0x54 p1=9 p2=4 src=4  round=4
CORE!  accu=0xf4 p1=10 p2=7 src=5  round=4
CORE!  accu=0xf7 p1=11 p2=10 src=6  round=4
CORE-K   k[14]= 0x63
CORE-K   k[15]= 0xfb
CORE!  accu=0xb8 p1=12 p2=13 src=0  round=4
CORE!  accu=0x2a p1=13 p2=0 src=1  round=4
CORE!  accu=0x51 p1=14 p2=3 src=2  round=4
CORE!  accu=0xa7 p1=15 p2=6 src=3  round=4
CORE!  accu=0xf9 p1=16 p2=9 src=4  round=4
CORE!  accu=0x40 p1=0 p2=12 src=5  round=4
CORE!  accu=0x1b p1=1 p2=15 src=6  round=5
CORE-K   k[ 3]= 0x63
CORE-K   k[ 4]= 0xfb
CORE!  accu=0x50 p1=2 p2=2 src=0  round=5
CORE!  accu=0xd4 p1=3 p2=5 src=1  round=5
CORE!  accu=0x52 p1=4 p2=8 src=2  round=5
CORE!  accu=0x d p1=5 p2=11 src=3  round=5
CORE!  accu=0x56 p1=6 p2=14 src=4  round=5
CORE!  accu=0xfc p1=7 p2=1 src=5  round=5
CORE!  accu=0x15 p1=8 p2=4 src=6  round=5
CORE-K   k[ 8]= 0xfb
CORE-K   k[ 9]= 0x63
CORE!  accu=0xf4 p1=9 p2=7 src=0  round=5
CORE!  accu=0xfb p1=10 p2=10 src=1  round=5
CORE!  accu=0x68 p1=11 p2=13 src=2  round=5
CORE!  accu=0x70 p1=12 p2=0 src=3  round=5
CORE!  accu=0x32 p1=13 p2=3 src=4  round=5
CORE!  accu=0x63 p1=14 p2=6 src=5  round=5
CORE!  accu=0x75 p1=15 p2=9 src=6  round=5
CORE-K   k[13]= 0xfb
CORE-K   k[14]= 0x63
CORE!  accu=0xdf p1=16 p2=12 src=0  round=5
CORE!  accu=0xb0 p1=0 p2=15 src=1  round=5

Key: 0x
 fb 63  0 63 fb fb fb 63 fb 63 fb fb 63 fb 63 fb

State: 0x
 1b 50 d4 52  d 56 fc 15 f4 fb 68 70 32 63 75 df b0
```

J Computational efficiency in software

J1 Primitive Setup Part

1 cycle per byte loading the IV, padding with constant
4 cycles initialize pointers, counters, accu
1 cycle reset round counter
2 cycles loop control for INIT_ROUNDS
- 1 cycle increment round counter
- **2 cycles loop control for nx sub-rounds**
- **2 cycles 2 times EXOR**
- **1 cycle S-BOX accecss**
- **1 cycle new state byte**
- **3 cycles update p1**
- **3 cycles update p2**
- **1 cycle increment src**
- **1 cycle conditional key modification**
  - ***1 cycle decrement src***
  - ***3 cycles calculate p3***
  - ***3 cycles calculate p4***
  - ***3 cycles new k[p3]***
  - ***3 cycles new k[p4]***

2 cycles average conditional increment p2

J2 Streaming Part

2 cycles loop control for MAX_ROUNDS
2 cycles loop control for STREAM_ROUNDS
- 1 cycle increment round counter
- **2 cycles loop control for nx sub-rounds**
- **2 cycles 2 times EXOR**
- **1 cycle S-BOX accecss**
- **1 cycle new state byte**
- **3 cycles update p1**
- **3 cycles update p2**
- **1 cycle increment src**
- **1 cycle conditional key modification**
  - ***1 cycle decrement src***
  - ***3 cycles calculate p3***
  - ***3 cycles calculate p4***
  - ***3 cycles new k[p3]***
  - ***3 cycles new k[p4]***

2 cycles average conditional increment p2

2 cycles loop control for encryption
- **1 cycle EXOR operation on plaintext byte**

1 cycle	**increment P/C pointer**
3 cycle	**increment po pointer**

K Computational efficiency in hardware

CLK rising edge operations:

```
53        round ← round + 1
              /* the following three lines, if output is required */
81            ciphertext[pc] ← plaintext[pc] exor state[po]  /* enc. */
82            pc ← pc + 1
83            po ← (po + 2) mod nx
58a       accu ← sbox_out
59        state[p1] ← sbox_out
57        address ← accu  exor  state[p1]  exor  k[p2]
```

CLK falling edge operations:

```
58b       sbox_out ← S-BOX-TABLE[ address ]
60        p1 ← ( p1 + 1 ) mod nx
61        p2 ← ( p2 + KEY_STEP1 ) mod nk
82        src ← src + 1
73        if ( round mod KEY_STEP2 equal 0) then p2 ← ( p2 + 1 ) mod nk
```

The operations above are executed 7 times (KEY_STEP3); then the following has to be inserted :

```
66        src ← src – KEY_STEP3
67        p3 ← ( p2 + 1 ) mod nk
68        p4 ← ( p3 + 1 ) mod nk
69        k[p3] ← SBOX[ k[p3] exor k[p2] ]
70        k[p4] ← SBOX[ k[p4] exor k[p2] ]
```

that means

CLK rising edge operations:

```
          /* p3 and p4 are always calculated in parallel to p2, line 61 */
69a       address ← k[p2] exor k[p3]
```

CLK falling edge operations:

```
69b       sbox_out ← S-BOX-TABLE[ address ]
50        src ← src - 7
```

CLK rising edge operations:

```
69c       k[p3] ← sbox_out
70a       address ← k[p2] exor k[p4]
```

CLK falling edge operations:

```
70b       sbox_out ← S-BOX-TABLE[ address ]
```

CLK rising edge operations:

```
70c       k[p4] ← sbox_out
57        address ← accu  exor  state[p1]  exor  k[p2]
```

a.s.o.