From: zunic@us.ibm.com
X-Lotus-FromDomain: IBMUS
To: aesround2@nist.gov
Date: Mon, 15 May 2000 13:36:24 -0400
Subject: Final Comments

Attached are the final comments for Round 2 from the MARS team.
Nev Zunic

Internet: zunic@us.ibm.com
IBM Crypto Solutions
(914) 435-6949 (T/L 295)

---------------------- Forwarded by Nev Zunic/Poughkeepsie/IBM on
05/15/2000 01:37 PM ---------------------------

Nev Zunic
05/15/2000 01:30 PM

To: jfoti@nist.gov
cc: David Safford/Watson/IBM@IBMUS, shaih@watson.ibm.com@IBMUS
From: Nev Zunic/Poughkeepsie/IBM@ibmus
Subject: Final Comments

Jim,
Attached are our final comments for Round 2. I've also attached two
additional documents (one on key agility and the other on linear analysis)
which are referenced in the Final Comments. These are complementary
documents. I'm attaching three different (doc, pdf, and postscript)
filetypes of the Final Comments:

(See attached file: Final Comments.doc)(See attached file: Final
Comments.pdf)(See attached file: Final Comments.ps)(See attached file:
linear.ps)(See attached file: key-agil.ps)

If you have any questions, please let me know.
Nev

Internet: zunic@us.ibm.com
IBM Crypto Solutions
(914) 435-6949 (T/L 295)

Key Agility in MARS

Shai Halevi

May 12, 2000

Abstract

We discuss some technical solutions to obtain key agility for the MARS key-setup procedure. (A discussion of the economics of the key agility issue appears in separate note.) We show three techniques for achieving key agility in MARS, which offer the following tradeoffs:

1. Pre-computing and storing an average of 4-8 additional bytes per key, makes it possible to eliminate the "key fixing" part of the key-setup. In some hardware implementations, this can reduce the key setup time by as much as a factor of two.

2. By pre-computing and storing 18 additional bytes per key (for a total of 22 additional bytes), makes it possible to reduce the key setup time to about 2-3 block encryptions.

3. Pre-computing and storing about 40 more bytes per key (for a total of 60 additional bytes), makes it possible to reduce the key setup time to around one clock encryption.

1 The key agility issue

This issue was recently raised as an argument against the key-setup routines of MARS and RC6. The underlying architecture consists of a hardware encryption chip which is connected to a high-capacity server. The premise is that the server keeps many different keys, and need to switch often between these keys. Hence, these keys are sent back and forth between the chip and the server. The "benchmark" which was put forward for this architecture, is of a server with about half-a-million different keys, where the context-switch occurs every four encryptions.

It was argued that in MARS and RC6, the only options that one has, is either to send the expanded key back and forth – which requires an additional memory on the server to store all the expanded keys, or to compute the key setup in the chip, which takes some time. The economics of this architecture are discussed in a separate note, where it is suggested that even these two choices do not seem to pose a serious problem for real-life systems. In this note, however, we demonstrate that at least for MARS, there are a few additional tradeoffs that one can use, which add just a few bytes per key of storage, but significantly reduce the key-setup time.

2 The MARS key-setup procedure

The MARS key-setup procedure consists of keeping a temporary array T of fifteen 32-bit words, and manipulating it to get the expanded key. The array T is initialized from the key, and then manipulated in four iterations, where each iteration generates ten of the forty words needed for the expanded key.

In each iteration, a linear transformation is first applied to T, followed by four rounds of "stirring", in which every entry in T is modified by adding to it an entry of the S-box, which is determined by the previous entry of T. Finally, the key words which are used in the cipher for multiplication are checked for some conditions, and modified if they do not meet these conditions. A pseudo-code for this procedure is shown in Figure 1.

2.1 Hardware implementation of the key-setup

Since the key setup procedure is very serialized, with not much room for parallelism, it seems that the critical path of any implementation in hardware would include nearly the entire procedure. Specifically, a simple implementation would have for each of the four iterations fifteen cycles for the linear transformation, followed by 60 cycles for the S-box stirring. Computing the mask to use in the "key fixing" part may take up to 15 cycles, so depending on the size of the circuit, the entire "key fixing" can take between 15 and 240 cycles.[1]

All in all, we have a critical path of anywhere between 315 and 540 cycles for the MARS key setup, which is roughly equivalent to encryption of 10-17 blocks. With respect to the key-agility "benchmark", this represents a slowdown of 250%-450% over the raw hardware encryption speed.

Below we persent three optimizations, which can be used in an architecture that needs key agility, to reduce the critical path to just about 1-3 block encryptions (20-80 cycles), while requiering only a small amount of additional storage. With respect to the key-agility "benchmark" from above, this represents a slowdown of only 15%-65% over the raw hardware encryption speed.

3 Three optimizations

The principle behind the three optimizations that we show below, is to first have the server run the key-setup procedure "off line", and store some of the information that is obtained during the procedure. This stored information is then used to speed up the computation of the "on line" key setup on the chip.

3.1 Eliminating the "key fixing"

The first observation is that we can compute and store ahead of time all the masks that are used in the "key fixing" part. This would save the time (and the hardware) needed of that part. The problem, of course, is that there are 16 masks per key, so storing them all would cost additional 64 bytes of memory. However, we note that it is sufficient to store only the non-zero masks, and that a typical key would have either zero or one such non-zero masks. It is therefore possible to store 16 flags, where the i'th flag is 1 if the corresponding mask is non-zero and 0 otherwise, and then to store only the non-zero masks.

The probability that a specific mask is non-zero is about 1/41, and so over 500,000 keys and 16 masks per key, we expect to have about $500,000 \times 16/41 \approx 195,000$ non-zero masks. It follows from the Chernoff bound that the probability of getting more than 250,000 non-zero masks is negligible (less than 2^{-10000}, see Appendix A. In fact, even the probability of getting more than 200,000

[1]Note that the "key fixing" for one iteration can be done in parallel with the next iteration, so it is possible to get down to 15 cycles with only 5-wise parallelizm.

```
Key-Expansion(input: $k[\,], n$; output: $K[\,]$)
1. // $n$ is the number of words in the key buffer $k[\,], (4 \le n \le 14)$
2. // $K[\,]$ is the expanded key array, consisting of 40 words
3. // $T[\,]$ is a temporary array, consisting of 15 words
4. // $B[\,]$ is a fixed table of four words

5. // Initialize $B[\,]$
6. $B[\,] = \{$0xa4a8d57b, 0x5b5d193b, 0xc8a8309b, 0x73f9a978$\}$

7. // Initialize $T[\,]$ with key data
8. $T[0 \ldots n-1] = k[0 \ldots n-1], \quad T[n] = n, \quad T[n+1 \ldots 14] = 0$

9. // Four iterations, computing 10 words of $K[\,]$ in each
10. for $j = 0$ to 3 do
11.     for $i = 0$ to 14 do                          // Linear transformation
12.         $T[i] = T[i] \oplus ((T[i-7 \mod 15] \oplus T[i-2 \mod 15]) \lll 3) \oplus (4i+j)$

13.     repeat four times                             // Four stirring rounds
14.         for $i = 0$ to 14 do
15.             $T[i] = (T[i] + S[\text{low 9 bits of } T[i-1 \mod 15]]) \lll 9$
16.     end-repeat

17.     for $i = 0$ to 9 do                           // store next 10 words into $K[\,]$
18.         $K[10j+i] = T[4i \mod 15]$
19. end-for

20. // Modify multiplication keys
21. for $i = 5, 7, \ldots 35$ do
22.     $j =$least two bits of $K[i]$
23.     $w = K[i]$ with both of the least two bits set to 1

24.     // Generate a bit-mask $M$
25.     $M_\ell = 1$ iff $w_\ell$ belongs to a sequence of ten consecutive 0's or 1's in $w$
26.                 and also $2 \le \ell \le 30$ and $w_{\ell-1} = w_\ell = w_{\ell+1}$

27.     // Select a pattern from the fixed table and rotate it
28.     $r =$least five bits of $K[i-1]$              // Rotation amount
29.     $p = B[j] \lll r$

30.     // Modify $K[i]$ with $p$ under the control of the mask $M$
31.     $K[i] = w \oplus (p \wedge M)$
32. end-for
```

Figure 1: The key-setup procedure in MARS

$T_{0,4}$: the initial array T
$T_{1,0}$: the array T after the 1st linear transformation
$T_{1,1}$: The array T after the 1st stirring round of the 1st iteration
$T_{1,2}$: The array T after the 2nd stirring round of the 1st iteration
$T_{1,3}$: The array T after the 3rd stirring round of the 1st iteration
$T_{1,4}$: The array T after the 4th stirring round of the 1st iteration
$T_{2,0}$: the array T after the second linear transformation
$T_{2,1}$: The array T after the 1st stirring round of the 2nd iteration
$\vdots$
$\vdots$
$T_{4,4}$: The array T after the 4th stirring round of the 4th iteration

Figure 2: The dynamic-programming notations of the key-setup procedure

non-zero masks is already as low as 2^{-90}). Hence, it is sufficient to store two bytes per key for the flags and an average of less than two bytes per key for the non-zero masks.[2]

3.2 A dynammic-programming approach to the key setup

Next, we show how to speed up the S-box lookup part of the key-setup procedure. To do that, we adopt a dynammic-programming approach to this procedure. Instead of viewing the procedure as operating on one array T, we view it as filling rows in a two-dimesional table. Different rows in this table correspond to the temporary array T during different times in the key setup procedure. That is, the first row in the table consists of the initial array T, the second row consists of T after the first linear transformation, the third row consists of T after the first stirring, and so on.

In the description below it is convenient to use the following notations: the array after the linear transformation in the i'th iteration is denoted by $T_{i,0}$, and the array after the j'th stirring round in this iteration by $T_{i,j}$ $(i, j = 1 \ldots 4)$. Using these notations, $T_{i,0}$ is obtained as a linear transformation of $T_{i-1,4}$, and $T_{i,j}$ is obtained from $T_{i,j-1}$ by setting, namely,

$$\begin{aligned} T_{i,j}[0] &= (T_{i,j-1}[0] + S[\text{low 9 bits of } T_{i,j-1}[14]]) \lll 9; \quad \text{and} \\ T_{i,j}[k] &= (T_{i,j-1}[k] + S[\text{low 9 bits of } T_{i,j}[k-1]]) \lll 9 \quad (k = 1 \ldots 14) \end{aligned}$$

The initial array which is computed from the key is denoted $T_{0,4}$. The 40 words of the expanded key are computed from rows $T_{1,4}, T_{2,4}, T_{3,4}, T_{4,4}$ of the table. See Figure 2 for an illustration of these notations.

We now observe that each "internal entry" $T_{i,j}[k]$ in this table that correspond to a stirring operation, is computed from the entry above it $T_{i,j-1}[k]$ and the low nine bits of the entry to its left $T_{i,j}[k-1]$. Similarly, each entry $T_{i,j}[0]$ on the edge of the table that correspond to a stirring operation, is computed from the entry above it $T_{i,j-1}[0]$ and the low nine bits of the last entry in the row above $T_{i,j-1}[14]$.

[2]In reality, you may need to also store with each key a four-byte index into the table of non-zero masks, so you get a total of eight additional bytes per key. It is possible to decrease this back to four bytes per key using slightly more sophisitcated data structures.

Consider now what happens if we pre-compute and store the low nine bits of the entries $T_{i,j}[14]$ for $i = 1 \dots 4, j = 0 \dots 3$. Assume that we initialized the first row $T_{0,4}$, and now we want to compute the rows $T_{1,0}$ through $T_{1,4}$, which correspond to the first iteration. In the first clock cycle we can compute the top-left corner $T_{1,0}[0] = T_{0,4}[0] \oplus ((T_{0,4}[8] \oplus T[13]) \lll 3) \oplus 0$. In the next clock, we can compute $T_{1,0}[1]$ and also $T_{1,1}[0]$ (since now we have $T_{1,0}[0]$ as well as the low nine bits of $T_{1,0}[14]$ which we stored ahead of time). In the next clock we compute $T_{1,0}[2], T_{1,1}[1]$ and $T_{1,2}[0]$, then $T_{1,0}[3], T_{1,1}[2], T_{1,2}[1]$ and $T_{1,3}[0]$. From there on, in each cycle we compute one entry in each row.

Hence, we can compute the five rows $T_{1,0} - T_{1,4}$ in just 19 cycles, and we can repeat this for the other three iterations, and compute the entire table in $4 \times 19 = 76$ cycles. If we also used the previous optimization to eliminate the "key fixing" part, we can potentially carry out the key setup procedure in just under 80 cycles. The amount of storage that is needed for this optimization is 9 bits times 16 rows, for a total of 144 bits (18 bytes).

3.3 Additional speed-ups

Finally, we note that if we can afford some more storage, we can pre-compute and store one full row of the table from above, instead of storing the original key and computing all the rows. The simplest thing in this case would be to store the row $T_{1,4}$, which is the first row that is used to compute the resulting expanded key. This way, there are only three more iterations to compute (rather than four), so we can reduce the computation time to just 60 cycles.

Even more, we can store some row in the middle of the table, and compute first half of the table "backwards", at the same time that we compute the second part forward. This can even be pipelined with the encryption/decryption of the first block. For example, when the key is used for encryption we can store the row $T_{2,4}$. Then, in 20 cycles we can compute both $T_{1,4}$ and $T_{3,4}$, at which time we can start encrypting the first block (since we already have the first 30 words of the expanded key). In an additional 20 cycles we compute also $T_{4,4}$, so by the time we need the last 10 words, they are already available.

This last optimization can potentially reduce the latency due to key setup to about 20 cycles, well below the time of encrypting one block. The cost in storage is to keep one row of the table (60 bytes) instead of the original key (16-32 bytes). Since we also need twelve 9-bit pointers, the total additional memory (as compared to storing only the key) is 42-58 bytes.

3.4 Summary

We presented above three optimizations, which allow a wide range of memory-speed tradeoffs in the MARS key-setup, provided that we can compute and store some information ahead of time. Table 1 summarizes the features of these optimizations. The "slowdown" column describes the slowdown which is encountered by the key agility application, as expressed in the "four encryption per key benchmark", vs. the raw encryption speed.

We note that the last two optimizations may require additional hardware since they require that several operations be done in parallel. However, it seems that even with the last, most aggressive optimization, the area needed for key setup is not more than what is needed for the encryption/decryption hardware.

Implementation	storage needed	key setup cycles	slowdown
Full setup in hardware	key only (16-32 bytes)	315-540	250%-450%
First two optimizations	key + 22 bytes	80	65%
All three optimizations	80 bytes	20-60	15%-50%
Storing expanded key	160 bytes	0	0%

Table 1: Storage vs. speed in the MARS key setup

A Bounding the number of non-zero masks

Recall the Chernoff bound for the sum of 0-1 random variables. Assume that you have n independent random variables, $X_1 \ldots X_n$, each of them is 1 with probability p and 0 with probability $1-p$. Then, for every $\delta > 0$, it holds that

$$\Pr\left[\left|\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) - p\right| > \delta\right] < 2e^{-(\delta^2 n)/(2p(1-p))}$$

In our case, we have $n = 500,000 \times 16 = 8,000,000$ masks, each of them is non-zero with probability $p = 1/41$, and we want to bound the probability that we get more than 250,000 non-zero masks. Namely, $\Pr[\sum X_i > 250,000]$. We first note that

$$\begin{aligned}\Pr\left[\sum_{i=1}^{8{,}000{,}000} X_i > 250,000\right] &= \Pr\left[\left(\frac{1}{8,000,000}\cdot\sum_{i=1}^{8{,}000{,}000} X_i\right) > \frac{250,000}{8,000,000}\right] \\ &= \Pr\left[\left(\frac{1}{8,000,000}\cdot\sum_{i=1}^{8{,}000{,}000} X_i\right) - \frac{1}{41} > \frac{250,000}{8,000,000} - \frac{1}{41}\right] \\ &< \Pr\left[\left|\left(\frac{1}{8,000,000}\cdot\sum_{i=1}^{8{,}000{,}000} X_i\right) - \frac{1}{41}\right| > \frac{250,000}{8,000,000} - \frac{1}{41}\right]\end{aligned}$$

So we can use $\delta = \frac{250{,}000}{8{,}000{,}000} - \frac{1}{41} \approx .00686$ in the bound. Plugging in these values for n, p and δ, we get $\delta^2 n/2p(1-p) \approx 7910$, so we can bound the probability of getting more than 250,000 non-zero masks by

$$\Pr\left[\sum X_i > 250,000\right] < 2e^{-7910} \approx 2^{-11411}$$