Performance Evaluation of AES Finalists on the High-End Smart Card

Fumihiko Sano* Masanobu Koike* Shinichi Kawamura† Masue Shiba*

* Toshiba System Integration Technology Center
3-22, Katamachi Fuchu-shi, Tokyo, 183-8512, JAPAN
† Toshiba Research and Development Center
1, Komukai Toshiba-cho, Saiwai-ku, Kawasaki, 210-8582, JAPAN
{fumihiko.sano, masanobu2.koike, shinichi2.kawamura, masue.shiba}
@toshiba.co.jp

Abstract. This paper reports the performance of the AES finalists, MARS, RC6, Rijndael, Serpent, and Twofish, on the high-end smart card that has a Z80 core with Toshiba's arithmetic coprocessor.

1 Introduction

During the first round of AES candidate assessment, some reported the performance evaluation of the algorithms on low-end smart cards. Their reports are important for understanding performance of each AES candidates in memory and computing resource-restricted environments. However, there are, so called high-end smart cards, which are equipped with a specific hardware for accelerating cryptographic processing. In general these cards are less restricted in their resource than low-end smart cards. So, it is important for better understanding of the AES candidates to evaluate the performance on high-end smart cards. NIST as well expressed their interests in such evaluation in [11]. This paper describes our experience in implementing five AES finalists, and summarizes the performances on our high-end smart card available from Toshiba[17].

The high-end smart card is substantially different from low-end one in that its core is equipped with a crypto coprocessor. It may usually correct to say that the amount of memory for a high-end card is larger than that of low-end one. In some cases, however, venders supply cards with large memory amount suitable for their specific purposes regardless of the core. Therefore, we distinguish between high-end and low-end cards based on the type of core.

At first, we present the architectures of the core on our smart card that includes a CPU and a coprocessor architecture. Next, we describe coding rules for our implementation and then, present experiences of five AES finalists accompanied by results of 64-bit ciphers such that DES[10] and MISTY1[9] on our smart card for reference purpose. Finally, we summarize advantages and disadvantages for each implementation.

2 Platform

High-end smart cards available now are usually equipped with 8/16-bit microprocessor and a crypto coprocessor, or accelerator for cryptographic operations[7]. To evaluate the AES finalists' performance on high-end smart cards, we choose Toshiba's T6N55 chip shown in table 1. The chip is equipped with Z80 microprocessor and a coprocessor. The coprocessor is under the control of Z80 and it carries out arithmetic/logical operations when Z80 asks to do so. The coprocessor is originally designed to accelerate the large integer arithmetics. As will described shortly, it is also suitable to accelerate some operations required to implement AES finalists.

Table 1. Features of Toshiba's T6N55 chip

CPU	Z80
ROM	48KB
RAM	1KB
EEPROM	8KB
Max. of Modulus	2,048-bit
Internal Clock Frequency	5MHz

2.1 Z80 Architecture

The Z80 is a famous 8-bit architectured microprocessor developed by ZiLOG[15]. It has an 8-bit accumulator and a flag register, six 8-bit general-purpose registers, two 16-bit index registers, a stack pointer (SP), and a program counter (PC). An accumulator A and a flag register F can be paired and dealt with as if it is a 16-bit register AF. Similarly, 8-bit registers can be paired with particular registers as BC, DE, and HL. Z80 incorporates dual register banks. Each register bank has each register sets such as an accumulator, a flag register, and six 8-bit registers. Note that one can use only one side of the banks at a time. If one want to use registers belonging to the other side of the bank, he should change the contexts with an EXX operation.

The instruction set includes the following classes:

- Load 8-bit values to registers or an accumulator.
- Load 16-bit values to registers.
- Arithmetic or logical instructions for the accumulator with registers.
- A single bit shift or rotate instructions.
- Compare, block transfer, and search instructions.
- Branch instructions.
- Subroutine calls and returns from them.
- I/O instructions.

– Checking or setting a single bit in registers.

There are some particular instructions for extended registers or control instructions of processor. Z80 can execute addition, subtraction, AND, OR, exclusive or (XOR), and single-bit rotation and shift. It does not have instructions for multiplication and division.

On using the ordinary Z80 core, we should take some features of its architecture into account. It needs four clocks even for the basic instructions, such as a no operation (NOP) or a load instructions between registers (LD r, r'). The next fastest instructions, such as for loading a value to a register (LD r, n) consume seven clocks. Operations for 16-bit register sets are more time consuming. Although we try to use faster operations, the average number of clocks needed for an instruction is about six.

2.2 Crypto Coprocessor

The coprocessor is developed mainly to accelerate the processing of the public key cryptosystem. It has 512-byte RAM area (we call it the 'CRAM' area). That area is segregated into two 256-byte RAM areas. The coprocessor can execute various operations between the 256-byte RAM areas or on the 512-byte RAM. Each maximum size of arithmetical operations supported by the crypto coprocessor is shown in table 2.

It can execute the following classes of calculations:

– Addition, subtraction, multiplication, division, and logical operations.
– Modular multiplication.
– Modular exponentiation.
– Montgomery multiplication.
– Extended Euclidean algorithm.
– Memory transfer in CRAM area.

Here, the logical operations mean AND, OR, and exclusive OR(XOR). The memory transfer is used to transfer data on CRAM area efficiently. So, the feature is similar to the direct memory access (DMA). The most time consuming operation is a modular exponentiation with a large exponent. Other operations, when used in implementing AES finalists, are very fast and finish within a time for the minimum execution time of a Z80 instruction.

The coprocessor executes logical operations between operands located on each CRAM areas. Before executing these operations, Z80 have to put several bytes of control words on the CRAM area in addition to the operands. Since Z80 does not perform so fast to the data on memory, using coprocessor operations are efficient for large data, but not so much for small data.

3 Implementations

3.1 Coding Rules

When we implement the AES finalist, we apply the following rules for the coding.

Table 2. Features of Toshiba's Crypto Coprocessor

Instruction	Max. of Operands (bits)
Addition	2,048
Subtraction	2,048
Multiplication	1,024
Division	2,048
Modular Multiplication	1,024
Exponentiation	1,024

– Program codes are located on the ROM area, and we do not change the code at any time.
– We can use all registers, i.e., registers on both sides of the banks.
– The codes run in constant time not depend on the data to avoid timing analysis.
– We can use memory on the CRAM area if necessary.
– We write codes that generate the extension keys with on-the-fly, if possible.

A time constancy of a code is an imprecise term. We try to give more precise idea behind the third rule. If we have only to realize the time-constancy, we may choose an easy way to stretch the execution time by merely adding NOPs at the end of the code. But what we really have to do is to avoid timing analysis. So, we have to pay more attention not to leak meaningful information. If we can successfully apply the third rule, we can prevent simple power analysis as well as timing attack. The third rule is not sufficient, though it seems necessary, to prevent the differential power analysis. We don't discuss on the differential power analysis in this paper any further.

It is interesting that we may neglect the differences between rounds, for example the key expansion of DES need 2-bit rotations in some rounds. They may leak some information, but it seems useless for analysis.

In this section, we report the performance of AES finalists in alphabetic order. For comparison purpose, results for 64-bit block ciphers, such as DES, triple DES, and MISTY1, will be shown, as well. We describe the speed of each algorithm with clocks and RAM requirement: In each table, 'Int.' means that size of required CRAM for coprocessor's operations, and 'Ext.' means other work area. Note that 5,000 clocks at 5MHz correspond to 1 millisecond. For example, DES needs about 25,000 clocks, and thus it works in 5ms.

The code of DES does not necessarily obey the coding rules above since some permutations for DES are realized by hard wired logic. The triple DES is a two-keyed one, but it executes the key schedule three times with on-the-fly. Therefore, three-keyed triple DES will have the same performance result. MISTY means the MISTY1 algorithm[9] with eight rounds.

To apply our results easily for other processors that have similar features, we try to reduce the memory usage on the CRAM area. But, in this paper, we see that the memory usage is of little importance, since the platform chosen has sufficient memory for these implementations.

3.2 MARS

It is the most difficult task for us to implement MARS on smart cards or other limited resources. MARS has a complex high level structure such as eight rounds of unkeyed forward mixing, eight rounds of keyed forward transformation, eight rounds of keyed backward transformation, and eight rounds of unkeyed backward mixing. Each of the eight rounds consists of so called type-3 Feistel network. In a type-3 Feistel network, input data is segregated into four words. One of them is taken as a pseudo-random function's input and the output is used to modify three other data words. Since MARS has a block length of 128 bits, each word has 32 bit length.

There are three disadvantages of MARS when implemented on a smart card. The first is that it needs 2KB table for S-boxes, but it is not serious. The second is the weakness check of extended key on the key schedule. The last is the rotations with variable shift amount. We discuss the last two disadvantages here.

It is necessary for MARS to implement complicated "weak" measures on the key schedule[3]. The weak keys for MARS are different from those of DES. In the case of DES, you may disregard the problem of weak key because it only increases some potential threats caused by the weak key properties. However, in the case of MARS, since the weak key check procedure is a part of the algorithm specification, you have to check the weak on the key schedule certainly. Otherwise, you may see a terrible result, such as differences in cipher text, although it encrypts the same plain text with common key. As mentioned above, the function of checking the weak on the key schedule is primarily needed.

Although implementing weak key check is necessary, it is also true that this introduces another problem for smart card implementation. If we check the weak and regenerate extension keys, there is a risk of applying timing attack. The regeneration of extension keys causes difference in processing time and leaks some information on the key. Further study of coding is necessary to avoid this problem.

To save our time, our implementation just omits the weak key check. Therefore, it is not complete. Our implementation is not so slow because of customization for 256-bit key and omitting to check 'weak' on the key schedule. The codes for check 'weak' on the key schedule will increase the requirement of ROM and processing time.

The rotations depend on a key data or an internal data are crucial for Z80 or other 8-bit processors since we need to write codes that run in constant time, or else an attacker can get some information about the key. Fortunately, our coprocessor can operate modular multiplications over any modulus. We use them for rotations. Modular multiplications on our smart card are very fast, and finish within a single instruction of Z80. It means that we can operate modular multiplications and data dependent rotations in a constant time and avoid timing attack.

It seems that MARS is a prudent algorithm against cryptanalysis. But it causes some difficulties in implementing on smart cards or similar resource-restricted environments.

Table 3. MARS

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	60	36	24	3,977	45,588
Schedule	512	512	0	1,491	21,742
Total	512	512	24	5,468	67,330

3.3 RC6

RC6 has various parameters and is defined as RC6-$w/r/b$ where w means the word length, r means the number of rounds, and b means the length of key with bytes. We write the code with the recommended parameters for AES such as RC6-32/20/32.

RC6 has a simple structure, but the round function includes various operations such as, addition, subtraction, multiplication, and rotations depending on a variable data. Most part of RC6 constructed by arithmetical operation. Therefore, we operate almost all operations on the coprocessor. Furthermore, since the coprocessor can operate up to 1,024 bits for operand, we can execute the pair of rotations with constant shift amount in parallel. An n-bit rotations to two data is written as follows: We duplicate each of data and put them on corresponding CRAM area, then multiply them with 2^n. As a result, we can improve the performance and reduce the size of code.

The coprocessor can execute RC6 data encryption efficiently. RC6 has a simple key schedule but need much iterations and does not suitable with on-the-fly. The key schedule takes four times as long execution time as encryption.

There is an idea to improve the key schedule processing time. A precomputed table improves the speed, but increase the size of code. It omits the computation of 43 initial values (S[i]) with 32-bit word. The modified code copies S[i]s from precomputed ROM table to RAM area instead of computing S[i]s with constant values. It shall reduce about 4,000 clocks. It needs some extra code or table for precomputed table, thus the size of code increases about 150 bytes.

On the smart cards, RC6 has a moderate encryption speed among the finalists, but its key schedule is slower than Rijindael or Twofish. Note that it has been reported that on the 32-bit processor, RC6's performance is faster than Rijndael and Twofish[5].

Table 4. RC6

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	124	124	0	489	34,736
Schedule	90	90	0	571	138,851
Total	156	156	0	1,060	173,587

3.4 Rijndael

256-bit key is the fastest for on-the-fly key generation, since we can translate the internal key every two rounds. 128-bit key is a little slower than 256-bit key, since we need to make extension keys every round. In the case of 192-bit key, since the key length is not the multiple of the block length, it is not so easy to implement on-the-fly key generation.

The `xtime` is an important subroutine for time constancy. It needs modulus operation with the primitive polynomial. Here is an example of straightforward implementation of the `xtime(a)` algorithm where the original value is stored in A register.

```
          RLA
          JR    NC, SKIP
          AND PRI          ; PRI means the primitive polynomial.
     SKIP:
          ...              ; end.
```

This is a very dangerous code. Since 'AND *PRI*' operation is operated only when the carry is '1', an attacker can know whether the value excesses 2^8 or not in this code. We must avoid such an implementation. Therefore, we use some techniques to avoid differences of processing time and thus prevent cryptanalysis using timing attack. Here is an example of `xtime(a)` operation with constant time, where `a` is stored in A register.

```
RLA
LD      B, A
SBC     A, A
AND     PRI
XOR     B
```

RLA is a instruction of 1-bit leftward rotation for A register. If RLA is carried out, MSB of A register is set to the carry flag. 'SBC A, A' is an instruction which substract a value in A register and a carry from A register. It means that if the carry flag is '1' then A register has a value `0xff`, otherwise A register has a value `0x00`. Next we operate AND instruction with *PRI* for A register. Then we get *PRI* or a value `0x00` in A register, and we can operate whether 'XOR PRI' or 'NOP' with the same instructions and processing time.

The transformation MixColumn is implemented in an efficient way shown in section 5.1 in [4]. We implement the `AddRoundKey` and data transfers with the coprocessor. Other transformations in Rijndael are not so heavy even for only the Z80 core. Rijndael is the most efficient algorithm on the finalists on our smart card.

A disadvantage of Rijndael is that it needs another code for decryption because of the asymmetry of encryption and decryption. If you need both encryption and decryption algorithms, it takes twice ROM area for code since most part of it cannot be shared.

Table 5. Rijndael

	RAM (bytes)			ROM (bytes)	Time (clocks)
	Total	Int	Ext		
Encrypt	34	32	2	700	25,494
Schedule	32	32	0	280	10,318
Total	66	64	2	980	35,812

3.5 Serpent

There is two kinds of implementation of Serpent: ordinary implementation and bitsliced implementation. Here is the result of an ordinary implementation of Serpent. It is not a bitsliced implementation. It needs a 2,048-byte ROM table on the ordinary implementation.

Serpent has various rotational operations. As is described in MARS implementation, modular multiplication with coprocessor can be used if they improve the performance. Most of the rotations are, however, more efficient with the Z80 operations than with the coprocessor. 1-bit leftward or rightward rotations can be implemented with the Z80 operations, and shifts with multiplies of 8-bit are reorder of bytes. We use the coprocessor operations only for 11-bit rotations, XOR, and memory transfer. Due to the architecture of our coprocessor, it is not suitable to efficiently implement three-operand operation used in Serpent.

In [2], Serpent can be implemented using under 80 bytes of RAM with on-the-fly. Our implementation needs twice more RAM, because we write it with coprocessor's operation XOR between halves of CRAM with different offsets.

It has more rounds than other finalists do, so its performance is not so good as Rijndael or Twofish.

The bitsliced implementation will reduce the size of code and required RAM with a little degradation in speed. In memory-restricted environment, bitsliced implementation may be better than the ordinary coding. In this paper, we attach importance to the speed. So, we choose the ordinary implementation for performance comparison.

3.6 Twofish

In the case that the length of key is less than 256-bit, we need to pad out the original key until it becomes 256-bit. We implement Twofish with 128-bit key to

Table 6. Serpent

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	68	68	0	3,524	71,924
Schedule	96	96	0	413	147,972
Total	164	164	0	3,937	219,896

take the processing time for padding into account. It includes code for padding, and it is a little slower than 256-bit key.

There are two models for implementing Twofish, such as Feistel model and non Feistel model[14]. We implement it with non Feistel model. We assume that it is faster than Feistel. We use coprocessor's operations for additions with subkeys, XOR, and memory transfers on CRAM area, but rotations are implemented with Z80's rotations.

The performance of Twofish depends on the size of precomputed tables' [14]. We consider that the case of using some tables amounted to 1,536 bytes. This code is compact for processing the key schedule with precomputed tables. It seems be compatible with 2200 bytes for code and table size model in [14]. The size of precomputed tables is belongs to encryption code in table 7.

Twofish is as fast as DES on throughput. It does not have any exceptional advantages, but we have nothing to complain about the performance.

Table 7. Twofish

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	34	32	2	2,493	31,877
Schedule	56	32	24	315	28,512
Total	90	64	26	2,808	60,389

4 Summary

We summarize the performance and the required resources on our implementations in table 8. The RAM includes required byte in the RAM area and the CRAM area. Note that when using a coprocessor, the required amount of RAM increase, because of the alignment rules for CRAM area.

Some finalists are designed to have heavy key schedules. They are intended to prevent exhaustive search attacks, but resulting in speed reduction on smart cards. We consider that Rijndael is excellent on all aspects. RC6 is as good as Rijndael on the code point of view, but the key schedule consumes more time.

Twofish needs much ROM memory than RC6 and Rijndael because of the table. It is faster than Triple DES and equal to DES on the throughput. It will have good performance on any smart cards. MARS has disadvantages of its code size caused by four of eight round iterations and a 2,048-byte table. The speed is equal to Twofish's one. We consider MARS has some difficulties to check 'weak' on the key schedule and regenerate. Serpent has disadvantages of its performance caused by the iterations of rounds and the difficulty of key schedule. The bitsliced implementation will improve the requirement of ROM or RAM, but slower than others.

We tried to write all program codes to consume as little RAM area as possible. On the other hand, if we may regard the RAM area, especially CRAM area, as a kind of free work space, it will be unfair to compare finalists how little work area they consume. Nevertheless, notice that MARS consumes all the CRAM area, whereas others consume at most half of the area.

Table 8. Comparison of AES finalists and the algorithms

Cipher	RAM		ROM		Time (clock)						
	(bytes)		(bytes)		Encrypt		Schedule		Encrypt + Schedule		
MARS	572	5	5,468		45,588	4	21,742	2	67,330	3	∗
RC6	156	3	1,060	2	34,736	3	138,851	4	173,587	4	
Rijndael	66	1	980	1	25,494	1	10,318	1	35,812	1	only encryption
Serpent	164	4	3,937	4	71,924	5	147,972	5	219,896	5	
Twofish	90	2	2,808	3	31,877	2	28,512	3	60,389	2	
DES	17		772						25,398		
Triple DES	17		849						72,341		
MISTY	44		1,598						25,486		

∗: omit to check "weak" in the key schedule.

5 Conclusion

We have implemented AES finalists on a high-end smart card that is equipped with a crypto-coprocessor. The resulting code has higher performance than that on a low-end smart cards, since multiplication and rotation are efficiently implemented using the coprocessor's commands. Coprocessor's RAM are also useful for work memory, as well.

Regarding speed, Rijndael is the best one and is as fast as our DES implementation. It is twice faster than DES on the throughput. RC6 is suitable for our smart card same as on the 8051[6, 8], but not to be compared with Rijndael or Twofish because of the key schedule.

For smart card implementation, it is necessary to perform key schedule at least for every processing block, in order to save memory areas to store extended key. For the same reason, it is desirable for key schedule to be suitable for on-the-fly key generation. As a result, design concept for key schedule affects the performance very much, and those algorithms that have heavy key schedule are not advantageous for smart card implementation.

Finally, we report the performance of E2[12] that is a candidate on the first round in the appendix.

References

1. R. Anderson, E. Biham, and L. Knudsen, *"Serpent: A Proposal for the Advanced Encryption Standard"*, AES submission, 1998.
2. R. Anderson, E. Biham, and L. Knudsen, *"Serpent and Smartcards"*, CARDIS '98, 1999, available on http://www.cl.cam.ac.uk/˜rja14/serpent.html.
3. C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr., L. O'Connor, M. Peyravian, *"MARS -a candidate cipher for AES"*, AES submission, 1998.
4. J.Daemen, V.Rijmen, *"AES Proposal: Rijndael"*, AES submission, 1998.
5. B. Gladman, "AES Algorithm Efficiency", http://www.btinternet.com/˜brian.gladman/cryptography_technology/Aes/
6. G. Hachez, F. Koeune, and J. Quisquater, *"cAESar results: Implementation of Four AES Finalists on Two Smart Cards"*, The second AES conference, 1999, available on http://www.dice.ucl.ac.be/crypto/CAESAR/caesar.html.
7. H. Handschuh, and P. Paillier, *"Smart Card Crypto-Coprocessors for Public-Key Cryptography"*, CryptoBytes, Vol. 4, No. 1, RSA Laboratories, 1998.
8. G. Keating, *"Performance Analysis of AES candidates on the 6805 CPU core"*, The second AES conference, 1999, available on http://www.ozemail.com.au/˜geoffk/aes-6805/.
9. M. Matsui, *"New Block Encryption Algorithm MISTY"*, Fast Software Encryption, 4th International Workshop Proceeding, LNCS **1267**, Springer-Verlag, 1997, pp.54-68.
10. National Bureau of Standards, *"Data Encryption Standard"*, U.S.Department of Commerce, FIPS 46-3, October 1999.
11. J. Nechvatal, E. Barker, D. Dodson, M. Dworkin, J. Foti, and E. Roback, *"Status Report on the First Round of the Development of the Advanced Encryption Standard"*, http://csrc.nist. gov/encryption/aes/round1/r1report.pdf
12. Nippon Telegraph and Telephone Corporation, *"Specification of E2 – a 128-bit Block Cipher"*, AES submission, 1998.
13. R.L. Rivest, M.J.B. Robshaw, R. Sidney, Y.L. Yin, *"The RC6 Block Cipher"*, AES submission, 1998.
14. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, *"Twofish; A 128-Bit Block Cipher"*, AES submission, 1998.
15. ZiLOG, *"Z80 Microprocessor Products"*, available on http://www.zilog.com/products/z80.html
16. http://csrc.nist.gov/encryption/aes/round2/Round2WhitePaper.htm, 1999.
17. http://www.toshiba.co.jp/about/press/1999_02/pr_j0301.htm, (in Japanese).

A E2

E2 is not selected as a finalist for the second round review. But it has a good performance, especially encryption speed without key schedule. The serious disadvantages of E2 are that it has time consuming key schedule and can't execute it with on-the-fly. Fortunately, since the RAM usage fits on the half of CRAM area, we select a way to extend all round keys on the half of them, at first. In this case, E2 is efficient for encryption just like the report in [6]. The round function is designed as suitable for byte oriented operations. It is good for the Z80 architecture. It is, however, difficult for Z80 to execute multiplication on the IT and division on the FT. We use the coprocessor's commands for these operations. Those commands include XOR, memory transfer, multiplication, and inverse.

Table 9. E2

	RAM (byte)			ROM (byte)	clock
	Total	Int	Ext		
enc	26	24	2	1,519	17,018
key	548	512	36	296	79,358
Total	548	512	36	1,815	96,376