cAESar results: Implementation of Four AES Candidates on Two Smart Cards

Gaël Hachez François Koeune
Jean-Jacques Quisquater
Université catholique de Louvain, UCL Crypto Group,
Laboratoire de microélectronique (DICE),
Place du Levant 3, B-1348 Louvain-la-Neuve, Belgium.
E-mail: {hachez,fkoeune,jjq}@dice.ucl.ac.be
URL: http://www.dice.ucl.ac.be/crypto

September 26, 2000

Abstract

One of the important criteria the future cryptographic standard will have to meet is smart card suitability. However, the information provided in the official submission makes comparison rather difficult. This paper tries to partially answer this question, by describing our experience in implementing four of the candidates on two smart card: a typical low-cost one and a typical "sophisticated" one.

1 Introduction

In the requirements for an AES candidate submission, the NIST asked for an estimation of the performances on an 8-bit platform. This question was answered very differently by the 15 candidates. Some completed a full smart-card implementation, while others gave only general estimations of the consequences an 8-bit implementation would imply, or forgot the memory usage question, ...

Anyway, even the full implementations were performed on different smart cards, which makes performance comparison impossible.

In view of the increasing importance of smart cards in today's world, it appeared indispensable to perform an advanced, independent examination of the value of the candidates regarding smart cards.

This paper describes our experience in implementing four AES candidates on two very different smart cards. We hope it will be a good basis for direct comparison of these candidates, as well as a starting point for a more complete work in this area.

The next section presents the two smart cards we chose to work with and the reasons of our choice; we then briefly explain our implementation goals; section

5 describes in some details our implementation experience with each candidate; finally, section 6 summarizes the candidate's relative performances.

2 The smart cards

As an insight in smart card world, we decided to choose two very different platforms to port AES candidates to. The first one had to be a basic, low-cost device, typical of everyday life usage such as pay-TV application, etc; the second one was supposed to be representative of a powerful smart card, aimed at providing several services, among which security.

Below we briefly present the two smart cards we chose to use.

2.1 The 8051

The 8051 is a micro-controller developed by Intel at the time of the 8086. Many smart cards (Philips, ...) are based on this micro-controller. Its main characteristics are:

- Harvard architecture,
- 8 bit architecture,
- 34-1024 bytes of RAM. However, this RAM is organized in pages of 256 bytes, and accessing data located elsewhere than on the current page is a costfull operation. In this paper, we will thus limit ourselves to 256 bytes of available RAM, which is the most current case in actual smart cards,
- 2 KB or more of ROM,
- accumulator based architecture: data processing instructions specify only one operand; the other operand has to be stored in the accumulator before call, and this is also the place where the result will be put,
- 8 general-purpose registers (program counter, ... not included),
- depending on version, performance peek varying between 1 (at 12 MHz) and 2.5 (30 MHz) million 8-bit instructions per second; note, however, that the standard clock frequency to sample a smart card is 3.57 MHz,
- several addressing modes (immediate value, direct, indirect[1] addressing, bit-addressable zone, ...) supported. For many instructions, operating on memory is a fast as on registers
- CISC architecture: the 8051 disposes of a complex set of instructions. Some of them allow interesting spare of time (there is, for example, an instruction allowing to increment the value of one register without the need to pass through the accumulator; another instruction – decrement

[1] Note that only two of the registers can be used for indirect addressing

and jump if not zero – is very useful for efficient loops). Note, however, that not every addressing mode is available with every instruction type,

- variable-size instructions,
- byte-by-byte multiplier: the 8051 disposes of an instruction allowing to multiply two 8 bit values, yielding a 16 bit result (however, this instruction is not always implemented in hardware, and its execution time thus greatly varies among versions). A byte-by-byte divider is also available, but it was of no use in our work.

We believe the 8051 to be a good example – one of the most-used today – of a basic, low-cost, microprocessor-based smart card. The price of a smart card of course highly depends on many factors (amount of units needed, ...), but we estimate is at about US $ 2-4.

2.2 The ARM

ARM (Advanced Risc Machine) is a familly of processors developped by ARM Ltd. Some characteristics of these processors are:

- von Neuman architecture,
- 32 bit architecture,
- 4 GB of address space (of course, the memory actually available on a smart card is much smaller, typically 1 KB),
- 3-address instructions: every data processing instruction specifies both the operands and the destination place for the result. This allows much more efficient coding than for the 8051, where the first operand and destination are always forced to the accumulator,
- 16 registers, some of them with preassigned functions (program counter, stack pointer, ...),
- 17 MIPS sustained at 25 MHz,
- 3-stages pipeline,
- coprocessor interface (like UART (memory management coprocessor), ...),
- orthogonal instruction set: the ARM is a RISC machine, the instruction encoding size is therefore fixed; furthermore, the number of instructions is much smaller than on the 8051. On the other hand, these instructions are completely orthogonal: with a few exceptions, any operand can be used with any instruction,

- LOAD, STORE architecture: data processing instructions always operate either on direct values or on registers. Operating on memory thus requires pre- and post- transfer from and to memory. As these are rather slow operations, they should however be avoided (something that is helped by the big number of registers),

- conditional execution: every instruction can be conditionally executed according to the state of flags (carry, overflow, ...). Instructions that are not executed take a single clock cycle. Considering that every instruction can also decide whether to affect the flags or not, this feature allow very efficient coding of small "`if ...then ...else ...`" blocks, compared to the overhead of branch instructions[2] used on more conventional architectures,

- barrel shifter: the second operand in a data processing instruction can optionnaly be shifted or rotated before being processed. This feature also allows to code complex operations in very few instructions,

- multiplier: the ARM disposes of an instruction allowing to multiply two 32-bit values, yielding the 32 low-order bits of the product. There is a member of the ARM family – the ARM7M – which disposes of a multiplier yielding the full 64-bit result, but this feature did not appear usefull for our work, as every multiplication we implemented was modulus 2^{32}.

We believe the ARM to be a typical example of a current sophisticated smart card. Its price is estimated to about US $ 10.

3 Our objectives

Implementation implies to make various decisions. This section is an attempt to summarize them.

As for our design goals, they were, in decreasing order of importance:

1. *Low RAM usage:* the most sensitive resource on a smart card is certainly RAM. We saw that it is often reduced to 256 bytes on the 8051; it is a bit larger on the ARM, but, as this type of smart card is typically devoted to contain more than a simple encryption function, it would be a good thing not to use a too large part of the available RAM. We therefore decided to optimize the code towards RAM usage first, even if there is a speed drawback. For example, every time it was possible to compute round keys on-the-fly, we used this alternative.

2. *Speed:* of course, fast execution is one of the most important features we expect from a program, so, any decision not affecting RAM usage was taken in favour of speed.

[2] Mainly due to the stalls they introduce on the pipeline engine.

3. On a less extent, *code size:* ROM is not so scarce as RAM on a smart card, and we were therefore not as severe about code size than about RAM usage (some critical portions of the code, for example, were unrolled when the performance benefit was important). We simply tried to stay under reasonable limits, for example by preventing ourselves to unroll all and every loop and, when large (more than 1 KB) tables were necessary, to propose a non-table variant as well.

Flexibility was definely not one of our aims. For example, we do not expect a typical smart card application to have to support different key sizes; as 128 bits seems to offer nowadays a safe shield, we chose to implement this variant. In the same way, our goal was not to construct general building blocks that could be reused by other programs co-existing on the smart card and we therefore did not try to design general-purpose functions, nor to give them a standard interface; such features were always sacrificed in benefit of speed.

3.1 Other hypotheses

3.1.1 Scenario

Speed measurements of a cryptosystem depend on various parameters. For example, the amount of data to be encrypted will influence the relative importance of key schedule and encryption times : if the same key is to be used for a long time, then it may be worth storing the round keys in EEPROM rather than computing them on-the-fly.

In the scenario we are considering here, the secret key is likely to be changed often (we call this *key agility*). This is typical of a session key or of a system were several keys – depending on the interlocutor – are stored on the smart card. We therefore expect the system to be able to complete a key schedule as fast as possible, and prevent the use of non-volatile memory to store key-dependent values.

3.1.2 Endianness

The AES candidates were presented in either big or little-endian version (or both). This endianness question is of course irrelevant on an 8-bit architecture, as the programmer will have to implement it himself. One interesting feature of the ARM is that it can be configured on-the-fly to behave as a little or as a big endian machine. We therefore chose to implement the candidates in their native endianness.

We will not extend further on this consideration, as we do not believe it has a significant impact on performances.

3.1.3 Emulators

Our implementations were performed on emulators of the two smart cards. Therefore, actual results may slightly vary. We believe them to be although

rather accurate.

4 The candidates

After eliminating candidates for which significant flaws have already been discovered, we were left with a handfull of cryptosystems to implement. Somehow or other, we had to choose some of them to begin with. We chose six among those we believe to have a good chance to pass the first round. Two of these (Mars, Serpent) are subject of master's theses at UCL, and we hope their implementations to be completed soon. The results for the four others (E2, RC6, Rijndael, Twofish) are described here. Of course, implementation work continues; further results will be put on cAESar's page
(http://www.dice.ucl.ac.be/crypto/CAESAR/caesar.html).

5 Implementations

5.1 Implementing E2

5.1.1 8051-based smart card

E2 does not support on-the-fly key schedule and it is therefore necessary to allocate enough RAM space (256 bytes) for the round keys. This constraint makes E2 impossible to implement on the 8051, as well as on almost any low-cost smart card. For comparison purpose, we decided though to implement it using the external RAM to store the rounds keys. This is not a realistic implementation, as, to recover the key, it would suffice for an adversary to eyesdrop the wires connecting external RAM to the chip. We did this only to get an idea of the execution time.

Besides the extended key, 88 bytes of RAM were needed by our implementation. The original key is destroyed by the key extension process, so we have to add 16 bytes if we want to keep it for further use. Among the 88 bytes, there are 16 bytes for the message, and 72 which are needed by the function G (this is the size of the result of a single call to G). We therefore believe it is impossible to use less RAM. Other functions can easily reuse the space reserved for G.

The most time-consuming operation required by E2 is modular inversion modulus 2^{32}. As suggested by the authors, we have incorporated this operation in the key schedule (see [TC98] for more details); this does not slow down single block encryption and will greatly improve performances if more blocks are to be encrypted. Among the methods proposed by the authors to compute a modular inverse, the extended binary gcd [Knu97] appeared to be most efficient one, especially as, in the peculiar case where the modulus is a power of 2, this algorithm can be greatly improved: the outer of the two nested loops is always executed exactly once and, moreover, it is possible to consider only positive numbers[3]. The improved version of the algorithm is depicted in figure 1.

[3]This is especially interesting in this case as, if negative numbers had to be considered,

```
u = 1; v = 0;
for  k = 1 to  n
     if (u even) then
          u = u/2
          v = v/2
     else
          u = (u + x)/2
          v = (v + 2^n)/2
endfor
v = 2^n − v
return  v
```

Figure 1: Fast computation of $x^{-1} \bmod 2^n$

Our E2 implementation completes key schedule in 26147 clock cycles and encryption in 9725 cycles.

5.1.2 ARM-based smart card

256 bytes of RAM is not beyond ARM's capacities, and E2 can thus be implemented on it. There is not much to say about this implementation. Most of the comments of previous section remain valid (of course, the binary gcd is much easier to implement on a 32 bit machine !).

Our implementation performed the key schedule in 8712 cycles and encryption in 2180 cycles. The code size was 1004 bytes.

Several other tricks have been used in the implementation of E2, both on the 8051 and on the ARM, but we do not find it interesting to extend on them here. We refer the interested reader to the source code for further details.

5.2 Implementing RC6

5.2.1 ARM-based smart card

RC6 was beyond doubt the easiest candidate to implement on a 32 bits machine, as is illustrated by its incredibly short code (272 bytes).

On a speed point of view, RC6 is impressive too. The choice of operations show RC6 was carefully designed for good performance on a 32 bit CPU. Our implementation completes the key schedule in 3903 clock cycles, and an encryption in 790 cycles.

In view of the small code size, we fell allowed to try to unroll some more loops than usually, and do some tedious work in order to speed up the key

we would be forced to store variables in five bytes instead of four, which would significantly increase the overhead.

schedule; this "big" version of RC6 requires 460 bytes of code, and completes key schedule in 2231 clock cycles.

Unfortunately, the bad point about RC6 is that it does not allow on-the-fly key schedule, and thus requires rather much RAM: 176 bytes are necessary to store the extended key, to which we have to add 16 bytes to store the key if we want to preserve it; the message block can fit into registers.

A last point we would like to comment about RC6 is on rotations: as a rotate left x positions corresponds to a rotate right $32 - x$ popsitions, and in order to spare one instruction, the barrel shifter of the ARM only offers a rotate right instruction, However, RC6 performs its rotations left and, as the rotation amount is data-dependent, we loose two cycles per rotation to compute $32 - x \bmod 32$. Due to the big amount of data-dependent rotations performed by RC6, this two cycles difference has a big impact on global performance. On the other hand, decryption uses right rotations and is therefore faster than encryption on the ARM. This property should be taken into account for protocols where only one direction – encryption or decryption – has to be performed by the smart card.

Finally, note that the key schedule always uses rotations left, and is therefore handicapped on the ARM.

5.2.2 8051-based smart card

RC6 was mainly designed for 32 bits machine. It only relies on 32 bit operations (32 bit addition, multiplication, XOR and rotate).

Each operation must be decomposed on a 8 bits architecture. While the addition and the XOR can be, without too much overhead, made with 8 bits operation, the multiplication need more computations. With a little rewriting of the operations, we can transform this multiplication into a square which is easier to implement and faster to compute.

The last operation, the rotate, must be carefully designed. This is the most used operation in RC6 and a difficult one to implement with the instruction set of the 8051. The rotate instruction of the 8051 is only able to rotate a byte of one bit position (left or right). The original paper of RC6 [RSA98] describes some tricks to speed-up the implementation.

As stated for the ARM implementation, the key schedule cannot be computed on the fly. The additional RAM needed takes a large amount of the available memory on the 8051. This is the main drawback of the RC6 on systems with limited memory such as smart cards.

Designed for 32 bits operations, RC6 performs poorly on a 8 bits architecture. The key setup takes about 43000 cycles to be achieved and the encryption about 14500 cycles. The cycles given are message-dependent due to the rotate. These cycles must not be considered as absolute values but as indicative values.

One of the advantages ot RC6 is that it does not rely on tables. This leads to a compact code of less than 600 bytes.

5.3 Implementing Rijndael

5.3.1 ARM-based smart card

Rijndael is extremely efficient on the ARM. Regarding memory usage, it was possible to implement encryption, including on-the-fly key schedule, using only the registers of the smart card (as this process destroys the register copy of the key, some could prefer to say a Rijndael block encryption requires 16 bytes of RAM). This will of course have a positive impact on performance, as memory access is a rather slow operation on the ARM.

Regarding speed, an encryption (including key schedule) can be completed in as few as 2889 clock cycles. An interesting property of Rijndael is that, although it involves *in theory* multiplications in GF(2^8) - a rather costly operation -, these are in fact restricted to multiplications by x ('02'), which can be implemented by a shift followed by a conditional XOR, and by $x + 1$ ('03'), which reduces to the previous operation plus an additional XOR. These properties, together with the efficient structure of *MixColumn*, allow to implement this transformation very efficiently.

The rather big size of the code is partially due to the fact the implementation was done in registers. As the registers names must be fixed at compilation time, we had to use macros instead of functions. A more memory-consuming implementation would have much smaller code.

For even better performances, [DR98] suggests to use 1 KB of tables, which would allow to reduce the cost of one round to 16 table lookups (plus 16 ANDs and 16 shifts to isolate appropriate bytes), 12 rotations and 16 XORs. The code size grows a bit more[4], and we cannot avoid this time the use of 16 bytes of RAM, but the performance increase is worth the effort: encryption + key schedule are now completed in 1467 clock cycles!

An alternative, using 4 KB of tables, would give slightly better results, but the gain in speed appears this time rather small compared to the cost in ROM space.

5.3.2 8051-based smart card

Rijndael has been implemented on the 8051 by their authors themselves, and we therefore decided not to repeat their work. For comparison purposes we simply note that their implementation requires 49 bytes of RAM, and performs an encryption (including key schedule) in (depending on code size) between 3168 and 4065 clock cycles.

[4] Because this cannot be applied to the last round, which must still be computed in the standard way

5.4 Implementing Twofish

5.4.1 ARM-based smart card

Twofish also permits on-the-fly key schedule, and is therefore possible to implement with a very low RAM usage. Our implementation on the ARM required only 48 bytes of RAM: 4 words for the message block, 4 for the key and 6 to store the arrays M_o, M_e and S; the round keys were computed on-the-fly and stored in registers.

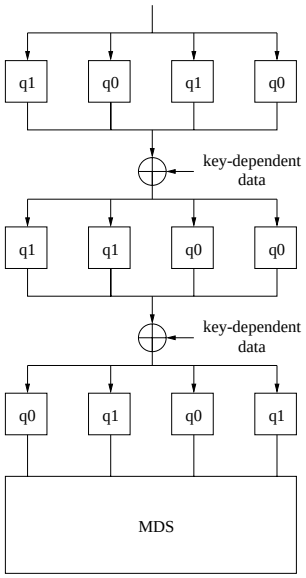

Figure 2: Twofish h function (128-bit version).

On the other hand, Twofish apperead to be much slower than RC6 or Rijndael on the ARM. A block encryption (including key schedule) required 8406 cycles to complete. This not-so-good performance may look surprising when compared to the excellent performances of Twofish on a standard 32-bit computer. It appears to be mainly due to the key-dependent nature of the S-boxes. The function h, whose structure is represented on figure 2, involves 12 passes through boxes $q_{0,1}$, followed by a matrix multiplication. [Sys98] proposes a way to speed up computations (*full keying*) by expanding each S-box to a 8-by-32 table that combines both the q_i boxes, the XORs and the multiply by the appropriate columns of the MDS matrix. However, as h is key-dependent, such a table would have to be stored in RAM[5], which is clearly impossible on a smart card. We are therefore left with another option, which is a compromize between

[5] An alternative, if a large amount of data is to be encrypted with the same key, would be to store such a table in EEPROM. This, however, contradicts the *key agility* hypothesis of our scenario.

the partial and zero keying alternatives of [Sys98]: we simply store (in ROM) a 4KB[6] table gathering MDS and the last "layer" of q_i, and implement other layers by independent table-lookups. But a table lookup (that is, a memory access) is a rather slow operation on the ARM, requiring three clock cycles to complete. Considering that h involves 12 table lookups and that this function is called 72 times we get a total of 2592 clock cycles devoted to table lookups, that is, almost one third of the total time.

Note that, if ROM space is scarce, it is possible to implement h without the 4KB of tables. Such an implementation completes a block encryption in 13662 cycles.

As for Rijndael, it is possible to implement the multiplication by matrix RS much more efficiently than by "dumb" matrix multiplication. We are gratefull to Dr. B. Gladman for this point. Dr. Gladman also proposes [Gla98] an efficient way to multiply by the matrix MDS, if the 4KB table is to be avoided. This method was used for the second version of our implementation.

5.4.2 8051-based smart card

Twofish behaves quite well on the 8051. Table-lookups are faster than on the ARM, and the key-dependent S-boxes, even implemented without the 4 KB table, are therefore no more a problem.

This implementation requires 68 bytes of RAM (the original key is not destroyed[7]), 1443 bytes of ROM, and performs an encryption (including key schedule) in about 25142 clock cycles. Note that the code size could be reduced at a small performance drawback.

6 Summary

The following tabulars summarize our results. The code size (in bytes) does not include decryption functions. The speed is in clock cycles. The "(+16)" notation means that 16 bytes of RAM must be added if the key is to be kept.

6.1 8051 implementations

Algorithm	Code size	Table size	RAM usage	Key setup time	Encryption time
E2	1188	256	344 (+16)	26147	9725
RC6	596	0	205 (+16)	43200	14400
Rijndael	512	256	49 (+16)	4065	
	760	256	49 (+16)	3168	
Twofish	931	512	68	24422	
	879	1024	48	18126	

[6] According to [Sys98], this table is only 1 KB large, but we believe it is a typo.

[7] We can, as preferred, recover it from the double-words M_o, M_e, or keep those in memory together with S, which will allow faster recomputation of the key schedule.

6.2 ARM implementations

Algorithm	Code size	Table size	RAM usage	Key setup time	Encryption time
E2	1004	256	336 (+16)	8172	2180
RC6	272	0	176 (+16)	3903	790
	460	0	176 (+16)	2231	790
Rijndael	1148	256	0 (+16)	2889	
	2620	1280	16 (+16)	1467	
Twofish	908	512	48	13662	
	696	4608	48	8406	

6.3 Achievable speeds

To get a more intuitive idea of what speeds can be achieved by the various candidates, we have translated the above data in terms of bytes per second. For each candidate, we have retained the fastest implementation, regardless of table sizes, memory usage, ... As we preferred, when possible, the implementations where the round keys are computed on-the-fly, and in order not to disadvantage candidates which allowed this choice regarding those which do not, the key schedule time is included for each block (in oher words, we are assuming here that the key changes for every block). The 8051 is supposed to be running at 3.57 MHz (6 oscillator periods per instruction); the ARM at 28.56 MHz (that is, we suppose the smart card is sampled at the standard frequence of 3.57 MHz and uses a clock multiplier).

Algorithm	8051 @ 3.57MHz	ARM @ 28.56 MHz
E2	267 bytes/sec[8]	44 142 bytes/sec
RC6	165 bytes/sec	151 260 bytes/sec
Rijndael	3005 bytes/sec	311 492 bytes/sec
Twofish	525 bytes/sec	56 289 bytes/sec

7 Conclusion

Regarding RAM usage, E2 is the only candidate to be ruled out for the 8051. All the other implementations are feasible, although RC6 is a rather big RAM consumer, which does not leave very much free space for other applications on a low-cost smart card. The very small RAM usage of Rijndael and Twofish (we have not completed Twofish implementation on 8051 yet, but we do not see any reason why it would need much more than the 48 bytes it needs on the ARM) clearly makes them front runners on this point of view.

Regarding speed, Rijndael is at no doubt the best candidate on both smart cards. On the ARM, it is followed by RC6, which is impressively fast too; then, but much slower, come Twofish and, finally, E2.

On the 8051, Rijndael has no competitive adversary (except maybe Twofish: we hope to complete its implementation soon and to answer this question at the

conference itself). E2, about ten times slower, comes in second position. RC6, apparently not designed for 8-bit processors, is this time the slowest candidate.

Note that the difference in performance between candidates is rather important, the ratio between the best and worst candidate being of about 1 to 7 on the ARM, 1 to 11 on the 8051.

That was for the first four candidates. In the future, we will put results for Mars, Serpent (and hopefully others !) on cAESar's page, at http://www.dice.ucl.ac.be/crypto/CAESAR/caesar.html.

8 Acknowledgements

The authors wish to thank Dr. Brian Gladman for his very efficient C implementation of Twofish and for his advices.

Note about the code: except for Twofish, were we used parts of Brian Gladman's code, and a few consultations (due to typos in reference paper) of Rijndael source code from its website, all the algorithms were coded from scratch on the basis of the candidates' description (AES CD-1).

References

[DR98] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. In *Proc. first AES conference*, August 1998. Available on-line from the official AES page: `http://csrc.nist.gov/encryption/aes/aes_home.htm`.

[Gla98] Brian Gladman. AES algorithm efficiency, 1998. Available on-line from `http://www.seven77.demon.co.uk/aes.htm`.

[Knu97] D.E. Knuth. *The art of computer programming*, volume Volume 2 Seminumerical Algorithms of *Computer science and information processing*. Addison-Wesley, 3rd. edition, 1997.

[Ltd97] Advanced RISC Machines Ltd. *ARM Software Developpment Toolkit version 2.11: User guide*. Advanced RISC Machines Ltd, 1997. Document number: ARM DUI 0040C.

[RSA98] RSA Laboratories. The RC6 block cipher. In *Proc. first AES conference*, August 1998. Available on-line from the official AES page: `http://csrc.nist.gov/encryption/aes/aes_home.htm`.

[Sem93] Philips Semiconductors. *80C51-Based 8-Bit microcontrollers: data handbook*. Philips, 1993.

[Sys98] Counterpane Systems. Twofish: A 128-bit block cipher. In *Proc. first AES conference*, August 1998. Available on-line from the official AES page: `http://csrc.nist.gov/encryption/aes/aes_home.htm`.

[TC98] Nippon Telegraph and Telephone Corporation. Specification of E2 - a 128-bit block cipher. In *Proc. first AES conference*, August 1998. Available on-line from the official AES page: `http://csrc.nist.gov/encryption/aes/aes_home.htm`.

[vSA93] Alex van Someren and Carol Attack. *The ARM RISC Chip: a programmer's guide*. Addison-Wesley, 1993.