Fast Software Implementations of SC2000

Helger Lipmaa

Laboratory for Theoretical Computer Science
Department of Computer Science and Engineering
Helsinki University of Technology
P.O.Box 5400, FIN-02015 HUT, Espoo, Finland
helger@tcs.hut.fi

Abstract. The block cipher SC2000 was recently proposed by a research group of Fujitsu Laboratories as a candidate cipher for the CRYPTREC and NESSIE projects. The cipher was designed so that it would be highly flexible and fast on many platforms. In this paper we show that the cipher is really fast on the Pentium III and AMD platforms: Our C implementation of SC2000 on Pentium III is only second to the best C implementations of RC6 on the same platform, and faster than for example the world fastest implementation of Twofish in assembly. In particular, we improve the bulk encryption and decryption times by almost 1.6 times as compared to the previous best implementation by Fujitsu. Finally, we report new Rijndael and RC6 implementation results that are slightly better than these of Aoki and Lipmaa.

Keywords: block cipher design, fast implementation, large S-boxes, SC2000.

1 Introduction

While Rijndael [DR02] has been recently approved as the new US governmental standard, Europe and Japan are in a quest for their own standards. SC2000 is one of the new block ciphers that was recently proposed by a research group from Fujitsu Laboratories [SYY+01] in the context of CRYPTREC in Japan and NESSIE in Europe. The cipher features many standard constructs of modern block ciphers, mixed in a way that makes it potentially fast in many different execution environments when using suitable implementation strategies. However, such design also makes the cipher more complex than some of the rivals.

An example design decision that makes it possible to code an implementation of SC2000 that takes into account the peculiarities of the target processor is the use of two 6-bit S-boxes and four 5-bit S-boxes. All six S-boxes can be accessed in parallel. One could then use the straightforward $(6,5,5,5,5,6)$-implementation for implementing SC2000 on low-end storage-constrained environments with only 256 different S-box entries. On high-end microprocessors with large cache size, one might instead want to use a $(16,16)$-implementation where six small S-boxes are combined into two large S-boxes. The number of table look-ups required by such an implementation would decrease by 48 per encrypted or decrypted block, while the S-box tables would have

$2^{17} = 131072$ entries. Another example of the flexibility of SC2000 is the B function that may be implemented without or with bit-slicing techniques, depending on the concrete processor.

On the other hand, although many strategic choices are available for the implementer it is not *a priori* clear whether SC2000 will perform well on any concrete platform. As an concrete example, the designers of SC2000 [SYY+01] have reported SC2000 implementations that at least on Pentium III are severely less efficient than corresponding implementations of Rijndael [DR02], RC6 [RRSY98], MARS [BCD+98] and Twofish [SKW+99], four of the five AES finalists. Complexity of SC2000 without any referable performance advantages was probably the main reason why this cipher did not pass to the second round of NESSIE.

We have chosen to implement SC2000 in the C language. For the sake of generality, we report the results of all implementations when compiled with gcc, icc or (when available) hand-coded in assembly on both Pentium III and AMD's Athlon. (See Table 1.) However, our implementations are specifically optimized for the $(16, 16)$-strategy for the icc 5.0 compiler and for the Pentium III. Chosen processors are sufficiently powerful to enable the usage of large S-boxes and to support efficient bit-slicing, but they are not as powerful as to make the resources used by our implementations unreasonable in other common processors. In particular this means that while the $(16, 16)$-implementation strategy might seem to be luxurious when using a Pentium III or an Athlon, it will become more realistic in the near future also in the commercial setting. Additionally, we focused our work on implementing the 128-bit key version of SC2000, SC2000-128, since this version seems to be commercially the most relevant for the foreseeable future.

We show that one can significantly improve upon the implementation results of [SYY+01]. On Pentium III, we report a pure C $(16, 16)$-implementation of SC2000 that is almost 1.6 times faster than the C implementation, and approximately 1.4 times faster than the combined C-assembly implementation of Fujitsu [SYY+01]. Such a large speed-up is mainly caused by the use of large S-boxes, and demonstrates the flexible design of SC2000. We also report fast implementations of SC2000 in different implementation strategies.

Additionally, we report fast implementations of Rijndael and RC6 that are somewhat faster than the ones described by Aoki and Lipmaa [AL00]. We then compare our implementations of Rijndael, RC6 and SC2000. On the Pentium III, our C implementation of SC2000 takes 270 cycles (or runs at 543 Mbit/s) that is faster than the best *assembly* implementation of Twofish on the same platform [AL00] and is only second to the assembly implementations of RC6 and Rijndael, reported in this paper. In particular, the only well-known block cipher that can be implemented faster in C on the same platform seems to be RC6. One of the main conclusions of this paper is that given current knowledge, it seems to be relatively easy to design fast (and secure) ciphers.

Road-map. In Section 2, we will give a short overview of the programming environment and previous best results. Then, in Section 3 we will describe our new implementations. There we will also give a summary of our final results. We conclude the paper with some recommendations, further work and acknowledgments.

Table 1. Comparison of Rijndael, RC6 and SC2000 implementations on Pentium III. N/A means that corresponding entry was not implemented. Enc means that the decryption key schedule is equal to the encryption key schedule. Subscript + means high optimization in terms of the spent time, with more pluses signifying better optimization

Cipher	Compiler	Encr.	Decr.	Key schedule	
				Encr.	Decr.
Rijndael	assembly	226^{+++}	239^{+}	N/A	N/A
	icc 5.0	430	453	185	319
	gcc 3.0.4	346^{++}	371^{++}	171^{++}	280^{++}
SC2000	icc 5.0	270^{++}	278^{++}	357^{++}	Enc
	gcc 3.0.4	269^{++}	376	357	Enc
RC6	assembly	219^{+}	205^{+}	N/A	N/A
	icc 5.0	257	289	1436	Enc
	gcc 3.0.4	234^{+}	265	1778	Enc

2 Environment and State-of-the-Art

2.1 Description of Target Processor

We decided to optimize our implementations for one concrete widely used processor but also test them on at least on another one. At the time of writing this paper mainstream computers shipped at least three different processor families that, while intercompatible, were internally sufficiently different to make it necessary to use specialized implementation strategies to achieve the best performance: Namely, Intel's P6 family (of which the Pentium III was the most high-end representative), Intel's P7 family (mainly, Pentium 4), and AMD's Athlon family.

We chose the P6 family (most precisely, the Pentium III) mostly because of the wide availability of comparison materials: For example, the results of Aoki and Lipmaa [AL00] on implementing four of five AES finalists can be directly compared to ours. Our main development and target processor was a 1200 MHz Pentium III Mobile that was used in high-end laptops at this point. (The performance of our implementations on Pentium III Mobile and Pentium III (Katmai) was the same, so we will mostly omit the word 'Mobile' in the next.)

A relevant introduction of the Pentium II for cipher designers and implementers is given by Aoki and Lipmaa [AL00]. The main difference of Pentium III, as compared to Pentium II (namely, inclusion of several new multimedia instructions) has no relevance to our work: Our implementations are coded in C, and neither the Gnu C or the Intel C Compiler generated any MMX technology instructions at all. As opposed to that, a quite relevant change is the increase in size of cache to 512 KB. Larger cache size made it feasible to use the $(16, 16)$-implementation strategy (see Section 3.1). On the other hand, increased cache size has not benefitted other block ciphers as greatly: our best implementations of AES finalists on the Pentium III, while being slightly faster, take approximately as much times as the implementations of Aoki and Lipmaa on the Pentium II [AL00]. (This can be seen from Table 1.) The main reason behind that seems

to be the fact that, as a rule, other block ciphers do not have specific implementation strategies that require $256 \ldots 512$ KB of memory.

In Section 3.2 we will also describe our results of implementing SC2000 on AMD's Athlon processor. The concrete Athlon had 256 KB cache and ran at 1400 MHz. Since we did not make any specific effort of optimization for Athlon—we stress that one could get a better performance on Athlon, after spending more time on implementation—, we will omit description of this processor family.

2.2 Description of General Environment

We have coded our implementations under the Linux operating system, by using two different compilers, gcc 3.0.4 (Gnu C Compiler) and icc 5.0 (Intel C Compiler). The choice of operating system was partially due to the envisioned scenario of using our implementations in (say) routers, firewalls and servers that often run a version of Linux. Such machines usually do not serve as workstations, they instead execute a few specialized tasks. Therefore, in such machines memory usage of $512 \ldots 600$ KB for one particular cipher seems to be quite reasonable if it results in 30% win in throughput. We also chose to use a high-level language with a highly optimizing compiler so as to be able to simply port our implementations to other machines and on the other hand, not to loose seriously in throughput compared to implementations coded in the assembly language.

More precisely, we tested our results on two different machines, one having a 1200 MHz Pentium III Mobile, 512 KB cache, 128 MB RAM, both gcc 3.0.4 and icc 5.0 compilers, and Linux 2.4.18 operating system. The second machine had a 1400 MHz Athlon, 256 KB cache, 1 GB RAM, gcc 3.0.3 compiler and Linux 2.4.17 operating system. The gcc compiler was used with flags

```
-O4 -fomit-frame-pointer -mcpu=XXX -march=XXX
-D__OPTIMIZE__ -fexpensive-optimizations
-funroll-loops -mpreferred-stack-boundary=2
```

where XXX corresponded to the processor type (pentiumpro or athlon). The icc compiler was used without any explicit optimization flags since their usage did not result in any gain in performance.

2.3 Overview of Related Results

As far as we know, the only optimized implementation of SC2000 on the Pentium III thus far is by the designers [SYY+01]. Their results, together with information from [Shi02], is summarized in Table 2.

Aoki and Lipmaa [AL00] implemented several AES finalists on the Pentium II, and described thoroughly their timing methods, implementation criteria (e.g., no self-modifying code), etc. We follow their sensible guidelines. This allows precise comparison of our results to theirs. We refer to [AL00] for description of the timing subroutines and other background information. Appendix A contains a short overview of the used time measurement procedures.

Table 2. The previously best implementations of SC2000-128 [Shi02], compiled with VC++ 6.0 and tested under Microsoft Windows

Processor	Strategy	Language	Encr.	Decr.	Key schedule
Pentium III (Katmai)	$(6, 10, 10, 6)$	C+assembly	383	403	427
		C	412	424	433
	$(11, 10, 11)$	C+assembly	525	535	488
		C	554	549	488
Athlon	$(6, 10, 10, 6)$	C+assembly	392	402	426
		C	383	402	426
	$(11, 10, 11)$	C+assembly	318	329	373
		C	337	358	373

3 Main Contributions

3.1 Choice of Implementation Strategy

General Strategy. We omit the full description of SC2000 and refer to [SYY$^+$01] instead. Our implementation and its strategy, described in this subsection, follows somewhat loosely the recommendations given in that paper about implementing SC2000 on 32-bit processors.

Cipher state. Cipher state consists of four 32-bit variables that are initialized by the plaintext and then get modified by the B, $\mathsf{B}^{(i)}$, I, M and R functions.

S-boxes. From numerous available possibilities of combining the S-boxes we chose the variant $(16, 16)$: That is, the case with two S-boxes that both contain 2^{16} elements, all elements being 32-bit integers. The total storage needed by such S-boxes, 512 KB, does not seem to be prohibitive in the target environment. However, when indeed one must save the memory, one can use also the $(11, 10, 11)$-implementation strategy when S-boxes have been partitioned into 11, 10 and 11-bit S-boxes. In this case, the S-boxes require only 20 KB of storage space, but the resulting implementation will also be 30% slower. Several intermediate strategies are possible, the most natural ones are summarized in Fig. 1. We implemented all four strategies.

Function M. We have chosen the strategy, outlined already in [SYY$^+$01], to combine the M function and S-boxes into single table. Therefore, M function does not add any additional overhead to the implementation, compared to the S-boxes alone. Indeed, our implementation features a function called SM.

Function R. Implementing R is straightforward, since it only includes two calls to the SM function and a few Boolean operations. There was basically no choices to be done while implementing this function.

Strategy	Look−ups	#Cells
6 | 5 | 5 | 5 | 5 | 6	6	256
6 | 10 | 10 | 6	4	2176
11 | 10 | 11	3	5120
16 | 16	2	131072

Fig. 1. Different implementation strategies for the S-boxes, with the number of table look-ups per SM function and the total number of elements in all S-boxes

Function I. Function I consists of XOR-ing of a part of the key schedule to the internal state of the function. An implementer has almost no freedom in optimizing this subroutine either.

Function B. We chose to implement B by using bit-slicing techniques, as also suggested in [SYY+01], but improved considerably upon the example code given in [SYY+01]. Our code for B function consists of 20 instructions that belong to one of the next five available primitive operations of the target processor: `a=b`, `a=~b`, `a|=b`, `a^=b` and `a&=b`. Our implementation of B also uses three temporary registers.

Function $B^{(i)}$. We also use bit-slicing to implement the function $B^{(i)}$. Our code for $B^{(i)}$ function consists of 23 instructions and uses four temporary registers. As we see later, the difference in the complexities of B and $B^{(i)}$ functions seem to be relevant under the gcc compiler.

Key schedule. Our key implementation proceeds first by creating intermediate keys and then the final key. Both parts are relatively straightforward. During creation of intermediate keys, one has to apply the SM function 16 times on different inputs; this part also includes some Boolean operations. During creation of the extended key schedule, we invoke a subroutine EKEY 56 times, where every invocation of EKEY consists of four table look-ups and a few simple instructions.

Encryption During encryption, we first invoke the "encryption meta-round" function $\phi_i = R_i \circ R_i \circ I \circ B \circ I$ six times, and then apply $I \circ B \circ I$ to the result. Here, by $\alpha \circ \beta$ we denote the serial composition of first β and then α; R_i denotes the R function with constant $\texttt{0x111111} \cdot (3 + 2 \cdot (i \mod 2))$. More precisely, every invocation of I

feeds a new part of the extended key schedule to I, and the four elements in the state of the cipher are permuted after each component function. The final state is stored as the ciphertext. Permuting is implemented by inputting a different permutation of the four state variables as arguments to different subroutines and therefore permuting is free. Permutations are chosen so as to minimize the amount of mov-type instructions in different functions.

Decryption During decryption, we first invoke the "decryption meta-round" function $\psi_i = \mathsf{R}_{i+1} \circ \mathsf{R}_{i+1} \circ \mathsf{I} \circ \mathsf{B}^{(i)} \circ \mathsf{I}$ six times, and then apply $\mathsf{I} \circ \mathsf{B}^{(i)} \circ \mathsf{I}$ to the result. Therefore, our code of decryption looks very similar to our code of encryption, except of the exchange in indexes of R functions and the replacement of B with $\mathsf{B}^{(i)}$. (Also, the permutations of $\mathsf{B}^{(i)}$ function are different of the permutations of B function.) Thus, in all our implemented strategies, the encryption and decryption routines have exactly the same complexity except that there are 3 additional instructions in every $\mathsf{B}^{(i)}$ function (and one additional temporary register in use) as compared to the B function.

3.2 Summary of Results

We implemented the SC2000 by using four possible strategies and compared the results obtained when using two different compilers (gcc and icc) on two different platforms (the Pentium III and Athlon). Summary results are given in Table 3 and in Table 4. We specifically optimized the $(16, 16)$-implementation for Pentium III under the icc compiler (the first row in Table 3), and no specific effort was made in optimizing results in any other row. If such effort will be made, numbers in many fields will most probably decrease significantly. We specifically expect that significantly improved performance can be achieved on Athlon.

Our results indicate that the above-mentioned difference between the B and $\mathsf{B}^{(i)}$ functions is substantial in the case of gcc that runs short of registers in decryption routine: This can result in decryption being almost twice slower. On the other hand, when icc is used, encryption and decryption will have almost identical timings. Thus, the small number of integer registers of the target processors makes implementations extremely sensitive to proper allocation of registers.

The Intel C Compiler tends to optimize better the key schedule algorithm, while the superiority of one compiler over the another one in producing better encryption code seems to depend very much on the concrete implementation strategy. From the implementation strategies, $(16, 16)$ seems to yield the best throughput on the Pentium III, while $(11, 10, 11)$ suits better the Athlon. The difference is caused mainly by the fact that the Pentium III Mobile and Katmai feature 512 KB cache, twice as much as the target Athlon. Really, when using the $(16, 16)$-strategy, the S-boxes have size 512 KB and therefore, when using the $(16, 16)$-strategy, accessing the S-boxes produces many cache misses on Athlon.

On the other hand, this problem with the $(16, 16)$-strategy non-withstanding, Athlon seems to be a slightly better processor than the Pentium III. Our implementations (except the $(16, 16)$-implementation) run faster on the Athlon even if no special optimization was made for Athlon! More precisely, icc-compiled implementations had often bet-

Table 3. Our SC2000 implementation results in the table form (cpb=cycles per block)

Processor	Strategy	Compiler	Encr. (cpb)	Decr. (cpb)	Key sch. (cpb)
Pentium III	$(16,16)$	icc 5.0	270	277	356
		gcc 3.0.4	269	489	359
	$(11,10,11)$	icc 5.0	349	356	427
		gcc 3.0.4	452	561	501
	$(6,10,10,6)$	icc 5.0	409	414	483
		gcc 3.0.4	388	609	511
	$(6,5,5,5,5,6)$	icc 5.0	521	527	519
		gcc 3.0.4	503	824	665
Athlon	$(16,16)$	icc 5.0	362	381	280
		gcc 3.0.3	374	543	309
	$(11,10,11)$	icc 5.0	319	327	361
		gcc 3.0.3	312	507	392
	$(6,10,10,6)$	icc 5.0	413	376	404
		gcc 3.0.3	366	606	438
	$(6,5,5,5,5,6)$	icc 5.0	471	478	427
		gcc 3.0.3	463	843	534

ter timings on Athlon than gcc-compiled implementations on Athlon (or icc-compiled implementations on Pentium III), even if icc does not optimize for Athlon! When the next versions of gcc will be properly tuned to perform heavy Athlon-specific optimizations, we can definitely expect to see our performance numbers to improve on Athlon.

3.3 Implementations of Rijndael and RC6

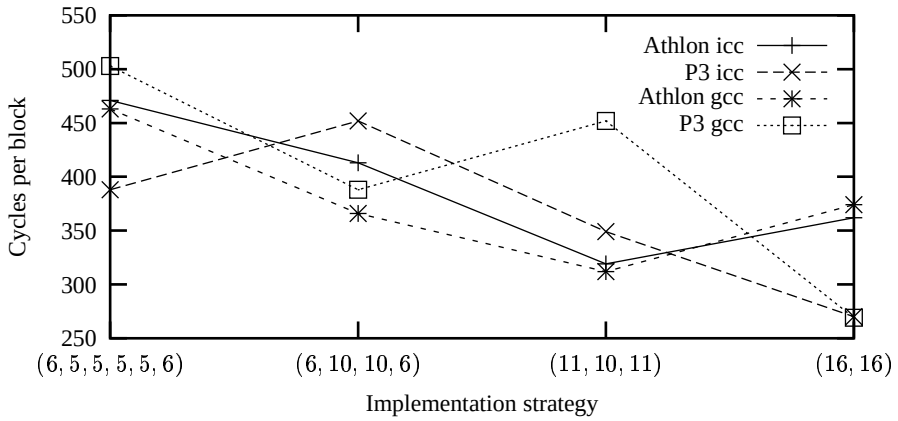
We have also implemented Rijndael, the new AES, and RC6 for both Pentium III. These implementations are slightly better than the implementations of Aoki and Lipmaa [AL00] for Pentium II. For example, instead of 237 cycles, our Rijndael encryption takes 226 cycles. Also, instead of 223 cycles, our RC6 encryption takes 219 cycles. We also implemented the key schedule algorithms of Rijndael, although in C, and achieved very good results. (The Rijndael key scheduling implementation that was available for the authors of [AL00] was substantially slower.)

4 Conclusions and Further Work

4.1 Conclusions

Following Section 3.2, we can conclude that from the depicted choices, a Pentium III running the icc-compiled $(16,16)$-implementation of SC2000 would be the best one when only raw throughput is important. If, additionally, the available memory is constrained, one might consider switching to Athlon and to the $(11,10,11)$-strategy. This,

Table 4. Our SC2000 implementation results as a bar graph. It is clearly seen that the $(16, 16)$-implementation strategy is more fit for the Pentium III than for the Athlon



in particular, demonstrates the importance of the cache size in software implementations.

Our results show that after additional scrutiny from the security viewpoint, the SC2000 might be a serious contender in the block cipher arena just because of excellent throughput in different environments, as shown by this paper and by [SYY+01]. (We however stress that we did not analyze the security of SC2000 at all.) Based on Table 1, on Section 3.3 and on [AL00], our C implementation of SC2000 is slower than the best assembly implementations of Rijndael and RC6, while being faster than the best C implementations of Rijndael and the best assembly implementations of Twofish and MARS [AL00] on Pentium III. This leaves us wondering how fast SC2000 could be in assembly, and also lets us to conclude that nowadays it is possible to create surprisingly fast ciphers.

Our improvement, 1.6 times in encryption speed compared to the previous best is quite significant. However, we used significantly more memory than the previous implementations and therefore the implementations are not directly comparable. It is an interesting open question how efficiently can one implement (say) Rijndael, given similar amount of memory—512 KB—for internal storage.

Note that in many environments, the large S-box tables can be stored in ROM. Since ROM is potentially both cheaper and faster than RAM, this would even more decrease the price per performance ratio. Alternatively, one can create the 16-bit S-box tables from a small seed of $\approx$ 128 elements. (The latter strategy was used in our implementations.)

4.2 Further Work

We did not implement SC2000 in assembly, but nevertheless we made some observations. While both used compilers seemed to generate a very good code, a few things can be certainly improved in assembly:

1. Use of MMX technology would benefit in at least two ways: First, it would make more internal registers available to the cipher implementer. This would eliminate some memory accesses. Second, the MMX technology features the nand instruction that could be used to somewhat speed up the implementations of B and $\mathsf{B}^{(i)}$.
2. Better register allocation: Even without using the MMX technology, one could reduce memory accesses by re-allocating some of the internal variables to correspond to registers.
3. Advanced optimization: Do complete optimization that would take into account trade-offs between different stages of the work of Pentium III, as described, say, in [AL00]. This would include careful instruction reordering, manipulating the length of individual instructions, etc.
4. Better bit-slicing: our implementations of B and $\mathsf{B}^{(i)}$ might well be suboptimal. To achieve a better throughput, one might have to write a brute force program for generating (at least heuristically) close-to-optimal implementation of B and $\mathsf{B}^{(i)}$. Here, such an implementation should both have a small number of Boolean operations but also a very small number of temporary registers. Most likely, optimal implementation depends on the processor family, chosen strategy but also on the compiler.

The fourth item in this list, finding better bit-slicing code, is also an interesting research question by itself, and does not only concern SC2000 but also some other ciphers like Serpent [ABK98].

Acknowledgments

We are thankful to Masahiko Takenaka, Takeshi Shimoyama and anonymous reviewers for useful comments. This work was partially supported by Fujitsu Laboratories. Our results were first presented at ISEC 2002 [TLT02].

References

[ABK98] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Flexible Block Cipher With Maximum Assurance. In *The First Advanced Encryption Standard Candidate Conference*, Ventura, California, USA, 20–22 August 1998.

[AL00] Kazumaro Aoki and Helger Lipmaa. Fast Implementations of AES Candidates. In *The Third Advanced Encryption Standard Candidate Conference*, pages 106–120, New York, NY, USA, 13–14 April 2000. National Institute of Standards and Technology. Entire proceedings available from the conference homepage `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3conf.htm`.

[BCD+98] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko Zunic. MARS — A Candidate Cipher for AES. Available from `http://www.research.ibm.com/security/mars.html`, June 1998.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael. AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.

[RRSY98] Ronald L. Rivest, Matt J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 Block Cipher. Available from `http://theory.lcs.mit.edu/~rivest/rc6.ps`, June 1998.

[Shi02] Takeshi Shimoyama. Personal communication. April 2002.

[SKW+99] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*. John Wiley & Sons, April 1999. ISBN: 0471353817.

[SYY+01] Takeshi Shimoyama, Hitoshi Yanami, Kazuhiro Yokohama, Masahiko Takenaka, Kouichi Itoh, Jun Yajima, Naoya Torii, and Hidema Tanaka. The Block Cipher SC2000. In Mitsuru Matsui, editor, *Fast Software Encryption '2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 312–327, Yokohama, Japan, 2–4 April 2001. Springer-Verlag, 2002.

[TLT02] Masahiko Takenaka, Helger Lipmaa, and Naoya Torii. The Implementation of The Block Cipher SC2000 (III). In *ISEC 2002*, Tohoku University, Sendai, Japan, 18–19 July 2002. In Japanese.

A Timing

We use exactly the same convention of measuring the time as in [AL00] and is described in Fig. 2. The inputs and key of the cipher are generated randomly before the measurement. The input variable `lenBlk` was chosen to be equal to 8000. Also, `time` is a work area of type `uint32`, used in later calculations.

We would get some overhead when outputting the result of this function alone due to both high latency of the `rdtsc` instructions and also the overhead caused by looping instructions like `jnz` which are not formally part of the cipher itself. We measure this overhead by using the null function shown in Fig. 3 obtaining a value `nulltime`, and then we subtract it from the value of `time` obtained by measuring the speeds of different encryption/decryption procedures. Finally, this result is divided by the number of blocks encrypted. Intuitively, by using this method we obtain the number of cycles corresponding to unrolled implementation of the block cipher, or to the implementation where we only care about the time it takes to encrypt one block, without adding any extra overhead. The subtracted overhead number was equal to ≈ 6 on the Pentium III and ≈ 7 on the Athlon in the case $n = 8000$. One could just add this number to those presented later to get the number of cycles *with* overhead.

Finally, we did a loop of 500 times over the described measurements and then chose the smallest number for every cipher, since that corresponds most likely to the case where most of the data and code are in L1 cache and the branch prediction works successfully: i.e., to the bulk encryption speed of the cipher itself.

```
movd mm0, dword ptr [time];  /* warm cache and set MMX state */
xor eax, eax;
cpuid;                       /* serialize instructions */
rdtsc;                       /* read time-stamp counter */
mov dword ptr [time], eax;   /* save counter */
xor eax, eax;
cpuid;                       /* serialize instructions */
/* call to xxEnc() or xxDec() */
xor eax, eax;
cpuid;                       /* serialize instructions */
rdtsc;                       /* read time-stamp counter */
sub dword ptr [time], eax;   /* compute the difference */
emms;                        /* empty MMX state */
```

Note that `time` is a 4 bytes work area.

Fig. 2. Time measurement code

```
   /* push all used registers */
   cmp dword ptr [lenBlk], 0;
   jz L1;
   align 16;
L0:
   dec dword ptr [lenBlk];
   jnz L0;
L1:
   /* pop these registers once more */
```

Fig. 3. Null function