NISTIR 6391

Efficiency Testing of ANSI C Implementations of Round 1 Candidate Algorithms for the Advanced Encryption Standard

Lawrence E. Bassham III

U.S. DEPARTMENT OF COMMERCE
Technology Administration
Computer Security Division
Information Technology Laboratory
National Institute of Standards
and Technology
100 Bureau Drive
Gaithersburg, MD 20899

NIST

NISTIR 6391

Efficiency Testing of ANSI C Implementations of Round 1 Candidate Algorithms for the Advanced Encryption Standard

Lawrence E. Bassham III

U.S. DEPARTMENT OF COMMERCE
Technology Administration
Computer Security Division
Information Technology Laboratory
National Institute of Standards
and Technology
100 Bureau Drive
Gaithersburg, MD 20899

October 1999

U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

TECHNOLOGY ADMINISTRATION
Gary R. Bachula, Acting Under Secretary
for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Raymond G. Kammer, Director

1. Introduction

The evaluation criteria for the Advanced Encryption Standard (AES) Round1 candidate algorithms, as specified in the "Request for Comments" [1], includes computational efficiency, among other criteria. Specifically, the "Call For AES Candidate Algorithms"[2] required both Reference ANSI[1] C code and Optimized ANSI C code, as well as Java™[2] code. Additionally, a "reference" hardware and software platform was specified for testing. NIST performed testing on this reference platform, as well as several others. Candidate algorithms were tested for computational efficiency using the Optimized ANSI C source code provided by the submitters.

This paper describes the testing methodology used in ANSI C efficiency testing, along with observations regarding the resulting measurements. Conclusions are provided regarding which algorithms have the most consistent performance across different platforms. This paper also includes an appendix containing tables of timing and cycle counting values obtained from testing the algorithms. Some knowledge regarding compilation and processor architectures is useful in understanding how the data was derived. However, the raw data in the document can also be useful without necessarily understanding how it was derived.

2. Scope

Performance measurements were taken on multiple platforms. These measurements were analyzed to determine the general rankings of the candidate algorithms with respect to one another. At this point in the AES development effort, NIST is not interested in the absolute value of the performance measurement, but in the relational value of one algorithm's speed when compared with the rest. From an efficiency point of view, NIST does not intend to rank one algorithm as "better" because it is relatively faster than another algorithm by .5%. However, if one algorithm was faster than another algorithm by 50%, then that would be considered a significant difference. NIST is interested in finding the consistent "top performers" on the test platforms by analyzing the performance data for the algorithms and observing natural breaks.

3. Methodology

In the "Call for AES Candidate Algorithms" [2], NIST cited a specific hardware and software platform as the "NIST Analysis Platform" (referred to in this document as the "reference platform") for testing candidate algorithms. This platform consists of an IBM-compatible PC with an Intel® Pentium® Pro™ Processor, 200 MHz-clock speed, 64MB RAM, running Microsoft® Windows® 95, and the ANSI C compiler in the Borland® C++ Development Suite 5.0. Performance measurements were taken on this platform and a large number of additional hardware and software platform combinations. The platforms tested are detailed in Table 1.

---

[1] ANSI – American National Standards Institute

[2] Certain commercial products are identified in this paper. In no case does such identification imply recommendation or endorsement by the National Institute of Standards an Technology, nor does it imply that material identified is necessarily the best for the purpose.

Performance measurements were conducted in two different ways. The first method of testing performance deals with determining the amount of time required to perform cryptographic operations (e.g., how many bits of data can be encrypted in a second, or how many keys can be setup in a second). This type of test is referred to as a "Timing Test" in this document. The second method of testing performance deals with counting the number of clock cycles required to perform cryptographic operations (e.g., how many cycles are consumed in encrypting a block of data, or how many cycles are consumed in setting up a key). This type of test is referred to as a "Cycle Count Test" in this document.

The Timing Tests utilized the `clock()` timing mechanism in the ANSI C library to calculate the processor time consumed in the execution of the API call and underlying cryptographic operation under test (i.e., `makeKey()`, `blockEncrypt()`, and `blockDecrypt()`). The time consumed to perform a particular operation was then used to calculate the bits/second or keys/second speed measure. The Cycle Count Tests counted the actual clock cycles consumed in performing the operation under test (for more information on counting clock cycles see [3]). Because cycle counting utilizes assembly language code in the testing program, interrupts could be turned off during testing[3]. This results in a very accurate measure of the performance of the API calls and the underlying cryptographic operations. Additionally, cycle counting eliminates the variability of the processor speed. The same number of clock cycles are

Table 1: System Platforms (Hardware/Software) and Compilers Used in Efficiency Testing

Processor/Hardware	Operating System	Compiler
200MHz Pentium Pro Processor, 64MB RAM	Windows95	Borland C++ 5.01 (time&cycles)
		Visual C++® 6.0 (time&cycles)
		DJGPP – gcc version pgcc – 2.90.23 980102 (egcs-1.0.1)
	Linux	GCC 2.8.1
450MHz Pentium II Processor, 128 MB RAM	Windows98 4.10.1998	Borland C++ 5.01 (time&cycles)
		Visual C 6.0 (time&cycles)
		DJGPP – gcc version pgcc – 2.90.23 980102 (egcs-1.0.1)
500MHz Pentium III Processor, 128 MB RAM	Windows98 4.10.1998	Borland C++ 5.01 (cycles only)
		Visual C 6.0 (cycles only)
Sun™: 300MHz UltraSPARC-II™ w/ 2MB Cache, 128 MB RAM	Solaris™ 2.7 (a 64 bit operating system)	GCC 2.8.1
		Sun Workshop Compiler C™ 4.2
		(Workshop Professional C™ 5.0) – 64 bit executables in Round2
Silicon Graphics™: 250MHz R10000™ w/ 2MB Cache, 512 MB RAM	IRIX64™ 6.5.2 (a 64 bit operating system)	GCC 2.8.1
		(MIPSpro C Compiler) – 64 bit executables in Round2
Sun: 2*360MHz UltraSPARC-II w/ 4MB Cache, 256 MB RAM	Solaris 2.7	GCC 2.8.1
		Sun Workshop Compiler C 4.2

[3] Interrupts occur, for example, when the operating system decides it needs to perform some action unrelated to the process that is running. If an interrupt were to occur during cycle count testing, the time spent performing the operating system activity would be included in the time spent on the cryptographic operation. This would lead to inflated and erroneous values for the cycles necessary to perform the cryptographic operation.

required to perform an operation on a 300 MHz Pentium II processor as it does on a 450 MHz Pentium II processor; there are simply more clock cycles in a second on a 450 MHz-based system. Cycle counting could only be performed on the Intel processor based systems. This is the only processor used by NIST during Round1 testing that provides access to a true cycle counting mechanism.

3.1 Timing Program

For each key size required by [2] (128 bits, 192 bits, and 256 bits) three values are calculated:

- The time to make 65536 calls to `blockEncrypt()` (65536calls*128bits/call=8388608 bits = 1Mbyte = 8 Mbits)[4];
- The time to make 65536 calls to `blockDecrypt()` (65536calls*128bits/call=8388608 bits = 1Mbyte = 8 Mbits); and
- The time to setup 1000 key pairs (a key pair consists of one encrypt key and one decrypt key).[5]

The test program generates 1000 triples (i.e., sets of timing information) as described above for each key size. The time values in each category are then sorted, and the median value is determined. A standard deviation is calculated for each test category. Finally, the average of all values that fall within three standard deviations of the median is determined. This value is the reported average time to perform the specific operation (encrypt, decrypt, or key setup) for a particular key size. Using these time values, speed values of Kbits/sec are calculated for encryption and decryption, and keys/sec is calculated for key setup. Time values in this test program are calculated around the NIST API calls. Results for the Timing Program can be found in Appendix A.1. Pseudo code for the generation of timing information for the `blockEncrypt()` operation is included in Figure 1.

```
(r=0; r<1000; r++);
      makeKey();
      cipherInit();
      (Start Timer)
      (i=0; i<256; i++);
            (j=0; j<256; j++);
            blockEncrypt(1 block);
      (Stop Timer)
```

Fig. 1: Pseudo code for Time Testing for `blockEncrypt()`

3.2 Cycle Counting Program

For each key size required by [2] (128 bits, 192 bits, and 256 bits) four values are calculated:

[4] More blocks of data were encrypted or decrypted on some of the faster algorithms in order to get a better resolution on the timer. The timer has a resolution of 5 msec. If the timer reports values of 5 or 10 msec (100% change) it is not as useful as having it report values of 55 or 60 msec (9% change).

[5] Due to the speed of the computation of keys, only 100 pairs of keys were setup for FROG. Due to memory constraints and speed, only 100 pairs of keys were setup for HPC.

- The number of cycles needed to setup a key for encryption;
- The number of cycles needed to setup a key for decryption;
- The number of cycles needed to encrypt block(s) of data; and,
- The number of cycles needed to decrypt block(s) of data.

These values were measured by placing the CPUID and RDTSC assembly language instructions around the NIST API. These instructions were called twice before the cryptographic operation to "flush" the instruction cache (see [3, §3.1]). Additionally, the CLI and STI instructions were used to switch interrupts off before the test and back on after the test. This eliminates extraneous interrupts that would skew results. Analysis of this data was performed in the same way as the timing program listed above in Section 3.1 (calculation of standard deviation, median, etc.) Results for the Cycle Counting Program can be found in Appendix A.2. Pseudo code for the generation of cycle counting information for the `blockEncrypt()` operation is included in Figure 2.

```
(r=0; r<1000; r++) {
      makeKey();
      cipherInit();
      cli;                /* Clear Interrupt Flag  */
      cpuid;              /* Clears instruction cache  */
      rdtsc;              /* Read Time Stamp Counter  */
      save counter;
      blockEncrypt();     /*  Perform operation being timed  */
      cpuid;
      rdtsc;              /* Read Time Stamp Counter  */
      subtract counter;
      save counter
      sti;                /* Set Interrupt Flag  */
      }
```

Fig. 2: Pseudo Code for Cycle Counting for blockEncrypt()

The Cycle Counting Program was run several times with different lengths of data for encryption and decryption to determine if size had any effect on the `blockEncrypt()` and `blockDecrypt()` speeds.

4. Observations

4.1 General

Some of the algorithms use flags to determine which compiler is used. By checking which compiler is used, an algorithm may substitute commands that direct the compiler to insert code to make use of instructions available on the CPU. The most common example of this is the use of the ROTL and ROTR instructions to perform left and right logical rotations, respectively. Using the machine instruction to perform these rotations is two cycles faster than performing the equivalent sequence of using a pair of shifts and an OR operation. This can provide a performance enhancement on various compilers that other algorithms do not enjoy because they do not perform this type of compiler dependent compilation. The Borland compiler does not

make use of the machine instructions of ROTL and ROTR. The Visual C compiler can make use of the machine instructions by using the routines `_rotl()` and `_rotr()` to perform the rotation. The DJGPP compiler will detect that the two shift operations and the OR are being used and will automatically substitute the ROTL or ROTR machine instructions during compilation.

The Timing Tests Program was not used to perform a timing test on the `cipherInit()` function. The only performance evaluation of this function was performed with the Cycle Count Program on the reference platform. None of the algorithms perform pre-computation in the `cipherInit()` function except LOKI97 and MAGENTA. Additionally, performance measurements for all algorithms were done using the Electronic Codebook (ECB) mode. In this mode, some algorithms convert the Initialization Vector (IV) from ASCII to binary even though the IV will not be used[6]. This makes the timing test for `cipherInit()` extremely variable, based on whether the conversion was performed or not. The timing test for `cipherInit()` essentially becomes a test of the API overhead instead of the cryptographic algorithm. As a result, cycle counts for `cipherInit()` are only included for the reference platform.

The `blockEncrypt()` and `blockDecrypt()` times improved as the numbers of blocks passed to the algorithm at the same time increased, because the API overhead is averaged over more blocks, and more data is available in the cache. The larger amounts of data are still encrypted and decrypted in ECB mode; however, in operational use, Cipher-Block Chaining (CBC) mode would likely be used. Efficiency testing was not performed in CBC mode because this would add another layer of data processing that has no real impact on the performance of the algorithm. This data processing would be similar across the algorithms and would involve pre- and post-processing the data before calling the algorithms' internal ciphering routines. In addition, there may be some performance characteristics from one algorithm to another, based on whether data is treated as two 64-bit chunks or four 32-bit chunks, but this effect should be marginal.

The algorithm software was used as it was submitted with the following exceptions:

i) The CRYPTON algorithm did not follow the API correctly with regard to the input of `keyMaterial` to the `makeKey()` function and the IV (Initialization Vector) to the `cipherInit()` function. The CRYPTON algorithm provided a binary stream instead of an ASCII stream. Since all other algorithms would have to do a conversion from ASCII to binary, the CRYPTON code for `makeKey()` and `cipherInit()` was altered to accept an ASCII stream and convert the stream to binary. Testing was performed using the altered code.

ii) The FROG algorithm would only handle lower case hex characters in its ASCII to binary conversion routine. This routine was altered to handle both upper and lower case hex characters.

[6] The IV would be used in Cipher-Block Chaining mode.

These changes were made in order to maintain consistency across the algorithms. The same test programs were used for all algorithms; therefore, all algorithms needed to accept inputs in the same manner.

4.2 Algorithm Specific

Rijndael uses `sscanf()` to convert from ASCII to binary. This is a very slow method of performing the conversion and results in inefficient `makeKey()` and `cipherInit()`.

Values for the FROG algorithm in the cycle count tables are copies of the "one block" times. FROG can only handle one 128-bit block at a time. This is not a deficiency, since the API only requires handling one block at a time.

On systems where a "long long" data type (a 64-bit integer) was available, the DFC and HPC algorithms were run in 64-bit mode. These algorithms are inherently 64-bit algorithms and should run much more efficiently in 64-bit mode. On the PC, with its 32-bit processor, using Linux and the GCC compiler, the 64-bit math is simulated in software. However, on the Sun™ and SGI™ systems, which have 64-bit processors, the math operations are performed as 64-bit hardware instructions. NIST is aware that these 64-bit implementations do not meet the API since they are not ANSI C compliant. Because these two candidate algorithms are inherently 64-bit in nature, NIST felt it was appropriate to consider the performance of this non-compliant code in addition to the performance of the compliant code provided by the submitters.

5. Conclusions

5.1 PC

Due to the testing mechanisms used in obtaining data, the most reliable and accurate values obtained for performance measurement of the candidate algorithms are the cycle counting measurements on the PC, which focus on the 128-bit key length. Additionally, cycle count values for encryption and decryption were obtained for various message block lengths. These values provide interesting results. For the most part, once the message length was greater than one block (128 bits), the encryption and decryption speeds were consistent within each algorithm. For this reason, NIST focused on the message block length of 128 blocks (2046 bytes), which is a typical size for an electronic mail message. The graph in Figure 3[7] shows the encryption speed average of the Borland and Visual C compiled executables (see the tables in Appendix A). There appears to be a clear separation in the performance of the top six algorithms from the rest. Please note that the chart has two algorithms omitted. These omitted algorithms, HPC and Magenta, have encryption speeds that are over 3 times longer than the slowest algorithm included on the chart. The fastest six algorithms on the Pentium Pro are RC6, CRYPTON, Twofish, MARS, E2, and Rijndael. For the Pentium II and Pentium III, the same six algorithms are re-ordered as RC6, CRYPTON, E2, Twofish, Rijndael, and MARS.

[7] The relative uncertainty for values in all graphs and tables is ≤ 1%.

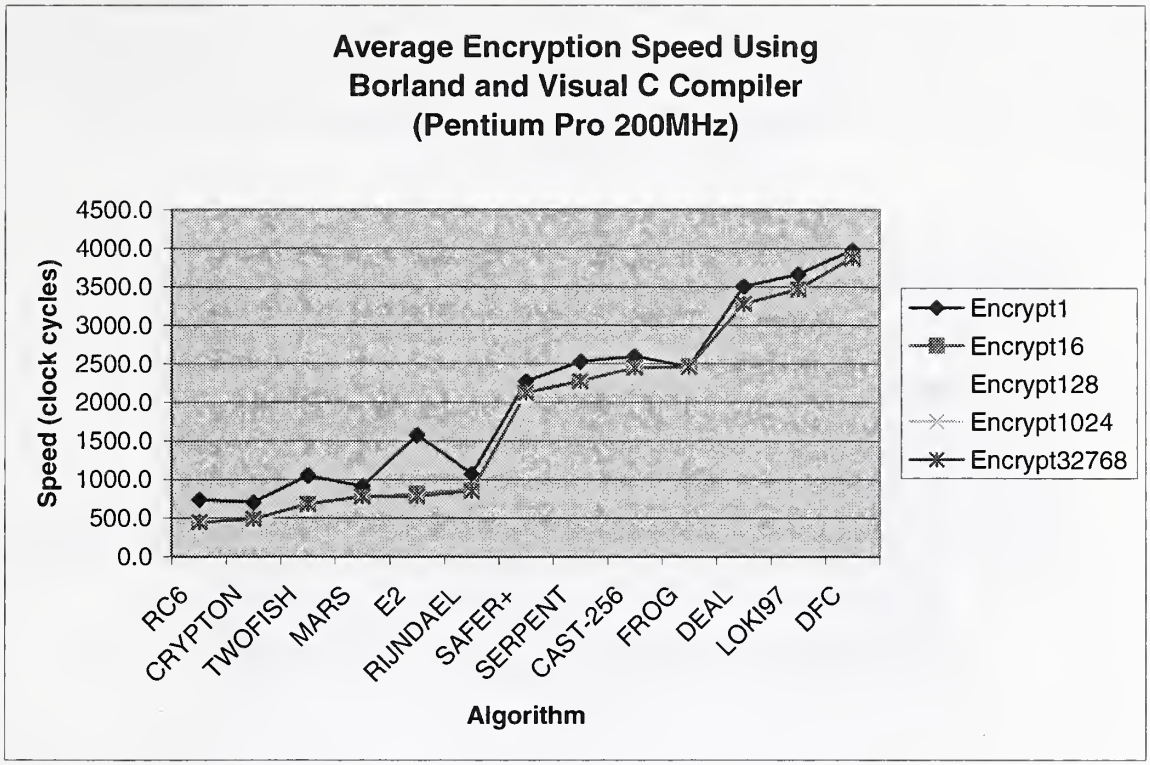

Fig. 3 Average Encryption Speed on Reference Platform

Brian Gladman [4] has performed similar efficiency experiments, the results of which were submitted during the Round1 public comment period. The tests that Gladman conducted used a 200 MHz Pentium Pro, and code that he developed independently from the submitters' code compiled with Visual C. Gladman omitted any data conversion or byte-ordering that the algorithms may have required to make them portable to other platforms. In spite of these significant differences, the overall outcome in terms of relative speed is similar to the NIST values detailed in Figure 3. Gladman's fastest seven algorithms (as determined by the first natural break in his values) are RC6, Rijndael, MARS, Twofish, CRYPTON, CAST-256, and E2. This set includes the fastest algorithms from the testing performed by NIST on the Pentium processor, with the addition of CAST-256.

Since key-dependent mixing can be performed in many ways, key setup on some algorithms may be much simpler than for other algorithms. For example, RC6 has a simpler key setup design. On the other hand, the FROG algorithm has a very complicated key setup design. In order to incorporate all key-mixing into one metric, the key setup time and encryption speed were combined into one value for each algorithm. The values must be normalized to provide equal weight to each component (average key setup time and average encryption speed). To normalize the values, each value was divided by the smallest in the category. That is, all key setup times were divided by the smallest key setup time, and all encryption speeds were divided by the lowest encryption speed. The two normalized values for each algorithm were then added together. The top six algorithms for this metric are CRYPTON, RC6, E2, SAFER+, MARS, and Rijndael. This is the same set of six algorithms shown to be the fastest when encryption speed

alone was considered, with the substitution of SAFER+ for Twofish. It can be noted that Twofish is the seventh algorithm by this metric. The graph in Figure 4 has three algorithms missing: DEAL, HPC, and FROG. The value of this metric for DEAL is approximately twice the value of the largest value charted, and HPC and FROG have values that are orders of magnitude larger.

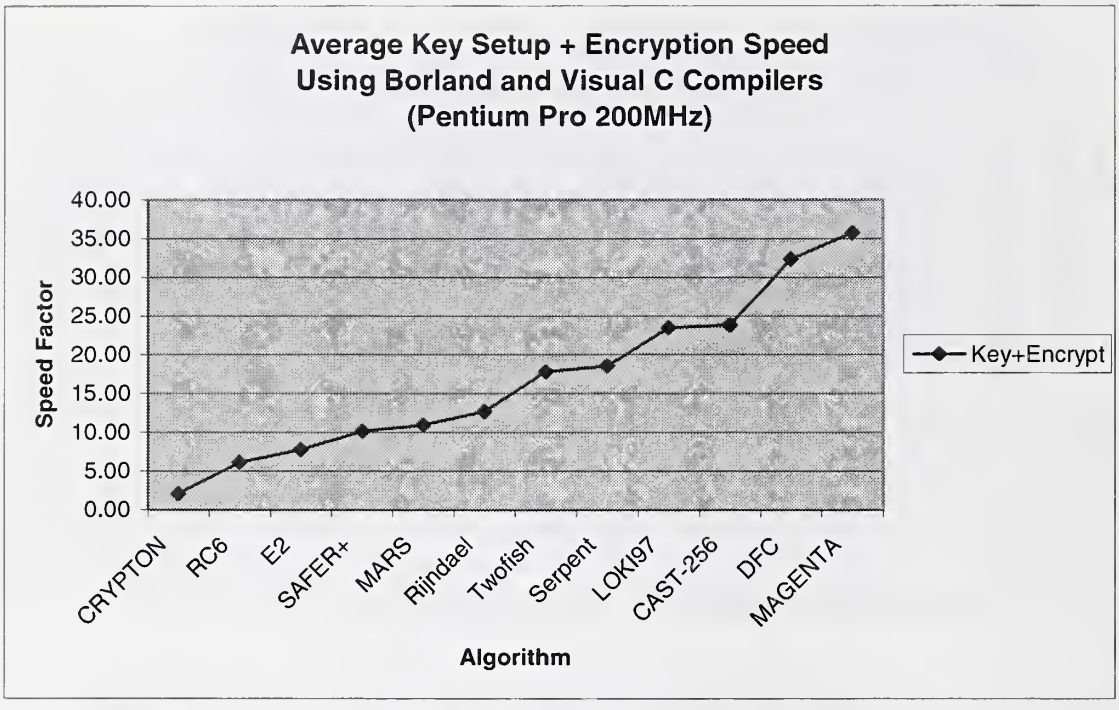

Fig. 4 Average Key Setup + Encryption Speed on Reference Platform

5.2 Sun

The UltraSPARC™ CPU found in the Sun systems on which testing was performed did not allow access to a cycle count mechanism. Performance numbers on these systems are based on the Timing Test Program. Additionally, the Sun systems have 64-bit processors. Therefore, two of the algorithms, DFC and HPC, that included code for 64-bit math operations, were run in both 64-bit mode (listed as DFC-64 and HPC-64) and 32-bit mode (listed as DFC and HPC). Two different compilers were used on the Sun, and the results from these were averaged. The graph in Figure 5 shows three separate groups of algorithms: the fastest six (CRYPTON, Rijndael, Twofish, Serpent, MARS, and CAST-256) processed at speeds approaching or exceeding 30 Mbits/s, HPC-64 and RC6 represent the next fastest group, followed by the rest of the algorithms. The E2 results are not an accurate reflection of the algorithm's true potential performance on this system. The code provided for the E2 algorithm by the submitter was not designed to be executed on big endian systems. E2 compiles but terminates abnormally when executed.

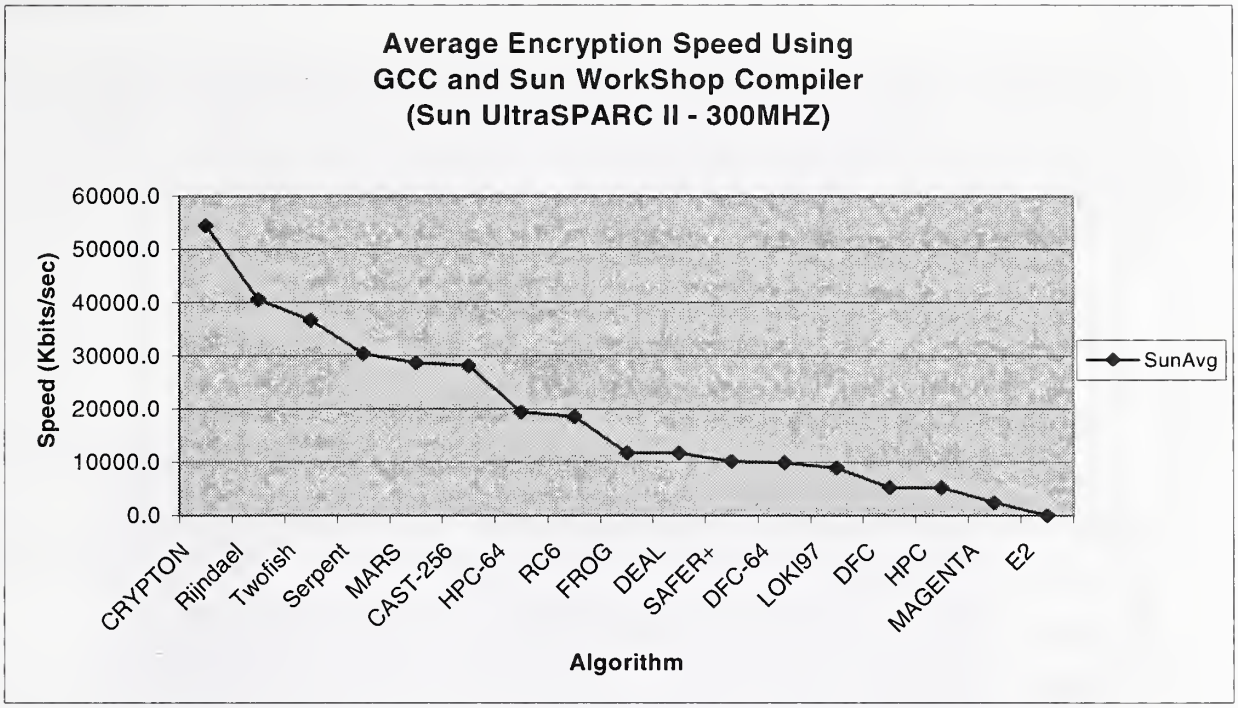

Fig. 5 Average Encryption Speed on Sun Platform

Louis Granboulan of ENS, France found very similar results on an UltraSPARC platform [5]. His fastest six algorithms are CRYPTON, Rijndael, Twofish, Serpent, CAST-256, and RC6. These six algorithms are a subset of the fastest eight algorithms listed above, with the following exception. Granboulan was unable to compile MARS, and he did not compile HPC in 64-bit mode. Therefore, these algorithms dropped out of the list of fastest performers. Compared with NIST's results, his fastest six algorithms were more closely grouped and had a much larger separation between the sixth and seventh algorithms.

5.3 SGI

The SGI system provides another 64-bit processor running the same version of the GCC compiler used for the Sun testing described in Section 5.2. Again, HPC and DFC were run in both 64-bit mode and 32-bit mode. The graph in Figure 6 shows that the top six algorithms in this environment were RC6, CRYPTON, MARS, Twofish, Rijndael, and HPC. As before, the E2 algorithm code compiled on this system, but did not execute properly.

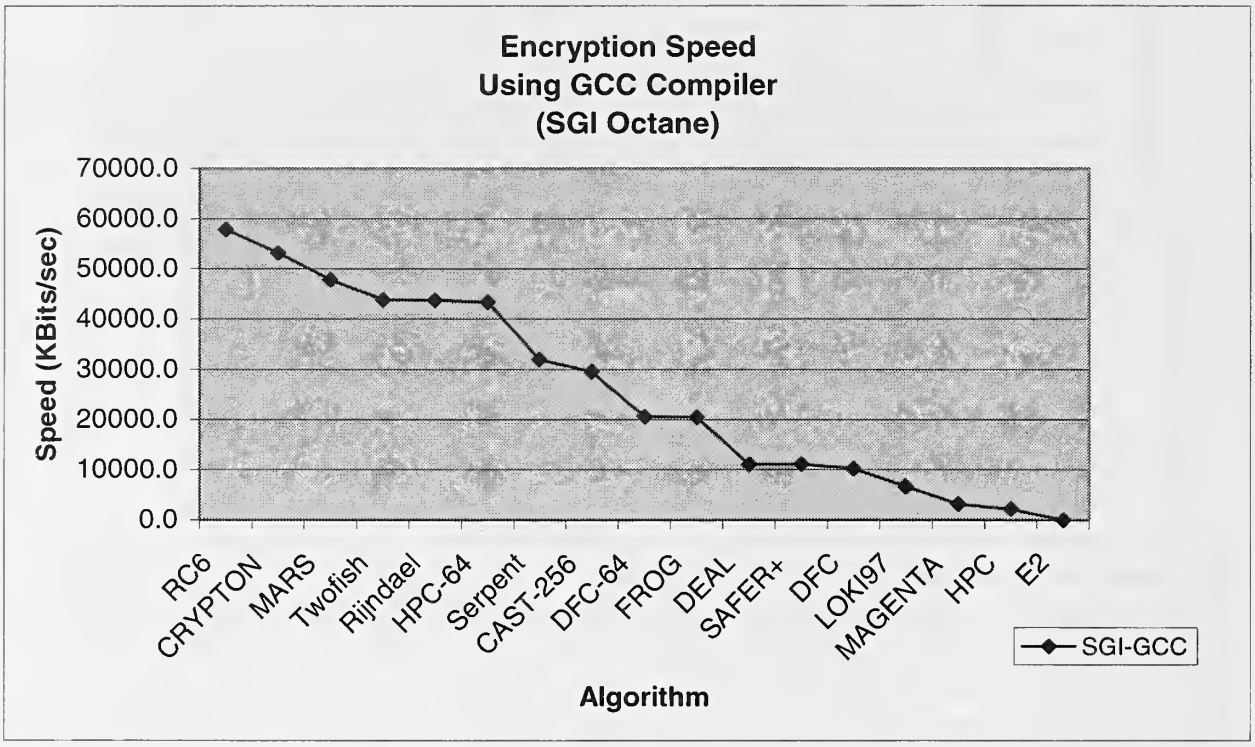

Fig. 6 Encryption Speed on SGI Platform

5.4 Overall Performance

The consistent top performers on all platforms tested are CRYPTON, RC6, Rijndael, Twofish, and MARS. Because E2 did not compile on the Sun and SGI platforms, the overall performance of E2 is difficult to assess. However, the analysis of Gladman and Granboulan indicate that E2 will consistently perform in the sixth or seventh position. Therefore, the five algorithms listed above should be categorized as the most consistent "top performers" across the platforms tested with E2, Serpent, and CAST-256 as the next three. HPC in 64-bit mode does provide sufficient encryption performance to place it in the second group of three, but this is only on 64-bit machines, and key setup speed remains poor. All other algorithms have consistently slow performance.

6. References

[1] "Request for Comments on Candidate Algorithms for the Advanced Encryption Standard (AES)", Federal Register, Volume 63, Number 177, pp. 49091-49093, Sept 14, 1998.

[2] "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)", Federal Register, Volume 62, Number 177, pp. 48051-48058, Sept 12, 1997.

[3] "Using the RDTSC Instruction for performance monitoring", http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM, Intel Corporation, 1997.

[4] Brian Gladman, "Implementation Experience with AES Candidate Algorithms", Proceedings of the Second Advanced Encryption Standard Candidate Conference, March 1999.

[5] Louis Granboulan, "AES: Analysis of the RefCode, OptCCode, and AddCode submissions", http://www.dmi.ens.fr/~granboul/recherche/AES/analysis.html, 1999.

Appendix A – Timing and Cycle Count Tables

A.1 Timing Tables

Values in the tables are as follow:

- Enc128 (Encryption using 128-bit key), Enc192 (Encryption using 192-bit key), and Enc256 (Encryption using 256-bit key) are in Kbits/sec
- Dec128 (Decryption using 128-bit key), Dec192 (Decryption using 192-bit key), and Dec256 (Decryption using 256-bit key) are in Kbits/sec
- Key128 (Setup a 128-bit key), Key192 (Setup a 192-bit key), and Key256 (Setup a 256-bit key) are in Keys/sec. This "Key" is the average time to setup an encryption key and a decryption key.

Some tables in this section contain values for DFC and HPC compiled using 64-bit integers (see Section 4). These additional values are listed in the tables as DFC-64 and HPC-64. The original values, using 32-bit integers, are listed as DFC and HPC.

Borland C++ 5.01 – 200 MHz Pentium Pro, 64MB RAM, Windows95

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	8490.5	8490.5	13157.8	8490.5	8490.5	12903.1	8490.5	8490.5	12658.6
CRYPTON	40524.7	39756.4	333332.8	40329.8	40136.9	333333.3	40329.8	39945.8	333333.6
DEAL	6864.7	6864.7	4048.4	6864.7	6864.7	3960.4	5275.9	5282.5	3025.8
DFC	5817.3	5874.4	10581.3	5853.9	5858	10256.3	5809.3	5817.3	9900.8
E2	28630.1	27324.5	48781.3	29852.7	27060.0	43478.1	30954.3	27869.1	40000.0
FROG	7936.2	14899.8	73.4	7936.2	14794.7	74.0	7936.2	14794.7	74.2
HPC	1743.6	1524.4	270.3	1769.4	1537.2	260.4	1736.8	1528.5	260.2
LOKI97	6186.3	6369.5	12578.1	6154.5	6297.8	12421.9	6190.9	6283.6	12195.3
Magenta	1368.0	1349.5	111110.9	1366.2	1352.1	74074.0	1025	1016.1	51282.0
MARS	25970.9	26800.7	26315.6	26214.4	26886.6	25640.6	26051.6	26886.6	25316.4
RC6	30283.8	31775	39215.6	30283.8	31775	38461.5	30283.8	31775	38461.7
Rijndael	31895.8	32263.9	27396.9	27235.7	28055.5	17699.0	23967.5	24745.2	13422.7
SAFER+	10631.9	10908.5	48781.3	7275.5	7503.2	29411.5	5537.0	5710.4	19230.5
Serpent	7307.1	7774.4	17093.8	7307.1	7774.4	14599.0	7307.1	7774.4	12422.7
Twofish	27324.5	29228.6	15384.4	26800.7	29228.6	12987.5	27324.5	29537.4	9661.7

Borland C++ 5.01 – 450 MHz Pentium II, 128MB RAM, Windows98

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	19239.9	19225.2	30303.1	19239.9	19239.9	29851.0	19254.6	19239.9	29411.7
CRYPTON	91429	92691.8	754717.2	89003.8	92691.8	701754.2	91180.5	91429	666666.4
DEAL	15635.8	15679.6	15151.6	15635.8	15679.6	14925.0	12035.3	12104.8	11235.9
DFC	13231.2	13189.6	24096.9	13325.8	13168.9	23256.3	13231.2	13294.1	22221.9
E2	77314.4	78398.2	129870.3	75403.2	78398.2	119047.9	77314.4	77852.5	109889.8
FROG	17635.5	33112.9	164.1	17647.8	33332.2	165.6	17647.8	33420.7	164.1
HPC	3729.9	3153.6	640.6	3684.1	3133.6	614.6	3692.2	3145.3	608.6
LOKI97	14218.0	14463.1	29850.0	14205.9	14794.7	29411.5	14500.6	14550.9	28985.2
Magenta	3078.4	3033.9	222221.9	3076.1	3035.0	151515.6	2313.5	2289.5	113960.2
MARS	58867.4	60677.1	58823.4	59705.4	60897.3	57970.8	59178.9	61008.1	56338.3
RC6	68061.7	71240.8	89285.9	68200.1	71240.8	87719.8	68200.1	71240.8	86956.3
Rijndael	72005.2	73423.3	57142.2	62601.6	63310.2	35714.6	55553.7	55553.7	26666.4
SAFER+	23876.5	24409.1	111110.9	16341.4	16732.6	62500.0	12421.4	12729.3	40000.0
Serpent	19988.7	21345.1	39215.6	19988.7	21327.0	31250.0	19988.7	21345.1	25974.2
Twofish	61008.1	65793.0	35087.5	60025.8	65664.3	30303.1	60241.4	66974.9	21978.1

Visual C 6.0 – 200 MHz Pentium Pro, 64MB RAM, Windows95

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	11618.6	11748.8	18348.4	11602.5	11748.8	18349.0	11602.5	11491.2	18348.4
CRYPTON	43690.7	43464.3	298507.8	43690.7	43919.4	281690.6	43464.3	43464.3	273972.7
DEAL	8120.6	8120.6	7273.4	8097.1	8128.5	7272.9	6246.2	6241.5	5449.2
DFC	7269.2	7275.5	14184.4	7175.9	7332.7	13699.0	7313.5	7275.5	13245.3
E2	26630.5	29026.3	52631.3	26630.5	29433.7	37037.5	26800.7	29026.3	32786.7
FROG	15709.0	19108.4	121.9	15917.7	18682.9	122.9	15391.9	19108.4	122.7
HPC	3062.7	2720.0	379.7	3097.7	2683.5	361.5	3101.1	2698.2	357.0
LOKI97	8338.6	8914.6	18348.4	8405.4	8811.6	18349.0	8208.0	8848.7	18348.4
Magenta	2730.7	2730.7	135134.4	2730.7	2730.7	91742.7	2056.0	2052.0	68493.0
MARS	30615.4	34952.5	37037.5	30615.4	34379.5	36363.5	30615.4	34521	37036.7
RC6	39756.4	40721.4	86956.3	39945.8	40920.0	74074.0	39016.8	40721.4	74074.2
Rijndael	22250.9	21845.3	22221.9	19108.4	18558.9	15625.0	16611.1	16131.9	12195.3
SAFER+	12000.9	11748.8	66667.2	8184.0	7898.9	37037.5	6181.7	5991.9	24691.4
Serpent	17810.2	20068.4	28571.9	17772.5	20068.4	22988.5	17772.5	20116.6	18348.4
Twofish	21845.3	28826.8	20407.8	21845.3	28728.1	15266.7	21845.3	28728.1	10989.1

Visual C 6.0 - 450 MHz Pentium II, 128MB RAM, Windows98

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	28149.7	28244.5	46510.9	28055.5	28244.5	45454.2	28244.5	28149.7	44444.5
CRYPTON	97542	98689.5	740740.6	98689.5	98689.5	655737.5	98689.5	98689.5	740740.6
DEAL	18315.7	18396.1	16393.8	18355.8	18355.8	15747.9	14122.2	14146.1	12195.3
DFC	16513.0	16480.6	32257.8	16352.1	16578.3	31250.0	16257.0	16480.6	30303.1
E2	76959.7	76959.7	142857.8	76959.7	76959.7	111111.5	76959.7	76959.7	90909.4
FROG	35246.3	42799.0	275.0	35098.8	42799	278.1	34952.5	43690.7	276.6
HPC	7109.0	6379.2	892.2	7133.2	6227.6	797.9	7067.1	6250.8	763.3
LOKI97	19784.5	20262.3	44443.8	19831.2	20460.0	44444.8	19831.2	20460.0	42553.1
Magenta	6181.7	6172.6	305343.8	6195.4	6181.7	222221.9	4655.2	4644.9	153846.1
MARS	66576.3	76959.7	100000.0	68759.1	76959.7	80000.0	68200.1	76959.7	80000.0
RC6	90200.1	93206.8	222221.9	91180.5	93206.8	200000.0	91180.5	93206.8	181818.0
Rijndael	76959.7	74235.5	48781.3	60787.0	62137.8	37037.5	53092.5	55553.7	27778.1
SAFER+	27147.6	26379.3	153846.9	18436.5	17924.4	86956.3	13888.4	13573.8	57143.0
Serpent	40136.9	45343.8	66667.2	40136.9	45343.8	54054.2	40136.9	45343.8	44444.5
Twofish	67650.1	76959.7	51282.8	68759.1	76959.7	37037.5	68200.1	76959.7	25974.2

DJGPP, gcc version pgcc 2.90.23 980102 (ecgs-1.0.1) –
200 MHz Pentium Pro, 64MB RAM, Windows95

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	21810.4	21810.4	22750.0	21810.4	21810.4	22750.0	21810.4	21810.4	22750.0
CRYPTON	30534.5	30534.5	303332.8	30534.5	30534.5	303333.3	30534.5	30534.5	260000.0
DEAL	7484.0	7484.0	7279.7	7484.0	7484.0	7000.0	5739.6	5783.1	5353.1
DFC	6206.2	6257.1	14000.0	6206.2	6206.2	14000.0	6257.1	6257.1	14000.0
E2	21810.4	23132.2	60667.2	23132.2	21810.4	36400.0	23855.1	23132.2	30333.6
FROG	7951.7	11744.1	87.5	7951.7	11744.1	87.5	7951.7	11744.1	87.5
HPC	107.9	107.2	700.0	91.8	91.3	758.3	91.8	91.3	758.6
LOKI97	5783.1	6010.7	12132.8	5739.6	6010.7	12133.3	5872.0	6058.4	12133.6
Magenta	875.4	874.4	364000.0	875.4	874.4	182000.0	656.9	656.4	182000.0
MARS	63613.6	63613.6	36400.0	63613.6	63613.6	36400.0	63613.6	63613.6	36400.0
RC6	42409.1	44903.7	91000.0	42409.1	44903.7	60666.7	42409.1	44903.7	60666.4
Rijndael	14680.1	14403.1	18200.0	12514.2	12312.3	12133.3	10751.6	10751.6	9100.0
SAFER+	4516.9	4989.3	91000.0	3041.3	3348.1	36400.0	2285.5	2519.4	26000.0
Serpent	16963.6	21810.4	26000.0	16963.6	21810.4	18200.0	16963.6	21810.4	15166.4
Twofish	14967.9	15267.3	7279.7	14967.9	15267.3	6066.7	14967.9	15267.3	4044.5

DJGPP, gcc version pgcc 2.90.23 980102 (ecgs-1.0.1) -
450 MHz Pentium II, 128MB RAM, Windows98

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	49784.6	49784.6	60667.2	49784.6	49784.6	60666.7	49784.6	49784.6	60666.4
CRYPTON	67854.5	67854.5	728000.0	67854.5	67854.5	606666.7	67854.5	67854.5	606666.4
DEAL	16963.6	16777.2	16545.3	16963.6	16777.2	16545.8	12938.4	13048.9	12133.6
DFC	14006.7	14136.4	36400.0	13879.3	14136.4	36400.0	14136.4	14136.4	30333.6
E2	61069.1	59871.6	151667.2	61069.1	59871.6	91000.0	61069.1	59871.6	82727.3
FROG	18032.2	26322.9	223.4	18032.2	26629.0	225.0	17891.3	26629.0	224.2
HPC	290.1	285.5	1517.2	201.3	199.0	1625.0	201.3	199.1	1610.9
HPC-64	29360.1	30534.5	3640.6	29360.1	30534.5	3639.6	29360.1	29360.1	3639.8
LOKI97	13392.3	14006.7	30332.8	13631.5	14136.4	30333.3	13275.9	14268.5	30333.6
Magenta	1982.8	1977.6	606667.2	1982.8	1977.6	364000.0	1488.0	1485.1	303333.6
MARS	138793.3	138793.3	121332.8	138793.3	138793.3	91000.0	138793.3	138793.3	91000.0
RC6	95420.4	98498.5	151667.2	95420.4	98498.5	151666.7	95420.4	98498.5	130000.0
Rijndael	33189.7	32832.8	36400.0	28013.3	27758.7	30333.3	24427.6	24427.6	20221.9
SAFER+	10223.6	11225.9	182000.0	6877.1	7583.1	91000.0	5169.5	5710.9	60666.4
Serpent	47710.2	50890.9	60667.2	47710.2	50890.9	45500.0	47710.2	50890.9	36400.0
Twofish	33927.3	34698.3	18200.0	33927.3	34698.3	14000.0	33927.3	34698.3	9100.0

Linux - 200 MHz Pentium Pro, 64MB RAM, Windows95

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	23301.7	23453.7	21978	23301.7	23453.7	21505.4	23301.7	23431.9	20833.3
CRYPTON	36684.9	36262.0	333333.3	35295.7	35797.8	285714.3	36105.9	36472.2	285714.3
DEAL	6601.7	6657.6	6968.6	6601.7	6657.6	6802.7	5049.3	5075.8	5194.8
DFC	7976.5	7703.0	14285.7	8047.9	7700.7	14084.5	7908.8	7707.8	13513.5
DFC-64	9241.9	9252.1	18867.9	9241.9	9252.1	18518.5	9245.3	9252.1	18018.0
FROG	10459.6	18752.5	90.0	10459.6	18696.7	89.6	10459.6	18369.2	89.3
HPC	5622.4	5033.2	787.4	5757.5	5007.1	787.4	5647.6	4975.4	793.7
HPC-64	10010.3	9397.2	1600.0	10192.7	9296.6	1538.5	10180.3	9303.4	1538.5
LOKI97	6035.0	6318.3	12422.4	6009.0	6288.3	12345.7	5999.0	6353.4	12345.7
Magenta	864.8	864.8	285714.3	864.8	865.0	181818.2	649.1	649.4	133333.3
MARS	38479.9	36792.1	40000.0	38362.5	36419.4	38461.5	38245.9	36792.1	37735.8
RC6	29262.6	37282.7	60606.1	29262.6	37282.7	58823.5	29262.6	37282.7	57142.9
Rijndael	36900.0	35951.2	9708.7	31695.0	31184.4	6410.3	28212.8	27413.8	4739.3
SAFER+	11224.7	10609.5	66666.7	7644.5	7217.0	40000.0	5794.6	5468.5	27777.8
Serpent	12684.4	15563.3	16528.9	12684.4	15563.3	12738.9	12684.4	15563.3	10256.4
Twofish	18236.1	19493.3	14492.8	18196.5	19478.2	11111.1	18236.1	19508.4	8333.3

GCC 2.8.1 - SGI 250 MHz R10000 w/2MB Cache, 512 MB RAM

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	29537.4	29746.8	46511.6	29537.4	29746.8	44444.4	29537.4	29746.8	42553.2
CRYPTON	53204.7	52538.3	500000.0	53773.1	53430.6	500000.0	52869.4	53204.7	400000.0
DEAL	11081.4	10894.3	8130.1	11076.5	10894.3	7936.5	8522.1	8366.3	6079.0
DFC	10284.4	10123.0	19607.8	10284.4	10176.2	19230.8	10267.6	10082.5	18691.6
DFC-64	20610.8	20594.0	37735.8	20610.8	20594.0	36363.6	20610.8	20594.0	35087.7
FROG	20476.7	19493.3	157.7	20476.7	19433.1	157.0	20443.4	19463.1	156.5
HPC	2171.3	1830.4	388.4	2180.9	1835.7	387.3	2163.3	1836.7	385.2
HPC-64	43389.4	44073.2	9174.3	43240.2	44228.2	9009.0	43389.4	44150.6	8849.6
LOKI97	6714.5	6768.6	13422.8	6716.3	6741.4	13245	6741.4	6748.7	13071.9
Magenta	3198.9	3456.8	333333.3	3213.2	3464.9	250000.0	2420.7	2606.8	181818.2
MARS	47843.8	51150.0	48780.5	47843.8	51150.0	47619.0	47843.8	51150.0	46511.6
RC6	57852.5	62601.6	125000.0	57852.5	62601.6	117647.1	57852.5	62601.6	117647.1
Rijndael	43766.7	41323.2	25316.5	36525.1	35746.9	17391.3	32598.2	31536.1	13071.9
SAFER+	11076.5	11165.0	90909.1	7485.4	7552.8	50000.0	5664.2	5702.7	32786.9
Serpent	32017.6	35746.9	38461.5	32017.6	35746.9	30769.2	32017.6	35746.9	25000.0
Twofish	43919.4	46007	26666.7	44073.2	46776.6	19230.8	43842.9	46345.9	13333.3

GCC 2.8.1 – 300 MHz UltraSPARC-II w/ 2MB Cache, 128 MB RAM

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	30840.5	30840.5	47619	30878.3	30878.3	46511.6	30840.5	30878.3	44444.4
CRYPTON	49441.7	45100.0	400000.0	49441.7	45100.0	400000.0	49539.0	45100.0	333333.3
DEAL	9872.8	9919.5	9302.3	9872.8	9923.4	9090.9	7550.5	7580.1	6968.6
DFC	5317.1	5323.8	10362.7	5340.8	5347.6	10309.3	5321.6	5327.2	10204.1
DFC-64	4634.6	4630.3	10695.2	4640.6	4635.4	10638.3	4638.9	4633.7	10582.0
FROG	13329.4	13610.5	109.1	13329.4	13617.9	108.5	13329.4	13759.3	108.2
HPC	8234.9	6703.7	251.4	8300.1	6693.0	242.1	8261.9	6661.1	233.8
HPC-64	13479.3	8487.6	2057.6	13472.1	8473.3	1586.0	13493.7	8467.6	1296.2
LOKI97	7265.0	7330.6	15384.6	7215.0	7300.8	15267.2	7231.6	7356.3	15384.6
Magenta	2481.8	2466.8	400000.0	2481.8	2466.5	285714.3	1865.1	1856.8	200000.0
MARS	27838.3	28181.2	47619.0	27838.3	28181.2	46511.6	27838.3	28181.2	45454.5
RC6	18275.8	18302.4	111111.1	18275.8	18302.4	111111.1	18275.8	18302.4	100000.0
Rijndael	37505.0	31378.8	27027.0	32768	27354.2	19047.6	28992.9	23497.5	14705.9
SAFER+	10551.7	10596.1	76923.1	7163.6	7196.4	43478.3	5422.5	5448.3	28169.0
Serpent	28826.8	30320.3	37037.0	28826.8	30320.3	29850.7	28826.8	30320.3	24390.2
Twofish	36002.6	37393.5	17391.3	36002.6	37393.5	13513.5	36002.6	37393.5	10000.0

Sun Workshop Compiler 4.2 – 300 MHz UltraSPARC-II w/ 2MB Cache, 128 MB RAM

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	25394.4	23431.9	40000.0	25394.4	23453.7	40000.0	25394.4	23453.7	38461.5
CRYPTON	59493.7	59918.6	400000.0	59634.7	59918.6	333333.3	59634.7	59918.6	333333.3
DEAL	13647.4	13654.8	10256.4	13647.4	13654.8	10000.0	10490.1	10507.7	7662.8
DFC	5224.4	5254.9	10362.7	5252.7	5206.0	10256.4	5234.2	5220.0	10256.4
DFC-64	15261.3	15233.5	32258.1	15317.0	15289.1	31746.0	15298.4	15279.8	30769.2
FROG	10259.2	15958	129.3	10284.4	16059.9	128.8	10242.5	15947.9	128.5
HPC	2251.8	1871.8	357.5	2244.3	1874.1	340.8	2240.3	1868.7	325.1
HPC-64	25471.5	23453.7	Inf	25627.1	23301.7	Inf	25653.2	23388.3	Inf
LOKI97	10677.1	10498.9	22222.2	10503.3	10507.7	22222.2	10468.3	10591.7	21978
Magenta	2462.4	2431.5	400000.0	2461.4	2429.6	250000.0	1849.2	1829.0	200000.0
MARS	29537.4	30174.8	66666.7	29676.7	30174.8	66666.7	29923.7	30030.8	64516.1
RC6	18822.6	18275.8	117647.1	18822.6	18289.1	111111.1	18822.6	18275.8	100000.0
Rijndael	43539.5	30504.0	26666.7	38014.8	26490.3	18181.8	34521.0	21883.3	14084.5
SAFER+	9841.9	9761.8	80000.0	6673.5	6627.8	46511.6	5049.3	5017.1	29850.7
Serpent	31936.3	33069.4	37735.8	31936.3	33069.4	29850.7	31936.3	33069.4	24691.4
Twofish	37282.7	37673.4	19230.8	37282.7	37673.4	14814.8	37282.7	37673.4	11363.6

Sun Workshop Compiler 4.2 – 2*360 MHz UltraSPARC-II w/ 4MB Cache, 256 MB RAM

Algorithm	Enc128	Dec128	Key128	Enc192	Dec192	Key192	Enc256	Dec256	Key256
CAST-256	30992.4	28597.5	50000.0	30992.4	28597.5	48780.5	30992.4	28597.5	46511.6
CRYPTON	72733.6	73156.5	500000.0	72733.6	73156.5	400000.0	72733.6	73156.5	400000.0
DEAL	16633.1	16644.1	12500.0	16644.1	16644.1	12195.1	12787.5	12807.0	9345.8
DFC	6218.4	6213.8	12345.7	6258.6	6255.5	12269.9	6219.9	6227.6	12121.2
DFC-64	18613.8	18586.3	40000.0	18669.0	18627.6	39215.7	18655.2	18627.6	37735.8
FROG	12507.9	19448.1	157.6	12539.0	19569.1	157.0	12489.2	19448.1	156.6
HPC	2764.3	2307.3	451.5	2755.2	2306.7	447.4	2744.4	2294.1	444.4
HPC-64	31378.8	28597.5	Inf	31339.8	28532.7	Inf	31418.0	28500.4	Inf
LOKI97	12905.6	12833.2	27027.0	12716.4	12794.0	27027	12710.0	12879.1	27027.0
Magenta	3005.2	2955.8	500000.0	3004.5	2956.2	285714.3	2258.6	2228.8	200000.0
MARS	36002.6	36792.1	83333.3	36157.8	36792.1	80000.0	36472.2	36631.5	80000.0
RC6	22940.6	22270.6	142857.1	22940.6	22270.6	133333.3	22940.6	22270.6	125000.0
Rijndael	53092.5	39444.9	33333.3	46345.9	32140.3	22988.5	42083.3	26715.3	17543.9
SAFER+	11995.1	11898.7	100000.0	8133.8	8078.9	55555.6	6153.0	6115.6	36363.6
Serpent	38956.4	40265.3	45454.5	38956.4	40329.8	36363.6	38956.4	40329.8	29850.7
Twofish	45507.8	45923.0	23529.4	45507.8	45923.0	18018.0	45507.8	46007.0	13888.9

A.2 Cycle Count Tables

The values in Ekey, Dkey, Encrypt-*n*, Decrypt-*n*, and Init are all in clock cycles. These values refer to:

- Ekey - The number of cycles needed to setup a 128-bit key for encryption;
- Dkey - The number of cycles needed to setup a 128-bit key for decryption;
- Encrypt-*n* - The number of cycles per block needed to encrypt *n* blocks of data using a 128-bit key;
- Decrypt-*n* - The number of cycles per block needed to decrypt *n* blocks of data using a 128-bit key; and,
- Init - The number of cycles needed to initialize the cipher (only included for the reference platform).

Values in the cycle count tables are sorted on the Encrypt-128 values. This length message is comparable to a typical electronic mail message. The values for Encrypt-*n* and Decrypt-*n* for the FROG algorithm are all duplicates of the value for Encrypt-1 and Decrypt-1. The `blockEncrypt()` function provided for the FROG algorithm can only accept one block at a time. In addition, the data encrypted and decrypted in the cycle count measurements was random data (as opposed to using all zero data blocks).

Cycles – Borland C++ 5.01 – 200 MHz Pentium Pro, 64MB RAM, Windows95

Algorithm	EKey	DKey	Encrypt1	Decrypt1	Encrypt16	Decrypt16
CRYPTON	689	768	753	912	492	500
RC6	5005	5006	840	787	518	462
TWOFISH	12950	12789	1000	967	622	571
RIJNDAEL	7667	8354	830	862	628	606
E2	4445	4386	1539	1599	850	844
MARS	7592	7592	993	970	834	809
SAFER+	4158	4158	2452	2341	2245	2219
CAST-256	15111	15108	3003	2983	2812	2812
SERPENT	11762	11786	3376	3185	3210	3012
FROG	2664417	2677195	3266	1818	3266	1818
DEAL	51311	51280	3845	3794	3547	3533
LOKI97	15507	15499	4177	4136	3997	3999
DFC	18926	19243	4460	4484	4315	4295
HPC	676803	676901	15736	17862	14356	16624
MAGENTA	2144	2148	18772	19014	18603	18859

Algorithm	Encrypt-128	Decrypt-128	Encrypt-1024	Decrypt-1024	Encrypt-32768	Decrypt-32768	Init
CRYPTON	479	480	505	505	523	523	7
RC6	497	446	498	445	520	460	220
TWOFISH	593	547	593	548	634	588	9
RIJNDAEL	610	592	627	603	621	593	5049
E2	795	778	789	775	814	797	7
MARS	823	797	827	800	847	820	16
SAFER+	2235	2211	2251	2231	2272	2252	27
CAST-256	2803	2802	2805	2807	2831	2833	447
SERPENT	3201	3006	3209	3013	3237	3053	9
FROG	3266	1818	3266	1818	3266	1818	1044
DEAL	3521	3509	3530	3514	3546	3543	923
LOKI97	3986	4019	3976	3991	4002	4022	1072
DFC	4324	4333	4312	4314	4338	4341	897
HPC	14297	16545	14299	16540	14309	16568	3804
MAGENTA	18589	18855	18607	18854	18627	18869	359

Cycles – Visual C 6.0 – 200 MHz Pentium Pro, 64MB RAM, Windows95

Algorithm	EKey	DKey	Encrypt1	Decrypt1	Encrypt16	Decrypt16
RC6	2272	2273	638	624	350	343
CRYPTON	734	807	659	691	442	446
MARS	5443	5443	838	729	685	597
TWOFISH	10206	10090	1106	1226	717	836
E2	4027	3978	1614	1624	788	792
RIJNDAEL	7488	7970	1326	1334	1103	1081
SERPENT	6963	6969	1675	1534	1298	1147
FROG	1611467	1622021	1662	1395	1662	1395
SAFER+	3094	3096	2098	2107	1953	1981
CAST-256	10568	10568	2196	2195	2049	2049
LOKI97	10844	10831	3157	2955	2930	2747
DEAL	28535	28232	3162	3184	2993	2986
DFC	13910	14568	3492	3583	3421	3409
HPC	473229	473437	9491	10636	8168	9573
MAGENTA	1465	1467	9259	9295	9120	9142

Algorithm	Encrypt-128	Decrypt-128	Encrypt-1024	Decrypt-1024	Encrypt-32768	Decrypt-32768	Init
RC6	336	326	335	329	376	347	138
CRYPTON	431	432	455	456	451	453	6
MARS	673	587	676	588	712	622	10
TWOFISH	690	811	693	813	726	825	12
E2	737	736	727	728	756	756	8
RIJNDAEL	1084	1064	1087	1064	1083	1063	7249
SERPENT	1279	1122	1278	1122	1317	1169	8
FROG	1662	1395	1662	1395	1662	1395	659
SAFER+	1949	1995	1974	2007	1977	2011	21
CAST-256	2041	2042	2051	2048	2075	2073	426
LOKI97	2920	2736	2905	2729	2940	2761	672
DEAL	2983	2979	2983	2979	3010	3007	778
DFC	3405	3411	3410	3407	3415	3414	549
HPC	8119	9391	8130	9403	8141	9427	2461
MAGENTA	9110	9128	9135	9145	9138	9163	103

Cycles – Borland C++ 5.01 – 450 MHz Pentium II, 128MB RAM, Windows98

Algorithm	EKey	DKey	Encrypt1	Decrypt1	Encrypt16	Decrypt16
CRYPTON	690	779	649	637	463	463
RC6	5009	5010	852	797	493	442
E2	3881	3865	900	871	573	572
TWOFISH	12692	12607	927	893	588	541
RIJNDAEL	7552	8216	805	852	604	582
MARS	7677	7677	973	961	809	781
SAFER+	4155	4153	2381	2338	2216	2198
SERPENT	10766	10760	6191	6433	2914	2596
CAST-256	15094	15091	2988	2991	2788	2791
FROG	2674949	2687678	3242	1816	3242	1816
DEAL	31155	31153	3673	3701	3510	3499
LOKI97	15497	15505	4051	4023	3811	3785
DFC	18916	19241	4458	4485	4292	4272
HPC	678651	679677	18521	20921	14736	17865
MAGENTA	2047	2050	18582	18836	18418	18665

Algorithm	Encrypt-128	Decrypt-128	Encrypt-1024	Decrypt-1024	Encrypt-32768	Decrypt-32768
CRYPTON	452	452	480	480	524	524
RC6	472	423	488	437	535	482
E2	556	553	568	566	600	599
TWOFISH	567	521	573	525	619	570
RIJNDAEL	586	567	584	566	582	567
MARS	798	772	813	786	862	836
SAFER+	2207	2189	2225	2207	2264	2246
SERPENT	2728	2375	2711	2350	2733	2385
CAST-256	2777	2779	2781	2782	2818	2818
FROG	3242	1816	3242	1816	3242	1816
DEAL	3502	3485	3506	3496	3548	3540
LOKI97	3786	3806	3798	3794	3850	3852
DFC	4299	4309	4288	4290	4306	4309
HPC	14600	17784	14605	17780	14663	17807
MAGENTA	18405	18655	18418	18660	18453	18708

Cycles – Visual C 6.0 - 450 MHz Pentium II, 128MB RAM, Windows98

Algorithm	EKey	DKey	Encrypt1	Decrypt1	Encrypt16	Decrypt16
RC6	2274	2275	642	616	329	321
CRYPTON	732	800	581	581	413	416
E2	3495	3455	1132	831	594	573
TWOFISH	8650	8575	909	760	587	443
RIJNDAEL	7497	7947	800	842	609	605
MARS	5423	5423	809	723	659	572
SERPENT	6808	6809	1709	1456	1279	1117
FROG	1611793	1622266	1654	1309	1654	1309
CAST-256	10051	10050	2031	2030	1872	1869
SAFER+	3085	3088	2093	2098	1928	1960
LOKI97	10402	10405	2976	2879	2720	2627
DEAL	28267	28267	3126	3130	2967	2962
DFC	13904	14563	3480	3563	3396	3384
HPC	474279	474464	9566	10671	7856	9204
MAGENTA	1463	1465	9248	9273	9096	9119

Algorithm	Encrypt-128	Decrypt-128	Encrypt-1024	Decrypt-1024	Encrypt-32768	Decrypt-32768
RC6	303	302	321	309	371	345
CRYPTON	403	405	434	433	491	492
E2	560	557	575	573	611	610
TWOFISH	567	421	582	429	626	468
RIJNDAEL	593	589	596	597	579	593
MARS	649	563	659	573	708	621
SERPENT	1257	1097	1273	1114	1311	1154
FROG	1654	1309	1654	1309	1654	1309
CAST-256	1861	1859	1879	1878	1921	1918
SAFER+	1923	1969	1961	1992	2008	2034
LOKI97	2711	2614	2726	2629	2761	2669
DEAL	2957	2952	2969	2964	3015	3011
DFC	3380	3386	3383	3381	3388	3386
HPC	7750	8999	7765	9007	7789	9039
MAGENTA	9087	9104	9114	9125	9168	9197

Cycles – Borland C++ 5.01 – 500 MHz Pentium III, 128MB RAM, Windows98

Algorithm	EKey	DKey	Encrypt1	Decrypt1	Encrypt16	Decrypt16
CRYPTON	656	745	646	637	430	430
RC6	4974	4975	849	794	461	414
E2	3842	3817	880	867	540	536
TWOFISH	12661	12601	924	890	554	506
RIJNDAEL	7519	8201	802	820	572	549
MARS	7546	7546	970	959	776	750
SAFER+	4110	4112	2388	2308	2183	2162
SERPENT	10726	10725	6188	6423	2882	2564
CAST-256	15050	15047	2990	2988	2754	2753
FROG	2674924	2687634	3182	1756	3182	1756
DEAL	31129	31122	3676	3697	3479	3464
LOKI97	15459	15499	4055	3967	3764	3746
DFC	18901	19211	4459	4498	4260	4240
HPC	677312	677366	17496	19682	14712	17832
MAGENTA	2015	2020	18579	18833	18386	18632

Algorithm	Encrypt-128	Decrypt-128	Encrypt-1024	Decrypt-1024	Encrypt-32768	Decrypt-32768
CRYPTON	418	418	455	455	527	527
RC6	438	389	446	390	487	425
E2	524	521	544	542	608	607
TWOFISH	534	487	546	502	597	551
RIJNDAEL	552	534	550	532	548	532
MARS	764	738	770	746	818	792
SAFER+	2173	2153	2204	2182	2266	2244
SERPENT	2693	2341	2687	2328	2733	2369
CAST-256	2741	2740	2753	2753	2796	2798
FROG	3182	1756	3182	1756	3182	1756
DEAL	3470	3453	3477	3465	3512	3500
LOKI97	3762	3776	3766	3758	3812	3810
DFC	4266	4275	4263	4265	4304	4307
HPC	14579	17739	14588	17734	14592	17771
MAGENTA	18370	18621	18394	18626	18450	18680

Cycles – Visual C 6.0 - 500 MHz Pentium III, 128MB RAM, Windows98

Algorithm	EKey	DKey	Encrypt1	Decrypt1	Encrypt16	Decrypt16
RC6	2238	2239	633	613	298	289
CRYPTON	698	766	578	578	381	384
E2	3463	3422	1127	833	563	541
TWOFISH	8655	8668	905	755	554	412
RIJNDAEL	7480	7929	796	837	577	573
MARS	5402	5402	804	718	627	540
SERPENT	6776	6776	1698	1452	1247	1085
FROG	1611678	1622175	1593	1240	1593	1240
CAST-256	10017	10014	2027	2026	1839	1837
SAFER+	3059	3060	2058	2067	1894	1926
LOKI97	10332	10344	2973	2852	2686	2592
DEAL	2857	2857	3123	3127	2934	2929
DFC	13881	14527	3476	3557	3364	3352
HPC	473937	474319	9566	10680	7823	9173
MAGENTA	1429	1431	9245	9270	9064	9087

Algorithm	Encrypt-128	Decrypt-128	Encrypt-1024	Decrypt-1024	Encrypt-32768	Decrypt-32768
RC6	269	269	284	276	340	314
CRYPTON	370	371	400	399	457	459
E2	527	523	540	538	579	578
TWOFISH	534	387	547	395	591	439
RIJNDAEL	559	555	562	563	545	559
MARS	616	529	639	543	684	589
SERPENT	1223	1063	1238	1079	1281	1125
FROG	1593	1240	1593	1240	1593	1240
CAST-256	1828	1826	1843	1842	1884	1881
SAFER+	1889	1935	1926	1957	1983	2013
LOKI97	2677	2580	2693	2595	2728	2636
DEAL	2923	2918	2934	2929	2964	2958
DFC	3346	3352	3351	3349	3353	3352
HPC	7719	8962	7738	8970	7762	9005
MAGENTA	9054	9070	9082	9091	9145	9172

Appendix B - Compiling Information

B.1 PC

On the three PCs used during testing, all algorithms were compiled using the same compiler options. Those options and their effect are:

- Borland:
  - -Oi Expand common intrinsic functions
  - –6 Generate Pentium Pro instructions
  - –v Source level debugging (does not effect speed)
  - –A Use only ANSI keywords
  - –a4 Align on 4 bytes
  - –O2 Generate fastest possible code
- Visual C:
  - /G6 Pentium Pro instructions
  - /Ox Best optimization for speed
- DJGPP:
  - –mcpu=pentiumpro Pentium Pro instructions and registers
  - –pedantic Warnings generated if non-ANSI
  - –fomit-frame-pointer If frame pointer is not need, it's not stored – frees a register
- Linux/GCC:
  - -O3 Best optimization for speed

The Borland programs were compiled on the 200 MHz Pentium Pro Reference machine. The Visual C and DJGPP programs were compiled on the 450 MHz Pentium II machine. The Linux operating system was installed on a Jaz drive attached to the 200 MHz Pentium Pro Reference machine. Compilations for GCC under Linux were performed on this machine.

B.2 Sun

All algorithms were compiled using the same compiler options. Those options and their effect are:

- GCC: -O3 Best optimization for speed
- Workshop: -xO5 Best optimization for speed

The compilations for the Sun systems were performed on the 300 MHz UltraSPARC II system. The Optimized C code for E2 appears to be designed in such a way that it only works on a little endian machine. The program compiled, but it resulted in a Bus Error and dumped core on execution.

B.3 SGI

All algorithms were compiled using the same compiler option. That option and its result is:

- GCC: -O3 Best optimization for speed

The compilations for the SGI were performed on the 250 MHz R10000 system. The Optimized C code for the E2 algorithm appears to be designed in such a way that it only works on a little endian machine. The program compiled, but it resulted in a Bus Error and dumped core on execution.