Efficiency Testing of ANSI C Implementations of Round 2 Candidate Algorithms for the Advanced Encryption Standard

Revised: April 27, 2000

Lawrence E. Bassham III

Computer Security Division
Information Technology Laboratory
National Institute of Standards and Technology

1. Introduction

The evaluation criteria for the Advanced Encryption Standard (AES) Round2 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. The results of the measurements are included followed by conclusions regarding which algorithms have the most consistent performance across different platforms. Some knowledge regarding compilation and processor architectures is useful in understanding how the data was derived. However, the raw data in the document may be useful without necessarily understanding the derivation.

The testing described in this paper is similar to that done in Round 1. The testing has obviously been restricted to the five Round 2 candidates. Additionally, Timing Tests for the Pentium based platforms has been omitted in favor of Cycle Count testing (see Section 3).

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. NIST is not interested in the absolute value of the performance measurement, but in the relative 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

[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 and Technology, nor does it imply that material identified is necessarily the best for the purpose.

than another algorithm by 0.5%. 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.

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 (cycles)
		Visual C++® 6.0 (cycles)
	Linux	GCC 2.8.1 (timing)
450MHz Pentium II Processor, 128 MB RAM	Windows98 4.10.1998	Borland C++ 5.01 (cycles)
		Visual C 6.0 (cycles)
600MHz Pentium III Processor, 128 MB RAM	Windows98 4.10.1998	Borland C++ 5.01 (cycles)
		Visual C 6.0 (cycles)
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
Sun: 2*360MHz UltraSPARC-II w/ 4MB Cache, 256 MB RAM	Solaris 2.7	GCC 2.8.1
		Sun Workshop Compiler C 4.2
Silicon Graphics™: 2*300MHz R12000™ w/ 4MB Cache, 512 MB RAM	IRIX64™ 6.5.4 (a 64 bit operating system)	GCC 2.8.1
		MIPSpro C Compiler 7.30

Performance measurements were conducted in two different ways. The first performance test method determines 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 performance testing method counts 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 required to perform an operation on a 300 MHz Pentium II processor as 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 Round 2 testing that provides access to a true cycle counting mechanism.

3.1 Cycle Counting Program

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

- The number of cycles needed to setup a key for encryption;
- The number of cycles needed to encrypt block(s) of data;
- The number of cycles needed to setup a key for decryption; 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 disable interrupts before testing and enable after testing. This eliminates extraneous interrupts that would skew results. The test program generates 1000 sets of cycle count information as described above for each key size. The values in each category are then sorted, and the median value is determined. A standard deviation is calculated for each test category.

```
makeKey();
cipherInit();
for (r=0; r<1000; r++) {
      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  */
      }
```

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. Values in this test program are calculated around

[3] Interrupts occur, for example, when the operating system 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.

the NIST API calls. Results for the Cycle Counting Program can be found in Section 5.1. Pseudo code for the generation of cycle counting information for the `blockEncrypt()` operation is included in Figure 1.

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.

3.2 Timing Program

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

- The time to setup 10,000 keys for encryption;
- The time to encrypt 8192 blocks of data (8192 blocks*128 bits/block=1048576 bits=1Mbit);
- The time to setup 10,000 keys for decryption; and,
- The time to decrypt 8192 blocks of data (8192 blocks*128 bits/block=1048576 bits=1Mbit).

Analysis of this data was performed in the same way as the cycle count program listed above in Section 3.1 (calculation of standard deviation, median, etc.) Results for the Timing Program can be found in Section 5.2. Pseudo code for the generation of timing information for the `blockEncrypt()` operation is included in Figure 2.

```
makeKey();
cipherInit();
for(r=0; r<1000; r++){
      (Start Timer)
      blockEncrypt(8192 blocks);
      (Stop Timer)
      }
```

Fig. 2: Pseudo code for Time Testing for `blockEncrypt()`

3.2 Compiler Options

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
- 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.

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.

SGI

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

• GCC:	-O3	Best optimization for speed
• MIPSpro:	-O3	Best optimization for speed

The Twofish algorithm compiles on the SGI using the MIPSpro compiler, but results in a Bus Error and a core dump when the `blockEncrypt()` and `blockDecrypt()` functions are invoked. This appears to be a problem with how the compiler is handling byte alignment in the optimized code.

4. Observations

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 results in code which 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 `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, i.e., pre- and post-processing the data before calling the algorithms' internal ciphering routines. In addition, there may be performance characteristics from one algorithm to another, based on whether data is treated as two 64-bit blocks or four 32-bit blocks, but this effect depends on the processor characteristics.

5. Results

5.1 Cycle Count Tables

The values[4] in Ekey, Dkey, Enc, and Dec 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;
- Enc - The number of cycles per block needed to encrypt *n* blocks of data; and,
- Dec - The number of cycles per block needed to decrypt *n* blocks of data.

Note: the data encrypted and decrypted in the cycle count measurements was random (as opposed to using all zero data blocks).

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

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	6815	6814	1097	1049	944	921	937	913	938	914	957	933
MARS-192	7001	7001	1094	1059	947	921	938	913	937	918	956	935
MARS-256	7222	7222	1081	1058	944	926	938	913	939	914	958	932
RC6-128	5171	5170	950	911	630	576	610	556	614	558	629	582
RC6-192	5254	5265	950	914	636	578	609	555	614	558	629	582
RC6-256	5330	5331	949	914	630	576	610	556	614	558	629	582
RIJNDAEL-128	2208	2870	826	836	690	690	685	686	682	681	704	714
RIJNDAEL-192	2972	3786	958	961	823	815	815	808	820	811	850	835
RIJNDAEL-256	3691	4684	1106	1137	982	996	939	946	939	947	961	968
SERPENT-128	12324	12291	3569	3273	3429	3158	3422	3155	3422	3163	3436	3178
SERPENT-192	14389	14398	3574	3301	3429	3159	3420	3147	3424	3165	3438	3176
SERPENT-256	16639	16644	3570	3214	3429	3074	3420	3064	3425	3163	3438	3175
TWOFISH-128	13544	13372	1052	1009	725	681	706	660	708	662	727	687
TWOFISH-192	15707	15544	1052	993	722	675	706	660	708	663	728	686
TWOFISH-256	21344	21181	1049	996	723	679	704	660	708	661	729	682

[4] The relative uncertainty for values in all tables is ≤ 1%.

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

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	4964	4964	837	754	687	598	681	593	684	595	718	629
MARS-192	4996	4996	821	737	686	601	680	593	683	596	719	629
MARS-256	5185	5185	823	743	689	601	680	593	682	595	720	629
RC6-128	2293	2294	640	627	351	351	340	332	343	334	382	355
RC6-192	2401	2402	640	627	352	351	340	332	343	334	382	355
RC6-256	2512	2513	642	629	352	351	343	332	343	334	382	355
RIJNDAEL-128	1278	1764	1277	1308	1138	1133	1125	1136	1134	1135	1149	1124
RIJNDAEL-192	2002	2566	1512	1574	1368	1362	1358	1365	1361	1372	1388	1365
RIJNDAEL-256	2591	3257	1732	1798	1604	1596	1591	1599	1596	1601	1614	1588
SERPENT-128	7092	7104	1439	1293	1298	1135	1286	1129	1285	1128	1326	1165
SERPENT-192	9048	9035	1455	1294	1295	1135	1285	1126	1285	1126	1326	1168
SERPENT-256	10861	10850	1454	1275	1292	1135	1285	1127	1286	1128	1326	1166
TWOFISH-128	9950	9790	1264	1024	965	725	947	707	950	711	967	740
TWOFISH-192	13298	13136	1265	1020	966	728	947	707	949	721	965	753
TWOFISH-256	18555	18394	1278	1016	965	726	947	707	950	710	966	743

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

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	6837	6837	1105	1082	947	924	939	913	941	920	986	963
MARS-192	7040	7038	1105	1092	949	919	939	913	937	921	985	961
MARS-256	7249	7249	1105	1082	949	922	936	914	941	921	992	966
RC6-128	5186	5183	984	944	631	578	610	556	617	560	651	598
RC6-192	5279	5279	984	943	631	577	609	555	617	560	651	598
RC6-256	5363	5364	984	944	631	578	609	555	617	560	651	598
RIJNDAEL-128	2254	2912	845	844	689	699	681	692	696	697	777	783
RIJNDAEL-192	2994	3778	983	993	818	814	811	807	826	820	892	896
RIJNDAEL-256	3722	4668	1099	1125	948	958	938	948	954	952	1021	1027
SERPENT-128	11767	11671	3108	2702	2855	2496	2842	2480	2847	2488	2868	2523
SERPENT-192	13872	13852	3108	2705	2856	2478	2842	2465	2847	2467	2868	2505
SERPENT-256	16073	15978	3108	2710	2857	2500	2842	2488	2847	2500	2868	2528
TWOFISH-128	12907	12816	1063	1034	726	677	702	657	708	662	755	708
TWOFISH-192	15311	15219	1061	1031	726	680	704	658	706	665	753	712
TWOFISH-256	20706	20645	1061	1018	727	679	703	657	708	663	754	713

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

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	4937	4938	825	734	669	582	658	571	669	583	715	628
MARS-192	4999	4999	825	734	669	578	658	572	667	582	716	629
MARS-256	5175	5175	825	734	668	582	658	572	667	583	716	628
RC6-128	2283	2284	638	622	339	327	321	310	330	320	379	354
RC6-192	2408	2409	638	622	339	327	321	310	330	320	379	354
RC6-256	2519	2520	638	622	339	327	321	310	330	320	379	354
RIJNDAEL-128	1292	1722	987	987	810	801	808	789	826	796	894	866
RIJNDAEL-192	2014	2553	1152	1135	987	969	983	957	1005	972	1079	1039
RIJNDAEL-256	2594	3241	1329	1311	1161	1135	1158	1124	1173	1132	1238	1202
SERPENT-128	6947	6935	1423	1262	1273	1116	1263	1107	1281	1122	1320	1162
SERPENT-192	8857	8857	1423	1280	1274	1117	1263	1107	1281	1122	1320	1162
SERPENT-256	10666	10683	1423	1256	1274	1117	1263	1108	1281	1122	1320	1162
TWOFISH-128	9266	9249	1126	952	802	636	782	615	800	628	831	669
TWOFISH-192	12707	12627	1130	952	802	634	782	616	795	622	832	673
TWOFISH-256	17942	17863	1126	955	802	635	782	616	795	622	832	672

Cycles – Borland C++ 5.01 – 600 MHz Pentium III, 128MB RAM, Windows98

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	6833	6833	1143	1120	951	924	938	913	947	921	976	959
MARS-192	7017	7017	1171	1131	951	926	938	914	940	917	980	959
MARS-256	7245	7245	1143	1120	950	927	939	913	943	918	978	959
RC6-128	5189	5186	1022	982	633	580	610	555	620	567	642	637
RC6-192	5272	5271	1022	982	633	580	610	556	620	567	642	637
RC6-256	5362	5363	1026	982	633	580	609	556	620	567	642	637
RIJNDAEL-128	2213	2862	908	890	692	694	681	681	700	687	757	747
RIJNDAEL-192	2981	3776	1031	1047	820	809	809	799	818	813	883	873
RIJNDAEL-256	3727	4672	1152	1140	959	950	935	937	947	944	1002	996
SERPENT-128	11850	11849	3161	2743	2859	2497	2842	2490	2855	2468	2870	2516
SERPENT-192	13937	13916	3164	2739	2861	2484	2841	2467	2856	2495	2870	2536
SERPENT-256	16133	16114	3165	2737	2859	2500	2841	2485	2849	2483	2869	2536
TWOFISH-128	12938	12861	1085	1057	724	682	704	658	712	667	763	718
TWOFISH-192	15347	15298	1085	1078	727	680	704	659	713	668	764	716
TWOFISH-256	20760	20689	1085	1053	729	681	704	658	718	664	764	713

Cycles – Visual C 6.0 - 600 MHz Pentium III, 128MB RAM, Windows98

			1 block		16 blocks		128 blocks		1024 blocks		32768blocks	
	Ekey	Dkey	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
MARS-128	4934	4936	860	769	668	581	656	569	683	585	708	617
MARS-192	4997	4997	860	769	668	578	656	569	682	585	709	618
MARS-256	5171	5171	860	769	669	581	656	569	682	586	709	617
RC6-128	2278	2279	672	657	339	327	318	307	325	318	366	346
RC6-192	2403	2404	672	657	339	327	319	307	325	318	366	346
RC6-256	2514	2515	672	657	339	327	319	307	325	318	366	346
RIJNDAEL-128	1289	1724	1007	1006	811	802	805	784	824	794	880	848
RIJNDAEL-192	2000	2553	1188	1169	987	966	981	955	1003	971	1069	1023
RIJNDAEL-256	2591	3255	1365	1347	1160	1138	1155	1121	1171	1131	1227	1187
SERPENT-128	6944	6933	1458	1315	1273	1113	1261	1104	1281	1120	1309	1150
SERPENT-192	8853	8853	1459	1297	1273	1116	1260	1102	1281	1123	1309	1151
SERPENT-256	10668	10668	1459	1315	1273	1115	1262	1103	1281	1120	1309	1150
TWOFISH-128	9263	9241	1161	987	802	635	780	613	797	625	828	664
TWOFISH-192	12722	12632	1165	987	802	633	779	613	791	619	828	666
TWOFISH-256	17954	17876	1161	990	802	635	780	613	792	622	828	665

5.2 Timing Tables

Values in the tables are as follow:

- Ekey (time to make a key for encryption) is in Keys/sec;
- Encrypt (time to encrypt) is in Kbits/sec;
- Dkey (time to make a key for decryption) are in Keys/sec; and,
- Decrypt (time to decrypt) is in Kbits/sec.

GCC 2.8.1 - 200 MHz Pentium Pro, 64MB RAM, Linux

	Ekey	Encrypt	Dkey	Decrypt
Mars-128	46729.0	39035.8	46511.6	37135.9
Mars-192	44444.4	39035.8	44642.9	37135.9
Mars-256	42918.5	38855.1	43103.4	37135.9
RC6-128	59523.8	37300.9	58823.5	52454.4
RC6-192	57142.9	37300.9	57803.5	52454.4
RC6-256	56818.2	37300.9	57142.9	52454.4
Rijndael-128	128205.1	42602.6	106383.0	41754.7
Rijndael-192	88495.6	36175.4	74074.1	35562.3
Rijndael-256	74074.1	31551.5	62500.0	30969.4
Serpent-128	16891.9	13052.4	16920.5	16328.2
Serpent-192	13123.4	13052.4	13140.6	16328.2
Serpent-256	10559.7	13052.4	10582.0	16328.2
Twofish-128	14471.8	20671.7	14450.9	22261.8
Twofish-192	11086.5	20671.7	11025.4	22261.8
Twofish-256	8305.6	20671.7	8291.9	22261.8

SGI 300 MHz R12000 w/4MB Cache, 512 MB RAM

	GCC 2.8.1				MIPSpro C Compiler Version 7.30			
	Ekey	Encrypt	Dkey	Decrypt	Ekey	Encrypt	Dkey	Decrypt
Mars-128	60975.6	63581.1	60975.6	66608.8	78125.0	67683.1	78125.0	71124.6
Mars-192	59171.6	63581.1	59523.8	67141.6	76923.1	67683.1	76923.1	70526.9
Mars-256	57803.5	63581.1	57803.5	66608.8	75188.0	67683.1	75188.0	70526.9
RC6-128	147058.8	86522.7	147058.8	98737.7	166666.7	80699.1	166666.7	87424.0
RC6-192	142857.1	86522.7	142857.1	98737.7	161290.3	80699.1	161290.3	87424.0
RC6-256	138888.9	86522.7	138888.9	98737.7	156250.0	80699.1	156250.0	87424.0
Rijndael-128	212766.0	58282.7	161290.3	58282.7	212766.0	74271.7	153846.2	79930.5
Rijndael-192	163934.4	49080.1	125000.0	49368.8	142857.1	63103.0	109890.1	68233.4
Rijndael-256	142857.1	42387.4	108695.7	42819.9	121951.2	54498.1	93457.9	58690.2
Serpent-128	47393.4	42174.4	47393.4	46113.8	57471.3	42819.9	57471.3	45612.5
Serpent-192	37878.8	41963.5	38022.8	46113.8	44247.8	42602.6	44247.8	45612.5
Serpent-256	31250.0	41963.5	31250.0	46113.8	35461.0	42602.6	35461.0	45612.5
Twofish-128	31055.9	59947.9	31055.9	63581.1	41493.8	N/A	41841.0	N/A
Twofish-192	23255.8	60379.2	23310.0	64066.4	32786.9	N/A	33112.6	N/A
Twofish-256	16420.4	59947.9	16447.4	63581.1	22321.4	N/A	22522.5	N/A

Sun 300 MHz UltraSPARC-II w/ 2MB Cache, 128 MB RAM

	GCC 2.95				Sun Workshop Compiler 4.2			
	Ekey	Encrypt	Dkey	Decrypt	Ekey	Encrypt	Dkey	Decrypt
Mars-128	48780.5	29867.3	48543.7	29242.9	52356.0	30081.4	53475.9	29973.9
Mars-192	47393.4	29867.3	46948.4	29141.3	52356.0	30081.4	52083.3	30081.4
Mars-256	46082.9	29867.3	45662.1	29242.9	51020.4	29973.9	51282.1	30081.4
RC6-128	111111.1	20981.8	113636.4	20981.8	111111.1	20470.0	*N/A*	20420.2
RC6-192	108695.7	20981.8	108695.7	20981.8	101010.1	20520.1	*N/A*	20470.0
RC6-256	105263.2	20981.8	106383.0	20981.8	*N/A*	20520.1	98039.2	20470.0
Rijndael-128	172413.8	45612.5	131578.9	38498.6	166666.7	49368.8	117647.1	50864.9
Rijndael-192	140845.1	37805.0	106383.0	32033.2	128205.1	41963.5	85470.1	43261.4
Rijndael-256	117647.1	33042.1	90090.1	27517.1	108695.7	36490.0	73529.4	37467.4
Serpent-128	30120.5	34537.9	30120.5	34969.6	33783.8	32156.0	33898.3	32912.6
Serpent-192	25000.0	34255.9	25000.0	34969.6	27173.9	32033.2	27248.0	32912.6
Serpent-256	21008.4	33841.5	21052.6	34824.5	22421.5	32156.0	22421.5	33042.1
Twofish-128	22321.4	36972.3	22321.4	36020.2	21739.1	41963.5	21739.1	*N/A*
Twofish-192	16366.6	36972.3	16366.6	36020.2	16447.4	41754.7	16420.4	*N/A*
Twofish-256	11547.3	37300.9	11560.7	36020.2	12285.0	42174.4	12300.1	*N/A*

NOTE: The italicized items in the Sun Workshop Compiler table above are corrections to the values found in the Proceedings of the 3rd AES Candidate Conference. The original values were incorrect and have been replaced by N/A (not available).

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

	GCC 2.95				Sun Workshop Compiler 4.2			
	Ekey	Encrypt	Dkey	Decrypt	Ekey	Encrypt	Dkey	Decrypt
Mars-128	59523.8	36332.1	59523.8	35562.3	65359.5	36649.4	65359.5	36810.1
Mars-192	57803.5	36175.4	57803.5	35412.3	64102.6	36649.4	64102.6	36810.1
Mars-256	56179.8	36175.4	56179.8	35562.3	62500.0	36649.4	62500.0	36810.1
RC6-128	138888.9	26227.2	138888.9	26227.2	142857.1	25587.5	142857.1	25587.5
RC6-192	133333.3	26227.2	135135.1	26227.2	136986.3	25587.5	138888.9	25587.5
RC6-256	129870.1	26227.2	129870.1	26227.2	131578.9	24978.3	131578.9	24978.3
Rijndael-128	217391.3	55215.2	161290.3	47958.3	200000.0	59522.7	142857.1	61260.6
Rijndael-192	172413.8	46886.6	129870.1	39965.3	158730.2	50864.9	107526.9	52454.4
Rijndael-256	142857.1	40940.0	109890.1	34396.3	133333.3	44405.8	88495.6	45612.5
Serpent-128	36101.1	41963.5	36231.9	42819.9	42372.9	39035.8	42372.9	39965.3
Serpent-192	30303.0	41963.5	30303.0	42819.9	34013.6	39035.8	34013.6	39965.3
Serpent-256	25641.0	41963.5	25641.0	42819.9	28328.6	39035.8	28328.6	39965.3
Twofish-128	27322.4	45122.1	27248.0	43039.5	26738.0	53118.4	26738.0	51489.0
Twofish-192	20080.3	44880.8	20080.3	43039.5	20120.7	53456.7	20120.7	51489.0
Twofish-256	14184.4	44880.8	14164.3	43261.4	15015.0	53456.7	15037.6	51806.8

6. Conclusions

6.1 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. Additionally, cycle count values for encryption and decryption were obtained for various data block lengths. These values provide interesting results. For the most part, once the data 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 fastest algorithm for key setup on the PC platform is Rijndael for all compiler and PC hardware/software configurations, followed closely by RC6 and then Mars. Serpent and Twofish are considerably slower than the other algorithms for key setup time. Encryption speed had more variability across compiler and hardware/software platforms. RC6 tends to fall near the top of PC encryption speed followed by Mars, Twofish, and Rijndael. Serpent is consistently at the bottom of the list for encryption speed.

Brian Gladman [4] has performed similar efficiency experiments, the results of which are available on a web page he maintains. The tests that Gladman conducted used code that he developed independently from the submitters' code. Gladman's results are similar to those listed above. Gladman's results for key setup time have the algorithms in basically the same order. The exception being the fact that Serpent's key setup time was greatly improved and ahead of Mars. Again, for encryption speed, Gladman's results coincide with the ordering of the algorithms listed above.

6.2 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. Two different compilers were used on the Sun. The data from both these compilers yielded similar results. The fastest algorithms with respect to encryption speed are Rijndael and Twofish, followed by Serpent and Mars, and finally by RC6. However, with respect to key setup Rijndael and RC6 are the fastest followed by Mars which is separated by a wide margin. Serpent and Twofish are last after another wide margin.

Helger Lipmaa reports very similar results on an UltraSPARC-II platform [5]. Lipmaa's table only reports encryption speed. The most noticeable difference is that on his table, the value for the encryption speed of RC6 is closer to those for Mars and Serpent.

6.3 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 6.2. Additionally, the MIPSpro compiler provided another configuration for comparison. The results for these compilers place RC6 as the fastest algorithm for encryption by a wide margin, followed by Mars, Twofish, Rijndael and

Serpent. For key setup, RC6 and Rijndael are the fastest, followed by Mars, Serpent, and Twofish, which are separated by a wide margin.

6.4 Overall Performance

The consistent top performers across all platforms with respect to key setup are Rijndael and RC6. Serpent and Twofish are usually significantly poorer performers; however, Gladman reports a much better value for Serpent key setup, placing Serpent ahead of Mars. Encryption speed values tend to vary much more depending on the platform being analyzed. Rijndael, Mars, and Twofish have the most even encryption performance across platforms – not always the fastest, but never near the bottom of the pack. RC6, on the other hand, was the slowest on the Sun systems but the fastest on the SGI and very nearly the fastest on the PC. Serpent is typically the slowest or towards the bottom of the list on encryption speed across platforms.

7. References

[1] "Request for Comments on the Finalist (Round 2) Candidate Algorithms for the Advanced Encryption Standard (AES)," Federal Register, Volume 64, Number 178, pp. 50058-50061, Sept. 15, 1999.

[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, "AES Second Round Implementation Experience," http://www.btinternet.com/~brian.gladman/cryptography_technology/aes2/index.htm, March 1999.

[5] Helger Lipmaa, "AES Ciphers: Speed," http://home.cyber.ee/helger/aes/table.html, 1999.