Performance of the AES Candidate Algorithms in Java

Andreas Sterbenz, Andreas.Sterbenz@iaik.at
Peter Lipp, Peter.Lipp@iaik.at
Institute for Applied Information Processing and Communications
Graz, University of Technology
Inffeldgasse 16, A-8010 Graz, Austria
http://www.iaik.at

Abstract

We analyze the five remaining AES candidate algorithms MARS, RC6, Rijndael, Serpent, and Twofish as well as DES, Triple DES, and IDEA by examining independently developed Java implementations. We give performance measurement results on several platforms, list the memory requirements, and present a subjective estimate for the implementation difficulty of the algorithms. Our results indicate that all AES ciphers offer reasonable performance in Java, the fastest algorithm being about twice as fast as the slowest.

1. Introduction

The performance of the AES candidates has been the subject of significant discussion, both in the authors' specifications as well as by other parties. Most of this discussion was focused on C and assembler implementations. Some attention has been given to Java implementations but the results were not fully conclusive. This was mostly caused by the fact that the authors' reference Java implementations were evaluated which vary significantly in their coding assumptions and in the degree to which they were subject to optimizations. We intend to fill this gap by evaluating independently developed, consistent Java implementations and comparing the AES candidates' performance to ciphers currently in use.

2. Implementation Notes

The code was developed at the IAIK by Andreas Sterbenz. The AES core code is available under a free license including source at [1] or with a JCE 1.2 compatible API as part of the IAIK JCE library. Serpent S-Box expressions and Rijndael and Twofish setup code are based on C code developed by Dr. Brian Gladman [2].

The design paradigm used is derived from the Java Cryptography Extension (JCE) defined by Javasoft and modified for use within the IAIK JCE library: for each cipher stream a Java object is created which is then initialized with a certain key in either encryption or decryption mode. Then the data to be encrypted is passed to the encrypt (decrypt) method one 128 bit block at a time. Buffering, block chaining, and padding are all performed on a higher level and do not influence the design of the core code. Therefore, for each AES cipher only three methods need to be provided: key setup, encryption, and decryption.

The algorithms have been subject to significant optimization work. The primary focus for the optimization was to maximize encryption and decryption throughput. Secondary and tertiary goals were key setup speed and memory usage, respectively.

3. Java

The Java programming language has become fairly popular in recent years. This is partly due to the fact that Java programs are platform (i.e. processor and operating system) independent in both source and binary form. This is possible by employing a compilation model different from that in most other languages. Instead of compiling source code into machine code for one particular processor family, the compiler produces machine code (called "bytecode") for an imaginary Java Virtual Machine (JVM). At runtime this bytecode is then translated into machine code by a JVM implementation for the particular platform.

This extra step has influences on the programming process when optimizing code. It takes you one step farther away from the hardware making some typical optimization tricks impossible, like for example directly using the processor rotation instruction. Another problem is that a sizable portion of the compilation is delayed until runtime and performed by the JVM. As they are not designed for optimizations this has the effect that those optimizations are not made.

Of course there are several options for the translation of bytecode to machine code. The simplest and most obvious is to use an interpreter: take one JVM instruction at a time and execute the corresponding machine code instruction(s). Much better performance is offered by so-called Just-In-Time (JIT) compilers. They take an entire method and translate it to machine code prior to its first execution, subsequently

the generated machine code is executed. JITs are now the common JVM type on most platforms and offer an approximately ten times performance improvement over interpreters. As a third type of JVMs there are hybrid variants aimed at reducing the initial delay caused when the JIT compilers translate a large number of methods at program startup, but this is not relevant for our application.

3.1. Java in Cryptographic Applications

Today the opinion that Java is not the language to be used for cryptographic applications still seems to be popular. Obviously we do not agree. While Java is of course slower than C the difference is typically less than a factor of two, heavily optimized C code excluded, as demonstrated by the results presented in this paper. Although this difference is of course significant Java on today's hardware is faster than C on two year old hardware. The point being that while Java will hardly be the language of choice for high load servers it may well be the choice for medium load servers and especially clients. Add to that handheld and other small devices and performance in Java becomes an issue.

One particular advantage of Java is that there is a well established standard cryptographic API, the JCA and JCE architecture from Javasoft. The success of cryptography libraries in Java including the libraries from the IAIK confirms this position.

4. Evaluation Parameters

The algorithms were implemented in Java. Those implementations were evaluated with respect to three criteria: execution speed, memory usage, and implementation difficulty.

4.1. Execution Speed

For symmetric ciphers there are three components that make up the time required to encrypt some data: static initialization time, key setup time, and data encryption time.

Static initialization is used to perform certain preparation steps, generate constant tables, etc. Because it takes very little time and is largely dependent on the code size vs. speed tradeoff chosen in the implementation it was not measured.

Key setup is used to initialize a cipher for a certain key, i.e. perform round key generation, etc. It is performed once per encryption stream. It may be dependent on whether encryption or decryption mode is chosen and on the key length. For the ciphers analyzed only Rijndael and IDEA have different key setup times for encryption and decryption modes and only Rijndael and Twofish significantly different setup times for different key lengths.

Data encryption time is of course the time it takes to encrypt data bits once the cipher has been properly initialized. The AES candidates are 128 bit block ciphers, that means one encryption operation is performed every 16 data bytes. Again it may vary with the cipher's mode and key length. For all ciphers analyzed the encryption and decryption times are virtually identical and only Rijndael's performance is dependent on the key length.

4.1.1. Key Setup Speed Measurement

Key setup speed was determined as described by the following pseudo code:

```
Repeat 128 times
  Generate 32 random keys
  Start timer
  For each key
    Repeat 1024 times
      Initialize cipher with key
  Stop timer
```

To obtain the final value the average of all measurements within three standard deviations was calculated.

4.1.2. Encryption Speed Measurement

Similarly encryption speed was measured:

```
Repeat 128 times
  Generate a random key
  Initialize cipher with key
  Start timer
  Repeat 2048 times
    Encrypt a 1024 byte array
  Stop timer
```

The same method as above was used to obtain the final value. Note that the same 1024 byte array is encrypted each time which takes full advantage of the CPU caches. In other words, the results presented here are upper boundaries for real world performance.

4.1.3. Environment

The code was compiled using Symantec Visual Cafe 2.5a with optimizations enabled. The results were obtained by running the tests on a machine with an Intel Pentium Pro 200 MHz CPU and 128 MB RAM running Windows NT 4.0 with Service Pack 4. Performance wise this is virtually identical to the NIST reference platform (64 MB RAM and running Windows 95).

However, it should be noted that the actual development and optimization was done on a machine using an AMD K6-2 processor. The optimizing process, which includes trial and error strategies was performed to maximize throughput on this machine and not the reference machine. This may in some cases

	DES	Triple DES	IDEA	MARS	RC6	Rijndael	Serpent	Twofish
Class File Size	n/a	n/a	n/a	9984	1931	4900	12483	5204
Per process memory	5120	5120	0	3220	0	20520	0	6816
Per instance memory	128	384	416	220	432	240	576	4400

Table 1: Class file size and memory usage in bytes.

lead to cases were the performance on the reference machine is not as good as it could be.

4.2. Memory Usage

We give an estimate of the memory required for each of the algorithms. The size of the class file (debugging information removed) is listed to give an idea of the total size, consisting of code size and data like S-Box tables, etc. This is only done for the AES candidates because the other algorithms use a slightly different API which would skew results.

Probably more interesting is the amount of memory required during execution. We list the data memory used obtained by counting the variables used in the source code. Overhead for arrays or data allocated on the stack is not counted as it is fairly small and approximately identical for all of the algorithms.

4.3. Implementation Difficulty

We also assign implementation difficulty "grades" to the algorithms. In difference to the other criteria these were not measured but are subjective estimates for the time it required to arrive at an acceptably fast implementation of the algorithm. If we want to look at it in a quasi formal way we identify the following factors:

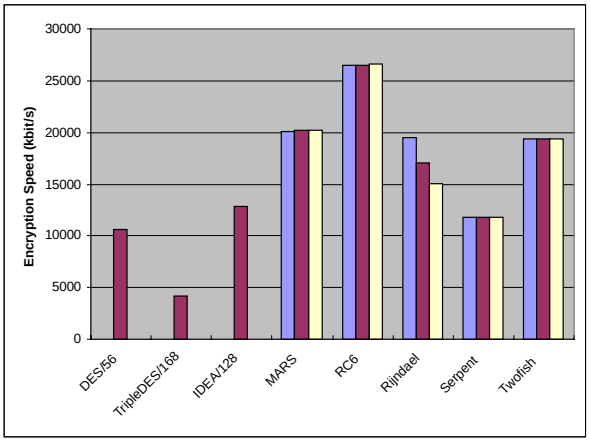
- Time taken to understand the algorithm (at least well enough to be able to implement it).
- Time taken to understand how to efficiently implement the algorithm on a 32 bit platform. As some algorithms need to be coded very differently from their specification in order to be efficient this part may constitute a significant part of the total time.
- Time taken to actually code the implementation.

The first two points are of course to some degree dependent on the documentation provided by the algorithm designers and other parties. Therefore, new or improved documentation may update the results given here.

5. Algorithms

5.1. DES

The Data Encryption Standard (DES) is the current US standard which the AES will eventually replace. It dates back from the 1970s and has become inadequate in particular because of its key length of only 56 bit. DES was designed for hardware implementations and requires tricks to operate moderately fast in 32 bit software implementations. These tricks are not obvious which is why DES only earns a B- for implementation difficulty. However, an advantage of DES over all other algorithms examined except Triple DES is that the encrypt and decrypt operations are identical save for the key schedule resulting in smaller code.



5.2. Triple DES

Triple DES overcomes the limitation of the short DES key length by using three DES cores with separate keys in sequence. This results in an effective strength of 112 bit (meet in the middle attacks) at the price a significant performance drop. Triple DES only performs somewhat faster than one third of the speed of DES (reduced overhead, leaving off the initial and final permutations), which means it is very slow in software. Implementation difficulty is B- as with DES.

5.3. IDEA

IDEA is a 16 bit oriented cipher which uses multiplication modulo 65537 for fast diffusion. Consequently it performs quite well compared to DES (depending on the processors multiplication unit). However, its key setup is quite slow in decryption mode as multiplicative inverses have to be calculated. It has also to be noted that a class of weak keys has been discovered. For implementation difficulty it earns B+ as that is fairly straight forward.

5.4. MARS

MARS is the first of the AES candidates we examine. It uses 8 rounds of unkeyed mixing before and after the core encryption rounds. One of its advantages is that a 32 bit implementation can be written exactly the way the algorithm is specified, also aided by the pseudo code given in the specification. Implementation difficulty B+.

5.5. RC6

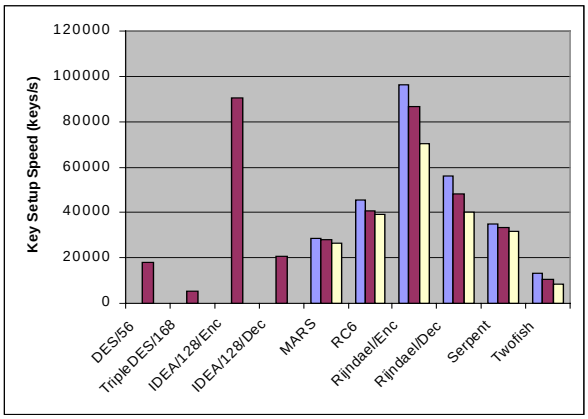
RC6 is a cipher that evolved from RC5. It is very simple to understand and implement and very fast on 32 bit processors; implementation difficulty A. Although the least time was spent on optimizing RC6 it still comes out as the fastest algorithm on almost all platforms.

5.6. Rijndael

Rijndael was designed based on strong mathematical foundations. Implemented on 32 bit processors only table lookup, XOR and shift operations are used. The number of rounds in the Rijndael cipher increases with the key length resulting in decreasing speed for both key setup and encryption. Key setup for Rijndael is very fast for in encryption mode but slower in decryption mode as an additional inversion step is required. Implementation difficulty B.

5.7. Serpent

Serpent was designed for so-called bitslice implementations. The idea is to view a 32 bit register as 32 one bit registers which are operated on by 32 one bit SIMD processors with e.g. logical operations. However, S-Boxes have to be implemented via logical expressions in this mode. Efficient expressions are not trivial to obtain and no expressions are given in the specification, contributing to the B- grade for implementation difficulty. Serpent was designed with a large safety margin of 32 rounds vs. about 20 minimum secure rounds. This results in lower speed, the



penalty depending on the JVM implementation and the processor.

5.8. Twofish

Twofish is a very flexible cipher that allows for several implementation options allowing a memory usage vs. key setup speed vs. encryption speed tradeoff. As maximum encryption throughput was desired the "full keying" option was chosen for the implementation. A special property of Twofish is that key dependent S-Boxes are used. This somewhat hurts performance on certain JVMs, in particular when using the Symantec JIT compiler that comes with the JDK on the Windows platform and which was used for the measurements. This means that Twofish may be somewhat faster compared to the other algorithms on other platforms. As Twofish is a quite complicated cipher it earns B- for implementation difficulty.

6. Conclusions

We have analyzed the performance of the AES candidate and other ciphers.

The results for encryption and decryption speed show that RC6 is about 25% faster than the other algorithms. Then MARS, Rijndael, and Twofish follow with virtually identical performance for 128 bit keys, Rijndael being slower for longer keys. Serpent is trailing behind but is still about as fast as IDEA. DES follows with Triple DES far behind. These results are similar to some tests made using C implementations but deviate much from Java studies. The results also show that Java is no more than a factor of 2-3 slower than optimized C code.

The key setup performance is more varied with the fastest AES candidate more than 7 times as fast as the slowest. This appears to be partly due to differing opinions about the purpose of the key schedule. It could be viewed as a one way hash function: accepting an arbitrarily long key, producing output of fixed length (the round keys). All round keys depend on all input bits and obtaining a round key (using some

attack) does not yield any information about the original key. Some algorithms try to approximate this ideal while others only generate the necessary key material in a straight forward way.

In any case Rijndael is the fastest algorithm with respect to key setup, although it is not that far ahead when keys longer than 128 bit are used and in decryption mode. Twofish has a fairly slow key setup using this implementation option.

In summary it can be said that if properly implemented all algorithm offer reasonable performance in Java. The results are mostly in line with those obtained by studies evaluating C implementations.

7. References

[1] IAIK. *The IAIK AES home page* http://jcewww.iaik.at/aes/

[2] Brian Gladman. *AES Implementations in C* http://www.btinternet.com/~brian.gladman/cryptography_technology/aes2/aes.r2.algs.zip

[3] X. Lai, J.L. Massey and S. Murphy. *Markov ciphers and differential cryptanalysis* Advances in Cryptology, Proceedings Eurocrypt'91, LNCS 547, D.W. Davies, Ed., Springer-Verlag, 1991, pp. 17-38.

[4] Jim Dray. *Report on the NIST Java AES Candidate Algorithm Analysis* Available from http://csrc.nist.gov/encryption/aes/round2/round2.htm

[5] Lawrence E. Bassham III. *Efficiency Testing of ANSI C Implementations of Round1 Candidate Algorithms for the Advanced Encryption Standard* http://csrc.nist.gov/encryption/aes/round2/round2.htm

[6]

Appendix

This appendix includes the full performance figures as obtained on the reference machine.

Encryption Speed (kbit/s)	DES (56 bit)	Triple DES (168 bit)	IDEA	MARS	RC6	Rijndael	Serpent	Twofish
128 bit key	10508	4178	12820	19718	26212	19321	11464	19265
192 bit key	n/a	n/a	n/a	19760	26192	16922	11474	19296
256 bit key	n/a	n/a	n/a	19737	26209	14957	11471	19275

Decryption Speed (kbit/s)	DES (56 bit)	Triple DES (168 bit)	IDEA	MARS	RC6	Rijndael	Serpent	Twofish
128 bit key	10519	4173	13018	19443	24338	18868	11519	18841
192 bit key	n/a	n/a	n/a	19670	24382	16484	11514	18841
256 bit key	n/a	n/a	n/a	19489	24279	14468	11533	18806

Encryption Key Setup (keys/s)	DES (56 bit)	Triple DES (168 bit)	IDEA	MARS	RC6	Rijndael	Serpent	Twofish
128 bit key	18128	5150	90571	28680	45603	96234	34729	13469
192 bit key	n/a	n/a	n/a	27928	40625	86773	33516	10556
256 bit key	n/a	n/a	n/a	26683	29069	70494	31973	8500

Decryption Key Setup (keys/s)	DES (56 bit)	Triple DES (168 bit)	IDEA	MARS	RC6	Rijndael	Serpent	Twofish
128 bit key	18039	5136	20737	28743	45709	56017	34687	13469
192 bit key	n/a	n/a	n/a	27917	40625	48324	33560	10550
256 bit key	n/a	n/a	n/a	26731	39028	39963	31973	8531