NIST Performance Analysis of the Final Round Java™ AES Candidates

Jim Dray
Computer Security Division
The National Institute of Standards and Technology
james.dray@nist.gov

March 15, 2000

1. Introduction

NIST solicited candidate algorithms for the Advanced Encryption Standard (AES) in a Federal Register Announcement dated September 12, 1997[1]. Fifteen of the submissions were deemed "complete and proper" as defined in the Announcement, and entered the first round of the AES selection process in August 1998. Since that time, NIST has been working with a worldwide community of cryptographers to evaluate the submissions according to the criteria established in[1]. Five candidates were subsequently chosen to enter the final round of the selection process: MARS, RC6, Rijndael, Serpent, and Twofish.

A previous NIST publication entitled "Report on the NIST Java™ AES Candidate Algorithm Analysis"[2] documents the first round analysis performed by NIST, using the Java Development Kit (JDK) Version 1.1.6. Only IBM has submitted official modifications to their candidate (MARS) prior to the final round. Results of the first round analysis using the JDK1.1.6 are therefore still valid for the other four candidates. The revised version of MARS was tested under both JDK1.1.6 and JDK1.3, to ensure an accurate comparison of the modified algorithm's performance in both environments. Performance data for 128, 192, and 256-bit keysizes are also included in the second round analysis.

The JDK itself has gone through two major revisions since the first round. This paper documents additional performance data for the five AES finalists obtained under JDK1.3, and should be used in combination with the first round NIST Java AES analysis to obtain a complete picture of the characteristics of the finalists in different Java environments. Some background information from the first round analysis is repeated herein for convenience. Comments should be addressed to the author at the email address above.

2. Java Platform

AES candidate algorithm submitters were required to provide optimized implementations of their algorithms in Java and the C language. The rationale for this was to provide more information than could be obtained by testing implementations in a single language, and to take advantage of the hardware independence of the Java virtual machine.

The Java virtual machine presents a uniform abstraction of the underlying hardware platform to a Java application or applet. A Java programmer compiles source code into byte code files, which are then interpreted by the Java virtual machine at runtime (byte code files are also known as class files). In theory, a Java byte code file can be interpreted on any hardware platform running the Java virtual machine without recompilation. Since the virtual machine isolates the Java programmer from the underlying hardware, Java programmers cannot write machine-specific code to take advantage of the unique features of a particular platform. Machine-specific code allows for optimization on a given computing platform, but also eliminates the code portability that is a cornerstone of the Java philosophy.

The Java environment has two characteristics that facilitate the AES evaluation process. First, candidate algorithms written in Java can be easily moved from one platform to another to compare performance on different processors at different system clock speeds. Second, submitters cannot write machine-specific code and so all implementations are on a level playing field.

Java does not provide the level of performance that can be attained in some other languages (C or assembler, for example). However, many applications do not require high-speed encryption of large amounts of data, and cryptoalgorithms implemented in Java are easier to integrate into Java applications. Other languages and hardware implementations will be used for applications where absolute performance is an issue, but there will also be a broad range of applications where the ease of implementing, integrating, and maintaining Java AES code outweighs the performance issue.

3. Evaluation Criteria

The NIST Java AES evaluation process is designed to directly address the criteria published in the Federal Register Announcement[1], Section 4. The goal is to provide objective results that can be clearly quantified for use in the selection process. Sections of the Announcement that describe selection critera relevant to the Java AES analysis are repeated here for convenience:

COST

ii. *Computational Efficiency:* "…Computational efficiency essentially refers to the speed of the algorithm. NIST's analysis of computational efficiency will be made using each submission's mathematically optimized implementations on the platform specified under Round 1 Technical Evaluation below."

iii. *Memory Requirements:* "Memory requirements will include such factors as gate counts for hardware implementations, and code size and RAM requirements for software implementations."

ALGORITHM AND IMPLEMENTATION CHARACTERISTICS

i. *Flexibility:*

b. "The algorithm can be implemented securely and efficiently in a wide variety of platforms and applications (e.g. 8-bit processors, ATM networks, voice & satellite communications, HDTV, B-ISDN, etc.)."

ii. *Simplicity:* "A candidate algorithm shall be judged according to relative simplicity of design."

Additionally, in Section 6.B (**Round I Technical Evaluation**):

iii. *Efficiency testing:* "Using the submitted mathematically optimized implementations, NIST intends to perform various computational efficiency tests for the 128-128 key-block combination, including the calculation of the time required to perform:

o Algorithm setup,
o Key setup,
o Key change, and
o Encryption and decryption.

NIST may perform efficiency testing on other platforms."

In condensed form, the published NIST criteria require testing of speed for a set of cryptographic operations, code size and RAM requirements, flexibility, and simplicity of design. Since the candidates have been implemented in Java, flexibility is a given for the reasons discussed in the previous section. The Java AES candidates will run on any device containing a Java virtual machine and adequate memory, although performance will obviously vary depending on the processing power of the underlying hardware.

4. Test Procedures

4.1 Overview

The test results presented here were obtained from the NIST-specified hardware platform and the most recent version of the Java environment available at the time of this writing (JDK1.3, beta release). Results for other hardware/Java virtual machine combinations will be made available on the AES home page at http://www.nist.gov/aes, and in papers submitted to NIST by other organizations[3,4,5]. Detailed test results are presented in tabular form in Appendices A and B, and chart form in Appendix C. All NIST testing was performed through the Applications Programming Interface (API) specified in the NIST/Cryptix Java AES Toolkit. Links to the Toolkit and the Java AES API specification can be found at http://csrc.nist.gov/encryption/aes/earlyaes.htm.

The Java compiler provided with JDK1.3 accepts a command line code optimization switch (-O). However, the JDK1.3 documentation[6] states that this switch "does nothing in the current implementation". Presumably the compiler accepts the optimization switch for reasons of backward compatibility.

4.2 Procedures

Candidate algorithms were compiled from source files provided by submitters using the JDK1.3 compiler. The resulting bytecode files were packaged into a standard Java ARchive (JAR) file named AESCLASSES.jar.

A Java application was developed to allow testing of any candidate/ keysize/operation combination. The test application instantiates the desired candidate from AESCLASSES.jar, and uses the Java reflection API to invoke the Basic API methods.

500,000 cycles of each candidate/keysize/crypto operation were executed, and the total time was recorded for each combination. Start and stop times were obtained through calls to the System.time.millis() method provided in the Java core library, immediately before and after starting the loop that executed the crypto operations. Charts 1, 2, and 3 present performance data for key setup, encrypt, and decrypt operations, respectively. Data points are included for 128, 192, and 256-bit key sizes. For the majority of candidates, encryption and decryption speed is approximately equal for all three key sizes. Rijndael is a minor exception: encryption speed decreases by approximately three percent for each stepwise increase in key size.

5. Results

In comparison to the JDK1.1.6 performance data presented in NIST's previous paper[2], the results obtained with JDK1.3 show a striking increase in execution speed for all candidates. On average, the five candidates perform 128-bit key setup operations eleven times faster. The average speed for encrypt and decrypt operations has increased by a factor of five. The same hardware platform and program code (except for MARS) were used for both first round and final round testing, so the overall increase in performance can be attributed to differences in the Java environment. In particular, JIT (Just-In-Time) compilation was not used during the first round performance analysis due to a bug in the JDK1.1.6 JIT implementation that caused problems with certain candidates. Usage of the JIT compiler under JDK1.1.6 increases performance by a factor of ten for most Java programs.

Performance data for the new version of MARS under JDK1.1.6 are presented separately in Appendix A. The test setup for the MARS/JDK1.1.6 analysis was exactly the same as for the other algorithms during the first round, and is described in[2].

In addition to the overall performance increase of the finalists under JDK1.3, there were some changes in the relative ordering of candidates. Most of these changes in order were due to relatively small performance differences, as shown in Appendices B and C. The results for 128-bit keysize operations are summarized below, with candidates ordered from fastest to slowest:

128-bit Key Setup:

JDK1.1.6:	Rijndael, RC6, MARS, Twofish, Serpent
JDK1.3:	RC6, MARS, Rijndael, Serpent, Twofish

128-bit Encrypt:

JDK1.1.6:	RC6, Rijndael, MARS, Serpent, Twofish
JDK1.3:	Rijndael, RC6, MARS, Serpent, Twofish

128-bit Decrypt Operations:

JDK1.1.6:	RC6, Rijndael, MARS, Serpent, Twofish
JDK1.3:	Rijndael, RC6, MARS, Twofish, Serpent

REFERENCES

1. "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)", Federal Register: September 12, 1997 (Volume 62, Number 177), Pages 48051-48058.

2. J. Dray, "Report on the NIST Java™ AES Candidate Algorithm Analysis", http://csrc.nist.gov/encryption/aes/round1/r1-java.pdf , November 8, 1999.

3. A. Folmsbee, "AES Java™ Technology Comparisons", Proceedings of the Second Advanced Encryption Standard Candidate Conference, March 22, 1999, Pages 35-50.

4. K. Aoki, "Java Performance of AES Candidates", Submitted to NIST via email in response to the call for public comments on the AES candidates, April 15, 1999.

5. A. Sterbenz, P. Lipp, "Performance Analysis of Java Implementations of the Second Round AES Candidate Algorithms", Submitted to NIST via email, January 2000.

6. Java™ 2 SDK, Standard Edition Documentation (Version 1.3), http://java.sun.com/products/jdk/1.3/docs/.

APPENDIX A: JDK1.1.6 DATA FOR MARS

Key Size	Key Setup	Encrypt	Decrypt
128 bits	165	462	444
192 bits	244	466	444
256 bits	324	465	445

Table data are presented in kilobits per second.

APPENDIX B: RAW DATA TABLES

Algorithm	setKey128	setKey192	setKey256
RC6	2233	3335	4444
MARS	2110	3131	4131
Rijndael	1191	1574	1733
Serpent	487	734	979
Twofish	286	327	361

Algorithm	Encrypt128	Encrypt192	Encrypt256
Rijndael	4855	4664	4481
RC6	4698	4740	4733
MARS	3738	3707	3733
Serpent	1843	1855	1861
Twofish	1749	1749	1744

Algorithm	Decrypt128	Decrypt192	Decrypt256
Rijndael	4819	4624	4444
RC6	4733	4698	4740
MARS	3965	3965	3936
Serpent	1873	1897	1896
Twofish	1781	1775	1781

Table data are presented in kilobits per second.

APPENDIX C: PERFORMANCE DATA CHARTS

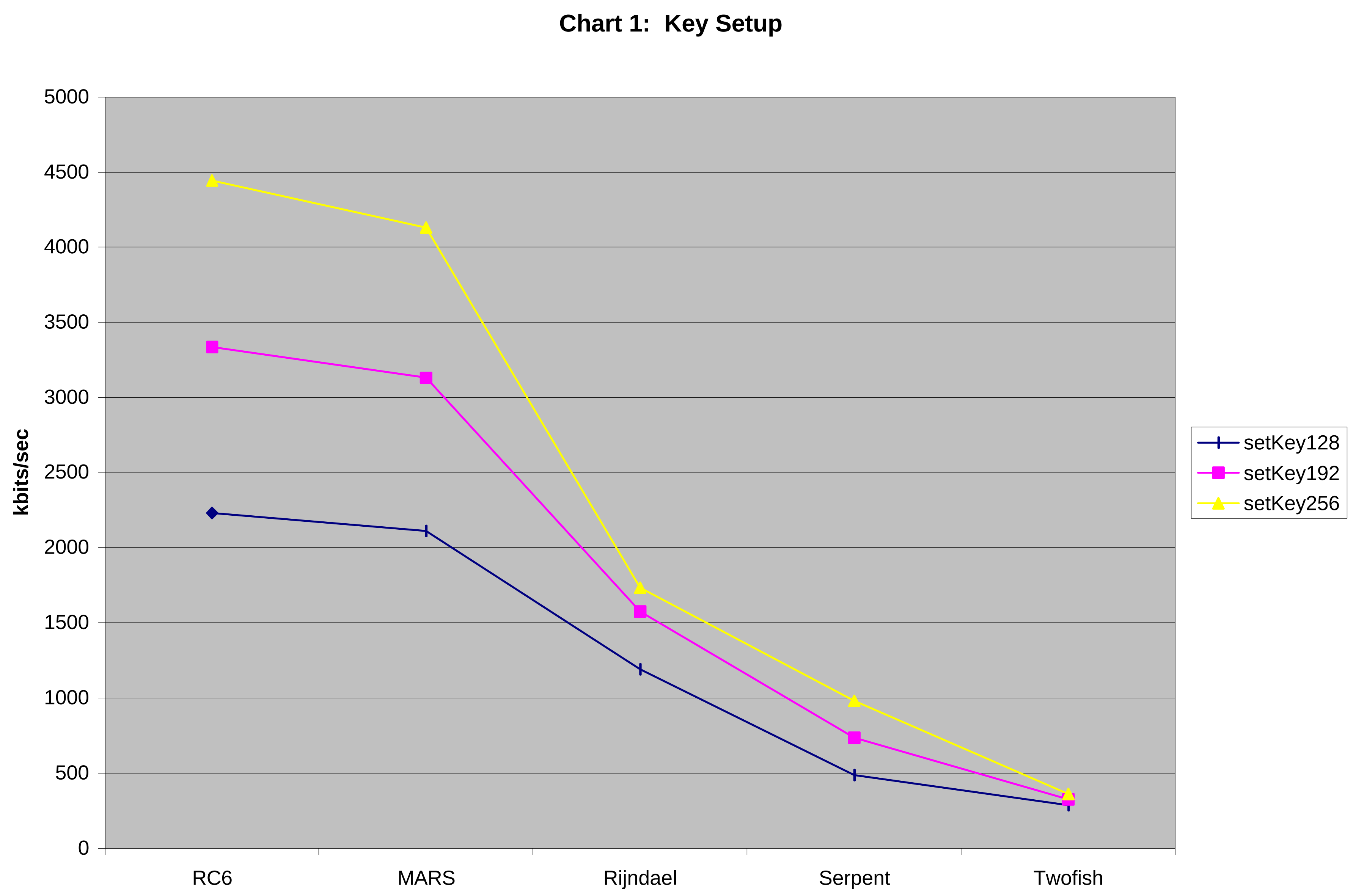


Chart 1: Key Setup
kbits/sec
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
RC6
MARS
Rijndael
Serpent
Twofish
setKey128
setKey192
setKey256

Chart 2: Encrypt

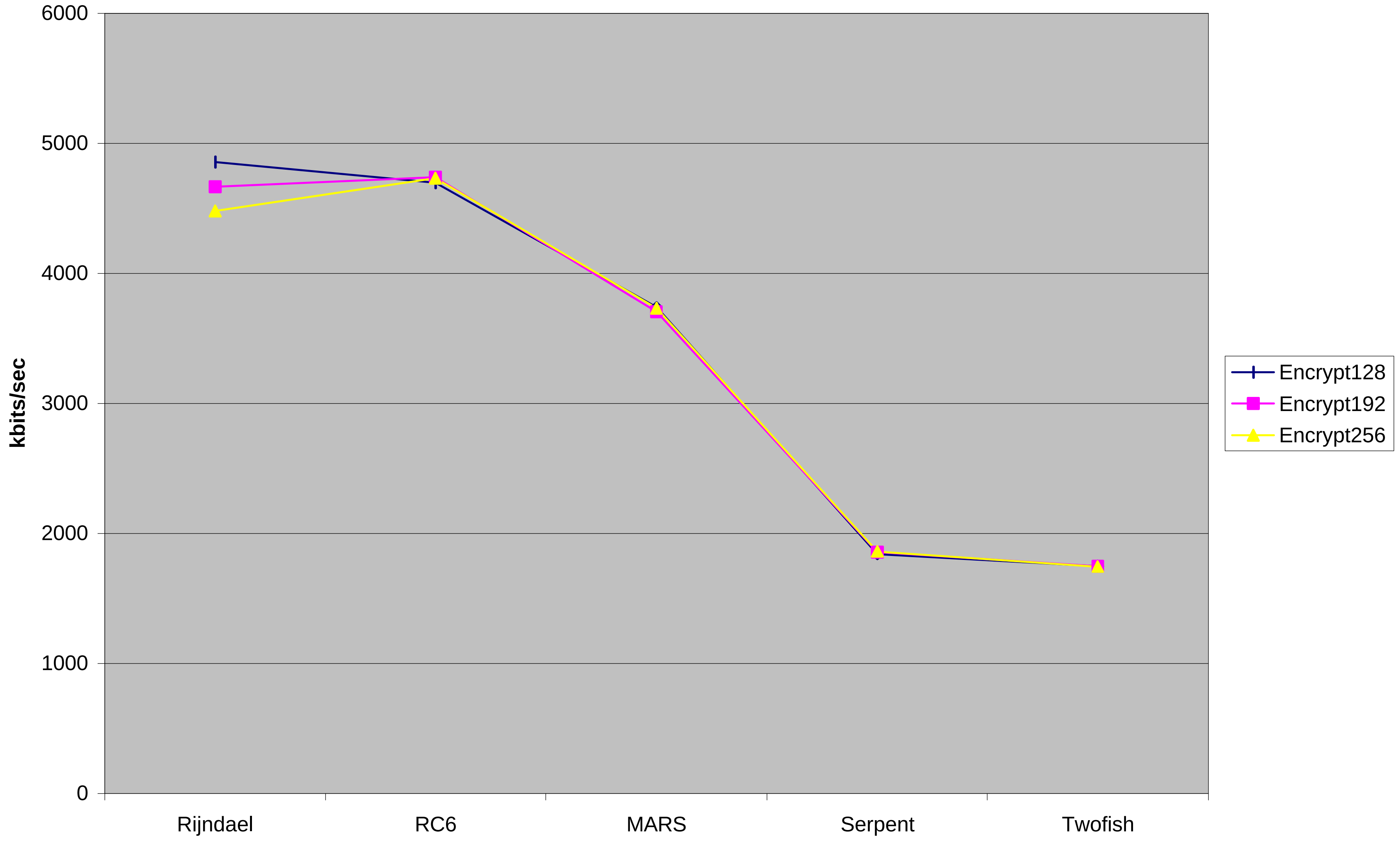
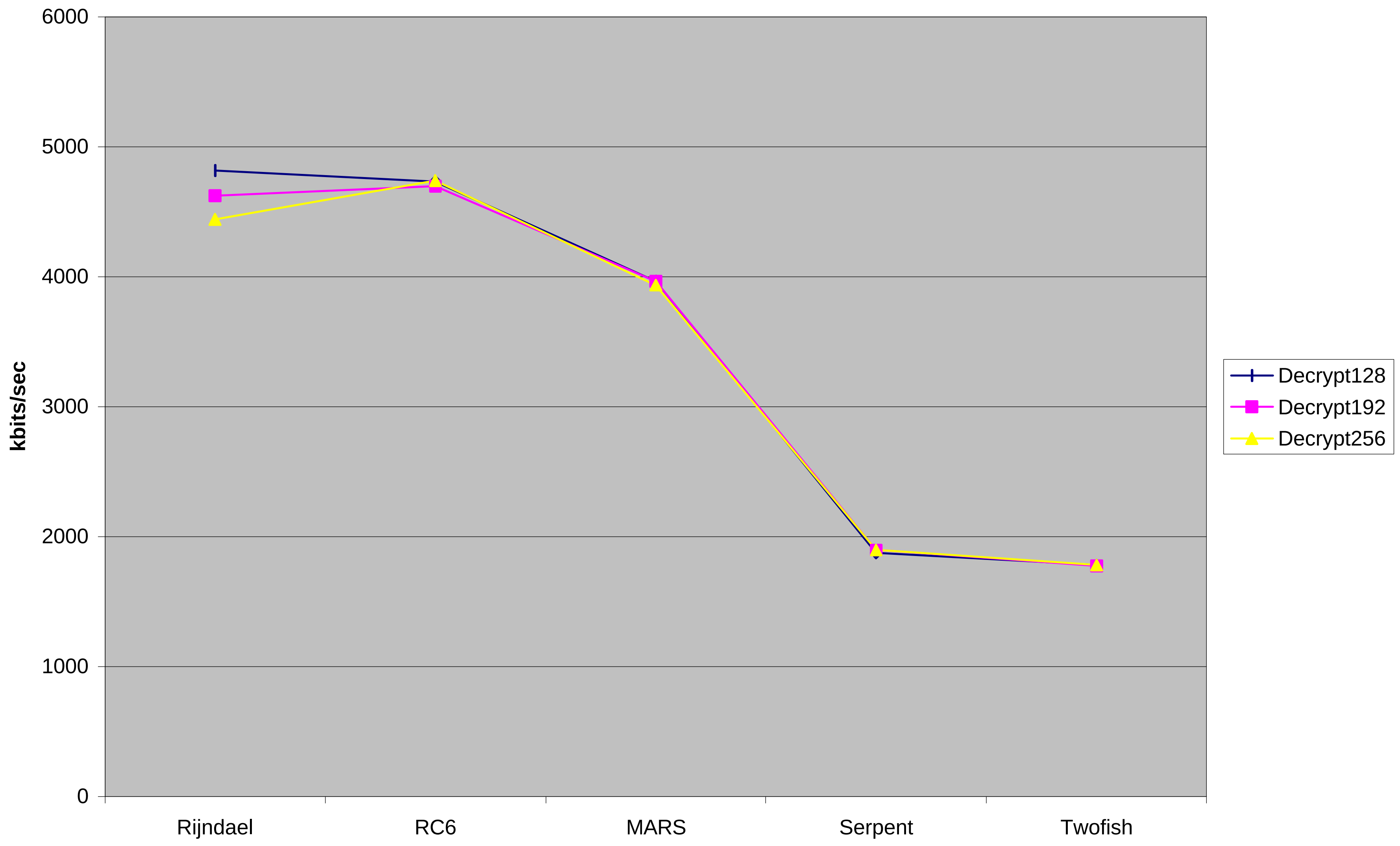


Chart 2.3: Decrypt


kbits/sec
Rijndael
RC6
MARS
Serpent
Twofish
Decrypt128
Decrypt192
Decrypt256