Report on the NIST Java™ AES Candidate Algorithm Analysis

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

November 8, 1999

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 subsequently 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]. This paper documents the test environment created by NIST to evaluate the optimized Java™ implementations provided by submitters, and the results of this evaluation to date. 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 first-round 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:

- Algorithm setup,
- Key setup,
- Key change, and
- 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 1.1 virtual machine and adequate memory, although performance will obviously vary depending on the processing power of the underlying hardware. Test results for the remaining selection criteria are presented in the next section.

4. Test Procedures and Results

4.1 Overview

The test results presented here were obtained from the NIST-specified hardware platform and Java environment (JDK1.1.6). 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]. Test results for DES were generated by using the Java implementation of the DES algorithm included in the DEAL candidate submission package. For each test category, results are summarized by dividing the candidates into three groups based on performance. Detailed test results are presented in tabular form in Appendix A and bar chart form in Appendix B. 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 provides an option to optimize code during compilation via a command-line flag (-•). The primary mechanism for optimization is to include classes in the compiled Java bytecode files that would normally be loaded dynamically at runtime. This speeds up program execution, but also makes the compiled bytecode files larger.

Java AES candidates were compiled without optimization for memory usage measurements, and with optimization for speed measurements. This provides optimal results for each test category. However, readers should note that the compiled class files used to measure dynamic memory allocation were therefore different from those used to measure speed, although they were compiled from the same source files (the –O option was used for speed measurements).

4.2 Static Memory Usage

Static code sizes were taken from a DOS directory listing. Properties files were not included in size estimates, but required external "helper" classes were. In some cases implementers chose to build extra functionality into the class files that implement the basic NIST/Cryptix API methods, making it difficult to determine an exact file size for just the core API methods. For example, the Java source file HPC_Algorithm.java contains a large number of methods that perform utility and/or test functions in addition to the basic API methods. In these cases, the total class file size was used. Candidates can be categorized as follows, in kilobytes:

20 to 40kb:	Serpent, HPC, CAST256
10 to 19kb:	MARS, Twofish, DEAL, E2, Rijndael, Crypton, Safer+
Under 10kb:	LOKI97, DFC, RC6, MAGENTA, frog

Detailed results are presented in Chart 1.

4.3 Dynamic (Heap) Memory Usage

A Java test harness was developed to measure heap usage. This test program instantiates any AES candidate algorithm and exercises setkey/encrypt/decrypt operations for any of three key sizes. The -O (optimize) option was not used because it prevents generation of debug tables needed by the Java profiler.

A DOS batch file was then created to run the Java test class for each algorithm/keysize/ operation combination using the java_g -prof -noasyncgc options to collect runtime profile statistics. The –noasyncgc option turns asynchronous garbage collection off. This allows measurement of the total dynamic memory used by an algorithm during its execution.

Heap usage statistics were extracted from the raw data files created by the Java profiler, and assembled into an ASCII comma-delimited file format. This file was imported into a Microsoft Excel™ worksheet, and used to generate Chart 2.

Dynamic memory usage statistics were normalized by subtracting the heap usage of a minimal NULL cipher from each entry, since the differences between candidates were minor relative to the total heap used by most candidates. The NULL cipher implementation allocates 110,408 bytes of heap for key setup, and 111,240 bytes for cipher operations. The amount of heap allocated by all candidates for encrypt and decrypt operations is the same, so Chart 2 simply lists these as cipher operations. The candidates fall into the following three ranges for adjusted heap usage, in kilobytes:

14 to 20kb:	Rijndael, DFC, and LOKI97
6 to 10kb:	HPC, DEAL, Twofish, Crypton, CAST256
Less than 6kb:	MARS, Serpent, frog, Safer+, MAGENTA, RC6

E2 is not listed above or on Chart 2 because it is an exceptional case. It uses 264kb of heap, roughly ten times the heapspace required by the next highest candidate (Rijndael). This can be attributed to the fact that the Java implementation of E2 allocates an array of 256 x 256 32-bit integers. It should be noted that E2 can be implemented in ways that consume far less memory, but the results in this paper are derived from the official implementation of E2 submitted to NIST.

4.4 Computational Efficiency (Speed)

Candidate algorithms were compiled from source files provided by submitters. The JDK1.1.6 compiler was used, with the -O (optimize) option. 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. The -nojit option was used to turn off the default JDK1.1.6 Just-In-Time (JIT) compiler during execution of the test application. Use of the JIT compiler increases the speed of most candidates by an order of magnitude, but a bug in this specific version of the compiler causes problems with some of the candidates. This compiler bug is documented at http://developer.java.sun.com, bugParade ID 4171185. Readers should note that the data presented by NIST at the Second AES Candidate Conference (AES2) differs from the data in this paper. The results presented at AES2 did not use the –O compiler optimization option, and did use the JIT compiler. The data in this paper will be used for the first-round selection process.

Fifty thousand 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 3,4, and 5 present performance data for key setup, encrypt, and decrypt operations respectively. Data points are included for 128, 192, and 256-bit key sizes (the first-round selection process will focus on performance for 128-bit key operations). For the majority of candidates, encryption speed is approximately equal for all three key sizes. Rijndael and Safer+ are the two exceptions: for these candidates, encryption speed decreases as the key size is increased.

MAGENTA is not shown on Chart 3 because it executes 128-bit key setup operations at 6.3mbps, over a thousand times faster than the next fastest candidate (Crypton).

Results for 128-bit key setup in kilobits per second are:

200 to 600kbps:	Crypton, Rijndael, RC6
50 to 199kbps:	Safer+, E2, MARS, LOKI97, CAST256, DEAL
0 to 49kbps:	Twofish, Serpent, HPC, DFC, frog

Results for 128-bit encryption in kilobits per second are:

600 to 1400kbps:	RC6, Rijndael, E2, Crypton
200 to 599kbps:	HPC, MARS, Serpent, CAST256, Twofish, LOKI97, frog
0 to 199kbps:	Safer+, DEAL, MAGENTA, DFC

4.5 Simplicity

One of the stated selection criteria for AES candidates is simplicity. It is more common to measure the complexity of source code. Since one is the inverse of the other, translating between the two domains is straightforward. Most analytic tools present results in terms of complexity, so this paper follows that convention.

Total program size in lines of code is one way to measure complexity. All other factors being equal, larger programs are more complicated than smaller programs because they contain more code. This measure is addressed in Section 4.1. To provide insights into code complexity at a finer level of detail, further analysis of the AES candidates was performed at the method level. International Software Automation's Panorama for Java tool[2] was used for this. Panorama claims to provide objective measures of code quality and complexity, based on a set of metrics standards defined by the test engineer.

The following definitions are summarized from the Panorama for Java$^{\text{TM}}$ reference manual[2]. More detailed descriptions can be found there:

- Size in lines: Average method size measured in lines of source code.
- Cyclomatic Complexity (with case): An initial complexity of 1 is assigned to each method. This initial complexity is incremented by 1 for every decision or loop statement. An N-way switch increments the complexity measure by N.
- Cyclomatic Complexity (without case): Calculated the same way as Cyclomatic Complexity (with case), except that each switch statement increments the complexity measure by 2.
- J-Complexity: Code analysis tools insert instrumentation points into the code to be analyzed, to provide test points for recording data. The number of test points is a measure of code complexity, since more complex code will require more test points for adequate coverage. J-Complexity measures the minimum number of instrumentation points required to record test data in three categories:
    - J-Complexity0: Block test coverage.
    - J-Complexity1: Basic segment test coverage.

- J-Complexity1+: All segment test coverage.

For the purposes of this analysis, standards for the Java AES software quality metrics were established by setting the upper limit for each metric to the highest value attained by any candidate. These metrics are applied at the method level (each value is an average taken across all the methods within a class), and all were assigned an equal weight of 1:

- Size in lines: 196
- Cyclomatic Complexity (with case): 48
- Cyclomatic Complexity (without case): 48
- J-Complexity0: 55
- J-Complexity1: 99
- J-Complexity1+: 122

Chart 6 summarizes the complexity analysis.

5. Related Java AES Analysis Efforts

Two independent Java AES analyses have been performed by groups outside NIST. The first is documented in a paper by Alan Folmsbee (Sun Microsystems), and published in the Proceedings of the Second Advanced Encryption Standard Candidate Conference[3]. A second paper has been submitted to NIST by Kazumaro Aoki (Nippon Telephone and Telegraph)[4].

The Folmsbee paper presents timing data in the context of Known Answer Test(KAT) and Monte Carlo Test(MCT) operations, so a direct comparision to the data presented in this paper in kbits/sec is not possible. Performance data in this paper roughly correlates to NIST data, but there are a number of exceptions. It is interesting to note that some candidates change position in the ranking by a significant amount when KAT and MCT results are compared.

Folmsbee measures memory usage in terms of ROM size and RAM size. These measurements are equivalent to static code size and dynamic memory (heap) usage, respectively. Again there is a rough correlation between data presented in the Folmsbee paper and NIST measurements, with some significant exceptions. Some of these exceptions are probably due to the use of different measurement techniques. Folmsbee measures dynamic memory usage by manually counting program variables, while NIST chose to use the heap statistics provided by the Java profiler.

The Aoki paper addresses computational efficiency, using a variety of hardware platforms and Java compilers/virtual machines. Aoki uses a more recent version of Java, JDK1.1.7. For 128-bit key setup, the order of the candidates is the same as that obtained by NIST except that E2 and Safer+ are reversed. These two differ by less than 5 percent in both assessments, however.

NIST and Aoki also rank the candidates in the same order for speed of encryption and decryption, except that Serpent and MARS are reversed. As was the case for E2 and Safer+ in the rankings for key setup, measurements for Serpent and MARS differ by less than 5 percent. Aoki's measurements are 3.5 percent higher on average than those obtained by NIST. This is not surprising since two different versions of the Java Development Kit were used.

6. Conclusions

The Java virtual machine provides a hardware-independent platform for testing Java implementations of the AES candidates. The syntax of the Java language is superficially similar to some other languages such as C and C++, but the Java execution environment is different in a number of fundamental ways from most other programming language environments. AES implementations in other languages may be faster or slower, and consume more or less memory. The ordering of candidates may also be different in other languages, depending on the efficiency with which a given language handles the data structures and operations required by each candidate. The comparative performance of a candidate implemented in different languages is of interest to system designers, but cross-language comparisons are less useful in terms of the AES evaluation.

NIST has used the results presented in this report as one set of decision points in the process of selecting the five AES finalists. These results will also be of interest to those wishing to use Java implementations of AES, or to create new ones. NIST has not attempted to assign a weight to each of the evaluation criteria because the importance of each criterion is application-dependent. For example, those implementing or using Java implementations on hardware platforms with limited memory will be most concerned with static and dynamic memory usage. In an application requiring high throughput where memory resources are not limited, encryption speed will be the most important factor. Key setup time will be an issue for applications requiring frequent key changes. Implementers can use these results to measure the characteristics of Java AES implementations against the requirements of various applications.

REFERENCES

All of the following documents except the Panorama for Java reference manual are available online at http://csrc.nist.gov/encryption/aes/aes_home.htm.

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. Panorama for Java™ Reference Manual, http://www.softwareautomation.com/java/index.htm.

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.

APPENDIX A: RAW DATA TABLE

Candidate	Code Size (bytes)	Heap Usage	Key Setup (128)	Encrypt (128)	Decrypt (128)	Avg Method Size	Cyclomatic Complexity	J-Complexity
	(bytes)	(bytes)	(kbits/sec)	(kbits/sec)	(kbits/sec)	(lines)		
CAST256	27531	7184	65	395	399	11	2	4
CRYPTON	12018	7513	601	653	653	14	2	4
DEAL	16965	8624	52	140	140	20	4	6
DFC	9623	16160	7	32	31	13	3	5
E2	14748	264840	123	897	918	47	5	10
Frog	4091	3984	0	204	233	14	3	7
HPC	38571	9680	17	506	489	16	6	12
LOKI97	9744	15016	96	294	294	19	4	7
MAGENTA	4975	3168	6295	38	38	15	4	7
Mars	18110	4808	107	492	469	19	3	7
RC6	7077	432	237	1371	1371	36	2	3
Rijndael	12158	18360	279	1129	1129	18	3	6
Safer+	11295	3952	124	180	181	18	3	4
Serpent	39290	4680	31	485	462	41	3	6
Twofish	17189	7600	37	379	379	13	3	5

APPENDIX B: BAR CHARTS

Chart 1: Static Code Size

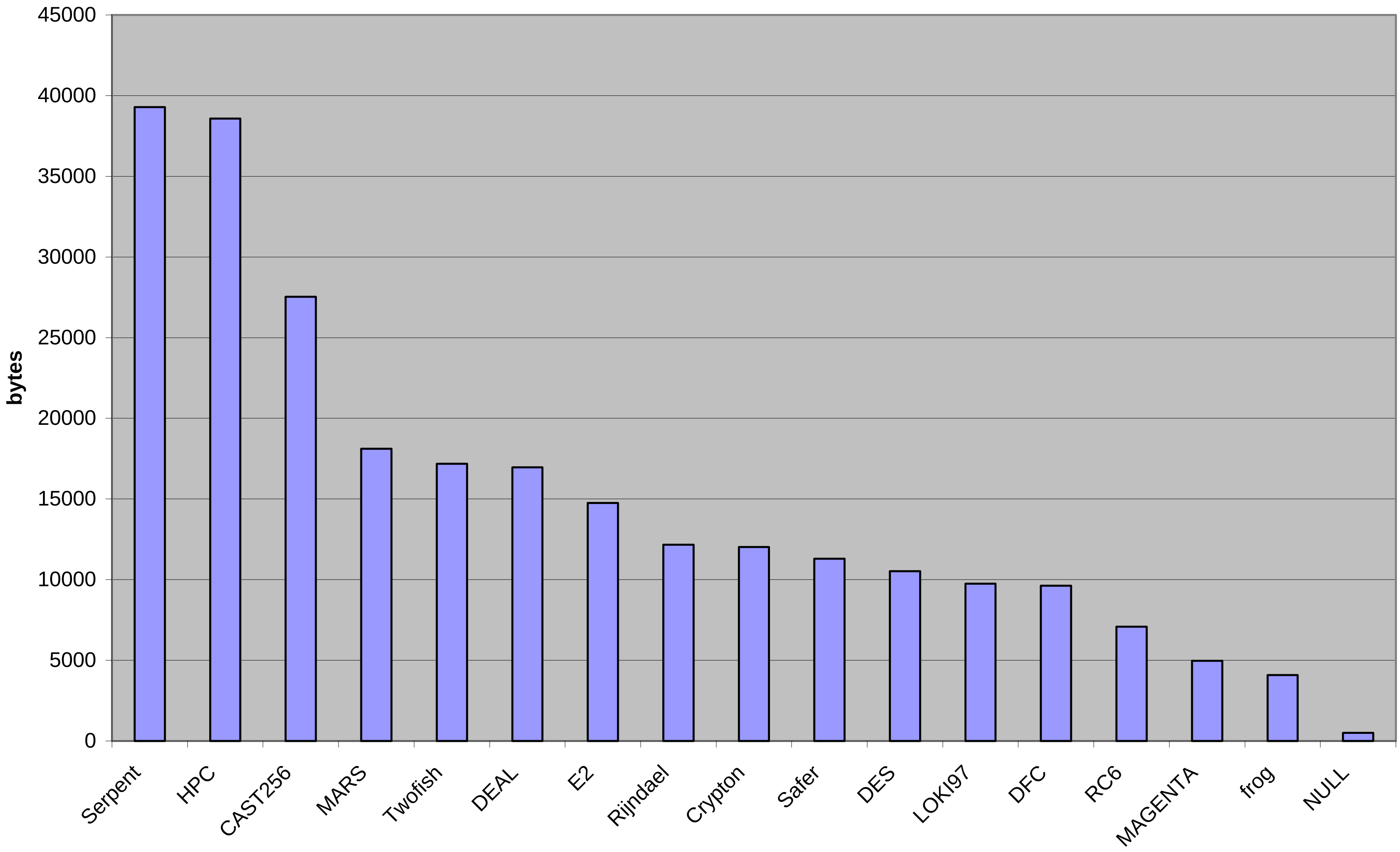
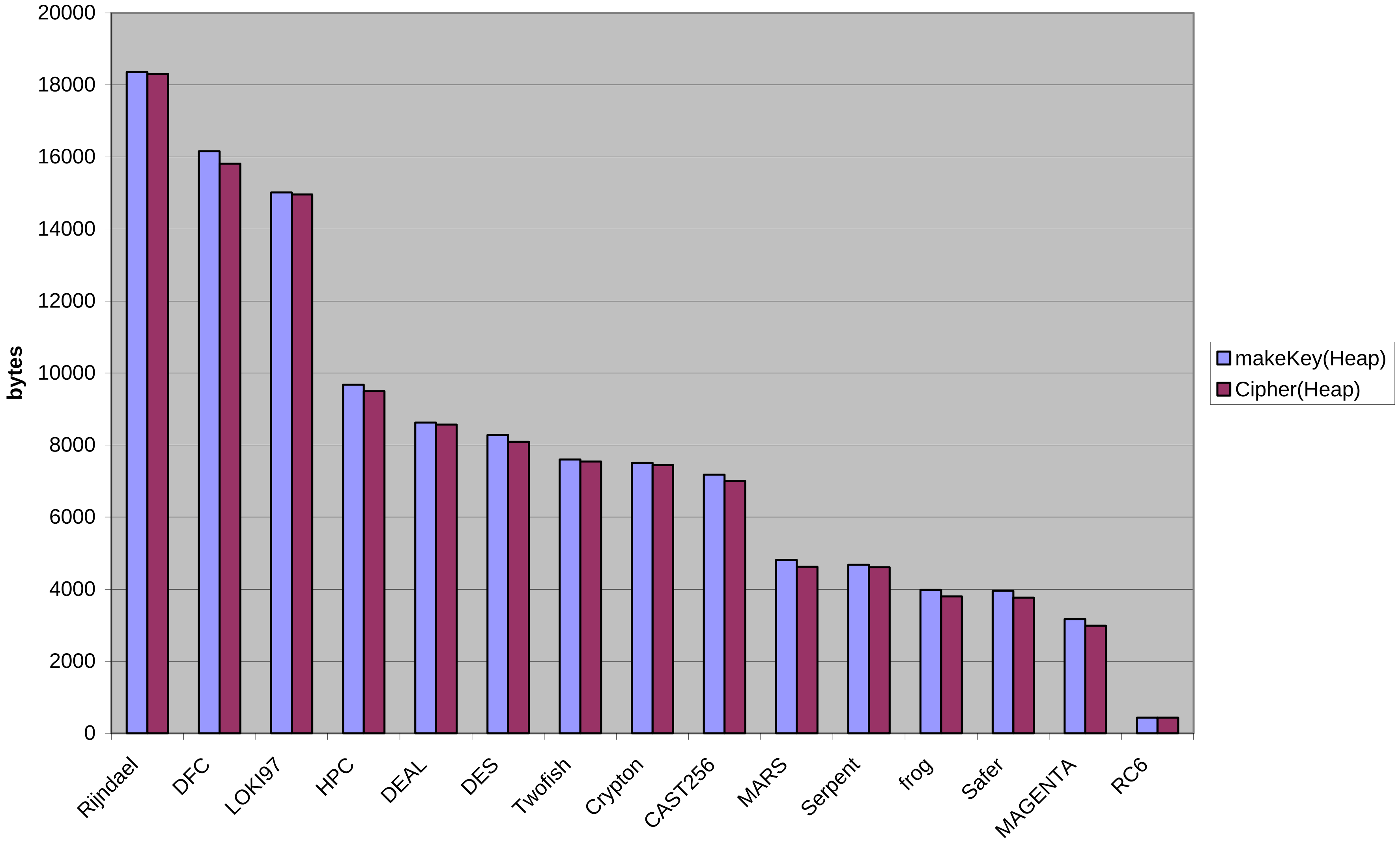
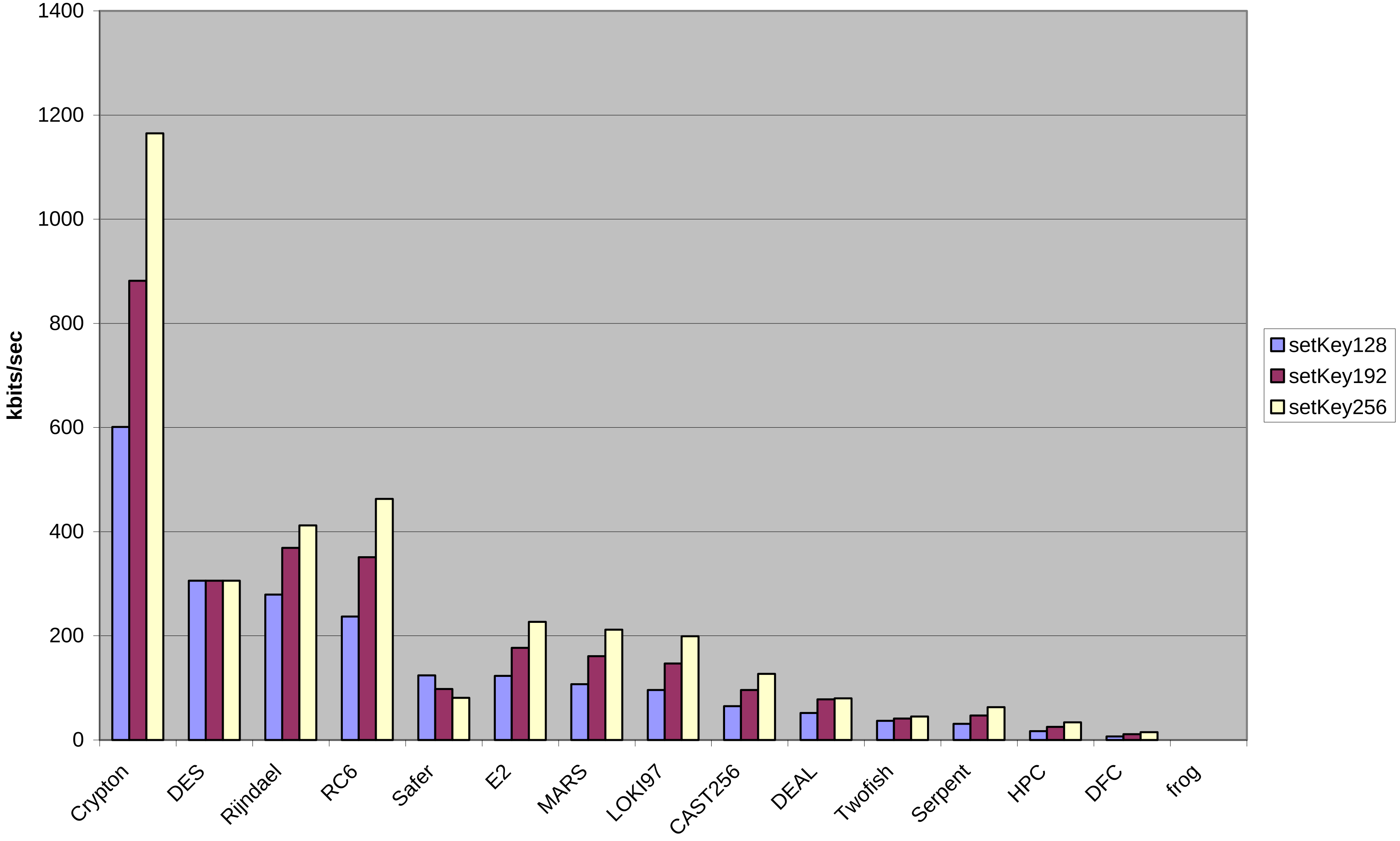


Chart 2: Heap Usage



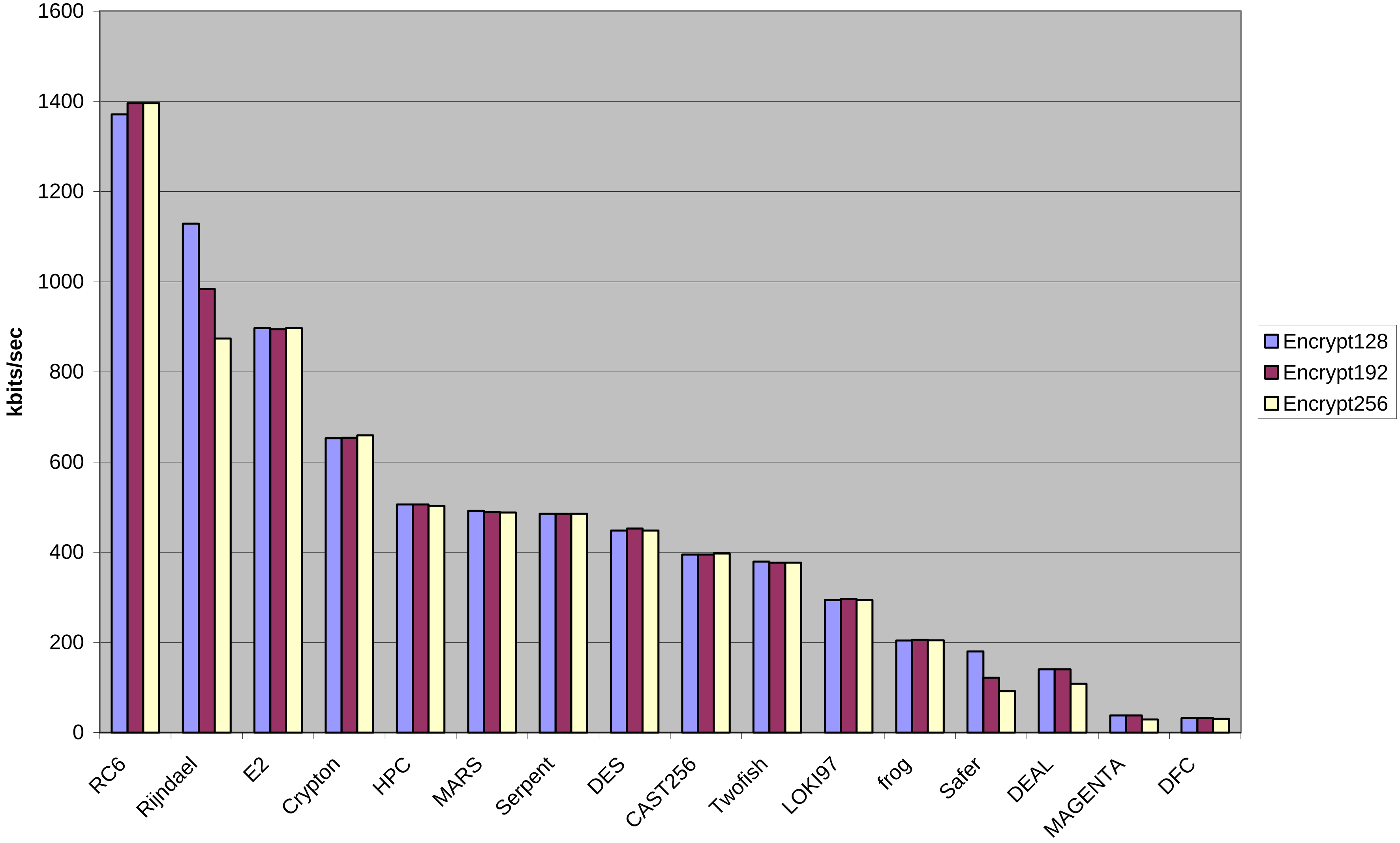
NOTE: E2 uses 264kbytes of heap.

Chart 3: SetKey (O/no JIT)



NOTE: MAGENTA performs 128-bit setKey operations at 6.3mbit/sec.

Chart 4: Encrypt (O/no JIT)



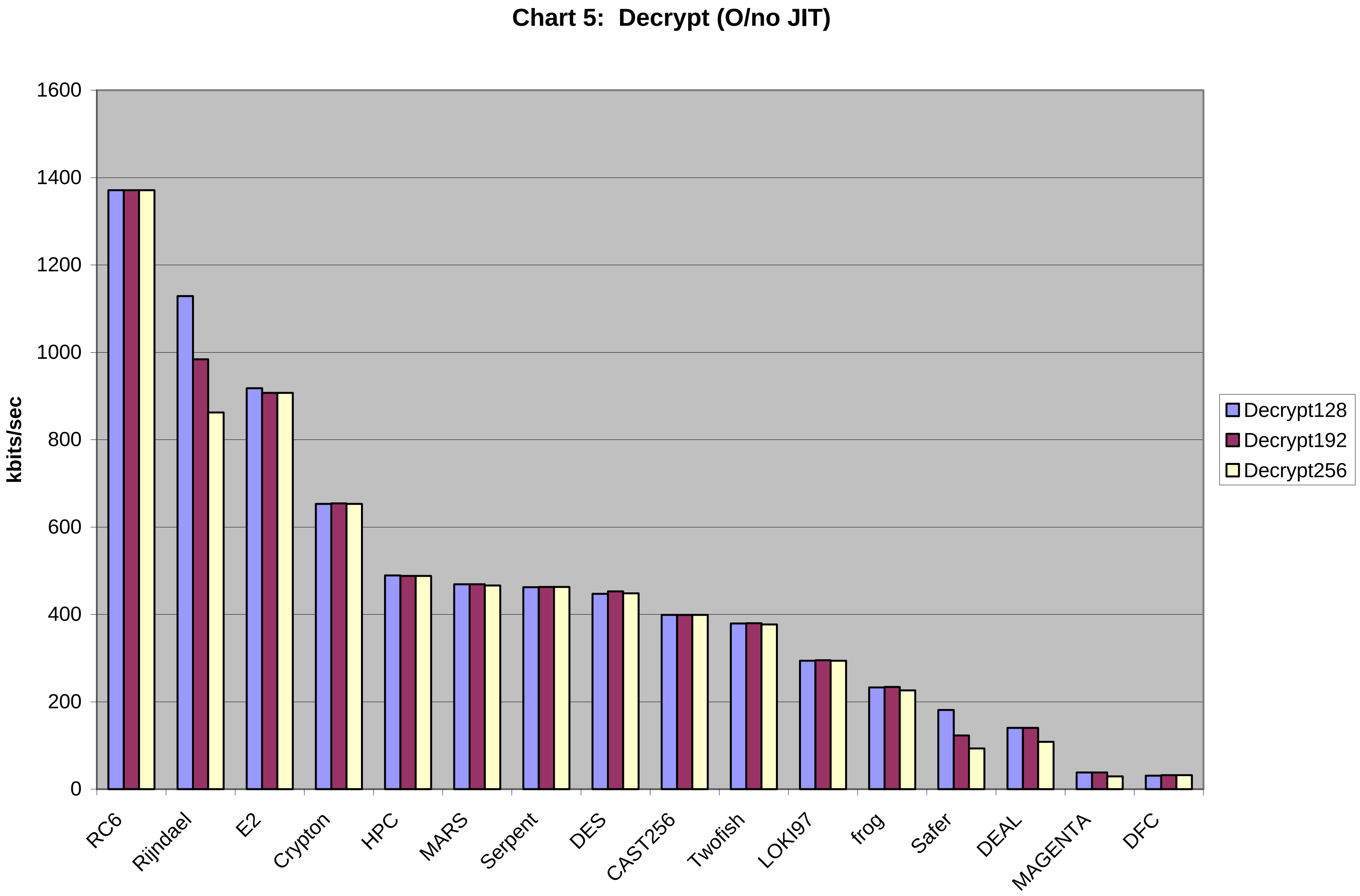


Chart 5: Decrypt (O/no JIT)
kbits/sec
1600
1400
1200
1000
800
600
400
200
0
RC6
Rijndael
E2
Crypton
HPC
MARS
Serpent
DES
CAST256
Twofish
LOKI97
frog
Safer
DEAL
MAGENTA
DFC
Decrypt128
Decrypt192
Decrypt256

Chart 6: Average Method Size in Lines

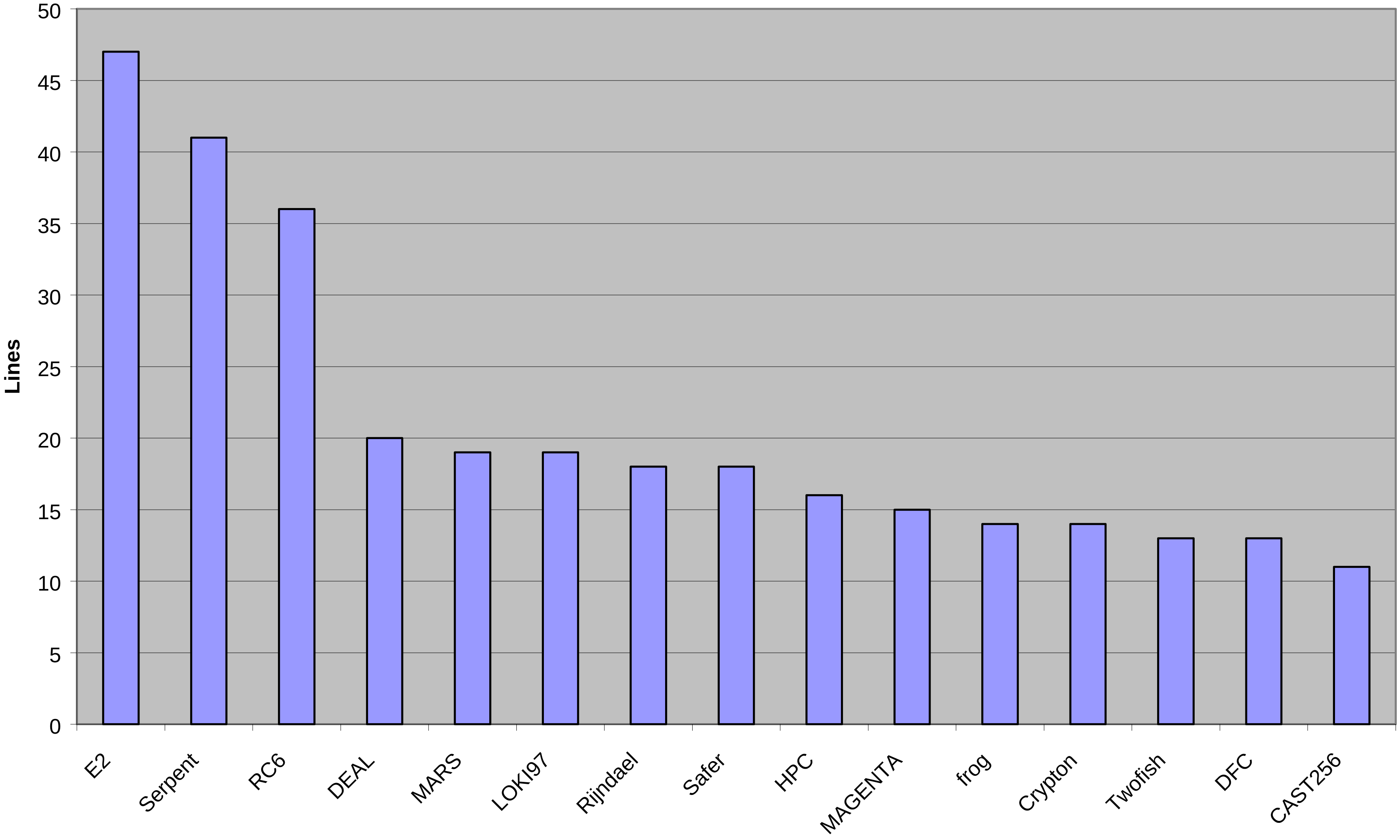


Chart 7: Cyclomatic Complexity

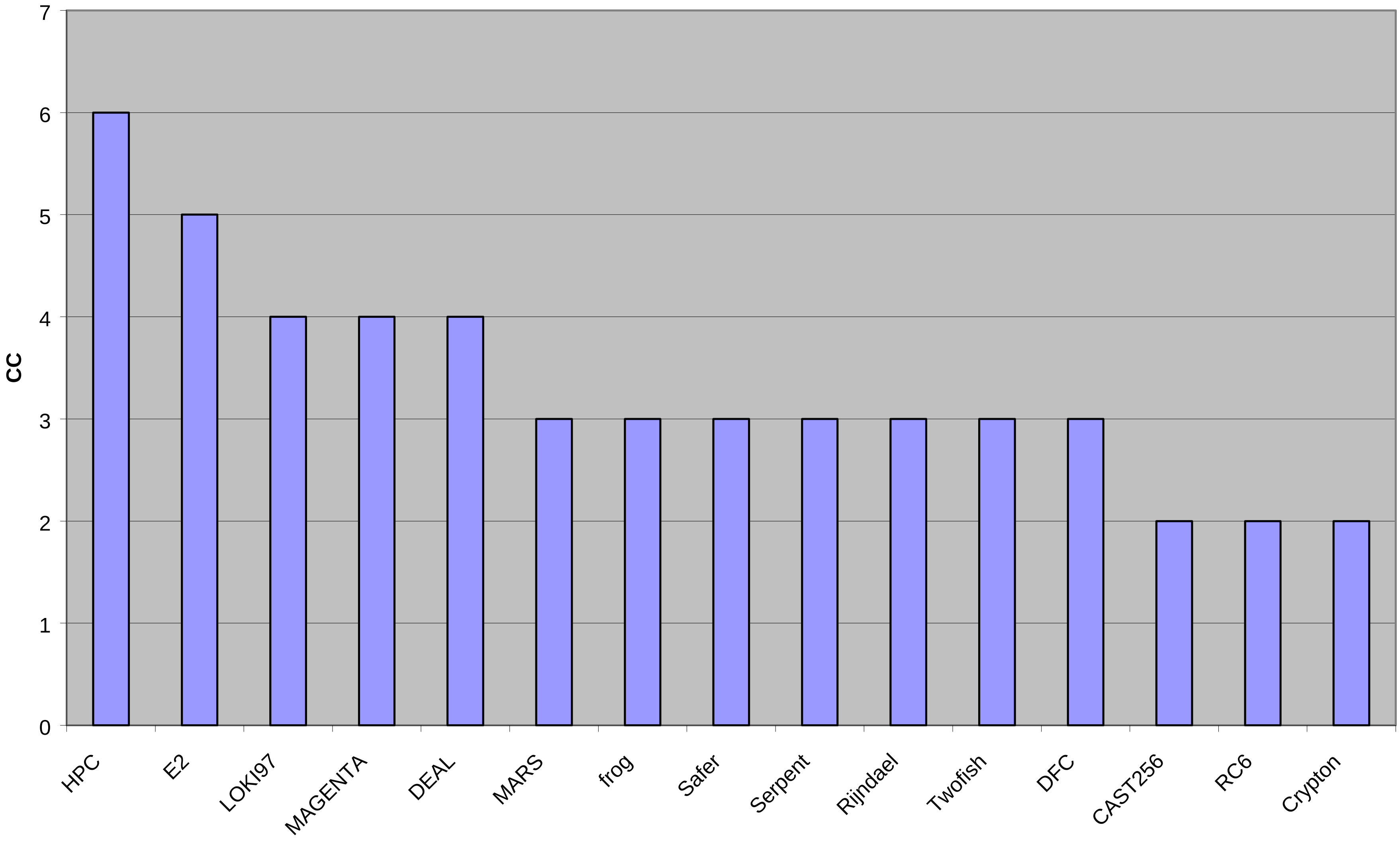
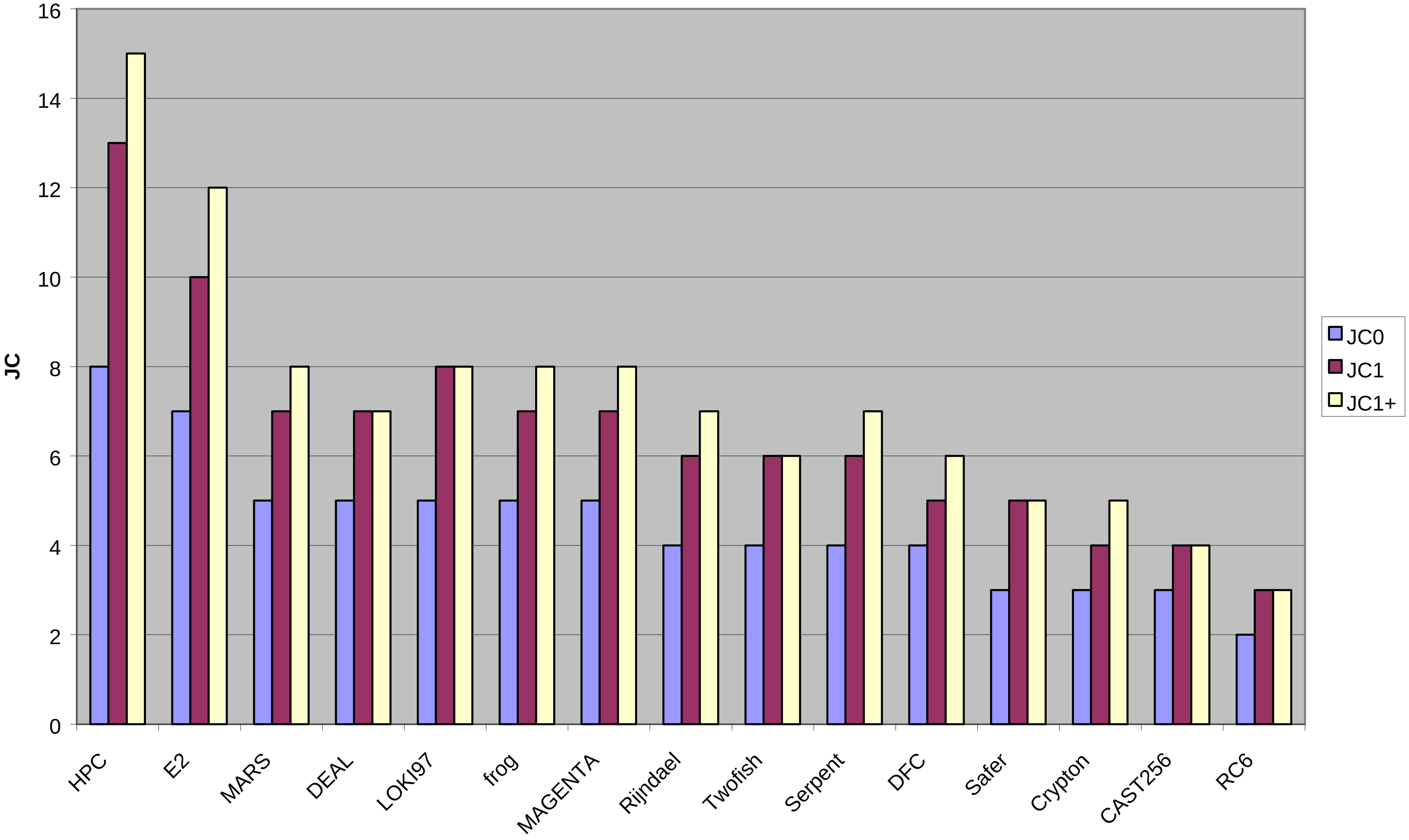


Chart 8: J-Complexity


JC
HPC
E2
MARS
DEAL
LOKI97
frog
MAGENTA
Rijndael
Twofish
Serpent
DFC
Safer
Crypton
CAST256
RC6
JC0
JC1
JC1+