Fast Implementations of AES Candidates

Kazumaro Aoki[1] and Helger Lipmaa[2]

[1] NTT Laboratories
1-1 Hikarinooka, Yokosuka-shi, Kanagawa-ken, 239-0847 Japan
maro@isl.ntt.co.jp
[2] Küberneetika AS
Akadeemia tee 21, 12618 Tallinn, Estonia
helger@cyber.ee

Abstract. Of the five AES finalists four—MARS, RC6, Rijndael, Twofish—have not only (expected) good security but also exceptional performance on the PC platforms, especially on those featuring the Pentium Pro, the NIST AES analysis platform. In the current paper we present new performance numbers of the mentioned four ciphers resulting from our carefully optimized assembly-language implementations on the Pentium II, the successor of the Pentium Pro. All our implementations follow well-defined API and timing conventions and sensible guidelines, like no using of self-modifying code and key-specific static data — i.e., tricks that speed up the implementation but at the same time restrict the field of application. Our implementations are up to 26% percent faster than previous implementations. Our work also shows how a simple change (inclusion of the MMX technology) in the analysis platform can influence the relative encryption speed of different ciphers. To enable everyone to compare their implementations to ours, we also fully specify our procedures used to obtain the speed numbers.

1 Introduction

For more than 20 years, DES [FIP77] has been a widely employed cryptographic standard. While the best cryptanalytic attacks against DES (differential and linear cryptanalysis) are still highly impractical, during the last years DES has became obsolete for its too short key and block sizes, not withstanding the current advances in computing technology. Motivated by this, NIST initiated a new effort to replace DES as a standard. 21 algorithms were submitted and 15 algorithms were accepted as AES (*Advanced Encryption Standard*) candidates, of which 5 candidates—MARS [BCD+98], RC6 [RRSY98], Rijndael [DR98], Serpent [ABK98], Twofish [SKW+99b]—were chosen to the second round.

However, the AES process was started not only due to the theoretical reasons: there are a few well-known constructions, including 3DES, that seem to have very good security margins. Unfortunately, 3DES, based on the hardware-oriented DES, is unsatisfyingly slow on the modern 32- and 64-bit computer architectures: modern block ciphers are up to 10 times faster than 3DES. Regardless of these ciphers having unproven (even by time) security properties, they are widely used in the industry by pragmatic reasons: hardware applications like 1 GBits/s Ethernet or on-the-fly encryption of 160 MByte/s

SCSI hard disks are requesting for faster ciphers. Clearly, the situation of having a (moderately) secure and (moderately) fast *de jure* standard DES, a (probably) secure and (clearly) slow *de facto* standard 3DES and some fast but with unknown security margin *de facto* standards is not acceptable: there should be a single standard that is both secure and fast. This is one of the reasons why, when inviting the public to propose candidates for the AES, NIST explicitly stated that the new standard should be both "more secure and faster" than 3DES.

While security of the candidates cannot be exactly quantified by the currently known methods, it seems to be easier to measure their speed. However, there is still a lot of ambiguity in answering the question what AES candidate is the fastest. Several papers (including [Lip99,SKW$^+$99a]) have compared AES candidates speed, but since the implementations quoted in them are often incomparable (or based on pure estimations), one cannot make direct conclusions about the efficiency of the ciphers based on the published papers. Incomparability stems from the different implementation assumptions, API's, hardware (e.g., processors) and software (e.g., compilers) used by implementers. Even more, some of the timings presented in previous papers correspond to "show-case" (as opposed to practically applicable) implementations, some examples of those being the fastest implementation of Twofish [SKW$^+$99b] that uses self-modifying code and Brian Gladman's implementations of AES candidates [Gla99] that use a number of key-specific static variables instead of allocating a register to address them, therefore effectively freeing some registers for other uses. Especially in the case of the Pentium family, where the number of available registers is very restricted, such implementations may result in a huge speed up. However, both types of implementation tricks restrict the application area of the implementation.

In the current paper we try to give a satisfactory answer to the question "what cipher is the fastest on the Pentium II" by carefully optimizing the 4 fastest AES candidates—MARS, RC6, Rijndael and Twofish—in Pentium II assembly, using for all implementations exactly the same, reasonable in practice, API and speed measurement conditions for all the ciphers. Due to this, our results are much fairer than most of the previously known ones: our implementations can be seen as black boxes applicable in almost any possible application of block ciphers on an environment featuring Pentium II. Additionally, careful optimization process resulted in implementations that are clearly faster than the previously known implementations. (Except for Twofish, which has still a faster "show-case" implementation.)

We start the paper by describing our platform of choice (Section 2), implementation philosophy and API (Section 3). Section 4 briefly surveys our results, and Section 5 gives more details on the problems encountered when implementing the ciphers. More information about the Pentium II is given in the Appendices.

2 Choice of the Platform

Our first principal choice was the decision what processor to use. By purely pragmatic reasons we decided that the implementation environment equips an Intel Pentium family CPU: while this family is not the most modern processor family available, it is the most widespread one at the moment of writing this paper and most probably also during the next few years. Therefore, since in the foreseeable future most of the software-based commercial security applications run on the Pentium family (as recognized also by the AES finalists designers), this family has the most direct impact on the choice of a cipher by security consumers.

At second, from the Pentium family we decided to choose the Pentium II processor. At first, it is a more advanced processor than Pentium Pro, the NIST AES analysis platform: the Pentium II provides (twice) larger register space due to the added MMX technology, and many new MMX-specific commands. Compared to the Pentium Pro, the Pentium II is also easier to obtain at the current stage, since Pentium Pro has been out of the manufacturing for a while. On the other hand, the Pentium II was preferred by the authors to the Pentium III since the latter is somewhat too new and controversial due to the privacy issues.

Another reason to choose Pentium II was that as the successor of the NIST AES analysis platform, implementing the AES candidates on the Pentium II could provide some insights on how generally suitable are the candidates, some of which were specifically optimized for the Pentium Pro, on future processors having features unpredicted by algorithm designers. While this is not as crucial as withstanding the "future attacks", it still gives some ideas about the possible longevity of the cipher. (We clearly would not want the AES in 20 years to have the role the 3DES has today!)

As shown in [Lip98], the MMX technology can seriously speed up IDEA ([LM90], [LMM94]), one of the believably most secure block ciphers with 64-bit block size. As stated in [Lip98], this can be done since IDEA has its key attributes similar to those of multimedia applications, for which the MMX technology was originally created. An open question posed in [Lip98] was how much would the MMX technology help implementing other ciphers, including the AES candidates. In the following we will partially answer to that question, showing that also some ciphers using only "simple" operations can greatly benefit from the added MMX technology. A short overview of Pentium II that is necessary for implementers and for cryptographers who design ciphers optimized for this platform is given in Appendix A. We refer for Intel manuals for a more complete overview.

3 Implementation Considerations

Several papers (including, in particular, [Lip99,SKW$^+$99a]) have compared AES candidates speed, but since the implementations quoted in them are often incomparable (or based on pure estimations), one cannot make direct conclusions about the efficiency of these algorithms based on the published papers. Incomparability stems from the different implementation assumptions, API's, hardware (processors) and software (compilers) platforms used by implementers. Even more, some of the numbers there correspond to the "show-case" (as opposed to practically applicable) implementations; including the bizarre case that one candidate was claimed to be the fastest on its inventors laptop under some suitable conditions.

As another example of the unsuitability of some "show-case" implementations, the fastest implementation of Twofish [SKW$^+$99b] uses self-modifying code and therefore cannot be used in a number of applications, while Brian Gladman's implementations of

AES candidates [Gla99] use a number of key-specific static variables instead of allocating a register to address them, therefore effectively freeing some registers for other uses. Especially in the case of the Pentium family, where the number of available registers is very restricted, such implementations may result in a huge speed up. On the other hand, Gladman's implementations cannot be used several applications, including multithreaded programs and SMP (symmetric multi-processing) systems.

Most of the security customers need however speed numbers applicable in whatever product they use in whatever environment in runs (for example, in a Linux kernel-supported IPSEC implementation, secure login or multithreaded access to encrypted storage arrays). For users it is necessary to know in what environment the measured speed numbers were obtained, to be able to calculate the possible efficiency of the ciphers in their own environments. Additionally, full specification is important for other implementers to be able to compare their implementations with ours. Hence, apart from providing "clean" implementations under some reasonable public assumptions, we shall also next fully specify these assumptions:

- We do not use self-modifying code ("code compilation" [SKW+99b]) since it makes the implementation inapplicable in a number of situations, e.g., in operation-system kernel and ROM-based applications.
- We additionally decided not to use key-specific static areas since then the implementation could not be used, e.g., in SMP-capable systems and multithreaded programs.
- We decided to maximally use the MMX technology since it should not be forbidden in any reasonable modern environment. (While using self-modifying code and key-specific static areas is generally considered to be a bad programming practice.)
- We decided to use exactly the same API (specified later in Section 3.1) in all our implementations.
- A number of well-understood assumptions that 1) improve the speed and can be easily followed by implementers or 2) are essential to even be able to measure the speed:
  - All codes and data are correctly aligned.
  - Input and output texts and codes are preloaded to L1 cache in the possible extent to reduce the number of cache misses.
  - Simplicity of code: we tried to reduce time spent during writing and optimizing the code. In particular, all our implementations use highly optimized but round-number independent round macros. (Hence, our results could be slightly bettered if every round would optimized separately to avoid, e.g., delays in fetching stage.)

3.1 API

Since a different API can be influence the speed of an implementation severely, we also decided to fully specify the API used by us to make for the other implementers easier to compare their implementations to the ours. We felt that this is necessary, since AES candidate implementations reported in [Lip99] vary greatly in their API's.

```
void xxKS(char *master, uint32 bitLen, char *eKey);
void xxEnc(char *inBlk, uint32 lenBlk, char *eKey,
           char *outBlk);
void xxDec(char *inBlk, uint32 lenBlk, char *eKey,
           char *outBlk);
```

where

`xx` is algorithm name (e.g., `Rijndael`).
`xxKS` is key scheduling subroutine.
`xxEnc` is encryption subroutine that encrypts `lenBlk` blocks of plaintext starting from the address `inBlk` to the ciphertext location `outBlk`, by using extended key `eKey`, in ECB block cipher mode.
`xxDec` is decryption subroutine with the same input conventions as `xxEnc`.
`uint32` is the type of 32-bit unsigned integers (in the case of Pentium II, equal to `unsigned long` in the case of most compilers).
`master` is pointer to the master key bits.
`bitLen` is the bit length of a master key.
`eKey` is pointer to subkeys and other initialization data, used later by encryption and decryption.
`inBlk` is pointer to input texts to be encrypted in the case of `xxEnc` and to be decrypted in the case of `xxDec`.
`outBlk` is pointer to the corresponding output texts.
`lenBlk` is number of blocks to be encrypted or decrypted.

Fig. 1. Specification of our API.

Note that our API, depicted in Figure 1, is essentially equivalent to the API's used in most of the commercial applications, specifying only those inputs and outputs to the algorithms that are really needed by the algorithms. (Names of the subroutines and their parameters of course do not affect the speed, of course.) Our API was fixed for the key length of 128-bits due to the feeling that at the time when greater key sizes become necessary, our implementation platform would already be a history.

Here, the key schedule and decryption subroutines are specified only for completeness. Since in the current paper we are not interested in the optimization of these subroutines, we almost do not mention decryption and key schedules hereafter.

3.2 How to Measure a Number of Cycles

Different time measurement methods may change the speed numbers quite dramatically. As in the case of the API's, we decided to use one, sensible published and *fully specified* convention (specified in Figure 2) for all the implementations. (Note that this wrapping corresponds almost exactly to the method specified in [Fog00], to which the reader is referred for a throughout explanation of the method.) The inputs and key of the cipher are generated randomly before the measurement begins, to prevent "optimization" for specific class of keys. The input variable `lenBlk` was chosen to be equal to 8000 so that the input and output texts would not fit in the L1 cache. Also, `time` is a work area of type `uint32`, used in later calculations.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.