# AES Software Implementations on ARM7TDMI

Matthew Darnall[1,⋆] and Doug Kuhlman[2]

[1] Department of Mathematics
University of Wisconsin, Madison, 480 Lincoln Drive Madison, WI 53706
darnall@math.wisc.edu
[2] Motorola Labs, 1301 E Algonquin Rd Schaumburg, IL 60173
Doug.Kuhlman@motorola.com

**Abstract.** Information security on small, embedded devices has become a necessity for high-speed business. ARM processors are the most common for use in embedded devices. In this paper, we analyze speed and memory tradeoffs of AES, the leading symmetric cipher, on an ARM7TDMI processor. We give cycle counts as well as RAM and ROM footprints for many implementation techniques. By analyzing the techniques, we give the options we found which are the most useful for certain purposes. We also introduce a new implementation of AES that saves ROM by not explicitly storing all the SBOX data.

## 1 Introduction

Rijndael was selected as the Advanced Encryption Standard (AES) in 2001 by the National Institute of Standards and Technology (NIST) for use in government cryptographic purposes [1]. Before and after its adoption by the NIST, Rijndael received much study in the area of optimization. Many papers give software methods for optimizing certain portions of the algorithm. These papers usually focus on only one part of the algorithm. However, real world implementations are often a trade-off between speed, program size, and memory available. Rarely is one aspect fully optimized at the expense of the others.

ARM processors are the most common for embedded systems, with over 1 billion sold worldwide [6]. Their use in applications like digital radios and PDAs capable of electronic funds transfers has necessitated software implementations of AES for the ARM core.

The contribution of this paper is a detailed study of software optimization options of the Rijndael cipher on ARM processors. Previous work has focused on specific ideas for limited resource environments. We bring together the previous methods and some new ideas to give accurate timings and memory calculations for the AES cipher. Our work covers the key setup and encryption/decryption of AES with all three key sizes approved by the NIST. This paper focuses on the 128-bit key size, but the methods extend in a natural way.

⋆ The first author was supported by the University of Wisconsin Mathematics Department's NSF VIGRE grant.

R. Barua and T. Lange (Eds.): INDOCRYPT 2006, LNCS 4329, pp. 424–435, 2006.

The rest of the paper has the following format. In section 2, we give a short description of AES and set some notation. In section 3, we describe the ARM7TDMI and the diagnostic tools we used to test our implementations. In section 4, we discuss options for the key expansion of AES. In section 5, we give code optimizations for the actual encryption and decryption. Section 6 details some results from combining various implementation tricks. In section 7, we summarize our results.

## 2 Description of AES

A full description of AES is included in [4], and we will maintain the same notation. AES is a 128-bit block cipher with 128-, 192-, or 256- bit keys. The data is operated on as a 16-byte state, which is thought of as four 32-bit column vectors. The number of rounds (Nr) is 10, 12, and 14 for the three respective key sizes (128, 192, and 256). The cipher starts with an XOR of the first 128 bits of the key with the plaintext. Then there are Nr - 1 rounds consisting of four parts: the S-BOX substitution, a byte permutation, a MixCol operation that operates on the state as four separate four-byte column vectors, and a key addition (XOR). The final round has no MixCol part. The MixCol operation is multiplication by a constant fourbyte vector, as described in [4]. The key for each round is determined by transformations of the key for the previous round, with the first key being the cipher key. Each part of a round is invertible. We shall denote the inverse of the MixCol operation as InvMixCol.

AES was designed to give excellent performance on many platforms. In particular, the cipher performs well on 32-bit platforms such as the ARM. The state and key can be implemented as an integer number of 32-bit variables. The key addition is a straightforward XOR with the state. The SBOX substitution has an algebraic description, but implementing it is more efficient in time and space when using a table with 256 one-byte entries. The byte permutation can be implemented when performing the S-BOX. As noted in [3], the best method for optimizing the algorithm lies in the implementation of the MixCol and InvMixCol operations.

## 3 ARM Information

The ARM processor family, created in 1985, has established itself as the leading embedded systems processor. The processors use RISC (Reduced Instruction Set Computer) architecture design philosophy. Due to the constrained environments the processors are used in, the processors have little power consumption and a high code density.

We tested all of our implementations for the ARM7TDMI. This processor is one of the most popular ARM designs, used mainly in mobile embedded systems such as cell phones, pagers, and mp3 players. The processor has a three stage pipeline. It uses a Von Neumann architecture design, so the data and the instructions use the same bus. This ARM core is particularly low power, using just .06 mW/Mhz. The lower power usage is partly due to the fact there is no