A More Compact AES

David Canright[1] and Dag Arne Osvik[2]

[1] Naval Postgraduate School, Monterey CA 93943, USA
dcanright@nps.edu
[2] École Polytechnique Fédérale de Lausanne
dagarne.osvik@epfl.ch

Abstract. We explore ways to reduce the number of bit operations required to implement AES. One way involves optimizing the composite field approach for entire rounds of AES. Another way is integrating the Galois multiplications of MixColumns with the linear transformations of the S-box. Combined with careful optimizations, these reduce the number of bit operations to encrypt one block by 9.0%, compared to earlier work that used the composite field only in the S-box. For decryption, the improvement is 13.5%. This work may be useful both as a starting point for a bit-sliced software implementation, where reducing operations increases speed, and also for hardware with limited resources.

Keywords: AES, tower field, composite Galois field, bitslice.

1 Introduction

There have been many implementations of the Advanced Encryption Standard, optimized for various criteria, for different applications. Some approaches seek to minimize circuitry, e.g., [1,2,3,4,5]. For this goal, Rijmen[6] suggested using subfield arithmetic in the crucial step of computing an inverse in the Galois Field of 256 elements. [Note: strictly speaking, the operation is not $x \to x^{-1}$ but rather $x \to x^{254}$ so $0 \to 0$, but we will refer to this as the inverse for convenience; similarly for subfields.] Rudra et al.[1] gave a detailed implementation using that subfield approach. This idea was further extended by Satoh et al.[2], using subsubfields (the "tower-field" representation of Paar[7], also called the "composite-field" approach), along with other innovative optimizations, which resulted in the smallest AES circuit at that point. The S-box architecture of Satoh was improved by Canright[8], mainly through carefully chosen normal bases, resulting in the most compact S-box to date. This S-box has been used in bit-sliced software implementations of AES, by Rebeiro et al.[9], and (slightly improved by [10]) by Käsper and Schwabe[11].

The present work seeks to further reduce the size of AES, in terms of the number of bit operations. While [8] showed that normal bases gave a more compact Galois inverter for the S-box, the specific basis chosen did not yield compact Galois multiplications by the constants used in the MixColumns step; hence that composite basis was used *only* for the S-box. Here we reconsider the approach of maintaining the composite-field representation throughout the rounds of encryption, as in Rudra et al.[1]. We find that a different choice of basis than in [8] does indeed give a smaller AES implementation with this approach, in part through combining the linear transformations of the S-box with the constant multiplications (or "scalings") of MixColumns. Moreover, applying optimization software to certain portions of the logic further reduces the number of operations. Together, these improvements give a 9.0% reduction in the number of bit operations needed to encrypt one block with a 128-bit key.

First we briefly review the AES algorithm in Section 2, then detail our method in Section 3, including choices of basis for the tower field and integration of the scalings of MixColumns with the linear transformations of the S-box. Finally, we summarize our results in Section 4 and briefly discuss conclusions in Section 5.

2 AES Algorithm

The Rijndael algorithm, as adopted for the Advanced Encryption Standard, is a symmetric block cipher with 128-bit blocks and three key sizes: 128, 192, or 256 bits[12]. Here, we give just enough detail to explain our method below.

For encryption, each block of 16 bytes is processed by several rounds: 10, 12, or 14, depending on key size. From the initial key, the key schedule generates a different round key for each round. Each round comprises the following steps.

1. *SubBytes* subjects each byte independently to a nonlinear function, often called the S-box, and substitutes the result for the original byte. The S-box function consists of two sequential operations:
   (a) first, *inversion* treats the byte as an element of $GF(2^8)$, where the bits are coefficients of a polynomial, and polynomial arithmetic is modulo the irreducible polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$; each nonzero byte is replaced by its multiplicative inverse in this field, while a zero byte remains unchanged.
   (b) then an *affine transformation* is applied: treating the byte as a vector of bits, the byte is multiplied by a constant bit matrix M and then a constant byte $\boldsymbol{b}$ is added (with bit arithmetic in $GF(2)$, where multiplication is AND and addition is XOR), so $\boldsymbol{x} \to M\,\boldsymbol{x} + \boldsymbol{b}$.

   In software, the S-box is often implemented as a table lookup.
2. *ShiftRows* considers the 16 bytes as a 4×4 array and rotates each row to the left by its position, so row #0 does not move, row #1 moves 1, etc.
3. *MixColumns* operates independently on each column of the 4×4 array: the column, as a vector of four bytes, is multiplied by a constant byte matrix, where the byte arithmetic is in $GF(2^8)$ as in the S-box inversion:

$$\begin{pmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \end{pmatrix} \to \begin{pmatrix} 2\,3\,1\,1 \\ 1\,2\,3\,1 \\ 1\,1\,2\,3 \\ 3\,1\,1\,2 \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \end{pmatrix}$$

4. *AddRoundKey* bitwise adds (XOR) a 128-bit round key to the 128-bit state.

The first round is preceded by an *AddRoundKey* step, and the last round skips the *MixColumns* step.

For decryption, the whole process is reversed, using the inverse operation for each step in the reverse order. AddRoundKey is its own inverse, and the inverse of ShiftRows rotates rows to the right instead of left. The inverse of MixColumns just multiplies each column by the inverse of the constant byte matrix, so

$$\begin{pmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \end{pmatrix} \to \begin{pmatrix} \mathrm{E} & \mathrm{B} & \mathrm{D} & 9 \\ 9 & \mathrm{E} & \mathrm{B} & \mathrm{D} \\ \mathrm{D} & 9 & \mathrm{E} & \mathrm{B} \\ \mathrm{B} & \mathrm{D} & 9 & \mathrm{E} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \end{pmatrix}$$

where the constant values are in hexadecimal (the leading 4 bits of each are 0). For the inverse of SubBytes, first the inverse affine transformation is applied, so $\boldsymbol{x} \to M^{-1}(\boldsymbol{x} + \boldsymbol{b})$, then the Galois inversion is its own inverse operation.

Some reordering of the steps in each round is possible. SubBytes commutes with ShiftRows, and MixColumns and AddRoundKey can be swapped by modifying the key schedule appropriately; similarly with the inverse operations for decryption. Commonly, fast software implementations, e.g., those of Bernstein and Schwabe[13], combine the S-box function with the Galois multiplications of MixColumns, using each input byte to index a table of 4-byte columns, as suggested in the Rijndael proposal[14], which called them "T-tables."

3 Method

Our goal was to develop an implementation of AES with a minimal number of bit operations. The result could be useful for a bit-sliced software implementation, or for hardware with limited resources. (Our original inspiration was considering a bit-sliced AES for the CellBE processor[15].) Our starting point, and baseline for comparison, was the compact AES of [2], with the improved S-box of [8].

These prior works used a tower-field representation of $GF(2^8)$ so that the Galois inversion in the S-box could be calculated compactly. But where [8] optimized the choice of basis for the S-box only, we sought to find the best basis for whole rounds of AES, as [1] did for a different composite-field representation.

One reason *not* to do this, i.e., to change back to the standard basis after the S-box, is that the Galois multiplication by the constant 2 byte, as required in MixColumns, is very compact in the standard basis: three bitwise XORs. Nonetheless, we found that overall the advantages of our approach overcame this disadvantage.

One way we reduced operations is by combining the constant Galois multiplications of MixColumns with the linear part of the affine transformation of the S-box. We tried different places to put these combined transformations, either earlier as parts of the S-box or later as parts of MixColumns, as we will describe in subsection 3.2.

Another way was by finding the tower-field basis that would be most compact not just for the inverter, but for an entire round of encryption or decryption.

There are actually many different tower-field representations possible; here we only need to examine a small subset of those considered in [16], as discussed in subsection 3.1 below.

Lastly, we applied state-of-the-art optimizing software to the transformation matrices and to parts of the inverter operation. The optimizing software employs heuristics to arrive at very efficient implementations.

3.1 Basis Choices

In [8], 432 different choices of basis were considered for the tower-field representation of $GF(2^8)$, where $GF(2^8)$ is considered as a quadratic extension of $GF(2^4)$, which in turn is considered as a quadratic extension of $GF(2^2)$. We will use the notation $GF(2^8)/GF(2^4)/GF(2^2)$ to indicate such a tower-field representation; of course, all representations of $GF(2^8)$ are isomorphic. Such a representation really involves three bases: one each for $GF(2^2)/GF(2)$, for $GF(2^4)/GF(2^2)$, and for $GF(2^8)/GF(2^4)$, where each basis consists of two elements linearly independent over the subfield.

Only polynomial bases (of the form $[r, 1]$) and normal bases (of the form $[r^q, r]$, where $q = 2^1$, 2^2, or 2^4 is the size of the subfield, and r, r^q are conjugates) were considered in [8]; other types are generally less efficient. And only choices with a trace of unity $\tau = r + r^q = 1$ were considered, that is, where the minimal polynomial has the form $x^2+x+\nu$ and $\nu = r \times r^q$ is the norm of r, since this choice eliminates some operations. Some other special forms would also eliminate some operations, such as where the norm is unity $\nu = 1$ or where the trace and norm are equal $\tau = \nu$, but these turned out to be less efficient for the Galois inverter of the S-box. Normal bases were shown to have a definite advantage for the inverter, since more factors are shared in the lower level operations. And one particular choice, #4 of the 432 in [16, App. E], gave the smallest optimized transformation matrices, and hence the smallest merged S-box (where encryption and decryption share a Galois inverter), as well as the smallest S-box for encryption only or for decryption only.

But basis #4 did not give a compact form for the Galois multiplications needed in MixColumns, so the tower-field representation was only used for the S-box, with the rest of each round using the standard basis. In particular, for encryption, MixColumns requires multiplying bytes by the constants 2 and 3 (in the standard representation), where multiplying by 2 only requires three bitwise XORs. (In the standard representation, “2” represents a root of the irreducible polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$, and “3” = “2” + 1, where 1 is the multiplicative identity as usual. So multiplication of a byte by 2 involves shifting left one bit, and if the msb was 1, then XOR with 0x1B.)

We explored whether a different approach might give a more compact implementation: using a tower-field representation throughout the rounds of encryption, similar to the approach of [1]. We sought an optimum basis for both the S-box and MixColumns steps; ShiftRows is just a re-ordering of bytes, and AddRoundKeys works the same in any basis. To keep the optimization tractable, we limited consideration to encryption only, or separately for decryption only;

we did *not* consider a merged encrypt/decrypt architecture as in [2]. And to keep the Galois inverter of the S-box small, we only looked at normal bases with unit trace.

This left 16 possibilities out of the 432 in [16, App. E]: basis numbers 1, 4, 19, 22, 37, 40, 55, 58, 73, 76, 91, 94, 109, 112, 127, and 130. It turns out that all these cases give a Galois inverter of the same size; though the specifics of the operations change, the total number of bit operations in the optimized inverter is the same.

Besides the inverter, the S-box includes the affine transformation, and Mix-Columns requires Galois multiplication by 2 and 3, or by four different constants for decryption. We will use the term "scaling" to indicate such Galois multiplying of a byte by a specified constant byte. Then both scaling and the affine part of the S-box (ignoring the additive constant for now; see subsection 3.3) can each be represented as a linear transformation: multiplication of an 8-bit vector by a bit matrix (with all bit arithmetic modulo 2). These transformations can be combined by simply multiplying the bit matrices. To see more precisely which matrices would be required in each round, we needed to consider how to implement MixColumns. Then we could choose the basis that gave the most compact versions of those matrices.

3.2 MixColumns

Satoh[2] gave an elegant implementation that combined MixColumns with its inverse, for the architecture with both encryption and decryption merged (with a selector signal). For just MixColumns, each column, as a vector of four bytes, is multiplied by a 4×4 matrix, where scalar multiplication is in $GF(2^8)$. Satoh effectively decomposed the matrix as below:

$$\begin{pmatrix} 2&3&1&1\\ 1&2&3&1\\ 1&1&2&3\\ 3&1&1&2 \end{pmatrix} = 2 \times \begin{pmatrix} 1&1&0&0\\ 0&1&1&0\\ 0&0&1&1\\ 1&0&0&1 \end{pmatrix} + \begin{pmatrix} 0&0&1&1\\ 0&0&1&1\\ 1&1&0&0\\ 1&1&0&0 \end{pmatrix} + \begin{pmatrix} 0&1&0&0\\ 1&0&0&0\\ 0&0&0&1\\ 0&0&1&0 \end{pmatrix}$$

This decomposition allowed reuse of certain combinations of bytes[2, (6)], and each byte multiplication by 2 took three XORs, so altogether each 4-byte column took 108 XORs.

For decryption, the inverse MixColumns matrix of [2] came from adding more terms to the MixColumns matrix:

$$\begin{pmatrix} E&B&D&9\\ 9&E&B&D\\ D&9&E&B\\ B&D&9&E \end{pmatrix} = \begin{pmatrix} 2&3&1&1\\ 1&2&3&1\\ 1&1&2&3\\ 3&1&1&2 \end{pmatrix} + 4 \times \begin{pmatrix} 1&0&1&0\\ 0&1&0&1\\ 1&0&1&0\\ 0&1&0&1 \end{pmatrix} + 8 \times \begin{pmatrix} 1&1&1&1\\ 1&1&1&1\\ 1&1&1&1\\ 1&1&1&1 \end{pmatrix}$$

where the 4 and 8 came from repeated multiplications of common terms by 2, at 3 XORs each. With the reuse of common terms, altogether each 4-byte column took 195 XORs.

We considered similar decompositions that could re-use some byte sums, but with the constant scaling combined with the affine transformation of the S-box. Let T_2 be the matrix (given in subsection 3.3) below that performs the "times 2" operation, and similarly for other constants. (Note: $T_1 = I$, the identity matrix.) So with the matrix M of the affine transformation, the combined transformations needed for encryption are M, $T_2\,M$, and $T_3\,M$.

Our approach uses the decomposition below:

$$\begin{pmatrix} 2&3&1&1\\ 1&2&3&1\\ 1&1&2&3\\ 3&1&1&2 \end{pmatrix} = \begin{pmatrix} 2&3&0&0\\ 0&3&2&0\\ 0&0&2&3\\ 2&0&0&3 \end{pmatrix} + \begin{pmatrix} 0&0&1&1\\ 1&1&1&1\\ 1&1&0&0\\ 1&1&1&1 \end{pmatrix}$$

Using the common terms in the last matrix, this approach has 11 byte additions (88 XORs) per column.

One way to do the transformations is for half the bytes, after the inverter of the prior S-box, to get transformed with both M and $T_2\,M$ separately, and the other half with M and $T_3\,M$; no byte needs both $T_2\,M$ and $T_3\,M$. Another way is to do half the bytes with just $T_2\,M$, the other half with $T_3\,M$, and later apply M only to two common terms: sums of untransformed bytes 0 & 1 and 2 & 3. While the latter ("later" transformations) way has fewer transformations overall, in the former ("early" transformations) way, pairs of transformations apply to each byte, allowing additional optimizations of the pairs. We explored both, and it turned out the early transformation approach was slightly better.

For decryption, the inverse MixColumns matrix has four different constants (hexadecimal E, B, D, & 9), linearly independent over $GF(2)$, so is more expensive. These scalings can be combined with the inverse affine transformation for the following inverse S-box, since inverse MixColumns is a linear operation.

We considered a direct, early approach, where after AddRoundKey, each byte is transformed by the four transformations $M^{-1}\,T_{\mathrm{E}}$, $M^{-1}\,T_{\mathrm{B}}$, $M^{-1}\,T_{\mathrm{D}}$, and $M^{-1}\,T_9$, followed by the 12 byte additions (96 XORs) for inverse MixColumns. We also considered a decomposition:

$$\begin{pmatrix} \mathrm{E}&\mathrm{B}&\mathrm{D}&9\\ 9&\mathrm{E}&\mathrm{B}&\mathrm{D}\\ \mathrm{D}&9&\mathrm{E}&\mathrm{B}\\ \mathrm{B}&\mathrm{D}&9&\mathrm{E} \end{pmatrix} = \begin{pmatrix} 3&2&0&0\\ 0&3&2&0\\ 0&0&3&2\\ 2&0&0&3 \end{pmatrix} + \mathrm{D} \times \begin{pmatrix} 1&0&1&0\\ 0&1&0&1\\ 1&0&1&0\\ 0&1&0&1 \end{pmatrix} + 9 \times \begin{pmatrix} 0&1&0&1\\ 1&0&1&0\\ 0&1&0&1\\ 1&0&1&0 \end{pmatrix}$$

Each byte gets transformed with both $M^{-1}\,T_2$ and $M^{-1}\,T_3$, and later the two common expressions, sums of untransformed bytes 0 & 2 and 1 & 3, each get both $M^{-1}\,T_{\mathrm{D}}$ and $M^{-1}\,T_9$. This still has 12 byte additions but fewer transformations. Again, we tried both, and this time the later approach was better.

3.3 Transformation Matrices

In the ShiftRows, S-box, and MixColumns steps of a normal encryption round, each byte is routed to the correct position in a column, is inverted in a Galois inverter, then goes through the affine transformation along with 2 or 3 times that result (as shown above). The affine transformation on a byte $\boldsymbol{x}$ looks like $\boldsymbol{y} = M\,\boldsymbol{x} + \boldsymbol{b}$, or in detail

$$\begin{pmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{pmatrix} = \begin{pmatrix} 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \\ 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \end{pmatrix} \begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

where bit #7 is the most significant and all bit operations are modulo 2.

To do the same operation in a different basis, we need to apply a similarity transformation to this matrix M (to account for the change of basis on both input and output vectors). Let X refer to the 8×8 bit matrix that converts a byte *from* the tower-field basis *to* the standard basis, and let $\boldsymbol{u}$ and $\boldsymbol{v}$ be the tower-field representations of $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively (so $\boldsymbol{x} = X\,\boldsymbol{u}$ and $\boldsymbol{y} = X\,\boldsymbol{v}$). Then the affine transformation becomes

$$\boldsymbol{v} = \left(X^{-1}\,M\,X\right)\,\boldsymbol{u} + \boldsymbol{c} \qquad \text{where} \qquad \boldsymbol{c} = X^{-1}\,\boldsymbol{b}$$

or equivalently

$$\boldsymbol{v} = \left(X^{-1}\,M\,X\right)\,(\boldsymbol{u} + \boldsymbol{d}) \qquad \text{where} \qquad \boldsymbol{d} = X^{-1}\,M^{-1}\,\boldsymbol{b}$$

Galois multiplication by the constants 2 and 3 can also be done by matrices; let T_2 and T_3 respectively be these matrices with respect to the standard representation. Then

$$2 \times \boldsymbol{x} = T_2\,\boldsymbol{x} = \begin{pmatrix} 0\,1\,0\,0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,0\,1\,0\,0\,0 \\ 1\,0\,0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,0\,0\,1\,0 \\ 1\,0\,0\,0\,0\,0\,0\,1 \\ 1\,0\,0\,0\,0\,0\,0\,0 \end{pmatrix} \begin{pmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{pmatrix}$$

and $T_3 = T_2 + I$. To get the scaling matrices for decryption, let $T_4 = (T_2)^2$, $T_8 = T_4\,T_2$, $T_\mathrm{C} = T_8 + T_4$; then $T_\mathrm{E} = T_\mathrm{C} + T_2$, $T_\mathrm{B} = T_8 + T_3$, $T_\mathrm{D} = T_\mathrm{C} + I$, $T_9 = T_8 + I$. Again, to do these same operations in the tower-field basis, we would apply a similarity transformation to these matrices. Or, if we combine with the affine transformation, for encryption we get

$$2 \times \boldsymbol{v} = \left(X^{-1}\,T_2\,M\,X\right)\,(\boldsymbol{u} + \boldsymbol{d}) \qquad \text{and} \qquad 3 \times \boldsymbol{v} = \left(X^{-1}\,T_3\,M\,X\right)\,(\boldsymbol{u} + \boldsymbol{d})$$

So for a given byte, with the early transformation strategy, first we apply the Galois inverter, then apply two transformations, either affine and 2×affine, or affine and 3×affine, depending on in which row of a column it ends up. Thus for a given basis X we need to optimize the matrix *pairs* $[(X^{-1}\,M\,X)\,,(X^{-1}\,T_2\,M\,X)]$ and $[(X^{-1}\,M\,X)\,,(X^{-1}\,T_3\,M\,X)]$, where each pair is considered as a single 16×8 matrix. For the later transformation strategy, the three separate matrices $(X^{-1}\,M\,X)$, $(X^{-1}\,T_2\,M\,X)$ and $(X^{-1}\,T_3\,M\,X)$ would be optimized.

The additive constant $\boldsymbol{c}$ can usually be included by simply replacing some XORs by XNORs, or it may be incorporated into the key schedule. Note that, because the row sum of the constants in the MixColumns matrix (or its inverse) is 1, then $\boldsymbol{c}$ really only needs to be added to any *one* of the four terms in the row.

For each of the 16 different normal bases, we applied our optimization software to minimize (smallest number of XORs) the two 16×8 bit matrices (each would apply to a pair of bytes per column) of the early transformation strategy, and also the three 8×8 matrices (again for a pair of input bytes) for the later strategy.

Basis #127 was the winner, with an early strategy optimized total of $17+18 = 35$ XORs, barely beating the later strategy at $11 + 13 + 12 = 36$ XORs. Here are the basis change matrices for basis #127:

$$X = \begin{pmatrix} 0&0&1&0&0&1&0&0\\ 0&1&1&0&0&0&1&1\\ 1&1&0&1&1&0&1&1\\ 0&1&0&1&0&1&1&0\\ 0&0&1&0&1&1&1&0\\ 1&0&1&1&0&1&1&1\\ 1&1&0&1&1&1&0&1\\ 1&1&0&0&0&0&1&0 \end{pmatrix} \quad , \quad X^{-1} = \begin{pmatrix} 0&0&1&0&1&1&0&1\\ 0&1&0&1&0&1&0&1\\ 1&1&0&1&1&0&1&1\\ 0&1&1&0&0&1&1&1\\ 1&1&1&1&0&0&0&1\\ 0&1&0&1&1&0&1&1\\ 0&1&1&1&1&0&0&1\\ 1&0&1&1&0&1&1&1 \end{pmatrix}$$

Besides the matrix transformations in normal rounds, this approach also uses two others. Before the first round, each byte must be transformed from the standard representation to the tower-field basis, by the matrix (X^{-1}) above; the optimized version requires 15 XORs per byte, which we treat as part of round 0, the initial AddRoundKey. And the last round of encryption skips MixColumns and needs to end up in the standard representation, so after the last inverter, the affine transformation is combined with the basis change in the matrix $(M\,X)$; this requires 13 XORs per byte (with the constant $\boldsymbol{b}$ incorporated into the last round key; otherwise a NOT is needed).

For decryption, again for each of the 16 normal bases, we optimized the single 32×8 bit matrix (four transformations) for the early transformation strategy, and also the two 16×8 matrices for the later strategy. In ranking the results, recall that the early approach applies the 32×8 matrix to each input byte; the later approach applies one 16×8 matrix to each input byte but applies the other only to the shared sums, half as many bytes. This time, basis #94 won, with the best later strategy result at $19 + \frac{1}{2} \times 18 = 28$ XORs per byte, better than the best early strategy result of #58, at 31 XORs. Here are the basis change matrices for basis #94:

$$X = \begin{pmatrix} 0&1&1&1&0&1&0&0 \\ 0&1&1&1&0&0&1&0 \\ 1&0&0&0&1&0&1&1 \\ 1&0&1&1&1&1&0&1 \\ 1&1&1&1&0&1&1&0 \\ 0&0&1&1&1&0&1&0 \\ 0&0&1&0&0&0&1&0 \\ 0&0&1&0&0&1&1&0 \end{pmatrix} \quad , \quad X^{-1} = \begin{pmatrix} 0&1&0&0&1&0&1&1 \\ 0&1&1&1&0&0&1&1 \\ 1&1&0&0&0&0&0&1 \\ 0&0&1&1&0&0&0&1 \\ 0&0&1&1&0&1&1&1 \\ 0&0&0&0&0&0&1&1 \\ 1&1&0&0&0&0&1&1 \\ 1&0&0&1&1&1&1&1 \end{pmatrix}$$

For the first round of decryption, with no MixColumns, we need the transformation $X^{-1}M^{-1}$, which takes 13 XORs. After the last decryption round, before the additional AddRoundKey corresponding to encryption round #0, we need to switch back to the standard basis with X, at 13 XORs.

3.4 Galois Inverter

For the inverter of basis #127, applying (by hand) the OR gate substitution reduced the inverter size to 56 XORs, 30 ANDs, 10 ORs. This is the same number of bit operations as the inverter for basis #4 of [8], which, like that for basis #94, is 56 XORs, 34 ANDs, 6 ORs.

But in the tower-field representation, each 8-bit Galois inverter includes a 4-bit Galois inverter in the subfield. The $GF(2^4)$ inverter performs a bijection function, as does a 4×4 S-box, and hence is a natural target for optimization with methods like those in [17]. The result reduced the 4-bit inverter from 9 XORs, 8 ANDs, 2 ORs down to 8 XORs, 5 ANDs, 2 ORs, a savings of 4 bit operations.

3.5 Round Keys

The AddRoundKey step of each round is a simple bitwise XOR in any basis. In our approach, each round key must be represented in the tower-field basis. One way to do this would be to pre-compute the usual round keys by some means, then apply the X^{-1} matrix transformation to each byte. Another approach is to do the whole key schedule in the tower-field representation. First the initial key needs to be transformed into the tower field. For the last round the round key needs to be transformed, either back to the standard representation and added after the data block is transformed back, or transformed by the inverse affine transformation and added before the data is transformed back.

In our comparisons below we do not consider the cost of the key schedule that generates the round keys. We assume the round keys have been pre-computed, including their tower-field representations. This is appropriate to using our approach for bit-sliced software, where the round keys can be stored and applied to many blocks. But this assumption is less appropriate to compact hardware implementations, comparable to that of [2], where storing keys in registers is expensive, so typically keys are computed on the fly.

3.6 Validation

We implemented our approach in the form of Verilog (hardware description language) code. This code was written mainly for testing purposes, and defines one module for each kind of round, including the key schedule. We successfully tested this implementation by compiling and running it on a FPGA in a SRC 6e computer system, with correct results. (The FPGA implementation was only to check correctness; true optimization for an FPGA would need to exploit their Look-Up-Table structure.)

4 Results

We have described our methods to reduce the number of bit operations needed for AES. Our original motivation was to explore bit-slice techniques to implement AES in software. For that, reducing the number of operations essentially translates into increasing the speed. Then an appropriate measure is the total number of bit operations needed to encrypt a block, as shown in Table 1. The reduction in operations we achieved might also be useful for compact hardware implementations, where area is limited. We do not propose a specific hardware design here.

Our baseline for comparison is a compact encryption-only (or decryption-only) AES implementation using the S-box of [8] and the MixColumns of [2], shown above in subsection 3.2, and our units of comparison are bit operations: XOR, AND, OR, NOT. These two implementations are compared in detail in Table 1.

One normal round of encryption took 155 ops/byte in the baseline; our new approach needs only 139.5 ops/byte, smaller by 10.0%. However, our approach requires an initial transformation into the composite field (15 ops/byte), which adds on to the cost of round #0, the initial AddRoundKey (8 ops/byte). The

Table 1. Results. The number of bit operations per byte is given for various operations up to the full 10-round AES, comparing our approach with a baseline compact implementation

	encryption		**decryption**	
	baseline	new approach	baseline	new approach
Galois inverter	96	92	96	92
initial transformation	0	15	0	13
round transformations	24	17.5	25	28
last transformation	24	13	25	13
MixColumns	27	22	48.75	24
AddRoundKey	8	8	8	8
round #0	8	23	8	21
normal round	155	139.5	177.75	152
last round	128	113	129	113
10-round AES	1531	1391.5	1736.75	1502

last round skips MixColumns: the baseline version takes 128 ops/byte; ours takes 113 ops/byte.

For a bit-sliced software approach, a reasonable basis for comparison is the total number of bit operations. Altogether, for 128-bit keys (10 rounds), the baseline requires 24496 bit operations to encrypt one block, while ours requires 22264, which is 9.0% smaller. For 256-bit keys, our approach is 9.4% smaller. For decryption, the improvement is even greater: 13.5% for 128-bit keys and 13.8% for 256-bit keys.

For a compact hardware approach, comparison is less clear, depending on the specific architecture. Suppose we assume an encryption-only version of the compact design in [2]. There, the 32-bit data path goes through four S-boxes including transformations, a MixColumns operation, and AddRoundKey. Selectors are used to skip MixColumns on the last round and also skip the S-box for round 0. Data register connections do the ShiftRows. To simply plug in our approach, the basic round has 10% fewer operations, but the paths for round 0 and the last round would need different transformations added; the result is actually 8.1% *larger* than the baseline (based only on the bit operations in the rounds, excluding selectors and registers). A different architecture would be needed in order to take advantage of our approach, such as one where just the initial transformation into the tower-field basis, and the last tranformation to the standard representation, have 8-bit data paths instead; this approach would make those byte-serial rounds slower, but would reduce the total operations for rounds by 5.5%.

5 Conclusions

We have reduced the number of bit operations for 10-round AES by 9.0%. We achieved this reduction partly through finding a tower-field representation that compactly calculates both the Galois inversion and the constant scaling of MixColumns (when combined with the affine transformation). The other contribution to increased efficiency comes from very effective optimization, both of the 4-bit inverter (within the 8-bit Galois inverter) and of the various transformation matrices.

Our more compact AES approach may be useful for software bit-slice implementations, or for hardware with limited resources. Of course, in developing a bit-sliced program for a specific target processor, then parallelism and register constraints need to be taken into account, as well as the cost of slicing and unslicing. In fact, soon next-generation Intel and AMD processors will include single instructions to perform whole AES rounds[18], which may render bit-sliced implementations uncompetitive on such targets. However, current and older Intel and AMD processors may be promising targets for some time, as well as other possibilities such as the CellBE processor[15]. And further optimization may be possible for a particular processor, using a suitable slicing arrangement and taking advantage of specific instructions; for example, on a CellBE processor, the $GF(2^4)$ inverter takes only 8 instructions, compared to the 15 bit operations mentioned in subsection 3.4. Also, Intel's coming AVX technology, with 256-bit registers and RISC-style SSE instructions, may make bit-slicing competitive with the native AES instructions.

Future work includes developing a bitsliced AES implementation for the CellBE processor, and possibly for others.

Acknowledgements

We would like to thank the reviewers for several helpful suggestions, including pointing out some recent relevant work.

References

1. Rudra, A., Dubey, P.K., Jutla, C.S., Kumar, V., Rao, J.R., Rohatgi, P.: Efficient Rijndael encryption implementation with composite field arithmetic. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 171–184. Springer, Heidelberg (2001)
2. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact Rijndael hardware architecture with S-box optimization. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 239–254. Springer, Heidelberg (2001)
3. Wolkerstorfer, J., Oswald, E., Lamberger, M.: An ASIC implementation of the AES S-boxes. In: Preneel, B. (ed.) CT-RSA 2002. LNCS, vol. 2271, pp. 67–78. Springer, Heidelberg (2002)
4. Chodowiec, P., Gaj, K.: Very compact FPGA implementation of the AES algorithm. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 319–333. Springer, Heidelberg (2003)
5. Feldhofer, M., Wolkerstorfer, J., Rijmen, V.: AES implementation on a grain of sand. In: IEE Proceedings on Information Security, IEE, vol. 152, pp. 13–20 (2005)
6. Rijmen, V.: Efficient implementation of the Rijndael S-box (2001), `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/sbox.pdf`
7. Paar, C.: Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields. PhD thesis, Institute for Experimental Mathematics, University of Essen, Germany (1994)
8. Canright, D.: A very compact S-box for AES. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 441–455. Springer, Heidelberg (2005)
9. Rebeiro, C., Selvakumar, D., Devi, A.: Bitslice implementation of AES. In: Pointcheval, D., Mu, Y., Chen, K. (eds.) CANS 2006. LNCS, vol. 4301, pp. 203–212. Springer, Heidelberg (2006)
10. Boyar, J., Peralta, R.: New logic minimization techniques with applications to cryptology. Cryptology ePrint Archive, Report 2009/191 (2009), `http://eprint.iacr.org/`
11. Käsper, E., Schwabe, P.: Faster and timing-attack resistant aes-gcm. Cryptology ePrint Archive, Report 2009/129 (2009), `http://eprint.iacr.org/`
12. NIST: Specification for the Advanced Encryption Standard (AES), FIPS PUB 197 (2001)
13. Bernstein, D.J., Schwabe, P.: New aes software speed records. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 322–336. Springer, Heidelberg (2008)

14. Daemen, J., Rijmen, V.: AES proposal: Rijndael (1999), `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`
15. IBM: Introduction to the Cell Broadband Engine (2005), `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/D21E662845B95D4F872570AB0055404D`
16. Canright, D.: A very compact Rijndael S-box. Technical Report NPS-MA-05-001, Naval Postgraduate School (2005)
17. Osvik, D.A.: Speeding up Serpent. In: AES Candidate Conference, pp. 317–329 (2000)
18. Intel: Advanced encryption standard (AES) instructions set, rev. 2 (2009), `http://software.intel.com/en-us/articles/advanced-encryption-standard-aes-instructions-set/`