A Very Compact S-box for AES

D. Canright

Naval Postgraduate School, Monterey CA 93943, USA,
dcanright@nps.edu

Abstract. A key step in the Advanced Encryption Standard (AES) algorithm is the "S-box." Many implementations of AES have been proposed, for various goals, that effect the S-box in various ways. In particular, the most compact implementations to date of Satoh et al.[1] and Mentens et al.[2] perform the 8-bit Galois field inversion of the S-box using subfields of 4 bits and of 2 bits. Our work refines this approach to achieve a more compact S-box. We examined many choices of basis for each subfield, not only polynomial bases as in previous work, but also normal bases, giving 432 cases. The isomorphism bit matrices are fully optimized, improving on the "greedy algorithm." Introducing some NOR gates gives further savings. The best case improves on [1] by 20%. This decreased size could help for area-limited hardware implementations, e.g., smart cards, and to allow more copies of the S-box for parallelism and/or pipelining of AES.

1 Introduction

The Advanced Encryption Standard (AES) was specified in 2001 by the National Institute of Standards and Technology [3]. The purpose is to provide a standard algorithm for encryption, strong enough to keep U.S. government documents secure for at least the next 20 years. The earlier Data Encryption Standard (DES) had been rendered insecure by advances in computing power, and was effectively replaced by triple-DES. Now AES will largely replace triple-DES for government use, and will likely become widely adopted for a variety of encryption needs, such as secure transactions via the Internet.

A wide variety of approaches to implementing AES have appeared, to satisfy the varying criteria of different applications. Some approaches seek to maximize throughput, e.g., [4], [5] and [6]; others minimize power consumption, e.g., [7]; and yet others minimize circuitry, e.g., [8], [1], [9], and [10]. For the latter goal, Rijmen[11] suggested using subfield arithmetic in the crucial step of computing an inverse in the Galois Field of 256 elements—reducing an 8-bit calculation to several 4-bit ones. Satoh et al.[1] further extended this idea, using the "tower field" approach of Paar[12], breaking up the 4-bit calculations into 2-bit ones, which resulted in the smallest AES circuit to date.

Mentens et al.[2] recently examined whether the choice of representation (basis in each subfield) used by [1] was optimal. They compared 64 different choices (including that in [1]), based on the number of '1' entries in the two transformation matrices used in encryption and on the number of binary XOR operations used in one of the 4-bit operations in the subfield. Based on these criteria, they determined that a different choice is better than that in [1], and estimated the improvement at 5%.

The current work improves on the compact implementation of [1] and extends the work of [2] in the following ways. Many choices of representation (432 different isomorphisms) were compared, including all those in [2]. The cases in [2] use a *polynomial basis* in each subfield (as in [1]), while we also consider a *normal basis* for each subfield. It turns out the best case uses all normal bases. And while [1] used the popular "greedy algorithm" to reduce the number of gates in the bit matrices required in changing representations, we fully optimized each matrix by an exhaustive tree-search algorithm, resulting in the minimum number of gates. (Based on our fully optimized matrices, comparisons of matrices using the simple "number of '1' entries" criterion of [2] gives incorrect comparisons in 37% of the cases, and even the greedy algorithm gives incorrect comparisons in 20% of the cases.) We included logic optimizations both at the hierarchical level of the Galois arithmetic and at the low level of individual logic gates. We were thus able to replicate the very compact *merged* S-box reported in [1], which includes both the S-box function and its inverse, including a Galois inverter and all four transformation matrices as well as multiplexors for selecting which input and output transformations are used[13]. Hence our comparisons of the different cases are based on complete, optimized implementations of the merged S-box (rather than the two criteria of [2]), and it turns out the best case for the merged architecture is also the best for the architecture with a separate S-box and inverse S-box. Also, although the bit operations of Galois arithmetic correspond directly to XOR and AND (or NAND) gates, here certain combinations of operations are implemented more compactly using XOR and OR (or NOR) gates. These refinements combine to give a merged S-box circuit that is 20% smaller than in [1], a significant improvement.

1.1 The Advanced Encryption Standard Algorithm

The AES algorithm, also called the Rijndael algorithm, is a symmetric block cipher, where the data is encrypted/decrypted in blocks of 128 bits. Each data block is modified by several rounds of processing, where each round involves four steps. Three different key sizes are allowed: 128 bits, 192 bits, or 256 bits, and the corresponding number of rounds for each is 10 rounds, 12 rounds, or 14 rounds, respectively. From the original key, a different "round key" is computed for each of these rounds. For simplicity, the discussion below will use a key length of 128 bits and hence 10 rounds.

There are several different modes in which AES can be used [14]. Some of these, such as Cipher Block Chaining (CBC), use the result of encrypting one block for encrypting the next. These feedback modes effectively preclude pipelining (simultaneous processing of several blocks in the "pipeline"). Other modes, such as the "Electronic Code Book" mode or "Counter" modes, do not require feedback, and may be pipelined for greater throughput.

The four steps in each round of encryption, in order, are called *SubBytes* (byte substitution), *ShiftRows*, *MixColumns*, and *AddRoundKey.* Before the first round, the input block is processed by AddRoundKey. Also, the last round skips the MixColumns step. Otherwise, all rounds are the same, except each uses a different round key, and the output of one round becomes the input for the next. For decryption, the mathematical inverse of each step is used, in reverse order; certain manipulations allow this to appear like the same steps as encryption with certain constants changed. Each round key calculation also requires the *SubBytes* operation. (More complete descriptions of AES are available from several sources, e.g., [3].)

Of these four steps, three of them (*ShiftRows*, *MixColumns*, and *AddRoundKey*) are *linear*, in the sense that the output 128-bit block for such steps is just the linear combination (bitwise, modulo 2) of the outputs for each separate input bit. These three steps are all easy to implement by direct calculation in software or hardware.

The single *nonlinear* step is the *SubBytes* step, where each byte of the input is replaced by the result of applying the “S-box” function to that byte. This nonlinear function involves finding the inverse of the 8-bit number, considered as an element of the Galois field $GF(2^8)$. The Galois inverse is not a simple calculation, and so many current implementations use a table of the S-box function output. This table look-up method is fast and easy to implement.

But for hardware implementations of AES, there is one drawback of the table look-up approach to the S-box function: each copy of the table requires 256 bytes of storage, along with the circuitry to address the table and fetch the results. Each of the 16 bytes in a block can go through the S-box function independently, and so could be processed in parallel for the byte substitution step. This effectively requires 16 copies of the S-box table for one round. To fully pipeline the encryption would entail “unrolling” the loop of 10 rounds into 10 sequential copies of the round calculation. This would require 160 copies of the S-box table (200 if round keys are computed “on the fly”), a significant allocation of hardware resources.

In contrast, this work describes a direct calculation of the S-box function using sub-field arithmetic, similar to [1]. While the calculation is complicated to describe, the advantage is that the circuitry required to implement this in hardware is relatively simple, in terms of the number of logic gates required. This type of S-box implementation is significantly smaller (less area) than the table it replaces, especially with the optimizations in this work. Furthermore, when chip area is limited, this compact implementation may allow parallelism in each round and/or unrolling of the round loop, for a significant gain in speed.

The rest of the paper describes our specific algorithm in detail. (See [15] for a thorough, detailed presentation of the 432 different versions considered in finding the best one.) Section 2 explains the basic idea of the algorithm and the resulting structure of the Galois inverter. Section 3 discusses ways to optimize the calculation, Section 4 describes the changes of representation, and Section 5 describes the results. Finally, Section 6 summarizes the work.

2 The S-box Algorithm Using Subfield Arithmetic

The S-box function of an input byte (8-bit vector) $\boldsymbol{a}$ is defined by two substeps:

1. *Inverse*: Let $\boldsymbol{c} = \boldsymbol{a}^{-1}$, the multiplicative inverse in $GF(2^8)$ (except if $\boldsymbol{a} = \mathbf{0}$ then $\boldsymbol{c} = \mathbf{0}$).
2. *Affine Transformation*: Then the output is $\boldsymbol{s} = M\,\boldsymbol{c} \oplus \boldsymbol{b}$, with the constant bit matrix M and byte $\boldsymbol{b}$ shown below:

$$\begin{pmatrix} s_7 \\ s_6 \\ s_5 \\ s_4 \\ s_3 \\ s_2 \\ s_1 \\ s_0 \end{pmatrix} = \begin{pmatrix} 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \\ 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \end{pmatrix} \begin{pmatrix} c_7 \\ c_6 \\ c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

   where bit #7 is the most significant, with all bit operations modulo 2.

The second, affine substep is easy to implement; the algorithm for the first substep, finding the inverse, is described below. (Some familiarity with Galois arithmetic is assumed. A succinct introduction to Galois fields is given in [16]; for more depth and rigor, see [17]. Also, [15] conveys just enough theory to understand this algorithm.)

The AES algorithm uses the particular Galois field of 8-bit bytes where the bits are coefficients of a polynomial (this representation is called a *polynomial basis*) and multiplication is modulo the irreducible polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$, with addition of coefficients modulo 2. Let A be one root of $q(x)$; then the standard polynomial basis is $[A^7, A^6, A^5, A^4, A^3, A^2, A, 1]$. (Note: we will usually use uppercase Roman letters for specific elements of $GF(2^8)$, lowercase Greek letters for elements of the subfield $GF(2^4)$, uppercase Greek letters for the sub-subfield $GF(2^2)$, and lowercase Roman letters for bits in $GF(2)$.)

Direct calculation of the inverse (modulo an eighth-degree polynomial) of a seventh-degree polynomial is not easy. But calculation of the inverse (modulo a second-degree polynomial) of a first-degree polynomial is relatively easy, as pointed out by Rijmen [11]. This suggests the following changes of representation.

First, we represent a general element G of $GF(2^8)$ as a linear polynomial (in y) over $GF(2^4)$, as $G = \gamma_1 y + \gamma_0$, with multiplication modulo an irreducible polynomial $r(y) = y^2 + \tau y + \nu$. All the coefficients are in the 4-bit subfield $GF(2^4)$. So the pair $[\gamma_1, \gamma_0]$ represents G in terms of a polynomial basis $[Y, 1]$ where Y is one root of $r(y)$.

Alternatively, we could use the *normal basis* $[Y^{16}, Y]$ using both roots of $r(y)$. Note that

$$r(y) = y^2 + \tau y + \nu = (y + Y)(y + Y^{16}) \ , \qquad (1)$$

so $\tau = Y + Y^{16}$ is the *trace* and $\nu = (Y)(Y^{16})$ is the *norm* of Y.

Second, we can similarly represent $GF(2^4)$ as linear polynomials (in z) over $GF(2^2)$, as $\gamma = \mathit{\Gamma}_1 z + \mathit{\Gamma}_0$, with multiplication modulo an irreducible polynomial $s(z) = z^2 + Tz + N$, with all the coefficients in $GF(2^2)$. Again, this uses a polynomial basis $[Z, 1]$, where Z is one root of $s(z)$; or we could use the normal basis $[Z^4, Z]$. As above, T is the trace and N is the norm of Z.

Third we represent $GF(2^2)$ as linear polynomials (in w) over $GF(2)$, as $\mathit{\Gamma} = g_1 w + g_0$, with multiplication modulo $t(w) = w^2 + w + 1$, where g_1 and g_0 are single bits. This uses a polynomial basis $[W, 1]$, with W one root of $t(w)$; or a normal basis would be $[W^2, W]$. (Note that the trace and norm of W are 1.)

This allows operations in $GF(2^8)$ to be expressed in terms of simpler operations in $GF(2^4)$, which in turn are expressed in the simple operations of $GF(2^2)$. In each of these fields, addition (the same operation as subtraction) is just bitwise XOR, for any basis.

In $GF(2^8)$ with a *polynomial* basis, multiplication mod $y^2 + \tau y + \nu$ is given by

$$(\gamma_1 y + \gamma_0)(\delta_1 y + \delta_0) = (\gamma_1\delta_0 + \gamma_0\delta_1 + \gamma_1\delta_1\tau)y + (\gamma_0\delta_0 + \gamma_1\delta_1\nu) \ . \tag{2}$$

From this it is easy to verify that the inverse is given by

$$\begin{gathered}(\gamma_1 y + \gamma_0)^{-1} = [\theta^{-1}\ \gamma_1]\, y + [\theta^{-1}\ (\gamma_0 + \gamma_1\tau)] \\ \text{where } \theta = \gamma_1^2\nu + \gamma_1\gamma_0\tau + \gamma_0^2 \ .\end{gathered} \tag{3}$$

So finding an inverse in $GF(2^8)$ reduces to an inverse and several multiplications in $GF(2^4)$. Analogous formulas for multiplication and inversion apply in $GF(2^4)$. Simpler versions apply in $GF(2^2)$, where the inverse is the same as the square (for $\mathit{\Gamma} \in GF(2^2)$, $\mathit{\Gamma}^4 = \mathit{\Gamma}$); note then that a zero input gives a zero output, so that special case is handled automatically.

The details of these calculations change if we use a *normal* basis at each level. In $GF(2^8)$, recall that both Y and Y^{16} satisfy $y^2 + \tau y + \nu = 0$ where $\tau = Y^{16} + Y$ and $\nu = (Y^{16})Y$, so $1 = \tau^{-1}(Y^{16} + Y)$. Then multiplication becomes

$$\begin{gathered}(\gamma_1 Y^{16} + \gamma_0 Y)(\delta_1 Y^{16} + \delta_0 Y) = [\gamma_1\delta_1\tau + \theta]\, Y^{16} + [\gamma_0\delta_0\tau + \theta]\, Y \\ \text{where } \theta = (\gamma_1 + \gamma_0)(\delta_1 + \delta_0)\nu\tau^{-1} \ ,\end{gathered} \tag{4}$$

and the inverse is

$$\begin{gathered}(\gamma_1 Y^{16} + \gamma_0 Y)^{-1} = [\theta^{-1}\ \gamma_0]\, Y^{16} + [\theta^{-1}\ \gamma_1]\, Y \\ \text{where } \theta = \gamma_1\gamma_0\tau^2 + (\gamma_1^2 + \gamma_0^2)\nu \ .\end{gathered} \tag{5}$$

Again, finding an inverse in $GF(2^8)$ involves an inverse and several multiplications in $GF(2^4)$, and analogous formulas apply in the subfields.

These formulas can be simplified with specific choices for the coefficients in the minimal polynomials $r(y)$ and $s(z)$. The most efficient choice is to let the trace be unity, so from here on we let $\tau = 1$ and $T = 1$. (This is better than choosing the norm to be unity—we can't have both, and neither can be zero.)

The above shows that both polynomial bases and normal bases give comparable amounts of operations, at this level; both types remain roughly comparable at lower levels of optimization. (Of course, one could choose other types of basis at each level, but both polynomial and normal bases have structure that leads to efficient calculation, which is lacking in other bases.) We considered all of the subfield polynomial and normal bases that had a trace of unity. There are eight choices for the norm ν that make $r(y) = y^2 + y + \nu$ irreducible over $GF(2^4)$, and two choices for N that make the polynomial $s(z) = z^2 + z + N$ irreducible over $GF(2^2)$. Each of these polynomials $r(y)$, $s(z)$, and $t(w)$ has two distinct roots, and for a polynomial basis we may choose either, or for a normal basis we use both. So altogether there are $(8 \times 3) \times (2 \times 3) \times (1 \times 3) = 432$ possible cases (including the all-polynomial case used in [1]).

We compared all of these cases, in terms of complete implementations of the merged S-box architecture of [1], including all low-level optimizations appropriate to each case. The most compact was judged to be the one giving the least number of gates (using a 0.13-μm CMOS standard cell library[13]) for the merged S-box, where the encryptor and decryptor share a $GF(2^8)$ inverter. As it happens, this is also the best case for an architecture using a separate encryptor and decryptor (each with an inverter).

The most compact case uses normal bases for all subfields. Here we will give the relevant Galois elements as hexadecimal numbers, for bit vectors in terms of the standard polynomial basis for $GF(2^8)$ (powers of A). For $GF(2^8)$, the norm $\nu = 0\text{xEC}$, and $Y = 0\text{xFF}$, so the basis is [0xFE,0xFF] (recall that for each of the normal bases, the sum of the two elements is the trace, which is unity). For $GF(2^4)$, $N = 0\text{xBC}$ and $Z = 0\text{x5C}$, so the basis is [0x5D,0x5C]. (These two levels are related by $\nu = N^2 Z$.) And for $GF(2^2)$, $W = 0\text{xBD}$, and the basis is [0xBC,0xBD]. (Those two levels are related by $N = W^2$ and $W = N^2$.)

2.1 Hierarchical Structure

Here we show the structure of this best-case inverter. To clarify the subfield operations needed, we will use $\oplus$ and $\otimes$ for addition and multiplication in the subfield. In $GF(2^8)$ the only operation required is the inverse; the normal basis inverter is shown in Figure 1 and the polynomial basis inverter in Figure 2, for comparison. The operations required in the subfield $GF(2^4)$ include an inverter (same form as in $GF(2^8)$), three multipliers, two adders (bitwise XOR), and the combined operation of squaring then scaling (multiplying) by the norm ν. Note: in $GF(2^2)$ inversion is the same as squaring, which is free with a normal basis:

$$(g_1 W^2 + g_0 W)^{-1} = (g_1 W^2 + g_0 W)^2 = g_0 W^2 + g_1 W \ . \tag{6}$$

The $GF(2^4)$ multiplier is shown in Figure 3 for a normal basis; the polynomial basis version has the same operations in a slightly different arrangement. The operations required in the subfield $GF(2^2)$ include three multipliers, four adders, and scaling by the norm N. The $GF(2^2)$ multiplier has the same structure, except lacks scaling by the norm (since the norm of W is 1), and in $GF(2)$, $\otimes$ means AND.

The other operation needed in $GF(2^4)$ is the combined operation of squaring then scaling by ν (the "square-scale operation"). The form of this operation

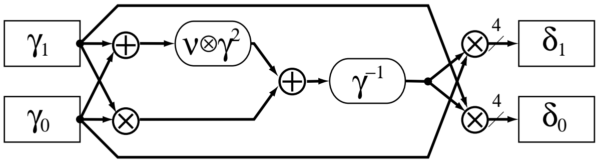

Fig. 1. Normal $GF(2^8)$ inverter: $(\gamma_1 Y^{16} + \gamma_0 Y)^{-1} = (\delta_1 Y^{16} + \delta_0 Y)$. The datapaths all have the same bit width, shown at the output (4 bits here); addition is bitwise exclusive-OR; and sub-field multipliers appear below. The $GF(2^4)$ inverter has the same structure. In $GF(2^2)$ inverting is free: a bit swap.

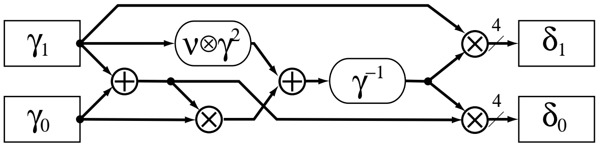

Fig. 2. Polynomial $GF(2^8)$ inverter: $(\gamma_1 y + \gamma_0)^{-1} = (\delta_1 y + \delta_0)$. The $GF(2^4)$ inverter has the same structure. In $GF(2^2)$ inverting (same as squaring) requires only one XOR.

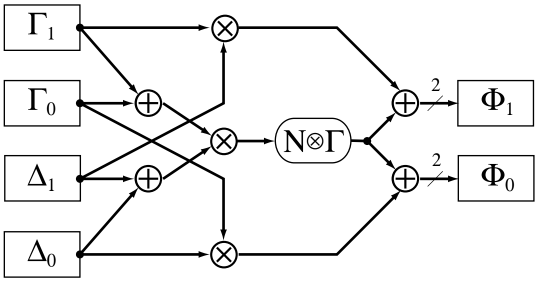

Fig. 3. Normal $GF(2^4)$ multiplier: $(\Gamma_1 Z^4 + \Gamma_0 Z) \otimes (\Delta_1 Z^4 + \Delta_0 Z) = (\Phi_1 Z^4 + \Phi_0 Z)$. The $GF(2^2)$ multiplier has the same structure except lacks the scaling by N, since the norm in the subfield is 1.

varies, depending not only on the type of basis in $GF(2^4)$, but also on the representation ν in that basis; there are a dozen different versions. Here scaling the square of $\gamma = \mathit{\Gamma}_1 Z^4 + \mathit{\Gamma}_0 Z$ by $\nu = N^2 Z$ gives

$$\nu \otimes (\mathit{\Gamma}_1 Z^4 + \mathit{\Gamma}_0 Z)^2 = [(\mathit{\Gamma}_1 \oplus \mathit{\Gamma}_0)^2] Z^4 + [(N \otimes \mathit{\Gamma}_0)^2] Z \ \ . \tag{7}$$

The only “new” operation required in the subfield $GF(2^2)$ is squaring, but this is the same as inversion, and for a normal basis is free.

The remaining operation needed in the subfield $GF(2^2)$ is scaling by $N = W^2$ (since squaring is free, this also give the square-scale operation in the $GF(2^4)$ inverter):

$$N \otimes (g_1 W^2 + g_0 W) = [g_0] W^2 + [g_0 \oplus g_1] W \ \ , \tag{8}$$

requiring a single XOR.

Also, combining the multiplication in $GF(2^2)$ with scaling by N gives a small improvement; this combination appears in the $GF(2^4)$ multiplier:

$$\begin{aligned} & N \otimes (g_1 W^2 + g_0 W) \otimes (d_1 W^2 + d_0 W) \\ & \quad = [f \oplus ((g_1 \oplus g_0) \otimes (d_1 \oplus d_0))] W^2 + [f \oplus (g_1 \otimes d_1)] W \\ & \text{where } f = g_0 \otimes d_0 \ \ . \end{aligned} \tag{9}$$

3 Inverter Optimizations

Here we will show the optimizations in the $GF(2^8)$ inverter for this best case. There are similar optimizations for other cases, described in [15]. All these optimizations were carefully calculated by hand, and so should be at least as good as versions given by automatic optimization tools.

3.1 Common Subexpressions

Eliminating redundancy where low-level subexpressions appear more than once in the above hierarchical structure reduces the size of the Galois inverter.

As [1] mentions, one place this occurs is when the same factor is input to two different multipliers. Each multiplier computes the sum of the high and low halves of each factor (see Figure 3), so when a factor is shared then this addition in the subfield can be removed from one of the multipliers. For example, a 2-bit factor shared by two $GF(2^2)$ multipliers saves one XOR (addition in the 1-bit subfield). Moreover, since each $GF(2^4)$ multiplier includes three $GF(2^2)$ multipliers, then a shared 4-bit factor implies three corresponding shared 2-bit factors in these subfield multipliers. So each shared 4-bit factor saves five XORs (one 2-bit addition and three 1-bit additions).

The normal-basis inverters for $GF(2^8)$ and $GF(2^4)$ share all three factors among the three multipliers; however, the corresponding polynomial-basis inverters each have only two shared factors (see Figures 1 and 2). This gives an advantage of five XORs to using a normal basis in $GF(2^8)$, from the additional shared factor.

A more subtle saving occurs in the $GF(2^4)$ inverter. There the bit sums computed for common factors can be used in the following square-scale operation, which saves one XOR. A similar optimization occurs in the $GF(2^8)$ inverter; combining the bit sums for shared input factors with parts of the square-scale operation saves three XORs.

3.2 Logic Gate Optimizations

Mathematically, computing the Galois inverse in $GF(2^8)$ breaks down into operations in $GF(2)$, i.e., the bitwise operations XOR and AND. However, it can be advantageous to consider other logical operations that give equivalent results.

For example, for the 0.13-μm CMOS standard cell library considered[13], a NAND gate is smaller than an AND gate. Since the AND output bits in the $GF(2^2)$ multiplier are always combined by pairs in a following XOR, then the AND gates can be replaced by NAND gates. That is, $[\,(a \otimes b) \oplus (c \otimes d)\,]$ is equivalent to $[\,(a \text{ NAND } b) \text{ XOR } (c \text{ NAND } d)\,]$. This gives a slight size saving.

Also, in this library an XNOR gate is the same size as an XOR gate. This is useful in the affine transformation of the S-box, where the addition of the constant $\boldsymbol{b} = $ 0x63 means applying a NOT to some output bits. In most cases, this can be done by replacing an XOR by an XNOR in the bit-matrix multiply, so is "free."

While the above logic optimizations are not original, here is one we have not seen elsewhere. Note that the combination $[\,a \oplus b \oplus (a \otimes b)\,]$ is equivalent to $[\,a \text{ OR } b\,]$. In the few places in the inverter where this combination occurs, we can replace 2 XORs and an AND by a single OR, a worthwhile substitution. (Actually, 2 XORs and a NAND are replaced by a NOR, smaller than an OR.) In fact, the NOR gate is smaller than an XOR gate, so even when some rearrangement is required to get that combination, it is worthwhile even if the NOR ends up replacing only a single XOR. Our implementation uses 6 NORs in the $GF(2^8)$ inverter (including two in the $GF(2^4)$ inverter).

4 Changes of Representation

This algorithm involves two different representations, or isomorphisms, of the Galois Field $GF(2^8)$. The standard AES form uses a vector of 8 bits (in $GF(2)$) as the coefficients of the 8 powers of A, the root of the defining polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$. The subfield form for $GF(2^8)$ uses a pair of 4-bit coefficients (in $GF(2^4)$) of Y^{16} and Y (for a normal basis), the roots of $r(y) = y^2 + y + \nu$. Then each element of $GF(2^4)$ is a pair of two-bit coefficients (in $GF(2^2)$) of Z^4 and Z, the roots of $s(z) = z^2 + z + N$. And in $GF(2^2)$, each element pair of one-bit coefficients (in $GF(2)$) of W^2 and W, the roots of $t(w) = w^2 + w + 1$. So the subfield representation uses pairs of pairs of pairs of bits.

One approach to using these two forms, as suggested by [8], is to convert each byte of the input block once, and do all of the AES algorithm in the new form, only converting back at the end of all the rounds. Since all the arithmetic in the AES algorithm is Galois arithmetic, this would work fine, provided the key was appropriately converted as well. However, the *MixColumns* step involves multiplying by constants that are simple in the standard basis (2 and 3, or A and $A+1$), but this simplicity is lost in the subfield basis (in our best basis, 2 and 3 become 0xA9 and 0x56). For example, scaling by 2 in the standard basis takes only 3 XORs; the most efficient normal-basis version of this scaling requires 18 XORs. Similar concerns arise in the inverse of *MixColumns*, used in decryption. This extra complication more than offsets the savings from delaying the basis change back to standard. Then, as in [1], the affine transformation can be combined with the basis change (see below). For these reasons, it is most efficient to change into the subfield basis on entering the S-box and to change back again on leaving it.

Each change of basis means multiplication by an 8×8 bit matrix. Letting X refer to the matrix that converts from the subfield basis to the standard basis, then to compute the S-box function of a given byte, first we do a bit-matrix multiply by X^{-1} to change into the subfield basis, then calculate the Galois inverse by subfield arithmetic, then change basis back again with another bit-matrix multiply, by X. This is followed directly by the affine transformation (substep 2), which includes another bit-matrix multiply by the constant matrix M. (This can be regarded another change of basis, since M is invertible.) So we can combine the matrices into the product MX to save one bit-matrix multiply, as pointed out by [1]. Then adding the constant $\boldsymbol{b}$ completes the S-box function.

The inverse S-box function is similar, except the XOR with constant $\boldsymbol{b}$ comes first, followed by multiplication by the bit matrix $(MX)^{-1}$. Then after finding the inverse, we convert back to the standard basis through multiplication by the matrix X.

For each such constant-matrix multiply, the gate count can be reduced by "factoring out" combinations of input bits that are shared between different output bits (rows). One way to do this is known as the "greedy algorithm," where at each stage one picks the combination of two input bits that is shared by the most output bits; that combination is then pre-computed in a single (XOR) gate, which output effectively becomes a new input to the remaining matrix multiply. The greedy algorithm is straightforward to implement, and generally gives good results.

But the greedy algorithm may not find the best result. We used a brute-force "tree search" approach to finding the optimal factoring. At each stage, each possible choice for factoring out a bit combination was tried, and the next stage examined recursively. (Some "pruning" of the tree is possible, when the bit-pair choice in the current stage is independent of that in the calling stage and had been checked previously. The C program is given in [15].) This method is guaranteed to find the minimal number of gates; the big drawback is that one cannot predict how long it will take, due to the combinatorial complexity of the algorithm.

The “merged” S-box and inverse S-box of [1] complicates this picture, but reduces the hardware overall when both encryption and decryption are needed. There, a block containing a single $GF(2^8)$ inverter can be used to compute either the S-box function or its inverse, depending on a selector signal. Given an input byte $\boldsymbol{a}$, both $X^{-1}\,\boldsymbol{a}$ and $(MX)^{-1}\,(\boldsymbol{a}+\boldsymbol{b})$ are computed, with the first selected for encryption, the second for decryption. That selection is input into the inverter, and from the output byte $\boldsymbol{c}$, both $(MX)\,\boldsymbol{c}+\boldsymbol{b}$ and $X\,\boldsymbol{c}$ are computed; again the first is selected for encryption, the second for decryption.

With this merged approach, these basis-change matrix pairs can be optimized together, considering X^{-1} and $(MX)^{-1}$ together as a 16×8 matrix, and similarly (MX) and X, each pair taking one byte as input and giving two bytes as output. (Then $(MX)^{-1}\,(\boldsymbol{a}+\boldsymbol{b})$ must be computed as $(MX)^{-1}\,\boldsymbol{a}+[(MX)^{-1}\,\boldsymbol{b}]$.) Combining in this way allows more commonality among rows (16 instead of 8) and so yields a more compact “factored” form. Of course, this also means the “tree search” optimizer has a much bigger task and longer run time. (Using an Intel Xeon processor under Linux, optimization times for a 16×8 matrix varied from a few minutes to many weeks.)

The additive constant $\boldsymbol{b}$ of the affine transformation requires negating specific bits of the output of the basis change. (Actually, for the merged S-box, the multiplexors we use are themselves negating, so it is the bits other than those in $\boldsymbol{b}$ that need negating first.) As mentioned in Section 3.2, this usually involves replacing an XOR by an XNOR in the basis change (both are the same size in the CMOS library we consider), but sometimes this is not possible and a NOT gate is required.

The change of basis matrix X for our best case is given below :

$$X = \begin{pmatrix} 0&0&0&1&0&0&1&0 \\ 1&1&1&0&1&0&1&1 \\ 1&1&1&0&1&1&0&1 \\ 0&1&0&0&0&0&1&0 \\ 0&1&1&1&1&1&1&0 \\ 1&0&1&1&0&0&1&0 \\ 0&0&1&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0 \end{pmatrix} \tag{10}$$

The other three matrices are easily computed from X. The combined 16×8 bit matrices for the merged architecture, fully optimized by our tree search algorithm, are given in [15]

At this time, not all of the matrices for all of the cases considered below have been fully optimized, but the data so far indicate how full optimization can improve on the greedy algorithm. For the architecture with separate encryptor and decryptor, all cases have been fully optimized: of 1728 matrices (8×8) optimized, 762 (44%) were improved by at least one XOR, and of those, 138 (18% of improved ones) were improved by two XORs, and 11 (1.4% of improved ones) were improved by three XORs. For the merged architecture, the top 27 cases have been optimized (we gave up on one matrix in case 28 after estimating optimization would take 5 years). Of 55 matrices (16×8) optimized, 24 (44%)

were improved by one XOR, 10 (18%) were improved by two XORs, and 6 (11%) were improved by three XORs, so altogether 73% were improved.

With so many optimized matrices, we could evaluate how well matrix comparisons based on the greedy algorithm or on the number of '1' entries correctly predicted the comparisons between the corresponding fully optimized matrices. We called a prediction incorrect when it predicted that one matrix was better than another, but the fully optimized version turned out worse or the same (or predicted same when one was better). For the 1492128 comparisons among the 1728 optimized 8×8 matrices, the greedy algorithm gave incorrect predictions for 19.9% of comparisons while the number of '1's incorrectly predicted 37.5%. The results for the 1485 comparisons among the 55 optimized 16×8 matrices were more dramatic: the greedy prediction was incorrect for 30.7% and the '1's prediction incorrect for 43.7%.

Since we have not yet fully optimized the (16×8) matrices for all of the 432 possible cases, it is remotely possible that some other case could turn out to be better than the case we call "best." We *have* optimized all cases whose estimated size, based on the greedy algorithm, was within 9 XORs of the actual size of our best case (except in one case, where only 1 of the 2 matrices was optimized; it improved by 2 XORs). So far, the best improvement in a single 16×8 matrix is 3 XORs, and the best improvement in the pair of matrices for a single case is 5 XORs. For some other case to be best, full optimization must improve a matrix pair, beyond what the greedy algorithm found, by at least 10 XORs. We consider this highly unlikely, and so are confident that we have indeed found the best of all 432 cases.

5 Implementation Results

The size of our best S-box is shown in Table 1, for three architectures: merged S-box and inverse S-box (one inverter, all four tranformation matrices, and two 8-bit selectors), only S-box (for just encrypting), and only inverse S-box (for just decrypting). Results are shown by number and type of logic operations, and also by total "gates," where the number refers to the equivalent number of NAND gates, using our standard cell library. We use the equivalencies 1 XOR/XNOR = $\frac{7}{4}$ NAND gates, 1 NOR = 1 NAND gate, 1 NOT = $\frac{3}{4}$ NAND gate, and 1 MUX21I = $\frac{7}{4}$ NAND gates [13]. Our merged S-Box, equivalent in size to 234 NANDs, is an improvement of 20% over that of Satoh et al. at 294 NANDs[1]. While Mentens et al.[2] use a different cell library, if we just compare equivalent NANDs our merged S-box is 14% smaller than their S-box at 272 NANDs.

Table 2 shows the effects of different levels of optimization of the inverter. Note in particular the the NOR substitution discussed in 3.2 further reduces the inverter by 9%. Table 3 show how different choices of basis affect the results. For fair comparisons, since we have not calculated fully optimized matrices and the NOR substitution improvements for all four bases shown, we show *our imple-*

mentations using greedy-algorithm matrices and exclude the NOR substitution. Our best basis is the only one of the four that uses normal bases.

Table 1. Best Case Results. Here are our best results for a complete implementation of a merged S-box & inverse, S-box alone, and inverse S-box alone. All use our best case basis with all optimizations.

best	XOR	NAND	NOR	NOT	MUX	total gates
merged	94	34	6	2	16	234
S-box	80	34	6	0	0	180
$(\text{S-box})^{-1}$	81	34	6	0	0	182

Table 2. Levels of Optimization. Here the first line shows the inverter based on the hierarchical structure (of 2.1); the next shows the improvement due to the removal of common subexpressions (of 3.1); the last shows the additional improvement from the NOR substitution (of 3.2). All use our best basis.

inverter	XOR	NAND	NOR	total gates
hierarchical	88	36	0	190
w/ low-level opt.	66	36	0	152
w/ NOR subst.	56	34	6	138

The merged S-box and inverse was implemented as a Verilog module, shown in [15], including all our optimizations. While this compact implementation is intended for ASICs, we tested this implementation using an FPGA. Specifically, we used an SRC-6E Reconfigurable Computer, which includes two Intel processors and two Virtex II FPGAs. As implemented on one FPGA, the function evaluation takes just one tick of the 100 MHz clock, the same amount of time needed for the table look-up approach.

We also implemented a complete AES encryptor/decryptor on this same system, using our S-box. Certain constraints (block RAM access) of this particular system prevent using table lookup for a fully unrolled pipelined version; 160 copies of the table (16 bytes/round × 10 rounds) would not fit (we precompute the round keys). So for this system, our compact S-box allowed us to implement a fully pipelined encryptor/decryptor, where in the FPGA, effectively one block is processed for each clock tick. (In fact, we could even fit all 14 rounds needed for 256-bit keys.)

6 Conclusion

The goal of this work is an algorithm to compute the S-box function of AES, that can be implemented in hardware with a minimal amount of circuitry. This

Table 3. Choice of Basis. Here we compare four different choices of basis: our best case, the best case of Mentens[2], the basis used by Satoh et al.[1], and our worst case. Each shows *our* complete implementation of a merged S-box & inverse, S-box alone, and inverse S-box alone. For comparison, all use the same level of optimization (using greedy-algorithm matrices and excluding the NOR substitution).

basis	type	XOR	NAND	NOT	MUX	total gates
ours	merged	107	36	2	16	253
	S-box	91	36	0	0	195
	(S-box)$^{-1}$	91	36	0	0	195
Mentens	merged	118	36	0	16	271
	S-box	96	36	0	0	204
	(S-box)$^{-1}$	97	36	0	0	206
Satoh	merged	119	36	3	16	275
	S-box	100	36	0	0	211
	(S-box)$^{-1}$	99	36	0	0	209
worst	merged	131	36	0	16	293
	S-box	107	36	0	0	223
	(S-box)$^{-1}$	106	36	0	0	222

should save a significant amount of chip area in ASIC hardware versions of AES. Moreover, this area savings could allow many copies of the S-box circuit to fit on a chip for parallelism within each round, and perhaps enough to “unroll” the loop of 10 rounds for full pipelining (for non-feedback modes of encryption), on smaller chips.

This algorithm employs the multi-level representation of arithmetic in $GF(2^8)$, similar to the previous compact implementation of Satoh et al[1]. Our work shows how this approach leads to a whole family of 432 implementations, depending on the particular isomorphism (basis) chosen, from which we found the best one. (A detailed exposition of this nested-subfield approach, including specification of all constants for each choice of representation, is given in [15].) Another improvement involves replacing some XORs and NANDs with NORs. And in factoring the transformation (basis change) matrices for compactness, rather than rely on the greedy algorithm as in prior work, we fully optimized the matrices, using our tree search algorithm with pruning of redundant cases. This gave an improvement over the greedy algorithm in 73% of the 16×8 matrices and 44% of the 8×8 matrices that we optimized.

Our best compact implementation gives a merged S-box that is 20% smaller than the previously most compact version of [1]. We have shown that none of the other 431 versions possible with this subfield approach is as small. (We did not examine issues of timing, latency and delay, but these should be comparable with [1].) This compact S-box could be useful for many future hardware implementations of AES, for a variety of security applications.

Acknowledgements

Many thanks to Akashi Satoh for his patient and very helpful discussions.

References

1. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact Rijndael hardware architecture with S-box optimization. In: Advances in Cryptology - ASIACRYPT 2001. Volume 2248 of Lecture Notes in Computer Science., Springer (2001) 239–254
2. Mentens, N., Batina, L., Preneel, B., Verbauwhede, I.: A systematic evaluation of compact hardware implementations for the Rijndael S-box. In: CT-RSA. Volume 3376 of Lecture Notes in Computer Science., Springer (2005) 323 333
3. NIST: Specification for the ADVANCED ENCRYPTION STANDARD (AES). Technical Report FIPS PUB 197, National Institute of Standards and Technology (NIST) (2001)
4. Morioka, S., Satoh, A.: A 10 Gbps full-AES crypto design with a twisted-BDD S-box architecture. In: IEEE International Conference on Computer Design, IEEE (2002)
5. Weaver, N., Wawrzynek, J.: High performance, compact AES implementations in Xilinx FPGAs. available at `http://www.cs.berkeley.edu/~nweaver/papers/AES_in_FPGAs.pdf` (2002)
6. Jarvinen, K.U., Tommiska, M.T., Skytta, J.O.: A fully pipelined memoryless 17.8 gbps AES128 encryptor. In: FPGA 03, ACM (2003)
7. Morioka, S., Satoh, A.: An optimized S-box circuit arthitecture for low power AES design. In: CHES2002. Volume 2523 of Lecture Notes in Computer Science., Springer (2003) 172–186
8. Rudra, A., Dubey, P.K., Jutla, C.S., Kumar, V., Rao, J.R., Rohatgi, P.: Efficient Rijndael encryption implementation with composite field arithmetic. In: CHES2001. Volume 2162 of Lecture Notes in Computer Science., Springer (2001) 171–184
9. Wolkerstorfer, J., Oswald, E., Lamberger, M.: An ASIC implementation of the AES Sboxes. In: CT-RSA. Volume 2271 of Lecture Notes in Computer Science., Springer (2002) 67–78
10. Chodowiec, P., Gaj, K.: Very compact FPGA implementation of the AES algorithm. In et al., C.W., ed.: CHES2003. Volume 2779 of Lecture Notes in Computer Science., Springer (2003) 319–333
11. Rijmen, V.: Efficient implementation of the Rijndael S-box. available at `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/sbox.pdf` (2001)
12. Paar, C.: Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields. PhD thesis, Institute for Experimental Mathematics, University of Essen, Germany (1994)
13. Satoh, A. personal communication (2004)
14. NIST: Recommendation for block cipher modes of operation. Technical Report SP 800-38A, National Institute of Standards and Technology (NIST) (2001)
15. Canright, D.: A very compact Rijndael S-box. Technical Report NPS-MA-04-001, Naval Postgraduate School (2004)
16. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland, New York (1977)
17. Lidl, R., Niederreiter, H.: Introduction to finite fields and their applications. Cambridge, New York (1986)