Accelerating AES with Vector Permute Instructions

Mike Hamburg

Computer Science Dept., Stanford University
`mhamburg@cs.stanford.edu`

Abstract. We demonstrate new techniques to speed up the Rijndael (AES) block cipher using vector permute instructions. Because these techniques avoid data- and key-dependent branches and memory references, they are immune to known timing attacks. This is the first constant-time software implementation of AES which is efficient for sequential modes of operation. This work can be adapted to several other primitives using the AES S-box such as the stream cipher LEX, the block cipher Camellia and the hash function Fugue. We focus on Intel's SSSE3 and Motorola's Altivec, but our techniques can be adapted to other systems with vector permute instructions, such as the IBM Xenon and Cell processors, the ARM Cortex series and the forthcoming AMD "Bulldozer" core.

Key words: AES, AltiVec, SSSE3, vector permute, composite fields, cache-timing attacks, fast implementations

1 Introduction

Since the 2001 selection of the Rijndael block cipher [6] as the Advanced Encryption Standard (AES), optimization of this cipher in hardware and in software has become a topic of significant interest.

Unfortunately, fast implementations of AES in software usually depend on a large table – 4kiB in the most common implementation – to perform the S-box and the round mixing function. While the table's size is problematic only on the most resource-constrained embedded platforms, the fact that the lookups are key- and data-dependent leads to potential vulnerabilities [2]. In an extreme case, Osvik et al. demonstrate how to extract the key from the Linux `dm-crypt` encrypted disk implementation with 65 milliseconds of measurements and 3 seconds of analysis [11].

This weakness seems to be intrinsic to the Rijndael algorithm itself. Except in bit-sliced designs, no known technique for computing the S-box is remotely competitive with table lookups on most processors, so that constant-time implementations are many times slower than table-based ones except in parallelizable modes of operation. Despite this issue, the Rijndael S-box' excellent cryptographic properties have led to its inclusion in other ciphers, including the LEX stream cipher [5], the Fugue hash function [7] and the Camellia block cipher [10].

Several processors — including the VIA C3 and higher, the AMD Geode LX and the forthcoming Intel "Sandy Bridge" and AMD "Bulldozer" cores — support hardware acceleration of Rijndael. This acceleration both speeds up the cipher and reduces its vulnerability to timing attacks. However, such hardware accelerators are processor-specific, and may not be useful in accelerating and protecting Fugue, Camellia or LEX.

We examine another hardware option for accelerating and protecting Rijndael: vector units with permutation instructions, such as the PowerPC AltiVec unit or Intel processors supporting the SSSE3 instruction set. Such units allow us to implement small, constant-time lookups with considerable parallelism. These vector units have attracted attention from AES implementors before: Bhaskar et al. considered a permutation-based implementation in 2001 [4], and Osvik et al. mention such an implementation as a possible defense against cache-based attacks [11].

To implement the S-box, we take advantage of its algebraic structure using composite-field arithmetic [12], that is, by writing $\mathbb{F}_{2^8}$ as a degree-2 field extension of $\mathbb{F}_{2^4}$. This allows efficient computation of the AES S-box without a large lookup table, and so is commonly used in hardware implementations of AES [14]. Käsper and Schwabe's software implementation in [8] takes this approach in a bit-sliced software implementation; this implementation holds the current PC processor speed record of 7.08 cycles/byte on the Intel Core i7 920 ("Nehalem") processor. Our technique achieves fewer cycles/byte on the PowerPC G4, but not on Intel processors.

As usual, hardware-specific optimizations are necessary to achieve optimal performance. This paper focuses on the PowerPC G4e[1] and Intel Core i7 920 "Nehalem", but the techniques can be used in other processors. To this end, we demonstrate techniques that are not optimal on the G4e or Nehalem, but might be preferable on other processors.

2 Preliminaries

2.1 Notation

Because we are working with fields of characteristic 2, addition of field elements amounts to a bitwise exclusive or. We will still write it as "+".

Over subfields of $\mathbb{F}_{2^8}$, we will write x/y for xy^{254}, which are equal when $y \neq 0$ because $y^{255} = 1$. This extension of $\cdot/\cdot$ adds some corner cases when dividing by 0. We will note such corner cases as they arise, and write $\approx$ instead of $=$ for formulae which are incorrect due to these corner cases.

The most frequent remedy for division by zero will be to set an "infinity flag". When dividing by a number with the infinity flag set, we will return 0 instead of the normal value. The flag is bit 4 on AltiVec and bit 7 in SSSE3, for simplicity

[1] "G4e" is an unofficial designation of the PowerPC 744x and 745x G4 processors, commonly used to distinguish them from the earlier and considerably different line of G4 processors, the PowerPC 7400 and 7410.

we simply set all 4 high bits. On AltiVec, use of an infinity flag requires extra masking to prevent the high bits of the input from interfering with the flag; in SSSE3, this masking is required anyway.

We write $a||b$ for the concatenation of a and b.

If $\boldsymbol{v}$ is a vector, then v_i is its ith component. By $\langle f(i)\rangle_{i=0}^{15}$, we mean the 16-element vector whose ith element is $f(i)$. For example, if $\boldsymbol{v}$ has 16 elements, then $\boldsymbol{v} = \langle v_i\rangle_{i=0}^{15}$.

We number bits from the right, so that bit 0 is the 2^0 place, bit 1 is the 2^1 place, and so on.

2.2 The Galois fields $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^4}$

AES is expressed in terms of operations on the Galois field $\mathbb{F}_{2^8}$. This field is written as

$$\mathbb{F}_{2^8} \cong \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$$

When we write a number in hexadecimal notation, we mean to use this representation of $\mathbb{F}_{2^8}$. For example, `0x63` $= x^6 + x^5 + x + 1$.

Because $\mathbb{F}_{2^8}$ is too large for convenient multiplication and division using AltiVec, we will work with the field $\mathbb{F}_{2^4}$, which we will write cyclotomically:

$$\mathbb{F}_{2^4} \cong \mathbb{F}_2[\zeta]/(\zeta^4 + \zeta^3 + \zeta^2 + \zeta + 1)$$

For this generator ζ, we will express $\mathbb{F}_{2^8}$ as

$$\mathbb{F}_{2^8} \cong \mathbb{F}_{2^4}[t]/(t^2 + t + \zeta)$$

The obvious way to represent elements of $\mathbb{F}_{2^8}$ is as $a + bt$ where $a, b \in \mathbb{F}_{2^4}$. A more symmetric, and for our purposes more convenient, representation is to set $\bar{t} := t + 1$ to be the other root of $t^2 + t + \zeta$, so that $t + \bar{t} = 1$ and $t\bar{t} = \zeta$. Then we may write elements of $\mathbb{F}_{2^8}$ uniquely as $xt + y\bar{t}$ with $x, y \in \mathbb{F}_{2^4}$ (here $y = a$ and $x = a + b$ from above). We will use these representations throughout this paper, and they will be reflected at the bit level: our implementations will compute with either $x||y$ or $y||(x + y)$, depending on timing constraints.

2.3 AltiVec and the PowerPC G4

We implemented AES on the PowerPC G4's AltiVec SIMD architecture, specifically the PowerPC 7447a G4e. We will be treating its 128-bit vectors as vectors of 16 bytes. In addition to bytewise arithmetic instructions, this processor has a vector permute instruction:

$$\texttt{vperm}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) := \langle(\boldsymbol{a}||\boldsymbol{b})_{c_i \bmod 32}\rangle_{i=0}^{15}$$

That is, $\texttt{vperm}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ replaces each element c_i of $\boldsymbol{c}$ with the element of the concatenation of $\boldsymbol{a}$ and $\boldsymbol{b}$ indexed by c_i's 5 low-order bits.

We will find two uses for `vperm`. The first is to permute the block for the `ShiftRows` and `MixColumns` steps of AES. In this case, $\boldsymbol{c}$ is a fixed permutation and $\boldsymbol{a} = \boldsymbol{b}$ is the input block. The second use is 16 simultaneous lookups in a 32-element table, or a 16-element lookup table when $\boldsymbol{a} = \boldsymbol{b}$.

The processor can dispatch and execute any two vector instructions of different types[2] per cycle, plus a load or store. The arithmetic operations that we will use have a 1-cycle effective latency, and the permute operations have a 2-cycle effective latency; both types have a 1/cycle throughput. Because we won't be saturating either the dispatcher or the load-store unit, loads and stores are effectively free in moderation.

2.4 Intel SSSE3

Intel's SSSE3 instruction set includes a weaker vector permute operation called `pshufb`. It differs from `vperm` in three ways. First, it only implements a 16-way shuffle, implicitly taking $\boldsymbol{a} = \boldsymbol{b}$. Second, if the highest-order bit of c_i is set, then the ith output will be 0 instead of $\boldsymbol{a}_{c_i \bmod 16}$. This is useful for implementing an infinity flag. Third, its operands follow a CISC 2-operand convention: its destination register is always the same register as $\boldsymbol{a}$, but $\boldsymbol{c}$ can be loaded from memory instead of from a register.

We will show benchmarks on three different Intel processors: a Core 2 Duo L7600 "Conroe", a Xeon E5420 "Harpertown" and a Core i7 920 "Nehalem". These processors have much more complicated pipelines than the G4e. All three can execute up to 3 instructions per cycle, all of which can be SSE logical instructions. Their shuffle units have different configurations as shown in Table 1 [1].

Core	SSE units	`pshufb` units	`pshufb` throughput	`pshufb` latency
Conroe	3	1	2 cycles	3 cycles
Harpertown	3	1	1 cycle	1 cycle
Nehalem	3	2	1 cycle	1 cycle

Table 1: Intel SSE configurations.

2.5 Log tables

Implementing multiplication and division with log tables is a well-known technique. However, it is not trivial to apply it in SIMD. AltiVec's `vperm` instruction only uses the low-order 5 bits of the permutation, so we must ensure that these 5 bits suffice to determine the result. Furthermore, when dividing we may wish to distinguish between $0/1, 0/0$ and $1/0$ in order to implement an infinity bit.

[2] There are 4 types of vector operations: floating-point operations; simple integer operations; complex integer operations such as multiplies; and permutations including whole-vector shifts, repacks and immediate loads.

Within these constraints, we worked out the following tables largely by trial and error.

For log tables over $\mathbb{F}_{2^4}$, we set

$$\log_{\text{num}}(x) = \begin{cases} \log(x) + 97, & x \neq 0 \\ -64 \equiv 192, & x = 0 \end{cases} \quad \text{and} \quad \log_{\text{denom}}(y) = \begin{cases} \log(1/y) - 95, & y \neq 0 \\ 65, & y = 0 \end{cases}$$

For multiplication, we perform an *unsigned addition with saturation*, defined as $a \uplus b := \min(a + b, 255)$ so that

$$\log_{\text{num}}(x) \uplus \log_{\text{num}}(y) = \begin{cases} 194 + \log(xy) \equiv 2 + \log(xy) \in [2, 30], & xy \neq 0 \\ 255 \equiv 31, & xy = 0 \end{cases}$$

For division, we perform a *signed addition with saturation*, defined as $a \mathbin{\cap\mkern-11mu{\scriptstyle +}} b := \min(\max(a + b, -128), 127)$ so that

$$\log_{\text{num}}(x) \mathbin{\cap\mkern-11mu{\scriptstyle +}} \log_{\text{denom}}(y) = \begin{cases} -128 \equiv 0, & x = 0 \neq y \\ 1, & x = 0 = y \\ 2 + \log(x/y) \in [2, 30], & x \neq 0 \neq y \\ 127 \equiv 31, & x \neq 0 = y \end{cases}$$

Because these sums' residues mod 32 depend only on xy or x/y (and in the same way for both), a lookup table on the output can extract xy or x/y. Furthermore, these log tables allow us to distinguish between $0/1, 1/0$ and $0/0$.

2.6 Cubic multiplication

Because `pshufb` operates on tables of size at most 16, it does not appear to admit a efficient implementation of log tables. It would be desirable to multiply instead using the "quarter-squares" identity $xy = (x+y)^2/4 - (x-y)^2/4$. Unfortunately, this identity does not work over fields of characteristic 2. We can instead set ω to a cube root of unity (so that $\omega^2 = \omega + 1$) and use an "omega-cubes" formula such as

$$xy^2 = \omega(x + \omega y)^3 + \omega^2(\omega x + y)^3 + (\omega^2 x + \omega^2 y)^3$$

which is not as horrible as it looks because the map $(x, y) \to (x + \omega\sqrt{y}, \omega x + \sqrt{y})$ is linear. If x and y are given in this basis, xy can be computed with 3 table lookups and 3 `xor`s, but transforming into and out of this basis will cost 4-6 instructions. Alternatively, the above formula can be used to compute x/y^2 and x^2/y in the high and low nibbles of a single register, but the lack of room for an infinity flag makes this strategy less useful.

On the processors we studied, cubic multiplication does not appear to be optimal technique for implementing AES, but it might be useful in other algorithms.

3 Implementing inversion

This section deals with algorithms for inverting an element $xt + y\bar{t}$ of $\mathbb{F}_{2^8}$.

3.1 Classical inversion

The simplest way to compute $1/(xt + y\bar{t})$ is to rationalize the denominator: multiplying top and bottom by $x\bar{t} + yt$ gives

$$\frac{1}{xt + y\bar{t}} = \frac{x\bar{t} + yt}{x^2 t\bar{t} + xyt^2 + xy\bar{t}^2 + y^2 t\bar{t}} = \frac{x\bar{t} + yt}{xy + (x^2 + y^2)\zeta} = \frac{x\bar{t} + yt}{(\sqrt{xy/\zeta} + x + y)^2\zeta}$$

This last, more complicated expression is how we actually perform the computation: the multiplications are computed with log tables; the squares, square roots and multiplications and divisions by ζ come for free. This technique requires few operations, but it has many lookups on the critical path. Therefore it is optimal on the G4e in parallel modes, but not in sequential modes.

3.2 Symmetric inversion

To improve parallelism, we can rearrange the above to the near-formula:

$$\frac{1}{xt + y\bar{t}} \approx \frac{t}{x + \zeta(x + y)^2/y} + \frac{\bar{t}}{y + \zeta(x + y)^2/x}$$

This formula is incorrect when $x = 0$ or $y = 0$, but this is easily fixed using an infinity flag. Since this technique can be computed in parallel, it is faster than classical inversion in sequential modes.

3.3 Nested inversion

The most efficient formula that we found for Intel processors uses the fact that

$$\frac{1}{1/x + 1/\zeta(x + y)} + y \approx \frac{xy + \zeta(x^2 + y^2)}{(1 + \zeta)x + \zeta y}$$

which leads to the monstrous near-formula

$$\frac{1}{xt + y\bar{t}} \approx \frac{t + \zeta}{\frac{1}{1/y + 1/\zeta(x+y)} + x} + \frac{\bar{t} + \zeta}{\frac{1}{1/x + 1/\zeta(x+y)} + y}$$

Division by zero is handled by using an infinity flag, which remarkably makes this formula correct in all cases. This technique performs comparably to symmetric inversion on the G4e[3], but much better on Intel because it does not require multiplication.

Figure 1 compares the parallelism of nested inversion and classical inversion. This diagram omits setup and architectural details, but is nonetheless representative: nested inversion completes faster despite having more instructions.

[3] In fact, their code is almost exactly the same. Nested inversion has different (and fewer) lookup tables, and some `xor`s replacing adds, but its dependencies are identical. It is due entirely to momentum that our G4e implementation uses symmetric inversion.

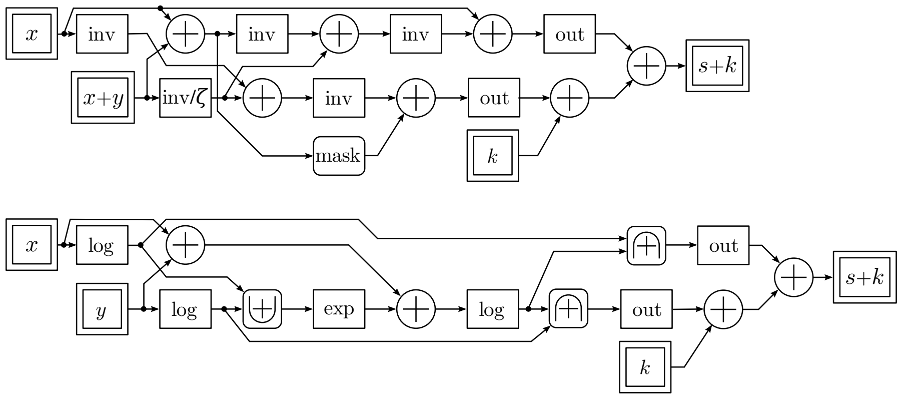

Fig. 1: Nested inversion (top) has more parallelism than classical inversion (bottom).

3.4 Factored inversion

Another approach is to separate the variables, for example:

$$\frac{1}{xt + y\bar{t}} \approx \frac{1}{x} \cdot \frac{1}{t + (y/x)\bar{t}}$$

Once we compute $\log(y/x)$, we can compute (the log of) the right term in the form $at + b\bar{t}$ with a pair of lookups. The formula is wrong when $x = 0$, but we can look up a correction for this case in parallel with the rest of the computation. This technique combines the low latency of symmetric inversion with the high throughput of classical inversion. However, its many different lookup tables cause register pressure, so we prefer to use the more specialized formulas above.

3.5 Brute force

The implementation in [4] uses a brute-force technique: a lookup in each of 8 tables of size 32 can emulate a lookup in a table of size 256. This technique is less efficient than symmetric or nested inversion on all the processors we tested. For example, nested inversion requires 7 lookups into 4 tables of size 16 (with an infinity flag) and 6 `xor`s.

4 Implementing AES

4.1 The S-box and the multiplication by `0x02`

Every inversion algorithm described above (other than brute force) ends by computing $f(a)+g(b)$ for some (f, g, a, b) using two shuffles and an `xor`. Therefore the S-box's linear skew can be folded into the tables for f and g. However, the use of infinity flags (which may force a lookup to return 0) prevents folding in the addition. Therefore, we use tables for $\texttt{skew}(f(a))$ and $\texttt{skew}(g(b))$; we add `0x63` to the key schedule instead. Similarly, we can multiply by `0x02` by computing $2\,\texttt{skew}(f(a)) + 2\,\texttt{skew}(g(b))$.

On the G4e, we make another modification to accomodate classical inversion. It happens that in the basis we use for classical inversion, $\texttt{skew}(at)$ and $\texttt{skew}(b\bar{t})$ are functions of the low-order 4 bits of $\texttt{0x2} \cdot \texttt{skew}(at)$ and $\texttt{0x2} \cdot \texttt{skew}(b\bar{t})$ but not vice-versa. As a result, we use fewer registers if we compute $\texttt{0x2} \cdot \texttt{skew}(at)$ first.

4.2 `ShiftRows` and `MixColumns`

We have three options to perform the `ShiftRows` step and the `MixColumns` rotations.

1. We could keep AES' natural alignment, with each column in 4 contiguous bytes. This would allow us to use an unmodified key schedule. On the G4e, this technique makes `MixColumns` fast at the expense of `ShiftRows`. On Intel, both require permutations.
2. We could align each row into 4 contiguous bytes. On the G4, this makes `ShiftRows` fast at the expense of `MixColumns`, but relieves register pressure. On Intel, it allows the use of `pshufd` for `MixColumns`, but the lack of SIMD rotations means that `ShiftRows` requires a permutation. Also, an both an input and an output permutation are required.
3. We could use permutations for the `MixColumns` step. We conjugate by the `ShiftRows` permutation, so we need not physically perform `ShiftRows` at all. There will be a different forward and backward permutation each round, with a period of 4 rounds. For 128- and 256-bit keys, the number of rounds is not a multiple of 4, so this technique requires either an input or an output permutation, but not both. (With the `MixColumns` technique used for classical inversion on the G4e, this method will always require an input or output permutation.) This technique seems to be the fastest both on the G4e and on Intel.

In any case, it is not advantageous to compute `ShiftRows` directly before computing `MixColumns`, because there are at least 2 registers live at all times. If `ShiftRows` is to be physically computed at all, this should be done at the beginning or end of the round, when only 1 register is live.

Let r_k denote a left rotation by k elements. To compute `MixColumns`, we compute $\boldsymbol{x} := (a, b, c, d)$ and $\texttt{0x02} \cdot \boldsymbol{x}$ as above. We compute

$$\boldsymbol{y} := r_1(\boldsymbol{x}) + \texttt{0x02} \cdot \boldsymbol{x} = (\texttt{0x02} \cdot a + b, \texttt{0x02} \cdot b + c, \texttt{0x02} \cdot c + d, \texttt{0x02} \cdot d + a)$$

We then compute the desired output

$$r_1(\boldsymbol{y}) + \boldsymbol{y} + r_3(\boldsymbol{x}) = (\texttt{0x02} \cdot a + \texttt{0x03} \cdot b + c + d,\ \ldots)$$

When using classical inversion, we compute $\texttt{0x02} \cdot \boldsymbol{x}$ before $\boldsymbol{x}$, so we use a similar addition chain that rotates $\texttt{0x02} \cdot \boldsymbol{x}$ first.

4.3 `AddRoundKey` and the modified key schedule

Naïvely we would add the round key either at the end of the round, or just before the end of the round, while waiting for a `MixColumns` permutation to finish executing. However, for some implementations, it is more convenient to add the round key during the S-box. Since every implementation of the S-box other than brute force ends with $f(a)+g(b)$ for some (f, g, a, b), we can compute $f(a)+k'+g(b)$, with the addition of k' in parallel with the computation of $g(b)$. This technique is more efficient on Intel and on the original G4, while performing the same on the G4e.

However, this trick makes the key schedule more complicated: the vector $\boldsymbol{k'}$ will be multiplied by

$$M := \begin{pmatrix} 0\,1\,1\,1 \\ 1\,0\,1\,1 \\ 1\,1\,0\,1 \\ 1\,1\,1\,0 \end{pmatrix}$$

during `MixColumns`. Since M is its own inverse, this means that we should set $\boldsymbol{k}' = M\boldsymbol{k}$. As a result, this trick may not be desirable when very high key agility is required.

Our key schedule also differs from the standard schedule in that keys must be transformed into the $(t, \bar{t})$ basis for $\mathbb{F}_{2^8}$ and rotated by the `ShiftRows` permutation.

5 Parallelism and related optimizations

5.1 Interleaving

On the G4e, the classical inversion algorithm gives few instructions, but a high latency. What is more, the instructions are balanced: they put an equal toll on the simple integer and permute units. As a result, they interleave very well: we can run 2 rounds in 24 cycles, compared to 1 round in 20 cycles using classical inversion or 17 cycles using symmetric inversion. For modes which allow parallel encryption, this boosts encryption speed by about 40%. Similiarly, we can interleave 4 rounds in 46 cycles, which should boost encryption speed by another 4% or so.

5.2 Byte-slicing

A more extreme transformation is to byte-slice encryption. This technique performs 16 encryptions in parallel. Instead of holding state for each of the 16 bytes in one encryption, a vector holds state for one byte in each of the 16 different encryptions. This allows two speedups. First, since each byte is in a separate register, no permutations are needed for `ShiftRows` and `MixColumns`. Second, the same key byte is added to every position of the vector. As a result, we can simply add this key byte to the exponentiation tables each round. Of course, to add this byte every round would save no time at all, but if we schedule each round's exponentiation table ahead of time, we will save an `xor` instruction.

We implemented this technique on the G4e, and found two difficulties. One is that we must split the data to be encrypted into byte-sliced form, and merge it back into ordinary form at the end of the encryption, which costs about 1/8 cycle per byte in each direction. A second problem is that the byte-sliced round function becomes limited by integer operations, so that saving permutations doesn't help. To alleviate this problem, we can replace a `vsrb` (element shift right) instruction with a `vsr` (whole-vector shift right) instruction, which uses the permutation unit on the G4e. In conjunction with double-triple mixing, this optimization reduces the cipher to 9.5 cycles per main round and 6.5 cycles for the final round.

Conveniently, Rogaway et al's OCB mode for authenticated encryption with associated data [13] is simpler in byte-sliced form. The offset $\varDelta$ may be stored for each slice; then most of the multiplication by 2^{16} – that is, shifting left by 2 bytes – consists of renumbering slice n as slice $n-2$. The lowest three slices will be a linear function of the upper two slices and the lowest slice; this can be computed with two shifts, 8 permutations and 6 `xor`s. This representation is highly redundant; it is possible that a more concise representation would allow a more efficient computation of the OCB offset $\varDelta$.

5.3 Double-triple mixing

Let (a, b, c, d) be the output of the S-box. Our implementation of mixing for standard round functions computed $2a$ then a. For highly parallel modes, we can do better by computing $2a$ and then $3a$. If we let

$$(\alpha, \beta, \gamma, \delta) := (3a + 2b, 3b + 2c, 3c + 2d, 3d + 2a)$$

then the output of the mixing function is

$$(\beta + \gamma + \delta, \alpha + \gamma + \delta, \alpha + \beta + \delta, \alpha + \beta + \gamma)$$

This output is easily computed using 10 `xor`s instead of 12. The technique we use computes

$$\begin{aligned}
&c_0 := \alpha, \qquad c_1 := \alpha + \beta, \qquad c_2 := \alpha + \beta + \gamma, \qquad c_3 := \alpha + \beta + \delta,\\
&c_4 := \beta + \gamma = c_2 + c_0,\\
&c_5 := \alpha + \gamma + \delta = c_3 + c_4\\
&c_6 := \beta + \gamma + \delta = c_1 + c_5
\end{aligned}$$

and outputs (c_6, c_5, c_3, c_2). In addition to taking only 10 `xor`s, this method takes its inputs in order $(\alpha, \beta, \gamma, \delta)$ and immediately `xor`s something into them. These features lead to significant savings in register usage, latency and complexity. We suspect that the savings would be even better on Intel hardware.

5.4 Counter-mode caching

Bernstein and Schwabe [3] call attention to a useful optimization in Hongjun Wu's eStream implementation of counter mode. Except in every 256th block, only the last byte of the input changes. As a result, only one S-box needs to be computed the first round. Its output affects only 4 bytes, so only 4 S-boxes need to be computed the second round. In a 10-round, 128-bit AES encryption, this saves about 1.7 rounds on average, or about 17% (slightly more, because the last round is shorter). What is more, it allows us to avoid transforming the input into byte-sliced form.

5.5 Scalar-unit assistance

Following [4], we considered using the G4e's scalar unit to perform two of the vector `xor`s, reducing the cipher to 9 cycles per round. However, the expense of shuttling data between the vector and scalar units nullifies any advantage from this technique.

6 Decryption

Decryption is more difficult than encryption, because the `MixColumns` step is more complicated: the coefficients are (`0x0E`, `0x09`, `0x0D`, `0x0B`), which are linearly independent over $\mathbb{F}_2$. As a result, all four coefficients must be looked up separately. This requires 4 more tables than encryption (minus one for permutations, because we can use only forward permutations with this method), and on Intel means that the lookup tables spill to memory.

7 Benchmarks

We initially tested several experimental implementations on the G4e. They include heavily optimized implementations of several modes several modes, but are also somewhat incomplete; in particular, we did not implement encryption of unaligned data or any sort of decryption.

After realizing that the same techniques are applicable to Intel using SSSE3, we set out to build a practical AES library for Intel machines. However, optimization on x86 processors is much more difficult than on the PowerPC, so our library does not yet approach its theoretical maximum performance. Our Intel library also does not yet implement as many modes or techniques, but it does implement encryption and decryption on aligned and unaligned data.

We tested our implementation on four machines, whose specifications are listed in Table 2.

Our byte-sliced implementations are experimental and so far incomplete. We benchmarked each individual component of the algorithm and added together the times. Similarly, on the G4e we benchmarked only the streaming steps of OCB mode, not the nonce generation and finalization. These consist of one

Machine	Processor	Core	Speed
`altacaca`	Motorola PowerPC G4 7447a	Apollo 7	1.67 GHz
`peppercorn`	Intel Core 2 Duo L7500	Conroe	1.60 GHz
`WhisperMoon`	Intel Xeon E5420	Harpertown	2.50 GHz
`lahmi`	Intel Core i7 920	Nehalem	2.67 GHz

Table 2: Bechmark machine specifications.

encryption each, so we have added the time required for two encryptions. We expect that a complete implementation would be slightly more efficient due to function call overhead.

We tested encryption and decryption on messages of size 32, 512 and 4096 bytes, with 128-, 192- and 256-bit keys.

Our classical encryption code was optimized for OCB mode; we expect that its ECB and CTR timings could be improved by 1-2% with further tuning. Due to cache effects, encryption of long messages is slightly slower than encryption of short messages in some cases.

Implementation	Par	Mode	128-bit key			192-bit key			256-bit key		
			32	512	long	32	512	long	32	512	long
Symmetric	1	ECB	11.3	10.6	10.7	14.1	12.7	12.8	17.0	14.8	14.9
	1	CBC	11.3	10.8	10.8	14.1	12.9	12.9	17.0	15.0	15.1
Classical	2	ECB	8.5	7.9	7.8	11.3	9.4	9.3	11.3	10.8	10.8
	2	CTR	9.4	7.9	7.9	11.3	9.4	9.3	11.3	10.8	10.8
	2	OCB	19.5	8.6	7.8	25.4	10.2	9.3	25.4	12.0	10.8
Classical (sliced)	16	CTR			5.4			6.7			7.9
	16	OCB			6.6			7.8			9.0
`openssl speed`	1	CBC			32.6			36.4			40.5

Table 3: Encryption timings on `altacaca` in cycles per byte.

7.1 Architecture-specific details

Alignment We tested with 16-byte-aligned input, output and key. Our Intel code supports unaligned input and output; our G4e code does not. Both implementations require 16-byte-aligned round keys, but this is enforced by our library. Our code currently only supports messages which are an integral number of blocks; we are intending to change this before release.

Implementation	Mode	128-bit key			192-bit key			256-bit key		
		32	512	long	32	512	long	32	512	long
Nested	ECB	22.0	21.6	21.5	26.3	25.3	25.5	30.6	29.9	30.1
	ECB^{-1}	27.0	26.5	26.3	32.3	31.7	31.4	37.8	37.1	37.0
	CBC	22.3	21.6	21.4	26.5	25.8	25.6	31.0	30.0	30.0
	CBC^{-1}	27.4	26.3	25.9	32.4	31.7	31.4	37.8	37.3	36.9
	CTR	22.2	21.5	21.8	26.5	25.8	25.8	30.7	30.1	29.9
	OCB	44.2	23.6	22.3	52.8	28.2	26.5	61.4	32.5	30.8
	OCB^{-1}	54.7	28.7	27.4	64.5	34.4	32.8	75.4	40.0	37.8
`openssl speed`	CBC			18.8			21.4			24.1

Table 4: Encryption timings on `peppercorn` in cycles per byte.

Implementation	Mode	128-bit key			192-bit key			256-bit key		
		32	512	long	32	512	long	32	512	long
Nested	ECB	11.8	11.1	11.0	13.9	13.2	13.3	16.1	15.4	15.4
	ECB^{-1}	14.7	14.3	14.4	17.7	17.0	17.1	20.4	19.9	19.9
	CBC	11.6	11.1	11.2	14.3	13.3	13.5	16.1	15.4	15.8
	CBC^{-1}	14.8	14.2	14.2	17.6	17.0	17.0	20.4	20.2	20.2
	CTR	11.9	11.1	11.1	14.1	13.3	13.4	16.4	15.4	15.8
	OCB	23.3	12.3	11.7	27.6	14.5	13.7	31.8	17.0	16.1
	OCB^{-1}	29.4	15.4	14.6	35.0	18.5	17.5	40.9	21.5	20.4
`openssl speed`	CBC			18.7			21.3			23.9

Table 5: Encryption timings on `WhisperMoon` in cycles per byte.

Implementation	Mode	128-bit key			192-bit key			256-bit key		
		32	512	long	32	512	long	32	512	long
Nested	ECB	10.3	10.0	9.9	12.4	11.9	11.9	14.8	13.9	13.9
	ECB^{-1}	12.9	12.4	12.4	15.3	15.0	15.0	18.0	17.6	17.6
	CBC	10.8	10.3	10.3	12.9	12.4	12.3	14.6	14.4	14.2
	CBC^{-1}	13.0	12.6	12.5	16.1	15.1	15.2	18.2	17.8	17.8
	CTR	10.4	10.0	10.0	12.4	12.0	11.9	14.2	13.9	13.9
	OCB	21.4	11.1	10.5	25.5	13.2	12.5	29.5	15.3	14.5
	OCB^{-1}	26.4	13.8	13.1	31.4	16.5	15.6	36.6	19.2	18.2
`openssl speed`	CBC			17.6			20.2			22.6

Table 6: Encryption timings on `lahmi` in cycles per byte.

Loop unrolling We did not unroll the round function at all, except in the byte-sliced case, in which we unrolled it 4 ways. Experiments showed that unrolling was generally unnecessary for optimum performance on the G4e, and our Intel code is still largely unoptimized.

8 Other processors

Our techniques are applicable to the PowerPC e600 (modern, embedded G4) with essentially no modification. Other processors have different instruction sets, pipelines and numbers of registers, and so our techniques will require modification for optimimum implementation.

The earlier PowerPC 7410 (original G4) can only issue 2 instructions per cycle instead of three. Because we no longer have "free" loads and branches, more unrolling and caching is necessary. However, the 7410's `vperm` instruction has an effective latency for only 1 cycle. After accounting for these differences, performance should be slightly faster than on the 7447 when using symmetric inversion, and about the same speed when using interleaved classical inversion. As a result, the interleaved case is less desirable.

The PowerPC 970 (G5) has much higher instruction latencies than the G4, and penalties for moving data between functional units. As a result, more parallelism is required to extract reasonable performance from the G5. It is possible that brute force is the best way to compute the S-box, due to its very high parallelism.

The IBM Cell's SPEs have many more registers than the G4e. Furthermore, their `spu_shuffle` instruction differs from `vperm` in that it assigns a meaning to the top 3 bits of their input, so inputs need to be masked before permuting. Furthermore, the SPEs lack a vector byte add with saturation, so a different log-table technique needs to be used. For multiplication, we suggest mapping 0 to `0x50` and nonzero x to $\texttt{0x30} + \log x$, so that $\log(0 \cdot 0) \rightarrow \texttt{0xA0}$ and $\log(0 \cdot x) \rightarrow [\texttt{0x80}, \texttt{0x8E}]$, all of which code for `0x00` in the `spu_shuffle` instruction. We estimate that with a byte-sliced 128-bit implementation, a Cell SPU would require approximately 8 clock cycles per byte encrypted.

The forthcoming AMD "Bulldozer" core will feature an SSE5 vector permute instruction similar to AltiVec's. Like the Cell's `spu_shuffle`, this instruction assigns additional meaning to the 3 bits of the input field, which means that more masking and different log tables will be needed. SSE5 has fewer registers than AltiVec, but its ability to take arguments from memory instead of from registers may make up for this if the latency penalty is low enough.

ARM's NEON vector instruction set features a vector permute instruction, but its performance is significantly worse than SSSE3 or AltiVec. Nested inversion is probably the most practical technique due to its smaller tables.

9 Conclusions and future work

We have presented a technique for accelerating AES using vector permute units, while simultaneously thwarting known timing- and cache-based attacks. Our technique is the first software design to yield a fast, constant-time implementation for sequential modes of operation. Our results include some 150% improvement over current implementations on the G4e[4]. On recent x86-64 processors, it is some 41% slower than Käsper and Schwabe's bitsliced implementation [8], but doesn't require a parallel mode to attain this speed.

We expect that microarchitectural optimization can improve the speed of our code significantly. This will be a major focus of future work. We also expect that this work can be applied to other primitives; it would be interesting to see if Camellia, Fugue or LEX can be implemented as efficiently.

References

1. Intel 64 and ia-32 architectures optimization reference manual, 2009.
2. Daniel Bernstein. Cache-timing attacks on AES. Technical report, 2005.
3. Daniel J. Bernstein and Peter Schwabe. New AES software speed records, 2008.
4. Raghav Bhaskar, Pradeep Dubey, Vijay Kumar, Atri Rudra, and Animesh Sharma. Efficient Galois field arithmetic on SIMD architectures. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 256–257, 2003.
5. Alex Biryukov. A new 128-bit-key stream cipher: LEX. In *eSTREAM, ECRYPT Stream Cipher Project, Report 2005/013*, 2005.
6. John Daemen and Vincent Rijmen. Aes proposal: Rijndael, 1999.
7. Shai Halevi, William Hall, and Charanjit Jutla. The hash function fugue, 2008.
8. Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant aes-gcm. In *Proceedings of CHES 2009*, 2009. to appear.
9. Helger Lipmaa. AES ciphers: speed, 2006.
10. Junko Nakajima, Kazumaro Aoki, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Tetsuya Ichikawa, and Toshio Tokita. Camellia: A 128-bit block cipher suitable for multiple platforms — design and analysis, 2000.
11. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2005.
12. Vincent Rijmen. Efficient implementation of the rijndael s-box, 2000.
13. Phillip Rogaway. Authenticated-encryption with associated-data. In *In Proc. 9th CCS*, pages 98–107. ACM Press, 2002.
14. Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 171–184, London, UK, 2001. Springer-Verlag.

[4] Bernstein and Schwabe claim 14.57 cycles/byte for CTR mode on a G4. We have not benchmarked their code on a G4e; Denis Ahrens claims a 4% speedup for AES on the G4e [9], so we estimate that Bernstein and Schwabe's code will also gain 4% on the G4e.