Cryptography, Finite Fields, and AltiVec

Bradley J. Lucier*

Abstract. The Tame Transformation Method (TTM) (`http://www.usdsi.com`) is a public-key cryptosystem based on quadratic forms in finite fields. The AltiVec instruction set in a PowerPC MPC7400 (G4) at 400 MHz allows encryption with knowledge only of the public key at a rate of over 18 million bits/second, and decoding, with knowledge of the private key, at over 50 million bits/second. In this report we explain how to exploit the AltiVec instruction set to speed the basic operations in vector spaces over a finite field, where the major computational resources of the TTM algorithm are needed.

I. Introduction

In this section we briefly examine the important connection between permutations and some ciphers (codes), and the role that finite fields can play in building and analyzing ciphers. For an extensive history of ciphers and their analysis, see [1]. In the next section we briefly describe the finite field with 256 elements, and in the final section we show how to implement vector (linear algebra) operations on the field with 256 elements very quickly using the AltiVec instruction set.

One of the simplest ciphers is a *substitution cipher*. In this cipher, each character in the original text, or *plain text*, is replaced by a fixed character in the encoded text, or *cipher text*. A substitution cipher given by the following cipher table

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BELGIUMJKNOPQRSTVWXYZACDFH
```

means that each `A` in the plain text is replaced by a `B` in the cipher text, `E` is replaced by `I`, etc. To invert the cipher, one simply replaces each occurrence of `B` in the cipher text with `A`, etc. The second line of letters, which contains all the letters of the first line in a different order, is called a *permutation* of the first line of letters.

Substitution ciphers are simple to break; one can use the known statistics of occurrences of the various letters in English text to guess the letter substitutions. For example, since `E` is the most commonly used letter in English, if `I` is the most-often-occurring letter in the cipher text, one might reasonably guess that it represents the letter `E`, etc.

The Enigma cipher used by the Germans in the Second World War was a substitution cipher, but with a cryptographically important twist—the permutation used to encode each letter of the plain text depended on the position of the letter in the plain text. For example, the first letter in a text might be permuted using the table above, while the second may use the table

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
FRANCETUVWXYZBDGHIJKLMOPQS
```

and the third letter would use yet another table, etc. A machine was built to keep track of which tables were used to encode each character, the Enigma machine, and messages encoded with this cipher were very difficult (but not impossible, it turned out) for the Allies to decipher.

One would like modern ciphers to be resistant, in a strong sense, to cryptographical analysis. In other words, one would like a mathematical analysis that shows that analysis of a cipher is difficult given the current state of mathematical knowledge. The RSA public-key cryptosystem is secure in this sense, since the only currently known way to break it involves factoring numbers that are the products of two large primes, and neither mathematicians nor computer scientists know

* Department of Mathematics, 1395 Mathematical Sciences Building, Purdue University, West Lafayette, IN 47907-1395. `lucier@math.purdue.edu`

how to do this quickly. Thus, people can be reasonably confident that RSA is cryptographically secure.

But RSA depends on modular arithmetic with large integers, which is relatively slow when implemented on modern microprocessors. Because of this, RSA is sometimes used only to pass the keys of a weaker cipher, such as DES, that can be implemented quickly for the bulk of the message.

It is difficult to reason mathematically about arbitrary permutations, and so gain this mathematical confidence that a cipher based on permutations is cryptographically secure. However, one finds that a restricted set of permutations, given as the addition and multiplication operations in a finite field, are permutations that satisfy the laws of algebra, much as rational numbers satisfy the laws of algebra. Thus, these two types of permutations can be combined in rather complicated ways, yet mathematicians are still able to reason about their effects and complexity of analysis. The Tame Transformation Method, a more recent public-key cryptosystem, is based on compositions of these types of permutations, and is (so far) cryptographically secure in the sense given above.

II. Arithmetic in Finite Fields

A *field* is a set of objects on which operations of addition and multiplication are defined; one requires that these operations satisfy the usual properties of addition and multiplication operating on real numbers, namely, that they be commutative ($x+y=y+x$) and associative ($(x+y)+z = x+(y+z)$), that multiplication distribute over addition, that they have identities (there are numbers 0 and 1 such that for all x, $x+0=x$ and $x\times 1=x$), that all numbers have additive inverses (for all x, there is a y such that $x+y=0$), and that nonzero numbers have multiplicative inverses (for all $x\neq 0$, there is a y such that $xy=1$).

A simple argument shows that the set with two symbols $\{0,1\}$ and operations defined by

$$\begin{array}{llll} 0+0=0, & 0+1=1, & 1+0=0, & 1+1=0; \\ 0\times 0=0, & 0\times 1=0, & 1\times 0=0, & 1\times 1=1; \end{array}$$

is a field. We denote this field by $\mathbb{Z}_2$. Note that these operations are those of Boolean logic when $0\equiv$ False, $1\equiv$ True, $+\equiv$ XOR (exclusive or), and $\times\equiv$ AND. $\mathbb{Z}_2$ is a field with two elements, 0 and 1.

Fields with 2^k elements can also be constructed. When $k=8$, then each element in the field can be associated with an 8-bit value, a byte, since there are $2^8=256$ different bytes. One construction of the 256-element field proceeds as follows.

Let us consider polynomials in one unknown (say x) of degree less than 8 and with coefficients in $\mathbb{Z}_2$. That is, the coefficient of each term in a polynomial is either 0 or 1. One example is

$$x^7+x^3+1 = 1\times x^7+0\times x^6+0\times x^5+0\times x^4+1\times x^3+0\times x^2+0\times x^1+1\times x^0.$$

There are 8 possible coefficients, each of which can be 0 or 1, so there are $2^8=256$ different polynomials of this form.

You add polynomials by adding their coefficients, taking into account that you must use the addition rule of $\mathbb{Z}^2$. So, for example, $(x^7+x^3+1)+(x^6+x^3+x^2+1)$ is

$$\begin{aligned} &1\times x^7+0\times x^6+0\times x^5+0\times x^4+1\times x^3+0\times x^2+0\times x^1+1\times x^0 \\ &+0\times x^7+1\times x^6+0\times x^5+0\times x^4+1\times x^3+1\times x^2+0\times x^1+1\times x^0 \\ &=1\times x^7+1\times x^6+0\times x^5+0\times x^4+0\times x^3+1\times x^2+0\times x^1+0\times x^0 \end{aligned}$$

The association we use between bytes and polynomials is particularly simple—the bits in the byte are simply the coefficients of the polynomial, so

$$10001001\equiv 1\times x^7+0\times x^6+0\times x^5+0\times x^4+1\times x^3+0\times x^2+0\times x^1+1\times x^0,$$

etc., so the above addition can be written as

$$\begin{aligned} & 10001001 \\ +\ & 01001101 \\ =\ & 11000100. \end{aligned}$$

Notice that addition is just the XOR of the two bytes, so it is very fast. Note also that XOR of one byte with another is invertible—(`A` XOR `B`) XOR `B`=`A`—so addition by a fixed byte `B` is a permutation of the 256 possible values of 8-bit bytes.

Multiplication is more complicated. Multiplying two polynomials of degree less than 8 can result in a polynomial of degree 14; the coefficients of this polynomial require 15 bits to store. We need a multiplication that maps the product of two 8-bit values to another 8-bit value.

You do this by fixing an *irreducible polynomial* of degree 8, and taking the remainder of the product polynomial when divided by this irreducible polynomial. We won't give the definition of irreducible here (see [2] for the definition); one such polynomial is $P(x) = x^8 + x^6 + x^5 + x + 1$. We have with the polynomials in our example of addition,

$$\begin{aligned} & (x^7 + x^3 + 1) \times (x^6 + x^3 + x^2 + 1) \\ & \quad = x^7 \times (x^6 + x^3 + x^2 + 1) + x^3 \times (x^6 + x^3 + x^2 + 1) + (x^6 + x^3 + x^2 + 1) \\ & \quad = x^{13} + x^{10} + x^9 + x^7 + x^9 + x^6 + x^5 + x^3 + x^6 + x^3 + x^2 + 1 \\ & \quad = x^{13} + x^{10} + x^7 + x^5 + x^2 + 1 \\ & \quad = (x^8 + x^6 + x^5 + x + 1) \times (x^5 + x^3 + x + 1) + (x^6 + x^5 + x^4 + x^3) \end{aligned}$$

or

$$(x^7 + x^3 + 1) \times (x^6 + x^3 + x^2 + 1) = x^6 + x^5 + x^4 + x^3 \mod (x^8 + x^6 + x^5 + x + 1).$$

In this field, each *nonzero* polynomial has a multiplicative inverse, so multiplication by a fixed nonzero field element is a permutation of the 256 possible values in the field.

Written in terms of bytes, we have $10001001 \times 01001101 = 01111000$. Since there are 2^8 possible values for the left multiplicand, and 2^8 for the right, one could precompute a table of results of all $2^8 \times 2^8 = 2^{16}$ possible products; each entry in the table would be a single byte containing the result. Then the product of bytes `a` and `b` is simply `multiplication_table[(a << 8) + b]`.

This is not so bad, but with AltiVec one can do better.

III. Fast Linear Algebra in Finite Fields

The two major linear algebra operations are vector addition and scalar multiplication. If these are fast, then standard techniques allow fast matrix-vector multiplication, etc. We now discuss how to do these basic operations with AltiVec.

Because addition in this field is just bytewise exclusive-or, fast addition is trivial. For example, if `a`, `b`, and `c` are pointers to (8-bit) unsigned chars that are aligned on 16-byte boundaries and `n` is a multiple of 16, we can simply write, using the notation of the Codewarrior C compiler AltiVec extensions,

```
(void)
finite_field_vector_addition (unsigned char *a,
                              unsigned char *b,
                              unsigned char *c,
                              int n)
{
  int i;
```

```
  vector unsigned char *vec_a = (vector unsigned char *) a;
  vector unsigned char *vec_b = (vector unsigned char *) b;
  vector unsigned char *vec_c = (vector unsigned char *) c;
  for (i = 0; i < n; i += 16)
    *vec_c++ = vec_xor (*vec_a++, *vec_b++);
}
```

See [3] for a description of the available instructions and the C interface to AltiVec.

Fast scalar multiplication is somewhat trickier. Here we want to calculate $y = \alpha x$ where α is a scalar (i.e., an element of the field with 256 elements) and x is a vector of n elements of the same field; again we assume that n is a multiple of 16. So each element $y_i = \alpha \times x_i$, $0 \le i < n$.

If we write the bit representation of x_i as $x_i = abcdefgh$, then we note that

$$\alpha \times x_i = \alpha \times abcdefgh = \alpha \times (0000efgh + abcd0000) = \alpha \times 0000efgh + \alpha \times abcd0000$$

where $\times$ is, of course, multiplication in the finite field, and $+$ is addition in the same field. Note that for α fixed, there are only 16 possible values for $\alpha \times 0000efgh$, i.e., $\alpha \times 00000000 = 00000000$, $\alpha \times 00000001 = \alpha$, etc., up to $\alpha \times 00001111$. Similarly, there are precisely 16 possible values of $\alpha \times abcd0000$.

Thus, for each of the 256 different values of α we precompute a table of 16 different low products $\alpha \times 0000efgh$, $efgh = 0000, 0001, \ldots, 1111$, and a table of the 16 different high products $\alpha \times abcd0000$, $abcd = 0000, 0001, \ldots, 1111$. In our scalar multiplication routine, we note the value of α and load these two 16 byte tables into AltiVec vector unsigned char variables. Multiplication by α of the low part of x_i is simply a table lookup into the table of low-part products; similarly for multiplication by α of the high part of x_i.

The vector permute instruction `vec_perm` found in the AltiVec instruction set is ideal for lookups into small byte tables. In general, one can have 16 simultaneous lookups from a 32-byte table; the 16 indices are specified by one vector unsigned char register, and the 32-byte table is specified by two vector unsigned char registers. The five lowest bits of each index byte are used as an index into the 32-byte table.

Since we are doing lookups in 16-byte tables instead of 32-byte tables, we specify the same 16-byte vector unsigned char register for each half of the 32-byte table. This has the added advantage that we can leave the fifth bit of each index byte unmasked for the lookup of α times the low part of each x_i, since the same table is used for both the first 16 bytes and the second 16 bytes of the 32-byte table. To do the high-part lookup, we need to shift each element x_i to the right by four bits, for which purpose we use the `vec_sr` AltiVec instruction.

A complete routine (except for the initialization of the tables, which proceeds using the scalar algorithm of the previous section) is

```
vector unsigned char low_products[256];
vector unsigned char high_products[256];

/* Initialize tables of low products and high products here. */

(void)
finite_field_scalar_multiplication (unsigned char alpha,
                                    unsigned char *x,
                                    unsigned char *y,
                                    int n)
{
  int i;
  vector unsigned char *vec_x = (vector unsigned char *) x;
```

```
    vector unsigned char *vec_y = (vector unsigned char *) y;
    vector unsigned char low  = low_products[(int) alpha];
    vector unsigned char high = high_products[(int) alpha];

    for (i = 0; i < n; i += 16, vec_x++, vec_y++)
      {
        vector unsigned char l, h;
        l = vec_perm (low, low, *vec_x);
        h = vec_perm (high,
                      high,
                      vec_sr (*vec_x, (vector unsigned char) (4)));
        *vec_y = vec_xor (l, h);
      }
  }
```

Here we have assumed again that `n` is a multiple of 16 and x and y are 16-byte aligned. The loop compiles to 8 instructions (unrolled) for each value of `i`.

IV. Performance

In this section we compare scalar and AltiVec-enabled programs for encoding data using the Tame Transformation Method (TTM), a recently developed public-key cryptosystem. We focus on the encoding process, which is slower than the decoding process (which uses the private key), and which is more amenable to vector processing with AltiVec.

The TTM procedure is a block cipher. An input block has length m bytes and an output block has length n bytes; for our particular ciphers, $n \geq m + 36$. The public key has two parts: a matrix key, consisting of n lower triangular matrices A_i, each of size $m \times m$ with entries in the field with 256 elements, and a vector key, consisting of n vectors v_i of length m, again with entries in the field of 256 elements. Given an input block x consisting of m bytes, the ith byte of the output is calculated as

$$x^T \cdot (v_i + A_i x)$$

where x^T denotes the transpose of the column vector x and $x \cdot y$ is the dot product of x and y.

The private key consists of two matrices, one $m \times m$ and the other $n \times n$, and one vector of length m. Thus, the number of bits in the private key is $m^2 + n^2 + m$ times the number of bits in each key entry.

It turns out that using 8 bits for each entry of the public and private keys yields a system that is cryptographically more secure than necessary—4-bit entries suffice. We see this even with our smallest example below, for which $m = 28$ and $n = 64$, so there are $(28^2 + 64^2 + 28) \times 4$ =19,632 bits in the private key, clearly much too large for an exhaustive search. Attacks based on higher mathematics have been proposed, but all these attacks require at least 2^{90} operations for success, which is more than enough to declare the codes cryptographically secure. So we restrict ourselves to 4-bit key entries. One can see this offers significant speedups in the vector operations outlined in the previous section, now requiring only one vector load, one vector permutation, and one vector exclusive-or for 16 scalar multiply-add operations.

Our experiments indicate that a well-written scalar program for TTM encryption requires about 2.56 cycles per multiply-add pair on a Motorola 7400 (PowerPC G4) processor; this multiplication, followed by an addition, is compiled to two load instructions and one exclusive-or operation, so one can see that a certain amount of instruction-level parallelism is exploited by the program.

Thus, one expects a scalar implementation of TTM encryption to take about

$$2.56 \times \left(\frac{m(m+1)}{2} + m \right) \times n$$

cycles to encode one input block. We use this formula in Table 1 below.

Our first example has $m = 28$ and $n = 64$. Here the coefficients of A_i and v_i, $i = 1, \ldots, 64$, together with the tables of high and low products and temporary storage, all fit in the 32 KByte level-one (L1) data cache on a PowerPC G3/G4 processor. One obtains an almost perfect speed-up of 14.9 over the scalar code. The output block of this code is $64/28 = 2.29$ times the size of the input block; this is the expansion factor of this code.

It is clear from the formula given above that the number of operations required for codes with larger m and n increase quadratically in m. On the other hand, codes with larger input blocks and $n = m + 36$ will have smaller expansion factors. Therefore, we examined the speed of a code with input blocks of 36 bytes and output blocks of 72 bytes, so that the expansion factor is two.

The programming of the 36/72 code was more difficult. We, in fact, computed a 36/80 code since AltiVec registers are 16 bytes wide, and we wanted the size of the ouput block to be a multiple of the AltiVec register width. Secondly, 80 half-byte matrix coefficients require 40 bytes to store, which is again not divisible by 16, so some padding of the public encryption key was necessary.

Finally, unlike our previous example, the public keys for this code do not fit in the L1 data cache with the multiplication tables, the temporary storage we used, and the input and output blocks. This caused some cycling of the contents of the L1 cache, resulting in relatively poor performance of the AltiVec code, with a speed-up of only 8.4 over the scalar code.

We got around this in the following way. The L1 data cache on the Motorola 7400 is 32 Kbytes in size and is eight-way associative. By using the `lvxl` instruction, which loads a vector register while marking it as the least-recently-used of the eight associated data lines in the L1 cache, we were able to specify that the only data we wanted to keep resident in the L1 cache were the matrix key, the input and output blocks, and the temporary memory we used (which was the same length as the vector key). This significantly reduced the level of cache thrashing and resulted in a speed-up of 11.1 over the scalar code. When considered as a 36/80 code, the speed-up of 12.3 is more impressive, because the scalar program is slower while the AltiVec program runs at the same speed.

Table 1. Encryption Speeds for TTM on a 400MHz PowerPC G4 7400

Input/Output block sizes (in bytes)	28/64	36/72
Scalar speed	1,260,080 bits/sec.	890,313 bits/sec.
AltiVec speed without cache hints	18,801,311 bits/sec.	7,488,117 bits/sec.
AltiVec speed with cache hints		9,841,225 bits/sec.

The final result is a secure, public-key cryptosystem with an encoding rate of over 18 million bits/second on a commodity microprocessor. Using AltiVec technology has a smaller effect on decoding speed, since the scalar implementation of decoding has fewer opportunities for vectorization, but the final speed is more impressive—over 50 million bits/second with the 28/64 code.

REFERENCES

[1] D. Kahn, *The Codebreakers*, 2nd ed., Scribner, New York, NY, 1996.
[2] S. MacLane and G. Birkoff, *Algebra*, 3rd ed., Chelsea Pub. Co., New York, NY, 1988.
[3] Motorola, *AltiVec Technology Programming Interface Manual*, `http://www.altivec.org/`.