A Fast Algorithm for Testing Irreducibility of Trinomials mod 2

(preliminary report)[1]

Richard P. Brent
Oxford University Computing Laboratory
Oxford OX1 3QD, UK
rpb@comlab.ox.ac.uk

Samuli Larvala
Helsinki University of Technology
Helsinki, Finland
slarvala@cc.hut.fi

Paul Zimmermann
INRIA Lorraine
615 rue du Jardin Botanique
BP 101, F-54600 Villers-les-Nancy
France
Paul.Zimmermann@loria.fr

Programming Research Group
Report PRG–TR–13–00
30 December 2000

Abstract

The standard algorithm for testing reducibility of a trinomial of prime degree r over GF(2) requires $2r+O(1)$ bits of memory and $\Theta(r^2)$ bit-operations. We describe an algorithm which requires only $3r/2 + O(1)$ bits of memory and significantly fewer bit-operations than the standard algorithm. Using the algorithm, we have found 18 new irreducible trinomials of degree r in the range $100151 \le r \le 700057$.

If r is a Mersenne exponent (i.e. $2^r - 1$ is a Mersenne prime), then an irreducible trinomial is primitive. Primitive trinomials are of interest because they can be used to give pseudo-random number generators with period at least $2^r - 1$. We give examples of primitive trinomials for $r = 756839$, 859433, and 3021377. The three results for $r = 756839$ are new. The results for $r = 859433$ extend and correct some computations of Kumada *et al.* [*Math. Comp.* 69 (2000), 811–814]. The two results for $r = 3021377$ are primitive trinomials of the highest known degree.

rpb199tr typeset using LaTeX

1 Introduction

Throughout this report all polynomials are assumed to be over the finite field GF(2). For the sake of brevity, we shall not repeat this below.

We say that a polynomial $P(x)$ is *reducible* if it has nontrivial factors; otherwise it is *irreducible*. We say that a polynomial $P(x)$ of degree $r > 1$ is *primitive* if $P(x)$ is irreducible and $x^j \neq 1 \bmod P(x)$ for $0 < j < 2^r - 1$. If $P(x)$ is primitive, then x is a generator for the multiplicative group of the field $Z_2[x]/(P(x))$, so we have a useful representation of GF(2^r). See Lidl and Niederreiter [16] or Menezes *et al.* [20, §§2.6.3,4.5,6.2] for background information.

There is an interest in discovering primitive trinomials of high degree r because of their connection with fast, high-quality pseudorandom number generators of period (at least) $2^r - 1$ and good statistical properties in all dimensions $D < r$, see [1, 3, 4, 5, 9, 11, 13, 17, 18, 21, 26].

To test if an irreducible polynomial of degree r is primitive, we need to know the factorization of $2^r - 1$. Zierler and Brillhart [28, 29] considered $r \leq 1000$ for which the factorization of $2^r - 1$ was known. We say that r is a *Mersenne exponent* if $2^r - 1$ is a (Mersenne) prime. In this case the factorization of $2^r - 1$ is trivial and an irreducible polynomial of degree r is necessarily primitive. Because several large Mersenne exponents are known [7], there is a possibility of finding primitive trinomials of high degree if we have an efficient algorithm for testing reducibility.

Zierler [30] gave all primitive trinomials of Mersenne exponent $r \leq 11213$. Kurita and Matsumoto [15] extended this to $r \leq 86243$, and Heringa *et al.* [9] to $r \leq 216091$.

Kumada *et al.* [14] did not consider the next Mersenne exponent ($r = 756839$), but conducted an exhaustive search for $r = 859433$ and found one primitive trinomial.

In this report we describe a new algorithm for testing reducibility of trinomials of prime degree, and give some results obtained by applying the algorithm to the Mersenne exponents $r \leq 3021377$. In particular, we have verified the published results for $r \leq 216091$, found three new primitive trinomials for $r = 756839$, found a primitive trinomial for $r = 859433$ which was missed by Kumada *et al.* [14], and found two new primitive trinomials for $r = 3021377$. At the time of writing (December 2000) the search for $r = 3021377$ is incomplete.

Sieving is discussed in §2 and §5. In §3 we describe the standard algorithm for testing reducibility, and in §4 we describe our new algorithm. Some performance figures are given in §6. The computational results are summarised in §7 and Tables 5–4.

2 Sieving

Testing a polynomial $P(x)$ for irreducibility is analogous to testing a number N for primality. We can save time by first checking if $P(x)$ is divisible by an irreducible polynomial of low degree, in the same way that we can save time by checking if N is divisible by a small prime.

The following theorem characterises the irreducible polynomials of given degree. The proof is well-known, see for example [16].

Theorem 1 *Let $\Phi_{d,1}, \Phi_{d,2}, \ldots$ be the irreducible polynomials of degree d. Then, for $n \geq 1$,*

$$\prod_{d|n} \prod_{j} \Phi_{d,j}(x) = x^{2^n} + x \; .$$

Example. Taking $n = 3$ in Theorem 1 we have

$$x^8 + x = x^8 - x = x(1 + x)(1 + x + x^3)(1 + x^2 + x^3) \; ,$$

so the irreducible polynomials of degree one are $\{x,\ 1 + x\}$, and those of degree three are $\Phi_{3,1}(x) = 1 + x + x^3$ and its *reciprocal* $\Phi_{3,2}(x) = x^3 \Phi_{3,1}(1/x) = 1 + x^2 + x^3$.

Corollary 1 *Let J_n be the number of irreducible polynomials of degree n. Then*

$$\sum_{d|n} dJ_d = 2^n \ ,$$

and by Möbius inversion

$$J_n = \frac{1}{n} \sum_{d|n} 2^d \mu(n/d) \ .$$

In particular, $J_n \sim 2^n/n$ and, if n is prime, then $J_n = (2^n - 2)/n$.

Remark. The number of primitive polynomials of degree n is $\phi(2^n - 1)/n$, where ϕ denotes Euler's phi function: see [20, Fact 2.230].

From Theorem 1, we can check if $P(x)$ is divisible by some irreducible polynomial $\Phi_{d,j}(x)$ of degree $d|n$ by computing $\mathrm{GCD}(P(x), x^{2^n} + x)$. By analogy with the process of sieving out small integer factors, this process will be called *sieving*, although the sieve is performed by a GCD computation[2]. We are interested in the case that $P(x) = x^r + x^s + 1$ is a trinomial, and we always assume that $r > s > 0$.

Consider the computation of $G = \mathrm{GCD}(x^r + x^s + 1, x^{2^n} + x)$. If $k = 2^n - 1$, then $G = \mathrm{GCD}(x^r + x^s + 1, x^k + 1)$. In practice, we distinguish two cases:

1. If $r \geq k$, we can use the fact that $G = \mathrm{GCD}(x^{r'} + x^{s'} + 1, x^k + 1)$ where $r' = r \bmod k$, $s' = s \bmod k$. Thus, for small k the computation of G is trivial (independent of r). We could precompute the possible G and store in a lookup table of size $\Omega(k^2)$, but our current program does not do this because the time taken for this case is negligible.

2. If $k > r$ the standard Euclidean algorithm would take time $O(k^2)$ and space $\Omega(k)$. We save time and space by first computing $x^{2^n} \bmod P(x)$ by squaring and reducing n times. Then we apply the Euclidean algorithm to compute $G = \mathrm{GCD}(x^{2^n} \bmod P(x) + x, P(x))$. The overall time is $O(r(n + r))$ and space $\Omega(r)$ (for details see §§3–4).

Sieving is performed with $n = 2, 3, 4, \ldots$ until one of the following holds:

1. We find a nontrivial GCD, in which case $P(x)$ is reducible.

2. The estimated time $T_s(n)$ which would be required to perform another sieving step satisfies $nT_s(n) \geq T_f(r)$, where $T_f(r)$ is the (estimated) time required for the full reducibility test of §4. The rationale for this criterion is given in §5.

3. $n \geq \lfloor r/2 \rfloor$, in which case $P(x)$ is irreducible (in practice, it is unlikely that we sieve this far).

3 The Standard Algorithm

The standard algorithm for testing reducibility of a polynomial $P(x)$ uses the following Theorem, which is an easy consequence of Theorem 1. In practice we first sieve out "small" factors of $P(x)$ as described in §2, but this is not essential and is only done for reasons of efficiency (for details see §5).

[2]In fact, sieving in this manner corresponds more closely to sieving out integer factors with n binary digits than to sieving out multiples of the n-th prime.

Theorem 2 *Let $P(x)$ be a polynomial of degree $r > 1$. Then $P(x)$ is irreducible iff*

$$x^{2^r} = x \bmod P(x)$$

and, for all d, $1 \le d < r$, if $d|r$ then

$$GCD(x^{2^d} + x, P(x)) = 1 \ .$$

Remark. The second condition in Theorem 2 is necessary, as the example

$$P(x) = (1 + x)(1 + x + x^2)(1 + x + x^3)$$

of degree 6 shows. However, it is trivial if the degree r is prime, which is the case in our applications (see §7). Hence, from now on we avoid complications by assuming that r is prime. In the case of interest to us, Theorem 2 reduces to:

Corollary 2 *If $r > s > 0$, where r is prime, then $P(x) = x^r + x^s + 1$ is irreducible iff*

$$x^{2^r} = x \bmod P(x) \ .$$

To implement Corollary 2 we have the following algorithm:

$A(x) \leftarrow x$;
for $j \leftarrow 1$ to r do
    $A(x) \leftarrow A(x)^2 \bmod P(x)$;
if $A(x) = x$ then return "irreducible"
    else return "reducible".

Full reducibility test

The inner loop, which is executed r times, consists of a *squaring* step $A(x) \leftarrow A(x)^2$, and a *reduction* step $A(x) \leftarrow A(x) \bmod P(x)$. Each of these steps can be implemented in $\Theta(r)$ bit-operations (as explained below) and $\Theta(r)$ space.

To consider the implementation in more detail, we assume that the polynomial $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ is represented as a bit-string[3] $a_0 a_1 \ldots a_d$. Because we are working over GF(2), the cross terms in $A(x)^2$ vanish and we have simply

$$A(x)^2 = a_0 + a_1 x^2 + \cdots + a_d x^{2d} \ ,$$

which is represented as a bit-string (with optional zero padding on the right):

$$a_0 0 a_1 0 a_2 0 \ldots a_{d-1} 0 a_d 0 \ .$$

In the following, we often describe algorithm in terms of bit-operations for the sake of clarity, but an efficient implementation should use word-operations so that several bit-operations can be performed with one machine instruction. We omit the details of conversion from bit-operations to word-operations.

3.1 Squaring

On a byte-addressable machine, the squaring operation can be performed eight bits at a time using a table lookup. We precompute a table of 256 16-bit integers representing the "squares" of the 256 possible 8-bit sequences (where the integers encode the coefficients of polynomials over GF(2)). Thus, squaring a polynomial of degree $r-1$ can be done with $\lceil r/8 \rceil$ table lookups.

We have found that, on some machines, the table lookup method is not the fastest, even if the table size is optimised to give the best results. An alternative on a machine with wordlength w bits is to perform squaring by a sequence of $\lceil 2r/w \rceil \log_2(w/2)$ iterations of the logical operations "shift right", "or" (written "$\vee$"), and "and" (written "$\wedge$").

[3] We identify the field element 0 with the bit 0 and the Boolean value "false", and the field element 1 with the bit 1 and the Boolean value "true".

We illustrate the idea with a small example ($r = 4$, $w = 8$). The masks $M_1 = 11001100$ and $M_0 = 10101010$ are precomputed, and "?" means "don't care".

$$\begin{array}{rcll}
\text{initial data} & & a_0a_1a_2a_3 & \\
\text{pad on right} & \to & a_0a_1a_2a_3\ 0\ \ 0\ \ 0\ \ 0 & \\
 & & & \\
\text{shift right 2} & \to & \ \ 0\ \ 0\ a_0a_1a_2a_3\ 0\ \ 0 & \Big\} \\
\vee \text{ last 2 words} & \to & a_0a_1\ ?\ \ ?\ a_2a_3\ ?\ \ ? & \Big\} \text{ iteration 1} \\
\wedge \text{ with mask } M_1 & \to & a_0a_1\ 0\ \ 0\ a_2a_3\ 0\ \ 0 & \Big\} \\
 & & & \\
\text{shift right 1} & \to & \ \ 0\ a_0a_1\ 0\ \ 0\ a_2a_3\ 0 & \Big\} \\
\vee \text{ last 2 words} & \to & a_0\ ?\ a_1\ ?\ a_2\ ?\ a_3\ ? & \Big\} \text{ iteration 2} \\
\wedge \text{ with mask } M_0 & \to & a_0\ 0\ a_1\ 0\ a_2\ 0\ a_3\ 0 & \Big\}
\end{array}$$

Squaring via logical operations

3.2 Reduction mod $P(x)$

Reduction of $A(x) = a_0 + a_1x + \cdots + a_dx^d \bmod P(x) = x^r + x^s + 1$ is performed by a sequence of "exclusive or" operations (written "$\oplus$"). We describe these as single-bit operations, but for efficiency they are implemented as w-bit-operations using word-length exclusive or operations (after appropriate shifting and masking).

While $d = \deg(A) \geq r$, we can replace

$$A(x) \ \text{ by } \ A(x) - (x^d + x^{d+s-r} + x^{d-r})$$

since

$$x^d + x^{d+s-r} + x^{d-r} = 0 \bmod P(x)\ .$$

In terms of bit-operations:

```
for d ← 2r − 2 downto r do
    begin
    a_{d−r} ← a_{d−r} ⊕ a_d;
    a_{d+s−r} ← a_{d+s−r} ⊕ a_d;
    a_d ← 0;   {This could be implicit}
    end.
```

Standard reduction algorithm

Overall, the reduction takes $2r + O(1)$ exclusive or bit-operations, or $2r/w + O(1)$ exclusive or word-operations.

Although complete descriptions are not always given in the original papers, previous computations involving the computation of irreducible/primitive trinomials (Kumada *et al.* [14], Heringa *et al.* [9], ...) seem to have used some variant of sieving combined with squaring and reduction, as described above. This is why we call it the *standard algorithm*. In §4 we describe an improvement which, although mathematically trivial, gives a significant reduction in computing time.

4 The New Algorithm

The standard algorithm is inefficient because many of the bit-operations are performed on bits which are necessarily zero. Our new algorithm avoids this. Before giving the algorithm in the general case, we illustrate with the example $r = 7, s = 3$.

We initialise $A(x) \leftarrow x$, i.e. $a_0 \dots a_6 \leftarrow 0100000$. The "squaring" operation is implicit: we keep the bit-vector 0100000 and regard this as representing

$$a_0 a_2 a_4 a_6 a_8 a_{10} a_{12}$$

(the odd-numbered coefficients $a_1, a_3, \dots$ in the square are necessarily zero, so need not be computed). We now reduce mod $P(x) = 1 + x^3 + x^7$. Observe that $x^{12} = x^5 + x^8 \bmod P(x)$, so we replace a_8 by $a_8 \oplus a_{12}$. We should also replace a_5 by $a_5 \oplus a_{12}$, but a_5 is currently zero, so we can simply regard the rightmost bit as representing a_5 rather than a_{12}. Thus, after the first step of the reduction we have a bit-vector representing

$$a_0 a_2 a_4 a_6 \underline{a_8} a_{10} a_5 \ ,$$

where the only bits which could have changed, because they depend on the result of an $\oplus$ operation, are underlined.

Proceeding in a similar fashion, we observe that $x^{10} = x^3 + x^6 \bmod P(x)$, but $a_3 = 0$, so we replace a_6 by $a_6 \oplus a_{10}$ and implicitly regard the second bit from the right as representing a_3 rather than a_{10}. Thus, after the reduction we have a bit-vector representing

$$a_0 a_2 a_4 \underline{a_6} a_8 a_3 a_5 \ .$$

One more step of reduction gives a bit-vector representing

$$a_0 a_2 \underline{a_4} a_6 a_1 a_3 a_5 \ .$$

Observe that this bit-vector contains the coefficients of $A(x)^2 \bmod P(x)$, but they are in a shuffled order. We need to apply an *interleave* permutation to get back to the natural order

$$a_0 a_1 a_2 a_3 a_4 a_5 a_6 \ .$$

Interleaving is closely related to the "squaring" operation described in §3.1. In fact, if we square $a_0 a_2 a_4 a_6 \rightarrow a_0 0 a_2 0 a_4 0 a_6 0$, square and rightshift $a_1 a_3 a_5 0 \rightarrow 0 a_1 0 a_3 0 a_5 00$, and apply a bitwise $\vee$ operation, we obtain $a_0 a_1 a_2 a_3 a_4 a_5 a_6 0$. Thus, interleaving can be implemented by squaring as in §3.1 and a few additional operations (shifting and $\vee$-ing). Although two squarings are necessary, the bit-vectors are only half as long as in §3.1, so the work involved is almost the same.

We give a complete example with $r = 7$, $s = 3$. The k-th operation of (implicitly) squaring and reducing mod $P(x)$ is denoted by S_k, and the k-th operation of interleaving by I_k. If we start with $A(x) = x$ and perform operations $S_1, I_1, S_2, I_2, \dots, S_7, I_7$ we obtain the following:

$$\begin{array}{llll}
S_1 \rightarrow 0100000, & I_1 \rightarrow 0010000 & \text{representing } x^2 \\
S_2 \rightarrow 0010000, & I_2 \rightarrow 0000100 & \text{representing } x^4 \\
S_3 \rightarrow 0010100, & I_3 \rightarrow 0100100 & \text{representing } x + x^4 \\
S_4 \rightarrow 0110100, & I_4 \rightarrow 0110100 & \text{representing } x + x^2 + x^4 \\
S_5 \rightarrow 0100100, & I_5 \rightarrow 0110000 & \text{representing } x + x^2 \\
S_6 \rightarrow 0110000, & I_6 \rightarrow 0010100 & \text{representing } x^2 + x^4 \\
S_7 \rightarrow 0000100, & I_7 \rightarrow 0100000 & \text{representing } x
\end{array}$$

Since the final result is x, we can deduce from Corollary 2 that $P(x) = 1 + x^3 + x^7$ is irreducible.

We now describe the new algorithm formally, in terms of bit-operations. It is necessary to assume that both r and s are odd. However, this is not a serious restriction. We already assumed that r is prime (and hence odd if $r > 2$). If s is even, we simply replace s by $r - s$, i.e. consider the reciprocal trinomial $x^r + x^{r-s} + 1$ instead of $x^r + x^s + 1$.

To avoid confusion, we denote the working bit-array by $b_0 b_1 \cdots b_{r-1}$. This bit-array is used to represent the coefficients $a_0 a_1 \cdots a_{r-1}$ of the polynomial $A(x)$, but not necessarily in the natural order. In a program, only the b-array is required.

Let $\alpha = (r-1)/2$ and $\delta = (r-s)/2$. Since r and s are odd, α and δ are integers. Initially we set $b_1 \leftarrow 1$ and the other $b_j \leftarrow 0$ to represent $A(x) = x$.

4.1 Implicit squaring and reduction

Each step S_k is implemented by

for $j \leftarrow r-1$ downto $\alpha + 1$ do
    $b_{j-\delta} \leftarrow b_{j-\delta} \oplus b_j$.

Squaring and reduction step S_k

Note that there are only $r/2 + O(1)$ "$\oplus$" bit-operations in the loop, which is a 75% reduction over the $2r + O(1)$ for the reduction step of the standard algorithm (§3.2).

4.2 Interleaving

The obvious implementation of the interleaving step I_k requires a temporary bit-array (say $c_0 c_1 \cdots c_{r-1}$). For example:

$c_0 \leftarrow b_0$;
for $j \leftarrow 1$ to α do {forward interleave}
    begin
    $c_{2j-1} \leftarrow b_{j+\alpha}$;
    $c_{2j} \leftarrow b_j$;
    end;
for $j \leftarrow 0$ to $r-1$ do $b_j \leftarrow c_j$.

Forward interleave I_k with copy

We call this a "forward interleave" because the first loop index j increases. We can avoid the final loop (copying the c array to b) by alternately using the array b and the array c (or by interchanging pointers appropriately). However, the space required is still $2r/w + O(1)$ words, so in terms of space requirements we have not yet improved on the standard algorithm.

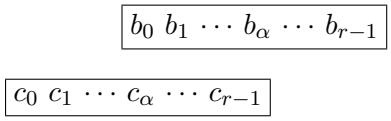
4.3 Combining steps to avoid copying and save space

We can interleave in the backward direction (replace "for $j \leftarrow 1$ to α" by "for $j \leftarrow \alpha$ downto 1" above). If we also interchange the roles of b and c to avoid the final copy, we obtain a program for steps $S_k, I_k, S_{k+1}, I_{k+1}$:

for $j \leftarrow r-1$ downto $\alpha+1$ do
    $b_{j-\delta} \leftarrow b_{j-\delta} \oplus b_j$.
$c_0 \leftarrow b_0$;
for $j \leftarrow 1$ to α do {forward interleave}
    begin
    $c_{2j-1} \leftarrow b_{j+\alpha}$;
    $c_{2j} \leftarrow b_j$;
    end;
for $j \leftarrow r-1$ downto $\alpha+1$ do
    $c_{j-\delta} \leftarrow c_{j-\delta} \oplus c_j$.
for $j \leftarrow \alpha$ downto 1 do {backward interleave}
    begin
    $b_{2j} \leftarrow c_j$;
    $b_{2j-1} \leftarrow c_{j+\alpha}$;
    end;
$b_0 \leftarrow c_0$

Combining steps $S_k, I_k, S_{k+1}, I_{k+1}$ with forward and backward interleaving

The point of interleaving alternately in the forward and backward directions is that we can save space by using a single working array of size $\frac{3r}{2w} + O(1)$ words. The b and c arrays can partially overlap – in fact b_j can occupy the same memory as $c_{j+\alpha}$ ($j = 0, 1, \ldots$), as shown:



Saving space by overlapping arrays

Note that the "forward interleave" transmits data from b to c (i.e. to the left) and the "backward interleave" transmits data from c to b (i.e. to the right)!

Partially overlapping the arrays b and c in this manner can improve performance dramatically on machines with memory hierarchies and cache sizes of less than $2r$ bits, because the working set is reduced in size by 25%. It has little effect on machines with much larger caches (see §6).

It is possible to perform interleaving using only $r/w + O(1)$ words, by splitting the required permutation into a product of cycles [12]. However, such an algorithm would be complicated and difficult to implement with word-operations, so we have not tried it.

To summarise, in comparison with the standard algorithm of §3, the new algorithm has 75% fewer $\oplus$ operations. Perhaps more significant than the number of operations is the number of memory references, which is reduced by 56%, from $8r/w + O(1)$ loads/stores to $\frac{7r}{2w} + O(1)$ loads/stores[4]. Also, the working set size is reduced by 25%, so memory references are more likely to be in the cache. In practice the improvement provided by the new algorithm depends on many factors: the values of r and (to a lesser extent) s, the cache size, the compiler and compiler options used, whether inner loops are written in assembler, etc, but it is generally at least a factor of two (see Table 2).

5 Effectiveness of Sieving

Suppose that we sieve by computing $\mathrm{GCD}(x^{2^n} + x, P(x))$ for $n = 1, 2, 3, \ldots$. Let f_n be the fraction of trinomials $P(x) = x^r + x^s + 1$ discarded at step n. Here, we are averaging over all s, $0 < s < r$, and then taking the limit (assumed to exist) as $r \to \infty$ through prime values. The fraction remaining after step n is

$$r_n = \prod_{j=1}^{n} (1 - f_j) \; .$$

It is easy to see that a trinomial is never discarded by the first sieve step ($n = 1$) because $P(x) = x^r + x^s + 1$ is not divisible by x or $1 + x$. (Thus, the first sieve step can be omitted.)

Consider the sieve with $n = 2$. There is precisely one irreducible polynomial of degree 2, namely $\Phi_{2,1}(x) = 1+x+x^2$. Note that $x^4 = x \bmod \Phi_{2,1}(x)$, so $x^r+x^s+1 \bmod \Phi_{2,1}(x)$ has period 3 as s varies with r fixed. Since r is a prime and we can assume $r > 3$, we have $r = \pm 1 \bmod 3$ and $x^r + x^s + 1$ is divisible by $\Phi_{2,1}(x)$ iff $r + s = 0 \bmod 3$. This occurs in one case out of three, so $f_2 = 1/3$. (The result is given in Swan [24, p. 1106].)

A generalization of this argument seems to be difficult, and we shall not attempt it here. The results of a program to evaluate f_n for $n \leq 11$ are given in Table 1 (columns headed "predicted") and are in good agreement with empirical data for $r = 859433$ (columns headed "observed"). The observed data for $r = 756839$ and $r = 3021377$ (incomplete) are similar.

In our program we estimate the time $T_s(n)$ for sieving at step $n > 2$ using the empirical approximation

$$T_s(n) \approx \sum_{j=1}^{n-1} T_s(j) \; ,$$

where the $T_s(j)$ values on the right hand side are known (because the sieving has already been performed). This approximation is plausible because the polynomials involved in the GCD are very sparse, but tend to fill in after a few steps of the Euclidean algorithm.

We also use the approximation $f_n \approx 1/n$ which is reasonable, in view of Table 1. The time $T_f(r)$ required for the full reducibility test of §4 can be estimated quite accurately by performing a small number of iterations of the square-reduce-interleave loop. If, using the estimated values of $T_s(n)$ and $T_f(r)$,

$$nT_s(n) < T_f(r) \; ,$$

then the expected benefit from sieving is greater than the expected cost, so we perform step n of the sieve. Otherwise we abandon sieving and perform the full reducibility test.

[4]More precisely, the standard algorithm loads $4r + O(1)$ bits and stores $4r + O(1)$ bits for one iteration of squaring and reduction; the new algorithm loads $2r + O(1)$ bits and stores $3r/2 + O(1)$ bits for one iteration of (implicit) squaring, reduction and interleaving; the exact number of word-level loads/stores depends on details of the implementations. In our Pentium implementation we use 64-bit loads/stores where possible, so long as these do not cross eight-byte boundaries.

Because of the dynamic criterion, the sieving cutoff varies with the trinomial being considered and the machine being used. For $r = 3021377$, in most cases we sieve for $n \leq 24$, and sieving takes about 8% of the overall time while eliminating about 93% of the trinomials from consideration. The remaining 7% of the trinomials require the full reducibility test, which takes about 92% of the total computing time.

n	$1/f_n$ predicted	$1/f_n$ predicted	$1/f_n$ observed	r_n predicted	r_n observed
2	3	3.0000	3.0000	0.6667	0.6667
3	7/2	3.5000	3.5000	0.4762	0.4762
4	5	5.0000	4.9999	0.3810	0.3809
5	31/6	5.1667	5.1666	0.3072	0.3072
6	15/2	7.5000	7.5001	0.2663	0.2663
7	127/18	7.0556	7.0574	0.2285	0.2285
8	34/3	11.333	11.345	0.2084	0.2084
9	949/118	8.0424	8.0325	0.1824	0.1824
10	275/26	10.577	10.596	0.1652	0.1652
11	2047/176	11.631	11.590	0.1510	0.1510
12			14.72		0.1407
13			12.80		0.1297
14			14.51		0.1208
15			15.59		0.1130
16			17.35		0.1065
17			18.13		0.1006
18			20.64		0.0958
19			19.82		0.0909
20			22.25		0.0868

Table 1: Predicted f_n and r_n, and observed values for $r = 859433$

6 Performance

We expect the running time of our program (excluding sieving) to be $T = 10^{-9}cr^2$ sec for a full reducibility test, where c is machine-dependent and approximately constant. In practice, because of cache effects, c is not independent of r. In Table 2 we give c for $r = 3021377$ on various machines. For IBM PCs (P-II and P-III) we give the size of the L2 cache. If the cache size is given as "large" this means that it is significantly larger than the working set size ($3r/2$ bits). Since $3r/2$ bits is 553KB, only slightly more than 512KB, the program performs much better on PCs with a 512KB cache than a 256KB cache. The program run on PCs had inner loops written in assembler[5], unless otherwise noted; for other machines the program was written purely in C (the times quoted are for the best compiler options, discovered by experiment).

processor	algorithm	compiler/assembler	cache size	c
300 Mhz P-II	standard	C code (gcc)	512KB	7.86
"	"	assembler (NASM)	"	6.31
"	new	C code (gcc)	"	3.54
"	"	assembler without overlap	"	2.60
"	"	assembler with overlap	"	1.64
400 Mhz P-II	new	assembler with overlap	512KB	1.24
500 Mhz P-III	"	"	"	0.77
833 Mhz P-III	"	"	256KB	1.66
300 Mhz Ultrasparc 10	"	C code (gcc)	large	2.90
195 Mhz SGI R10000	"	C code (cc -64 -Ofast)	"	1.80
300 Mhz SGI R12000	"	"	"	1.16
667 Mhz DEC Alpha	"	C code (gcc -O4)	"	0.60

Table 2: Normalised time to test reducibility, $c = \text{time(nsec)}/r^2$, $r = 3021377$

Table 2 is split into three sections:

1. We compare the standard and new algorithms (both C and assembler implementations) on the same machine. The version marked "without overlap" does not overlap arrays as described in §4.3, so its working set is about $2r$ bits instead of $3r/2$ bits. It avoids copying overhead by using two arrays alternately, as suggested at the end of §4.2.

2. We compare the new algorithm on different Pentium processors (note the dramatic influence of the cache size on performance – an 833 Mhz Pentium P-III with 256KB cache runs slower than a 300 Mhz Pentium P-II with a 512KB cache).

3. Finally we compare the C implementation on some other machines (Sparc, SGI R10000 and R12000, and DEC Alpha).

[5]The NASM [25] assembler routines use MMX instructions, which operate on 64-bit registers [10]. The speedup over pure C code is approximately a factor of two. This is due to the effective increase in wordlength from 32 to 64 bits, to careful use of the MMX registers to avoid inessential memory references and avoid alignment problems, to prefetching data well before it is needed, and to careful ordering of instructions to maximise the number of instructions executed per machine cycle [6].

In Table 3 we show the time for a full reducibility test with our new algorithm and various r on a machine (300 Mhz Pentium P-II) with 512KB L2 cache. The times given in the table do not include sieving, but this is relatively insignificant (e.g. sieving to $n = 24$ for $r = 3021377$ takes about 1200 seconds or 7.4% of the total time). The observed times depend to some extent on s and on whether other programs are running concurrently and competing for the cache.

r	time T (sec)	$c = 10^9 T/r^2$
19937	0.42	1.06
44497	2.10	1.06
110503	14.4	1.18
132049	21.7	1.24
756839	812	1.42
859433	1027	1.39
3021377	15010	1.64
6972593	198000	4.10

Table 3: Time to test reducibility on 300 Mhz P-II

Note that Table 3 gives the *worst case* time for testing reducibility, i.e. the time for the full reducibility test of §4. When testing many trinomials, the *average* time is less than 10% of the worst case time, because most of the trinomials are shown to be reducible by sieving (see §5) and do not require the full reducibility test.

7 Computational Results

In Table 5 we give a table of primitive trinomials $x^r + x^s + 1$ where r is a Mersenne exponent (i.e. $2^r - 1$ is prime). We assume that $0 < 2s \le r$ (so $x^r + x^{r-s} + 1$ is not listed). Entries with $r = \pm 3 \bmod 8$ are unlikely, since $s = 2$ (or $r - 2$) is then the only possibility, by Swan's theorem[6]. The only cases known are $r = 3$ and $r = 5$.

Early references are [8, 22, 23, 27]. The entries $r \le 11213$ are given by Zierler [30], and those for $11213 < r \le 216091$ are given by Heringa *et al.* [9]. We have confirmed these entries.

The entries for $r < 3021377$ have been checked by running at least two different programs on different machines. During this checking process, the entry with $r = 859433$, $s = 170340$ was found. This was surprising, because Kumada *et al.* [14] claimed to have searched the whole range for $r = 859433$. It turns out that Kumada *et al.* missed this entry because of a bug in the sieving routine [19].

The three entries for $r = 756839$ are new (Kumada *et al.* did not search for this r), as are the two entries for $r = 3021377$. At the time of writing, the search for $r = 3021377$ is incomplete[7] .

[6] See Swan [24, Corollary 5]. Swan's main result is due to Stickelberger (1897) – see [24, footnote on pg. 1099]. A generalization due to Blake *et al.* [2] is given in Menezes *et al.* [20, Fact 4.75].

[7] By 17 December 2000 we had searched about 60 percent of the range for $r = 3021377$. From the current rate of progress, we estimate that the search will be completed in April 2001. After completing the search for $r = 3021377$ we plan to continue with $r = 6972593$. For information on current status and results, see `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/trinom.html`

Since the average time for a single reducibility test is $O(r^2)$, the time required to test all trinomials of degree r is $O(r^3)$. Thus, ignoring the variability of c and the effect of different sieving cutoffs, we expect the search for $r = 3021377$ to take about 43 times as long as that for 859433, and the search for $r = 6972593$ to take about 12 times as long as that for $r = 3021377$.

There is a large gap between some of the Mersenne exponents r for which primitive trinomials exist. For example, in Table 5 there are no entries in the interval $132049 < r < 756839$. In Table 4 we give some irreducible trinomials to fill this gap. As usual, we only list $s \leq r/2$. The exponents r were chosen to be close to the arithmetic progression $10^5, 2 \times 10^5, 3 \times 10^5, \ldots$ with the constraints that:

1. r is prime.
2. $r = \pm 1 \bmod 8$.
3. $2^r - 1$ is composite, but no prime factors of $2^r - 1$ are known. Such factors are certainly larger than 2^{32} (see [7]).

Because of these constraints, we can be sure that the trinomials listed are irreducible. They are extremely likely to be primitive, but we can not prove this without knowing the factorisation of $2^r - 1$.

r	s	Search status
100151	4764, 15503	complete
200033	10175, 55224, 95397, 96236, 97575, 98763	complete
300073	–	complete
300151	49950, 87430	complete
400033	17865, 103623	complete
500231	4862, 10101, 203207, 205310	complete
600071	111503	in progress
700057	24829	in progress
800057	?	in progress
900217	?	in progress
1000121	?	in progress

Table 4: Some irreducible trinomials with prime exponent

8 Summary

In this preliminary report we have outlined our new algorithm for testing reducibility of trinomials over GF(2), and given our computational results up to the end of 2000. These include a correction to Kumada *et al.* [14] and two new primitive trinomials of record degree 3021377.

Acknowledgements

Some of the computations were performed by the third author on DEC alpha workstations at INRIA Lorraine. We thank the users of several workstations at INRIA Lorraine and Oxford University Computing Laboratory for making their idle time available. The Oxford Supercomputing Centre provided time to run the first author's programs on *Oscar*, an SGI Origin 2000, and *Tosca*, a PC cluster. Mark Rodenkirch contributed some CPU cycles, and Mike Yoder verified the new entries in Table 5 with his independently-written Ada program.

r	s	Notes
2	1	Only even r
3	1	Only known case $r = 3 \pmod 8$
5	2	Only known case $r = 5 \pmod 8$
7	1, 3	Watson [27]
13	–	Swan; Watson [27]
17	3, 5, 6	Watson [27]
19	–	Swan; Watson [27]
31	3, 6, 7, 13	Watson [27]
61	–	Swan; Watson [27]
89	38	Watson [27]
107	–	Swan; Stahnke [23]
127	1, 7, 15, 30, 63	Rodemich and Rumsey [22]
521	32, 48, 158, 168	Rodemich and Rumsey [22]
607	105, 147, 273	Rodemich and Rumsey [22]
1279	216, 418	Rodemich and Rumsey [22]
2203	–	Swan; Rodemich and Rumsey [22]
2281	715, 915, 1029	Zierler [30]
3217	67, 576	Zierler [30]
4253	–	Swan; Zierler [30]
4423	271, 369, 370, 649, 1393, 1419, 2098	Zierler [30]
9689	84, 471, 1836, 2444, 4187	Zierler [30]
9941	–	Swan; Zierler [30]
11213	–	Swan; Zierler [30]
19937	881, 7083, 9842	Kurita and Matsumoto [15]
21701	–	Swan; Kurita and Matsumoto [15]
23209	1530, 6619, 9739	Kurita and Matsumoto [15]
44497	8575, 21034	Kurita and Matsumoto [15]
86243	–	Swan; Kurita and Matsumoto [15]
110503	25230, 53719	Heringa *et al.* [9]
132049	7000, 33912, 41469, 52549, 54454	Heringa *et al.* [9]
216091	–	Swan; Heringa *et al.* [9]
756839	215747	Brent *et al.*, 14 June 2000
	267428	Brent *et al.*, 11 June 2000
	279695	Brent *et al.*, 9 June 2000
859433	170340	Brent *et al.*, 26 June 2000
	288477	Kumada *et al.* [14]
1257787	–	Swan; Kumada *et al.* [14]
1398269	–	Swan; Kumada *et al.* [14]
2976221	–	Swan; Kumada *et al.* [14]
3021377	361604	Brent *et al.*, 8 August 2000
	1010202	Brent *et al.*, 17 Dec 2000
	?	Search in progress
6972593	?	Not yet searched

Table 5: Primitive trinomials with Mersenne exponent

References

[1] S. L. Anderson, Random number generators on vector supercomputers and other advanced architectures, *SIAM Rev.* **32** (1990), 221–251.

[2] I. F. Blake, S. Gao and R. Lambert, Constructive problems for irreducible polynomials over finite fields, in *Information Theory and Applications*, *Lecture Notes in Computer Science* **793**, Springer-Verlag, Berlin, 1994, 1–23.

[3] R. P. Brent, Uniform random number generators for supercomputers, *Proc. Fifth Australian Supercomputer Conference*, Melbourne, December 1992, 95–104. `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub132.html`

[4] R. P. Brent, On the periods of generalized Fibonacci recurrences, *Math. Comp.* **63** (1994), 389–401. MR 94i:11012. `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub133.html`

[5] R. P. Brent, Random number generation and simulation on vector and parallel computers (extended abstract), *Proc. Fourth Euro-Par Conference*, *Lecture Notes in Computer Science* **1470**, Springer-Verlag, Berlin, 1998, 1–20. `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub185.html`

[6] A. Fog, *How to Optimize for Pentium Microprocessors*, version 2000-07-03, available from `http://www.agner.org/assem/` .

[7] GIMPS, The Great Internet Prime Search, `http://www.mersenne.org/`

[8] S. W. Golomb, *Shift register sequences,* Holden-Day, San Francisco, 1967. MR 39#3906.

[9] J. R. Heringa, H. W. J. Blöte and A. Compagner. New primitive trinomials of Mersenne-exponent degrees for random-number generation, *International J. of Modern Physics C* **3** (1992), 561–564.

[10] Intel Corporation, *MMX Technology Programmer's Reference Manual*. Available from `http://developer.intel.com` .

[11] F. James, A review of pseudorandom number generators, *Computer Physics Communications* **60** (1990), 329–344.

[12] A. Jones, Cycles of a simple permutation, *Australian Mathematical Society Gazette* **27**, 4 (October 2000), 158–160.

[13] D. E. Knuth, *The art of computer programming, Volume 2: Seminumerical algorithms* (third ed.), Addison-Wesley, Menlo Park, CA, 1997.

[14] T. Kumada, H. Leeb, Y. Kurita and M. Matsumoto, New primitive t-nomials ($t = 3$, 5) over GF(2) whose degree is a Mersenne exponent, *Math. Comp.* **69** (2000), 811–814.

[15] Y. Kurita and M. Matsumoto, Primitive t-nomials ($t = 3, 5$) over GF(2) whose degree is a Mersenne exponent ≤ 44497, *Math. Comp.* **56** (1991), 817–821.

[16] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their Applications*, Cambridge Univ. Press, Cambridge, second edition, 1994.

[17] G. Marsaglia, A current view of random number generators, in *Computer Science and Statistics: The Interface* (edited by L. Billard), Elsevier Science Publishers B. V. (North-Holland), 1985, 3–10.

[18] G. Marsaglia and L. H. Tsay, Matrices and the structure of random number sequences, *Linear Algebra Appl.* **67** (1985), 147–156.

[19] M. Matsumoto, Private communication by email, 17 July 2000.

[20] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997.

[21] J. F. Reiser, *Analysis of additive random number generators,* Ph. D. thesis, Department of Computer Science, Stanford University, Stanford, CA, 1977. Also Technical Report STAN-CS-77-601.

[22] E. R. Rodemich and H. Rumsey, Jr., Primitive trinomials of high degree, *Math. Comp.* **22** (1968), 863–865.

[23] W. Stahnke, Primitive binary polynomials, *Math. Comp.* **27** (1973), 977–980.

[24] R. G. Swan, Factorization of polynomials over finite fields, *Pacific J. Math.* **12** (1962), 1099–1106. MR 26#2432.

[25] S. Tatham and J. Hall, *NASM v0.98, the Netwide Assembler*, available from `http://www.web-sites.co.uk/nasm/docs/` .

[26] R. C. Tausworthe, Random numbers generated by linear recurrence modulo two, *Math. Comp.* **19** (1965), 201–209.

[27] E. J. Watson, Primitive polynomials (mod 2), *Math. Comp.* **16** (1962), 368–369. MR 26#5764.

[28] N. Zierler and J. Brillhart, On primitive trinomials (mod 2), *Inform. and Control* **13** (1968), 541–554. MR 38#5750.

[29] N. Zierler and J. Brillhart, On primitive trinomials (mod 2), II, *Inform. and Control* **14** (1969), 566–569. MR 39#5521.

[30] N. Zierler, Primitive trinomials whose degree is a Mersenne exponent, *Inform. and Control* **15** (1969), 67–69.