Primitive and Almost Primitive Trinomials over GF(2)*

Richard P. Brent
Computing Laboratory
University of Oxford

eccad03@rpbrent.co.uk

April 5, 2003

*East Coast Computer Algebra Day, Clemson, SC
Copyright ©2003, R. P. Brent. eccad03t

Abstract

We consider the problem of testing trinomials over GF(2) for irreducibility or primitivity. In particular, we consider trinomials whose degree r is a Mersenne exponent. We describe a new algorithm for testing primitivity of such trinomials. The algorithm has been used to find primitive trinomials of very high degree, e.g. $r = 6972593$.

For certain r, primitive trinomials of degree r do not exist. We show how to overcome this difficulty by finding *almost primitive* trinomials – polynomials with a primitive factor of degree r and overall degree slightly greater than r. In most applications, almost primitive trinomials are almost as useful as primitive trinomials.

Outline

- Introduction
- Definitions
- Sieving
- The standard algorithm
- The new algorithm
- Performance
- Some new primitive trinomials
- Almost primitive trinomials
- Implicit algorithms
- The Fermat connection
- Random number generators
- Acknowledgement
- Bibliography

Introduction

Irreducible and primitive polynomials over finite fields have many applications in cryptography, coding theory, random number generation etc. See, for example, the books by Golomb, Knuth (Vol. 2), and Menezes *et al.*

In this talk I will describe a new algorithm that has been used to find primitive polynomials of very high (in fact, "world record") degree over the field GF(2) of two elements.

The results can be generalised to *almost primitive* polynomials, which are (roughly speaking) polynomials with a large primitive factor.

Polynomials over GF(2)

GF(2) is just the set $\mathbf{Z}_2 = \{0, 1\}$ with operations of addition and multiplication modulo 2.

Equivalently, GF(2) is the set of Boolean values $\{F, T\}$ with operations $\oplus$ (exclusive or) and & (and).

We consider polynomials over GF(2), that is, polynomials whose coefficients are in GF(2). *For the sake of brevity, we won't repeat this statement every time!*

Note that, for polynomials u, v over GF(2),

$$2u = 2v = 0 \, .$$

This implies that $u - v = u + v$ and

$$(u + v)^2 = u^2 + v^2 \, .$$

Some Definitions

We say that a polynomial $P(x)$ is *reducible* if it has nontrivial factors; otherwise it is *irreducible*.

If $P(x)$ is irreducible of degree $r > 1$, then

$$\mathrm{GF}(2^r) \approx Z_2[x]/P(x) \, ,$$

so we have a representation of the finite field GF(2^r) with 2^r elements. If x is generator for the multiplicative group of $Z_2[x]/(P(x))$, then we say that $P(x)$ is *primitive*.

Since the multiplicative group has order $2^r - 1$, we need to know the complete factorisation of $2^r - 1$ in order to test if an irreducible polynomial is primitive. However, if r is a *Mersenne exponent*, i.e. $2^r - 1$ is prime, then irreducibility implies primitivity.

Some Well-Known Results

The following results can be found in texts such as Lidl, Menezes et al. Here μ is the Möbius function, and ϕ is Euler's phi function.

1. $x^{2^n} + x$ is the product of all irreducible polynomials of degree $d|n$. For example,

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

2. Let J_n be the number of irreducible polynomials of degree n. Then

   $$\sum_{d|n} dJ_d = 2^n \quad \text{and} \quad J_n = \frac{1}{n} \sum_{d|n} 2^d \mu(n/d) \, .$$

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

3. The number of primitive polynomials of degree n is $P_n = \phi(2^n - 1)/n \le J_n$.

   In particular, if n is a Mersenne exponent, then $P_n = J_n = (2^n - 2)/n$.

The Reciprocal Polynomial

If $P(x) = \sum_{j=0}^{r} a_j x^j$ is a polynomial of degree r, with $a_0 \neq 0$, then

$$P_R(x) = x^r P(1/x) = \sum_{j=0}^{r} a_j x^{r-j}$$

is the *reciprocal polynomial*. Clearly $P(x)$ is irreducible (or primitive) iff $P_R(x)$ is irreducible (or primitive).

In particular, if

$$P(x) = 1 + x^s + x^r \, , \quad 0 < s < r$$

is a trinomial, then the reciprocal trinomial is

$$P_R(x) = 1 + x^{r-s} + x^r \, .$$

If it is convenient, we can assume that $s \le r/2$ (else consider the reciprocal trinomial).

Similarly, if r is odd, we can assume that s is odd (this will be useful later).

Searching for Irreducible Polynomials

The irreducible polynomials (over GF(2), as usual) of degree r are analogous to primes with r digits. When searching for large primes we can quickly eliminate most candidates by sieving out multiples of small primes. Similarly, when searching for irreducible polynomials, we can eliminate candidates by checking if they are divisible by irreducible polynomials of low degree.

Sieving

Given a candidate $P(x)$, we check if $\mathrm{GCD}(P(x), x^{2^n} + x) \neq 1$, for $n = 1, 2, 3, \ldots, N$ where $N < \deg(P)$ is some bound. Since all irreducible polynomials of degree n are divisors of $x^{2^n} + x$, $P(x)$ passes these checks iff it has no irreducible factors of degree $\leq N$.

If $2^n > r = \deg(P)$, we can save time and space by computing $x^{2^n} \bmod P(x)$ by repeated squaring and reduction, before computing the GCD.

Irreducible Trinomials

For applications such as random number generation, we want irreducible (or better, primitive) polynomials of high degree r and with a small number of nonzero terms. Hence, we restrict attention to *trinomials* of the form

$$P(x) = P_{r,s}(x) = 1 + x^s + x^r \ , \quad 0 < s < r \ .$$

Swan's Theorem

Swan (1962) determines the parity of the number of irreducible factors by an argument involving the discriminant (actually, Swan's main result is due to Pellet (1878) and Stickelberger (1897)).

If r is an odd prime, then Swan's theorem implies that $P_{r,s}(x)$ has an even number of irreducible factors (and hence is reducible) if $r = \pm 3 \bmod 8$ and $s \neq 2$ or $r - 2$.

The condition on s can not be omitted, e.g. $x^{29} + x^2 + 1$ is irreducible.

Expectation of Success

For simplicity, assume r is an odd prime.

The probability that a randomly chosen polynomial of degree r is irreducible is of order $1/r$. Empirically, it seems that the same holds for trinomials of prime degree $r = \pm 1 \bmod 8$ (this condition implies that Swan's theorem does not rule out irreducibility).

Thus, if we consider all s in the range $0 < s < r/2$, we expect a small constant number c of irreducible trinomials of degree r. Empirical evidence suggests that $c \approx 3.2$.

For example, considering the 523 prime $r \in [1000, 10000]$ such that $r = \pm 1 \bmod 8$, we find exactly 1683 irreducible trinomials, giving an estimate $c = 3.22 \pm 0.08$.

Open Question. What is this constant c ?

Searching for Irreducible Trinomials

Suppose r is an odd prime, $r = \pm 1 \bmod 8$, and sieving has failed to show that $P(x) = P_{r,s}(x)$ is reducible. The *standard algorithm* computes

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

by r steps of squaring and reduction, then uses the result that $P(x)$ is irreducible iff

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

All authors of papers that give tables of irreducible trinomials (Watson, Rodemich, Zierler, Kurita, Heringa, Kumada, ...) seem to have used essentially this algorithm, which is why we call it the *standard algorithm*.

Complexity of the Standard Algorithm

Since we are working over GF(2),

$$\left(\sum_j a_j x^j\right)^2 = \sum_j a_j x^{2j} .$$

Thus, each squaring step takes $O(r)$ operations.

Each reduction step also takes $O(r)$ operations, since $P(x)$ is a trinomial and we can apply

$$x^{j+r} = x^{j+s} + x^j \bmod P(x)$$

for $j = r-2, r-3, \ldots, 0$ to reduce the result of squaring to a polynomial of degree less than r.

Thus, the complete test for reducibility of $P_{r,s}$ takes $O(r^2)$ operations, and to test all s takes $O(r^3)$ operations (assuming that sieving leaves a constant fraction of trinomials to test).

If the sieve limit is N and $2^N \approx r^c$ for some $c < 1$, then $N \approx c \log_2 r$ and we expect $O(r/N)$ trinomials to survive sieving, so the overall complexity might be reduced from $O(r^3)$ to $O(r^3/\log r)$.

Improving the Standard Algorithm

The standard algorithm uses $2r$ bits of memory for squaring, and $2r + O(1)$ $\oplus$ operations for each reduction (we count bit-operations but in practice we perform 32 or 64 bit-operations in parallel using word-operations; this also applies to our new algorithm). Many of these operations are on bits that are necessarily zero. There is a better algorithm that avoids these redundant operations.

Both algorithms represent a polynomial $A(x) = \sum_{j=0}^{r-1} a_j x^j$ as a bit-vector $a_0 \ldots a_{r-1}$.

Since r is odd and we can consider either $P_{r,s}$ or its reciprocal $P_{r,r-s}$, we can assume that s is odd.

The New Algorithm – Squaring

The first point is that there is no need to actually perform the squaring step! The standard algorithm would replace the bit vector

$$a_0 a_1 a_2 \ldots a_{r-2} a_{r-1}$$

by its "square"

$$a_0 0 a_1 0 a_2 \ldots 0 a_{r-2} 0 a_{r-1} .$$

However, we can simply keep the bit vector

$$a_0 a_1 a_2 \ldots a_{r-2} a_{r-1}$$

and regard it as *implicitly* representing a square (in other words, we do not store the coefficients of odd-degree terms, since they are known to be zero).

The New Algorithm – Reduction

To see how the reduction can be performed after our "implicit squaring", consider the example $r = 7, s = 3$.
We initialise $A(x) \leftarrow x$, i.e. $a_0 \ldots a_6 \leftarrow 0100000$.
The "squaring" operation is implicit: we keep the bit-vector 0100000 and regard this as representing

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

We now reduce mod $P(x) = 1 + x^3 + x^7$.
Observe that $x^{12} = x^5 + x^8 \bmod P(x)$, so we should replace a_8 by $a_8 \oplus a_{12}$ and 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

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

The only bit(s) that could have changed, because they depend on the result of an $\oplus$ operation, are underlined.

Example of Reduction continued

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

Interleaving is closely related to squaring. In fact, if we square $a_0 a_2 a_4 a_6$:

$$a_0 a_2 a_4 a_6 \to a_0 0 a_2 0 a_4 0 a_6 0 \ ,$$

square and rightshift $a_1 a_3 a_5 0$:

$$a_1 a_3 a_5 0 \to 0 a_1 0 a_3 0 a_5 0 0 \ ,$$

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 and a few additional operations (shifting and $\vee$-ing). Although two squarings are necessary, the bit-vectors are only half as long as before, so the work involved is almost the same.

A Complete Example

Consider the example $r = 7$, $s = 3$. 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 $S_1, I_1, S_2, I_2, \ldots, S_7, I_7$ we obtain the following:

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

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

The New Algorithm

We now describe the new algorithm formally, in terms of bit-operations. As before, assume that r and s are odd.

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.

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 $A(x) = x$.

Squaring and Reduction

Each step S_k is implemented by

$$\begin{array}{l}\text{for } j \leftarrow r-1 \text{ downto } \alpha+1 \text{ do}\\ \quad b_{j-\delta} \leftarrow b_{j-\delta} \oplus b_j.\end{array}$$

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.

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 $2r + O(1)$ bits, the same as for the standard algorithm.

A Refinement: Overlapping Arrays

We can interleave in the backward direction (replace "for $j \leftarrow 1$ to α" by "for $j \leftarrow \alpha$ downto 1" above). Suppose we also interchange the roles of b and c to avoid the final copy.

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 $3r/2 + O(1)$ bits. 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:

$$\boxed{b_0\ b_1 \cdots b_\alpha \cdots b_{r-1}}$$

$$\boxed{c_0\ c_1 \cdots c_\alpha \cdots c_{r-1}}$$

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)!

Effect of the Refinement

Partially overlapping the arrays b and c 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.

Generalisation of the New Algorithm

If we replace $1 + x^s + x^r$ by

$$1 + x^{s_1} + \cdots + x^{s_k} + x^r\ ,$$

then an obvious generalisation of our new algorithm is applicable *provided* that r is odd and the s_i all have the same parity (all odd or all even).

Comparison of the Algorithms

The new algorithm has 75% fewer $\oplus$ operations than the standard algorithm.

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, on a machine with wordlength w bits.

Also significant on some machines is that 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.

Performance of the New Algorithm

Table 1 gives normalised times for the standard and new algorithms on various processors, for $r = 3021377$. The third column is the "normalised time" $c = \text{time(nsec)}/r^2$.

processor	algorithm	c
300 Mhz P-II	standard	6.31
"	new (no overlap)	2.60
"	new (overlap)	1.64
500 Mhz P-III	"	0.77
833 Mhz P-III	"	1.66
300 Mhz SGI R12000	"	1.16
667 Mhz DEC Alpha	"	0.60

Table 1: Normalised time to test reducibility

Note that $3r/2$ bits is 553KB. The L2 cache size was 512KB on the P-II and P-III machines *except* only 256KB for the 833 Mhz P-III. The program was written in C, except that on PCs the inner loops were written in assembler to use the 64-bit MMX registers.

Times for Various Degrees

In Table 2 we show the time for a full reducibility test with our new algorithm and various degrees r on a machine (300 Mhz Pentium P-II) with 512KB L2 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 2: Time to test reducibility on a P-II

New Primitive Trinomials

In Table 3 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).

Results for $r < 756839$ are given by Heringa *et al.* [11]. We have confirmed these results.

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

$$r = 859433, \quad s = 170340$$

was found. This was surprising, because Kumada *et al.* [13] 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!

New Primitive Trinomials cont.

The entries in Table 3 are new, with the exception of one entry due to Kumada *et al.* The new entries except for the last one are from Brent, Larvala and Zimmermann (BLZ) [5].

r	s	Notes
756839	215747	BLZ, 14 June 2000
	267428	BLZ, 11 June 2000
	279695	BLZ, 9 June 2000
859433	170340	BLZ, 26 June 2000
	288477	Kumada *et al.* [13]
3021377	361604	BLZ, 8 August 2000
	1010202	BLZ, 17 Dec 2000
6972593	3037958	BLZ, 31 Aug 2002
		search 82% complete

Table 3: Primitive trinomials

"Almost Primitive" Trinomials

For about half the Mersenne exponents r (those with $r = \pm 3 \bmod 8$, $r > 5$), primitive trinomials of degree r probably do not exist. Examples are $r = 13, 19, 61, \ldots$.

In applications it would be almost as good to find trinomials of slightly higher degree, say $r + \delta$, having a primitive polynomial of degree r as a factor. Thus the period of the associated linear recurrence would be a small multiple of $2^r - 1$ (except for certain exceptional initial conditions that are easy to avoid). Blake, Gao and Lambert recently found some such trinomials of degree up to 500.

We can use a slight modification of our searching algorithm to find such trinomials. The sieving phase needs some modifications, and we have to allow the possibility of trinomials of even degree.

Definition of "Almost Primitive"

Let's make the concept "almost primitive" more precise.

Following Brent and Zimmermann [7], we say that a polynomial $P(x)$ of degree n is *almost primitive* if $P(0) \neq 0$ and $P(x)$ has a primitive factor of degree r, where $0 \leq n - r < r$.

We say that P has *exponent* r and *increment* $n - r$.

The restriction $\delta = n - r < r$ (equivalently, $r > n/2$) is convenient. In practice δ is usually a small constant.

Some examples are given in Table 4.

Some Almost Primitive Trinomials

For the values of r, δ and s given in Table 4,

$$x^{r+\delta} + x^s + 1$$

has a primitive factor of degree exactly r and period

$$(2^r - 1)f > 2^{r+\delta-1} .$$

The values of r are all the Mersenne exponents for which primitive trinomials of degree r do not exist, $107 < r < 2976221$.

r	δ	s	f
2203	3	355	7
4253	8	1806	255
9941	3	1077	7
11213	6	227	63
21701	3	6999	7
86243	2	2288	3
216091	12	42930	3937
1257787	3	74343	7
1398269	5	417719	21

Table 4: Some Almost Primitive Trinomials

Implicit Algorithms

Suppose we wish to work in the finite field GF(2^r) where r is the exponent of an almost primitive trinomial T. We can write

$$T = SD \ ,$$

where $\deg(S) = \delta$, $\deg(D) = r$. Roughly speaking, T is large and sparse, S is small, and D is large and dense. [We also know that D is primitive and $\gcd(S, D) = 1$, but this is not essential here.]

We have

$$GF(2^r) \equiv \mathbf{Z}_2[x]/D(x),$$

but because D is dense we wish to avoid working directly with D, or even explicitly computing D. We show that it is possible to work modulo the trinomial T.

Redundant Representations

We can regard $\mathbf{Z}_2[x]/T(x)$ as a redundant representation of $\mathbf{Z}_2[x]/D(x)$. Each element $A \in \mathbf{Z}_2[x]/T(x)$ can be represented as

$$A = A_c + A_d D,$$

where $A_c \in \mathbf{Z}_2[x]/D(x)$ is the "canonical representation" that would be obtained if we worked in $\mathbf{Z}_2[x]/D(x)$, and $A_d \in \mathbf{Z}_2[x]$ is some polynomial of degree less than δ. [In fact, A_c is the remainder, and A_d is the quotient, when A is divided by D, but we don't want to perform divisions by the large, dense polynomial D.]

$$\mathbf{Z}_2[x] \to \mathbf{Z}_2[x]/T(x) \to \mathbf{Z}_2/D(x)$$

Taking Advantage of Sparsity

We can perform computations such as addition, multiplication and exponentiation in $\mathbf{Z}_2[x]/T(x)$, taking advantage of the sparsity of T in the usual way.

If $A \in \mathbf{Z}_2[x]/T(x)$ and we wish to map A to its canonical representation A_c, we use the identity

$$A_c = (AS \bmod T)/S,$$

where the division by the (small) polynomial S is exact. A straightforward implementation requires only $O(\delta r)$ operations.

We avoid computing $A_c = A \bmod D$ directly; in fact we never compute the (large and dense) polynomial D: it is sufficient that D is determined by the trinomial T and the small polynomial S.

Example

An entry in Table 4 with $r = 1257787$, $\delta = 3$, gives

$$\begin{aligned} T(x) &= x^{1257790} + x^{74343} + 1 \\ &= (x^3 + x^2 + 1)D(x) \ , \end{aligned}$$

where $D(x)$ is a dense primitive polynomial of degree 1257787.

We can work in the field GF($2^{1257787}$) without ever computing or dividing by $D(x)$. All we need is operations mod $T(x)$ and multiplications/divisions by the small polynomial $S(x) = x^3 + x^2 + 1$.

The Fermat Connection

Let $F_j = 2^{2^j} + 1$ be the j-th Fermat number. If $r = 2^k$, then

$$2^r - 1 = F_0 F_1 \cdots F_{k-1} \ .$$

For example,

$$2^{2^3} - 1 = F_0 F_1 F_2 = 3 \cdot 5 \cdot 17 \ .$$

The complete factorizations of F_j are known for $j \leq 11$, so we can factor $2^{2^k} - 1$ for $k \leq 12$.

By Swan's theorem, a primitive trinomial of degree 2^k does not exist for $k \geq 3$. However, we can work efficiently in the finite fields $\mathrm{GF}(2^{2^k})$, $k \in [3, 12]$, using almost primitive trinomials and implicit algorithms.

For example, take $k = 12$, $r = 2^k = 4096$, $\delta = 3$, $s = 600$. The trinomial

$$T(x) = x^{4099} + x^{600} + 1$$

is almost primitive, with a small factor $x^3 + x + 1$ and a large primitive factor of degree 4096.

Existence Questions - Given Degree

It seems difficult to *prove* anything about the existence of almost primitive trinomials. We have shown by computation that an almost primitive trinomial of degree n exists for all $n \in [2, 2000] \backslash \{12\}$.

A probabilistic argument suggests that almost primitive trinomials exist for all degrees $n > 12$.

In the exceptional case (degree 12),

$$x^{12}+x+1 = (x^3+x^2+1)(x^4+x^3+1)(x^5+x^3+x^2+x+1)$$

has primitive factors of degrees 3, 4, and 5, but 5 is too small, so $x^{12} + x + 1$ is not "almost primitive" by our definition.

Another candidate is $x^{12} + x^5 + 1$; this is irreducible but not primitive, having period $(2^{12} - 1)/5$.

Existence Questions - Given Exponent

Instead of asking for an almost primitive trinomial of given degree, we can ask for one of given exponent. This is closer to the spirit of Blake et al.

For all $r \in [2, 712]$ there is an almost primitive trinomial $x^{r+\delta} + x^s + 1$ with exponent r and (minimal) increment $\delta \leq 43$. The extreme $\delta = 43$ occurs for $(r, s) = (544, 47)$. The mean value of δ is less than 4.

We can not go any further without knowing the factors of $2^{713} - 1$. However, a probabilistic argument suggests that almost primitive trinomials exist for all exponents $r \geq 2$.

Random Number Generators

Pseudo-random number generators (RNGs) are widely used in simulation.

A program running on a fast computer or cluster of PCs might use 10^9 random numbers per second for many hours. Small correlations or other deficiencies could easily lead to spurious results.

In order to have confidence in the results of simulations, we need to have confidence in the statistical properties of the random numbers used.

Primitive and almost primitive trinomials are useful for the construction of high quality random number generators, because the generating function for a 3-term linear recurrence corresponds to a trinomial.

Almost Primitive Trinomials in RNGs

If the almost primitive trinomial $T = x^n + x^s + 1 = SD$ is the denominator of the generating function for a linear recurrence $u_k = u_{k-s} + u_{k-n}$, it is possible (by choosing appropriate initial conditions that annihilate the unwanted component) to generate a sequence that satisfies the recurrence defined by the polynomial D.

This refinement is not necessary if all that matters is that the linear recurrence generates a sequence with period at least $2^r - 1$, where $r = \deg(D)$.

Acknowledgement

This talk describes joint work with Samuli Larvala (HUT) and Paul Zimmermann (INRIA Lorraine).

Thanks are also due to Nate Begeman, Nicolas Daminelli, Shuhong Gao, Robert Hedges, Brendan McKay, Barry Mead, Mark Rodenkirch, Juan Varona, and Mike Yoder for their assistance in various ways.

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, Construction and distribution problems for irreducible trinomials over finite fields, preprint, July 2001.

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

[4] R. P. Brent, Random number generation and simulation on vector and parallel computers, *Proc. Fourth Euro-Par Conference*, *LNCS* **1470**, Springer-Verlag, Berlin, 1998, 1–20. `···/pub185.html`

[5] R. P. Brent, S. Larvala and P. Zimmermann, A fast algorithm for testing reducibility of trinomials mod 2 and some new primitive trinomials of degree 3021377, *Math. Comp.*, to appear. Preprint and update at `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub199.html`

[6] R. P. Brent and P. Zimmermann, Random number generators with period divisible by a Mersenne prime, *Proc. ICCSA 2003*, Montreal, May 2003, to appear in *LNCS*. Preprint at `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub211.html`

[7] R. P. Brent and P. Zimmermann, Algorithms for finding almost irreducible and almost primitive trinomials, *Proc. Conference in Honour of Professor H. C. Williams*, Banff, Canada (May 2003), The Fields Institute, Toronto, to appear. Preprint available at `http://www.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub212.html`

[8] D. Coppersmith, H. Krawczyk and Y. Mansour, The shrinking generator, *Proc. CRYPTO'93, LNCS* **773** (1994), 22–39.

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

[10] S. W. Golomb, *Shift register sequences,* Aegean Park Press, revised edition, 1982.

[11] 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.

[12] D. E. Knuth, *The art of computer programming, Volume 2: Seminumerical algorithms* (third ed.), Addison-Wesley, Menlo Park, CA, 1998.

[13] 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.

[14] 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.

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

[16] G. Marsaglia, A current view of random number generators, in *Computer Science and Statistics: The Interface*, Elsevier Science Publishers B. V.,1985, 3–10.

[17] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997. `http://cacr.math.uwaterloo.ca/hac/`

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

[19] S. Tezuka, Efficient and portable combined Tausworthe random number generators, *ACM Trans. on Modeling and Computer Simulation* **1** (1991), 99–112.

[20] I. Vattulainen, T. Ala-Nissila and K. Kankaala, Physical tests for random numbers in simulations, *Phys. Rev. Lett.* **73** (1994), 2513–2516.

[21] N. Zierler and J. Brillhart, On primitive trinomials (mod 2), *Inform. and Control* **13** (1968), 541–554 and **14** (1969), 566–569.

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