Center for Reliable Computing

TECHNICAL REPORT

Primitive Polynomial Generation Algorithms Implementation and Performance Analysis

Nirmal R. Saxena and Edward J. McCluskey

04-03
(CRC TR 04-03)
April 2004

Center for Reliable Computing
Gates Bldg. 2A, Room #236
Stanford University
Stanford, California 94305-9020

Abstract:

Performance analysis of two algorithms, *MatrixPower* and *FactorPower*, that generate all $\varphi(2^r-1)/r$ degree-r primitive polynomials (φ is the Euler's totient function) is presented. MatrixPower generates each new degree-r primitive polynomial in $O(r^4) \sim O(r^4 \ln r)$ time. FactorPower generates each new degree-r primitive polynomial in $O(r^4) \sim O(k\, r^4 \ln r)$ time, where k is the number of distinct prime factors of 2^r-1. Both MatrixPower and FactorPower require $O(r^2)$ storage. Complexity analysis of generating primitive polynomials is presented. This work augments previously published list of primitive polynomials and provides a fast computational framework to search for primitive polynomials with special properties.

Keywords: *primitive polynomials*, *cryptography, key generation, error-correcting codes, built-in testing, cyclic redundancy checks* and *signature analysis*.

Funding:

This work has been supported by the Advanced Research Projects Agency under prime contract No. *DABT63-94-C-0045.

Imprimatur: Ahmad A. Al-Yamani

Primitive Polynomial Generation Algorithms–– Implementation and Performance Analysis

Nirmal R. Saxena & Edward J. McCluskey

CRC Techical Report No. 04-03
April 2004

CENTER FOR RELIABLE COMPUTING
Gates Bldg. 2A, Room #236
Computer System Laboratory
Department of Electrical Engineering
Stanford University
Stanford, California 94305-9020

ABSTRACT

Performance analysis of two algorithms, *MatrixPower* and *FactorPower*, that generate all $\varphi(2^r-1)/r$ degree-r primitive polynomials (φ is the Euler's totient function) is presented. MatrixPower generates each new degree-r primitive polynomial in $O(r^4) \sim O(r^4 \ln r)$ time. FactorPower generates each new degree-r primitive polynomial in $O(r^4) \sim O(k\, r^4 \ln r)$ time, where k is the number of distinct prime factors of 2^r-1. Both MatrixPower and FactorPower require $O(r^2)$ storage. Complexity analysis of generating primitive polynomials is presented. This work augments previously published list of primitive polynomials and provides a fast computational framework to search for primitive polynomials with special properties.

Keywords: *primitive polynomials*, *cryptography*, *key generation*, *error-correcting codes*, *built-in testing*, *cyclic redundancy checks* and *signature analysis*.

Table of Contents

List of Tables

List of Figures

1.0 Introduction

Linear feedback shift registers (LFSR) derived from primitive polynomials have been used for: random bit sequence generation [GOL67], cryptographic applications like stream ciphers [BEK82][SCH96][RSA94], error correction and detection codes [PET84][BER84][WIC95], and built-in testing for VLSI circuits [BAR87][SAX92][SAX97]. A degree-r polynomial is primitive if and only if its period is 2^r-1. The *period* of a polynomial is the maximum period realized by its corresponding LFSR implementation. Figure 1 illustrates that the polynomial x^4+x+1 is primitive. Like prime numbers, primitive polynomials have the property that they cannot be factored into smaller degree polynomials in the defining field. Polynomials that cannot be factored into smaller degree polynomials are called *irreducible*. All primitive polynomials are irreducible. However, all irreducible polynomials are not primitive. For example, $x^4+x^3+x^2+x+1$ is an irreducible polynomial but is not primitive (see Figure 2). If 2^r-1 is a prime number (Mersenne prime) then all degree-r irreducible polynomials are primitive [GOL67].

A partial list of primitive polynomials up to degree 828 appeared in [ZIE68][ZIE69]. This list has been augmented by primitive polynomials up to degree 500 in [BAR90][BAR92]. This augmented list was based on the Cunningham Project [BRI88] that involved the factorization of 2^r-1.

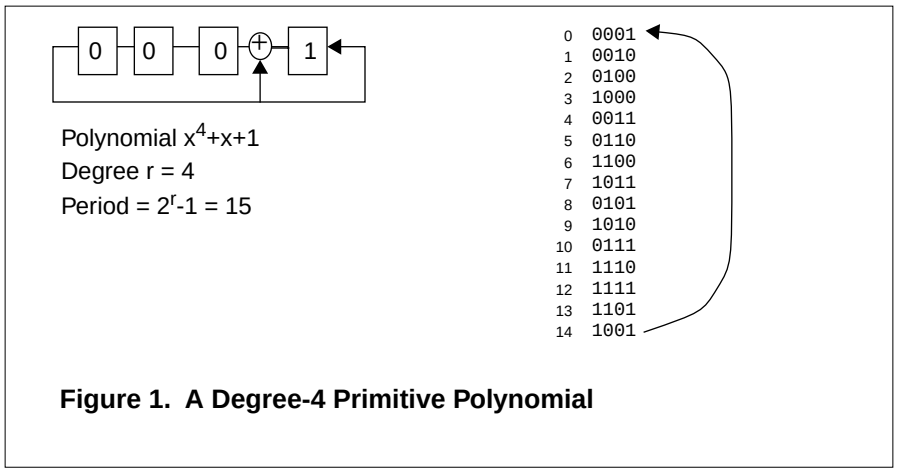

Figure 1. A Degree-4 Primitive Polynomial

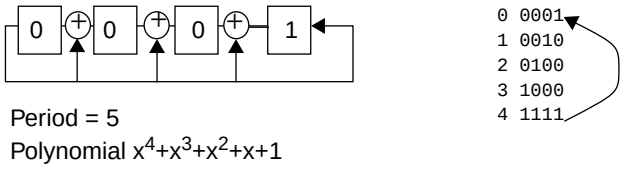

Figure 2. A Non-Primitive Degree-4 Irreducible Polynomial

In [ZIE69], a systematic procedure that generates all degree-*r* primitive trinomials, given a known factorization of 2^r-1, is presented. Trinomials are interesting in that they require only one feedback connection with the exclusive-or gate. However, primitive trinomials do not exist for all degrees. For example, there are no primitive trinomials when *r* is a multiple of 8 [GOL67][SWA62]. The published partial list [ZIE68][ZIE69][BAR90][BAR92] of primitive polynomials in most application situations may seem adequate. For example, results in [SAX92][SAX97] show that by using primitive polynomials tighter bounds on signature analysis aliasing probability are achieved. These bounds hold good as long as the polynomial is primitive and is independent of the type of primitive polynomial used. Also, in some applications (like built-in self test) if the only requirement is that the LFSR generate all possible non-zero patterns then it is not important as to what type of primitive polynomial is used. However, in situations where applications demand a greater choice in the selection of primitive polynomials this partial list may not be adequate. Some applications may require a choice in the selection of primitive polynomials. These applications are:

- *Cryptography*: Some stream ciphers like shift register cascade (e.g., Gollmann Cascade [SCH96][RSA94]) appear to have good security properties. A *shift register cascade* is a set of LFSRs connected together in such a way that the behavior of one particular LFSR depends on the behavior of the previous LFSRs in the cascade. This dependent behavior is usually achieved by using one LFSR to control the clock (or the enable signal) of the following LFSR. The parameters that define the behavior of these stream ciphers are the initial contents and the defining primitive polynomials of the LFSRs. For a given stream cipher architecture, we can think of initial contents and the defining polynomials of the LFSRs as keys. Having algorithms that pick different primitive polynomials for the LFSRs could contribute to increased security of the stream ciphers. For example, the Hughes XPD/KPD stream cipher algorithm [SCH96] uses a 61-bit LFSR. There are about 1024 different primitive polynomials (approved by NSA) stored in a table that are key selected for the XPD/KPD.

  There are in fact $\frac{\varphi(2^{61}-1)}{61}$ = 37, 800, 705, 069, 076, 950 degree-61 primitive polynomials not just 1024. The results in this work can algorithmically generate any arbitrary key selected

61-bit primitive polynomial (without the need for any table) in polynomial time. In order to search for more secure stream cipher implementations a greater choice in the selection of primitive polynomials is clearly a requirement.

- *Error Detecting*/*Correcting Codes*: Hamming and BCH codes can be constructed using primitive polynomials [PET84][BER84][WIC95]. In order to search for efficient implementations of these codes (in terms of area, cycle time, and physical design complexity) it is desirable to index through all possible choices of the code-defining primitive polynomials.
- *Built-In Self Test*: Implementation requirements and physical design constraints may limit the allowable sites for feedback connections for some LFSR stages. In these situations, it may be desirable to search for primitive polynomials that meet the requirements imposed by implementation and physical design constraints.

1.1 Primitive Polynomial Generation Algorithms: A Summary First

This section gives the reader a feel for the complexity associated with the generation of primitive polynomials. Section 2.0 and Section 3.0 present the actual analysis and description of the various algorithms. An algorithm (by Haluk Aydinoglu) that generates all possible primitive polynomials by exhaustively verifying the period property appears in [WIC95]. For reference, we will call this algorithm *PeriodA*. Similar to PeriodA, another unpublished algorithm due to the first author (we will call it *PeriodS* for reference) exhaustively enumerates all *feedback polynomials*† with their respective periods. PeriodS has been used for signature analysis related work in [SAX92]. In this work two algorithms, *FactorPower* and *MatrixPower*, are derived that generate all primitive polynomials. FactorPower is a generalization of the algorithm used in [ZIE69]. FactorPower requires the knowledge of factorization of 2^r-1. MatrixPower uses the isomorphism between primitive root powers, α^t, and powers of matrix, A^t, to generate all primitive polynomials. Matrix A represents the primitive polynomial whose root is α. MatrixPower requires as an input a single degree-r primitive polynomial to generate all degree-r primitive polynomials. Techniques of generating all indexing primitive polynoimials (such as used in MatrixPower) from a known primitive polynomial are fairly well-known [ALA64].

The average time to generate every new primitive polynomial of degree-r quantifies the performance of these algorithms. The unit for this average time is *Secs*/*Poly* and is calculated by the total time in seconds to generate all degree-r primitive polynomials over the total number of degree-r primitive polynomials. PeriodA, PeriodS, FactorPower, and MatrixPower were all coded in C/C++ language to study their relative performance in terms of the Secs/Poly metric. Table 1 shows the computation times to generate all primitive polynomials for degrees 10 through 20. The columns indicating *TotalSecs* show the time in seconds to generate the entire list of

†. In this paper we define feedback polynomials as those that are monic and have +1 term. In other words, a feedback polynomial is a monic polynomial without X as a factor.

polynomials for a given degree by the respective algorithms. Likewise, the columns indicating *Secs*/*Poly* show the average time in seconds to generate single primitive polynomials. Table 1 illustrates the distinguishing features of these algorithms in terms of computational complexity and relative performance. Section 2.0 and Section 3.0 present both the actual description and the derivation of computational complexity for the respective algorithms.

TABLE 1. Execution Time[a] Comparison

		PeriodA		PeriodS		FactorPower		MatrixPower	
r	$\varphi(2^r-1)/r$	TotalSecs	Secs/Poly	TotalSecs	Secs/Poly	TotalSecs	Secs/Poly	TotalSecs	Secs/Poly
10	60	0.17	0.003	0.04	0.0007	0.21	0.0035	0.05	0.0008
11	176	0.97	0.006	0.17	0.0010	0.36	0.0021	0.10	0.0006
12	144	2.31	0.016	0.42	0.0029	1.59	0.0110	0.13	0.0009
13	630	16.94	0.027	2.08	0.0033	0.16	0.00025	0.40	0.0006
14	756	50.05	0.066	7.02	0.0093	7.24	0.0096	0.84	0.0011
15	1800	216.17	0.120	26.57	0.0148	17.80	0.0099	2.05	0.0011
16	2048	897.43	0.438	102.66	0.0501	54.11	0.0264	4.43	0.0021
17	7710	4681.89	0.607	467.27	0.0606	3.50	0.0005	11.64	0.0015
18	7776	13643.91	1.755	1508.71	0.1940	299.40	0.0385	24.03	0.0039
19	27594			7185.75	0.2604	17.51	0.0006	61.45	0.0022
20	24000			22648.66	0.9437	1892.79	0.0789	124.79	0.0052

a. The computations were done on an upgraded Power Macintosh 8500/120 system (PowerPC 604e 233 Mhz processor upgrade card) with 176 megabytes of main memory. The programs were written in C/C++ and compiled using gcc -O2 optimizations. The operating system environment was MkLinux, Developer Release 2.1, Update 5.

The most practical aspect of FactorPower and MatrixPower is that every new and distinct primitive polynomial is generated enumeratively in polynomial-time ($O(r^4) \sim O(r^4 \ln r)$). This is in contrast to PeriodA and PeriodS that generate every new and distinct primitive polynomial in exponential-time (at least $O(r2^r)$).

PeriodA inefficiently simulates the LFSR and recursively indexes through the polynomials and therefore it is about eight to ten times slower than PeriodS. PeriodS for degrees up to 14 is faster than FactorPower and for degrees up to 10 is faster than MatrixPower. It is not surprising that for small values of *r* the exhaustive algorithm is more efficient than the polynomial-time algorithms FactorPower and MatrixPower. The listings of PeriodS and PeriodA programs appear in Appendix (Figure 13 and Figure 14).The code for PeriodS is extremely compact and almost requires O(1) storage complexity; whereas, both MatrixPower and FactorPower have larger code and require at least $O(r^2)$ storage. Therefore for small values of *r,* the disadvantage of bulky data structures in MatrixPower and FactorPower more than offset their polynomial-time advantages over PeriodS.

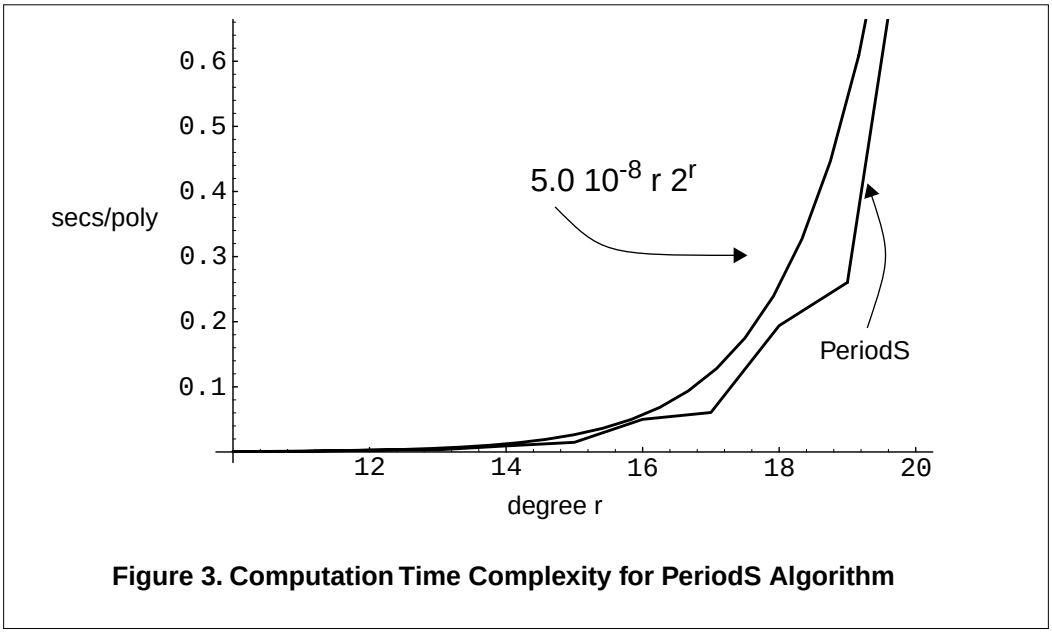

Figure 3. Computation Time Complexity for PeriodS Algorithm

The plot for Table 1 data in Figure 3 shows that PeriodS execution time follows $O(r2^r)$ computation time complexity. PeriodS becomes impractical for large degrees. For example, PeriodS takes 24 minutes to generate every new degree-30 primitive polynomial whereas MatrixPower takes only 0.026 secs to generate every new degree-30 primitive polynomial.

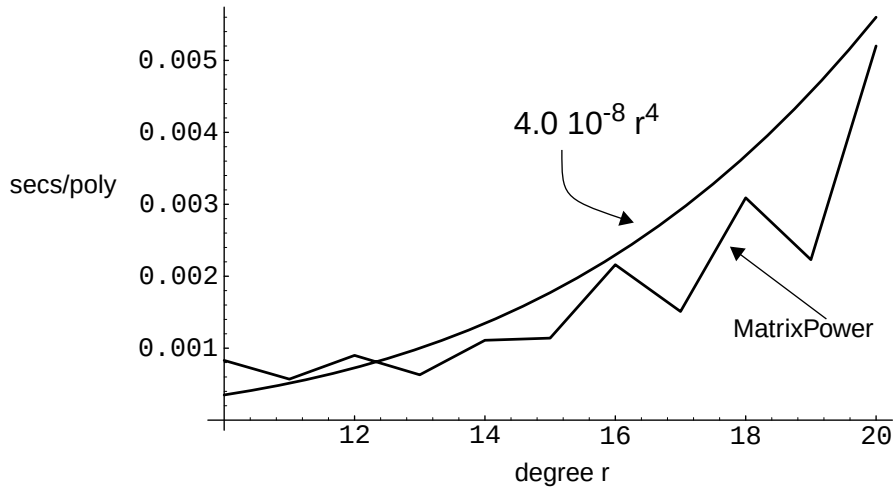

Figure 4. Computation Time Complexity for MatrixPower Algorithm

Figure 4 shows the $O(r^4)$ execution time complexity of MatrixPower. In general, for odd r the primitive polynomial generation algorithms are effectively faster. This is because for odd r, relative to 2^r there are effectively more degree-r primitive polynomials. In other words, primitive

polynomials are denser for odd *r*. For example, the probability of finding a degree-11 primitive polynomial by a random search over all degree-11 (there are 1024) feedback polynomials is about 17% (176/1024, there are 176 degree-11 primitive polynomials); whereas, the probability of finding a degree-12 primitive polynomial by a random search of all degree-12 feedback polynomials (there are 2048) is only 7% (144/2048, there are 144 degree-12 primitive polynomials).

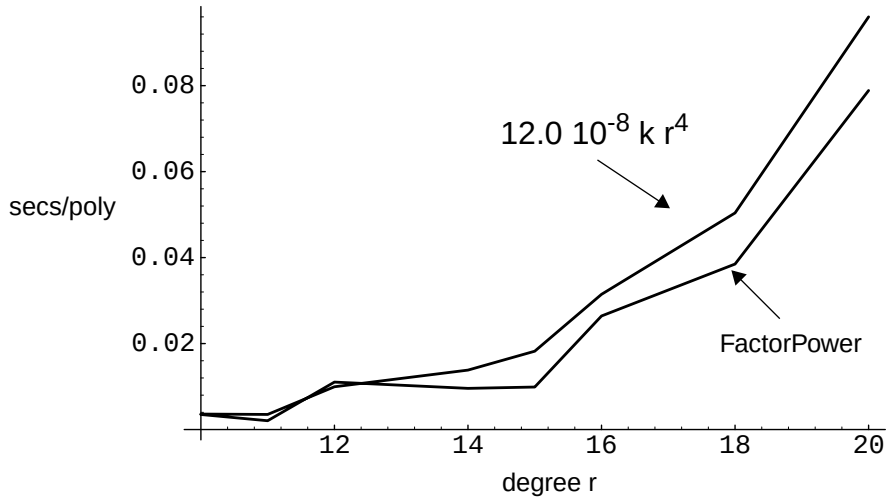

Figure 5. Computation Time Complexity for FactorPower Algorithm

FactorPower is fastest when 2^r-1 is prime. This is because all irreducible polynomials are primitive when 2^r-1 is prime. The complexity of verifying that a polynomial is irreducible is at most $O(r^3)$ [KNU97]. For Mersenne prime points, r=13, 17, and 19 (in Table 1) FactorPower is significantly faster than MatrixPower. In general, FactorPower computation complexity is $O(k\, r^4)$ (Figure 5), where k is the number of distinct prime factors of 2^r-1. A detailed list of factorization of 2^r-1 appears in [BRI88]. Listing of k for r=10 through 24 is shown in Table 2.

TABLE 2. Distinct Prime Factors of 2^r-1

r	Distinct Factors of 2^r-1	*k*
10	3, 11, 31	3
11	23, 89	2
12	3, 5, 7, 13	4
13	8191	1
14	3, 43, 127	3
15	7, 31, 151	3
16	3, 5, 17, 257	4
17	131071	1
18	3, 7, 19, 73	4
19	524287	1
20	3, 5, 11, 31, 41	5
21	7, 127, 337	3
22	3, 23, 89, 683	4
23	47, 178481	2
24	3, 5, 7, 13, 17, 241	6

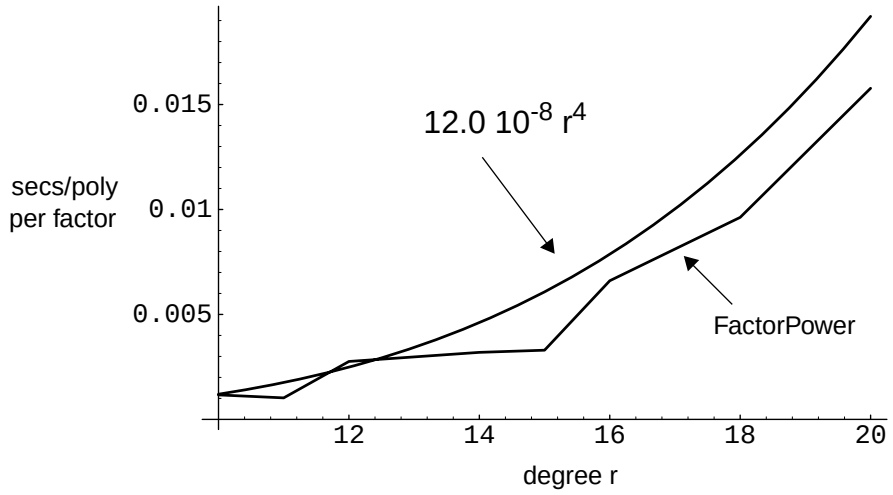

Figure 6. FactorPower Algorithm Performance Normalized Per Factor

FactorPower complexity normalized by *k* is shown in Figure 6. Plots in Figure 5 and Figure 6 exclude the Mersenne prime points, *r*=13, 17, and 19. In general, MatrixPower is faster than FactorPower. However, FactorPower can be used to do generate primitive polynomials with special properties. This is discussed in Section 4. Section 3 presents the description of PeriodA,

PeriodS, FactorPower, and MatrixPower. Section 2 presents some asymptotic results that are necessary to establish the computational complexity of the various primitive polynomial generation algorithms.

2.0 Primitive Polynomial Generation Complexity Analysis: Preliminaries

There are 2^{r-1} degree-r feedback polynomials. There are $\lambda(r) = \varphi(2^r-1)/r$ degree-r primitive polynomials. If the degree-r feedback polynomials are indexed in some order then the average number of steps required to find the next degree-r primitive polynomial is given by

$$\frac{r2^{r-1}}{\varphi(2^r-1)} \tag{1}$$

The expression (1) is the reciprocal of the probability of finding a degree-r primitive polynomial from an indexing set of degree-r feedback polynomials. In order to bound the complexity of primitive polynomial generation algorithms, an upper bound on the above expression is useful. In other words, we need a lower bound on $\varphi(2^r-1)$. Results from number theory are used in deriving and estimating these bounds.

In [HAR85], it is shown that

$$\liminf_{n \to \infty} \varphi(n) = \frac{ne^{-\gamma}}{\ln(\ln(n))} \qquad \gamma = 0.577... \text{ is Euler's constant} \tag{2}$$

An upper bound on the average number of steps to find the next t such that $t \perp n$.

$$\limsup_{n \to \infty} \frac{n}{\varphi(n)} = e^{\gamma} \ln(\ln(n)) = 1.781... \ln(\ln(n)) \tag{3}$$

For certain classes of n (for example $n = m!$) this upper bound seems reasonable. For example, Figure 7 shows the actual value of $n/\varphi(n)$ and its corresponding upper bound for $n=m!$.

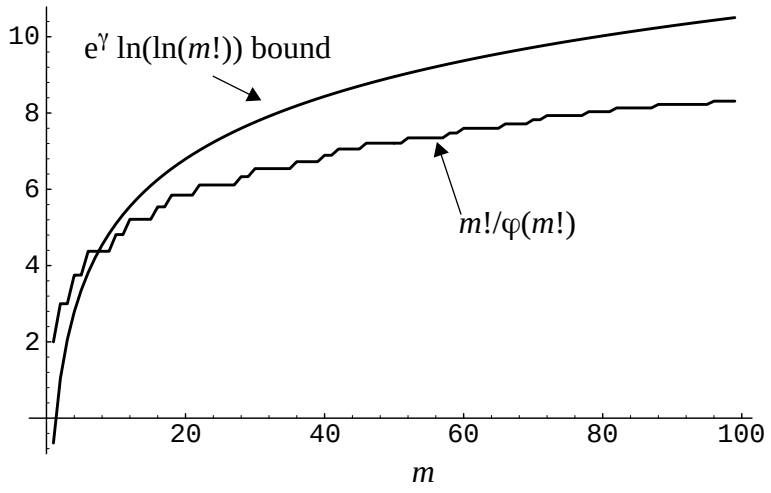

Figure 7. Bound on Average Number of Steps for *n* = *m*!

While the bound appears to be tight for values of $n = m!$, our interest is in values of $n=2^r-1$.

The average number of steps required to find the next relatively prime t such that $t \perp 2^r-1$ is:

$$\frac{2^r-1}{\varphi(2^r-1)} \tag{4}$$

Using the foregoing results, an upper bound (asymptotic) on average steps to relatively prime t is

$$\frac{2^r-1}{\varphi(2^r-1)} < e^{\gamma}\ln(r\ln(2)) \;=\; 1.781\ln r - 0.652 \tag{5}$$

Using the above result the bound on the average number of steps to relatively prime t is O(ln r).

However, the actual plot (Figure 8) of $\frac{2^r-1}{\varphi(2^r-1)}$ for values of r from 1 to 100 shows that

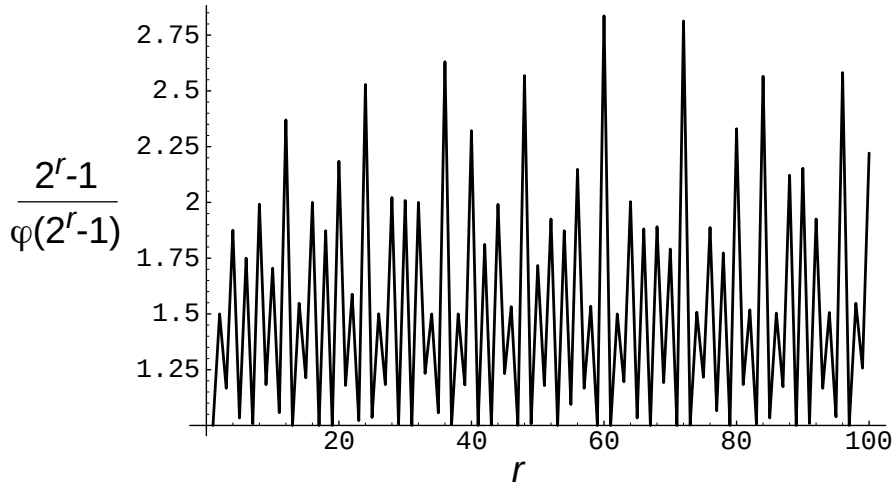

Figure 8. Average Steps to Relatively Prime Number *t*

the average steps is less than 3 (for odd r it is less than 1.25). The O(ln r) bound is not as tight for $n=2^r-1$ and there could in fact be a tighter O(1) upper bound (Figure 9). In the next section, we explore tighter estimates of the average number of steps to relatively prime t.

2.0.1 Average Value Estimation of $\varphi(2^r-1)$

In [APS84] it is shown that

$$E(\varphi(x)) = \frac{1}{x}\sum_{n \le x} \varphi(n) = \frac{3}{\pi^2}x + O(\log x) \tag{6}$$

By substituting (note that this substitution is not valid), $x=2^r-1$, we can heuristically estimate the average order of $\varphi(2^r-1)$ as

$$E(\varphi(2^r-1)) = \frac{3}{\pi^2}(2^r-1) + O(r) \tag{7}$$

Generally, $\varphi(n)$, is a an increasing function. Therefore, with probability,

$$\varphi(2^r-1) > \frac{1}{2^r-1}\sum_{n \le 2^r-1} \varphi(n) = E(\varphi(2^r-1)) = \frac{3}{\pi^2}(2^r-1) + O(r) \tag{8}$$

Using this heuristic argument, the average number of steps to relatively prime *t* appears to be bounded above, with probability, by $\pi^2/3$. Figure 9 illustrates that the $\pi^2/3$ bound is reasonably tight for values of *r* up to 100.

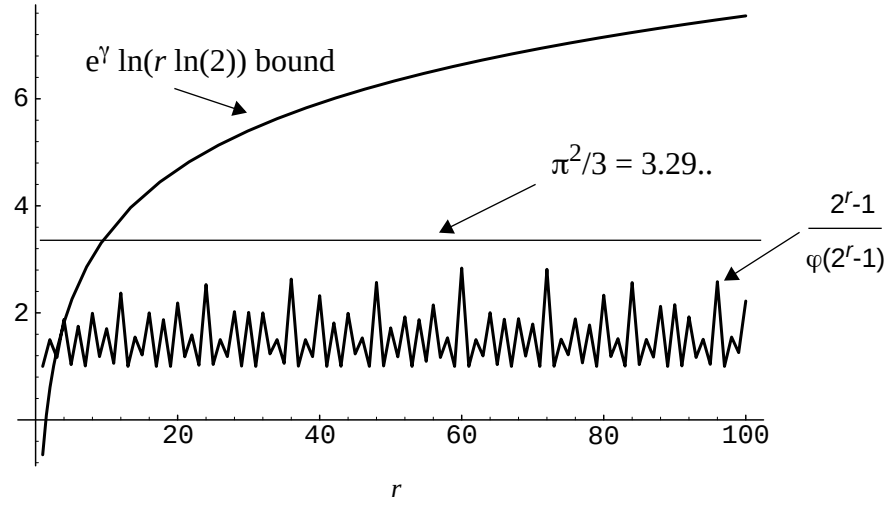

Figure 9. Bounds on the 2^r-1 over $\varphi(2^r$-1)

Figure 9 empirically suggests that is $\pi^2/3$ an upper bound on 2^r-1/$\varphi(2^r$-1); however, this does not last. Table 3 shows that for *r*=360, 2^r-1/$\varphi(2^r$-1) is 3.39 which is greater than $\pi^2/3 \sim 3.29$.

TABLE 3. Some Values of 2^r-1/$\varphi(2^r$-1)

r	2^r-1/ $\varphi(2^r$-1)
60	2.83645
120	3.02633
180	3.17153
240	3.07437
300	2.88622
360	3.39177
420	3.14824

While obtaining a tighter bound on 2^r-1/$\varphi(2^r$-1) may be a topic of theoretical interest; the practical importance of the results in Figure 9 and Table 3 is that the average behavior of 2^r-1/$\varphi(2^r$-1) appears to be more close to O(1) than O(ln *r*). The main result in this section is that the average number of steps required to find the next degree-*r* primitive polynomials from an indexed set of degree-*r* feedback polynomials is between O(*r*) and O(*r* ln *r*). Measurements on actual programs for a large range of values of *r* corroborate this behavior of finding the next degree-*r* primitive

polynomial. All of the plots and calculations of the Euler totient function were done using the *Mathematica* program [WOL92].

3.0 Primitive Polynomial Generation Algorithms: Description and Analysis

3.1 Verifying the Period: Exhaustive Algorithms PeriodA and PeriodS

The most simple way of finding a degree-r primitive polynomial is by simulating the corresponding LFSR and verifying that its period is 2^r-1. The algorithm to do this can be described in the following steps:

1. Pick a degree-r polynomial $P_i(X)$ from an indexed set $\{0, 2^{r-1} -1\}$
2. Verify the period of $P_i(X)$. If the period equals 2^r-1 then print primitive polynomial.
3. Continue while the indexed set lasts.

Both PeriodS and PeriodA programs implement this algorithm. Their listing appears in Appendix (Figure 13 and Figure 14 respectively). PeriodA indexes through the feedback polynomials recursively; whereas, PeriodS indexes through the feedback polynomials iteratively. PeriodS uses the machine shift operations and implements LFSR in machine registers; whereas, PeriodA implements LFSRs as arrays. These differences make PeriodS more efficient than PeriodA (see Table 1). The complexity to verify the period is $O(2^r)$. This is because the complexity of Step 2 in the algorithm is influenced by the average period of feedback plynomials. The *average period* for degree-r polynomials is the sum of periods of all degree-r feedback polynomials divided by 2^{r-1} (i.e., the number of degree-r feedback polynomials). The average period of degree-r feedback polynomials appears to be of order $O(2^r)$. Figure 10 shows the plot of average period for degrees up to 14. Also shown in Figure 10 is the plot of 2^{r-1} to illustrate the $O(2^r)$ growth of the average period. The average period for the plot in Figure 10 was calculated using PeriodS algorithm. The authors are not aware of a mathematical proof† that shows that the average period of degree-r feedback polynomials is $O(2^r)$. This paper does not investigate if such a proof exists because the main focus here is on FactorPower and MatrixPower algorithms. Also, there is enough experimental evidence that suggests that the average period is $O(2^r)$. However, interested readers might want to prove this.

†. It may be sufficient to prove that the average period of degree-r square-free (defined later) feedback polynomials is $O(2^r)$

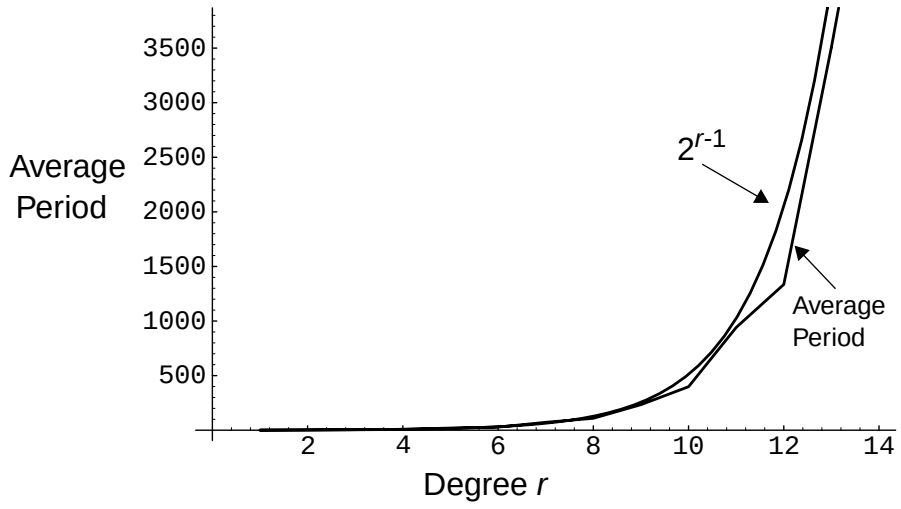

Figure 10. Average Period versus Degree-*r*

In the previous section, we saw that the average number of steps to find the next primitive polynomial is at most O(r ln r). Therefore, for PeriodA and PeriodS programs, the overall complexity to generate every new primitive polynomial is O(r ln r) x O(2^r) = O(r ln r 2^r). This is corroborated by the Secs/Poly plot for the PeriodS program in Figure 3.

3.2 Factorization of 2^r-1: Algorithm FactorPower

Exhaustive algorithms, PeriodA and PeriodS, become impractical for large values of r. Therefore, more efficient algorithms are desirable. If the factorization of 2^r-1 is known then degree-r primitive polynomials can be generated in polynomial time. However, factorization of 2^r-1 is in itself at most O($2^{r/2}$) complexity problem. Luckily, factorization of 2^r-1 has been done for reasonably large values of r [BRI88]. Primitive polynomial generation algorithms can benefit from this work. In this section, we generalize the primitive trinomial generation algorithm [ZIE69] to generate arbitrary primitive polynomials. We call this algorithm *FactorPower* to reflect the basis of the algorithm. FactorPower has the following steps:

1. Start with a known factorization of 2^r-1. Let $2^r - 1 = q_1^{w_1} q_2^{w_2} \ldots q_k^{w_k}$, where all w_i are greater than 0 and q_i are distinct primes.
2. Pick a degree-r feedback polynomial, $P_j(X)$, from an indexed set of 2^{r-1} degree-r feedback polynomials. The indexing can be done by assigning binary coefficients c_{r-1}, c_{r-2}, ..., c_1 for the feedback polynomial $x^r + c_{r-1}x^{r-1} + c_{r-2}x^{r-2} + \ldots + c_1 x + 1$. There are 2^{r-1} distinct assignments.

3. Verify if $P_j(X)$ is a square-free polynomial. That is $P_j(X)$ cannot be written in the form $V(X)^2W(X)$. If $P_j(X)$ is square-free then go to Step 4 else go to Step 7.
4. Verify if $P_j(X)$ is irreducible. If not irreducible then go to Step 7 else go to Step 5.
5. If 2^r-1 is prime then print primitive polynomial and go to Step 7 else go to Step 6.
6. Verify if $P_j(X)$ primitive. This is done as follows: Let $t = 2^r\text{-}1/q_i$. Check if $P_j(X)$ divides X^t+1 for all k distinct factors q_i. If $P_j(X)$ does not divide X^t+1 for all k distinct factors q_i then $P_j(X)$ is primitive. Go to Step 7.
7. Continue while the indexed set lasts.

3.2.1 Square-Free Polynomial Test

Step 3 is visited by all feedback polynomials. The complexity of verifying that a feedback polynomial is square-free is $O(r^2)$. A quick way of proving that $P_j(X)$ is square-free is to show that the greatest common divisor of $P_j(X)$ and the derivative of $P_j(X)$ is 1. This is described in [KNU97]. The function in FactorPower that implements the test for square-free polynomial has $O(r^2)$ computational complexity. There are at least $29*2^r/96$ square-free polynomials. Theorem 1 proves this.

Theorem 1: There are at least (29/96) 2^r degree-r square-free polynomials.

Proof: In order to derive a lower bound on the number of degree-r squarefree polynomials, $\eta(r)$, we derive an upper bound on the number of degree-r polynomials, $\rho(r)$, with square factors. Let $P(X)=V(X)^2W(X)$. Let S_d represent the set of polynomials $V(X)^2W(X)$ such that the degree of $V(X)$ is d. The cardinality of S_1, $|S_1|$, is $2^{r\text{-}3}$. This is because for d=1, $V(X)=1+X$ is the only irreducible polynomial and therfore the number of degree-(r-2) feedback polynomials, $W(X)$, would be $2^{r\text{-}3}$. Likewise, for d=2 the only degree-2 irreducible polynomial is, $V(X)=1+X+X^2$. So $|S_2| = 2^{r\text{-}5}$. In general, the number of degree-d irreducible polynomials $V(X)$ is bounded above by $(2^d\text{-}1)/d$. Therfore, $|S_d|$ is bounded above by $2^{r\text{-}2d\text{-}1}(2^d\text{-}1)/d$. Note that the sets S_j are not disjoint. This implies that the sum of the cardinalities of S_j will be an upperbound on the number of degree-r polynomials with square factors. Also degree-d cannot exceed floor of $r/2$.

$$\rho(r) < \sum_{d=1}^{\left\lfloor \frac{r}{2} \right\rfloor} |S_d| < \sum_{d=1}^{r} |S_d| < 2^{r-3} + 2^{r-5} + \sum_{d=3}^{r} \frac{2^d - 1}{d} 2^{r-2d-1} < 2^{r-3} + 2^{r-5} + \sum_{d=3}^{r} \frac{2^{r-d-1}}{3}$$

Simplifying, we have

$$\rho(r) < 2^{r-3} + 2^{r-5} + \sum_{d=3}^{r} \frac{2^{r-d-1}}{3} = 2^{r-3} + 2^{r-5} + \frac{2^{r-3}-1}{3} = \frac{4}{3}2^{r-3} + 2^{r-5} - \frac{1}{3}$$

Using the upperbound on $\rho(r)$ we have

$$\eta(r) > 2^{r-1} - \left(\frac{4}{3}2^{r-3} + 2^{r-5} - \frac{1}{3}\right) > 2^r\left(\frac{1}{2} - \frac{4}{24} - \frac{1}{32}\right) = \frac{29}{96}2^r \qquad \text{(Q.E.D)}$$

Table 4 lists the actual values, $\eta(r)$, with the lower bound. The actual values of $\eta(r)$ were calculated by the C++ program that implemented the FactorPower algorithm. Observation of the actual values in Table 4 suggests that

$$\eta(r) = \left\lceil \frac{2^r - 1}{3} \right\rceil \qquad (9)$$

Theorem 2 proves that equation (9) is indeed the closed-form formula for $\eta(r)$.

TABLE 4. Lower Bound and the Actual Number of Square-free Polynomials

r	**Lower Bound (29/96) 2^r**	**Actual $\eta(r)$**
10	309.	341
11	618.	683
12	1237.	1365
13	2474.	2731
14	4949.	5461
15	9898.	10923
16	19797.	21845
17	39594.	43691
18	79189.	87381
19	158379.	174763
20	316757.	349525

Theorem 2: There are $\eta(r) = \left\lceil \frac{2^r - 1}{3} \right\rceil$ degree-*r* square-free feedback polynomials.

Proof: Let $\psi(r)$ be the number of degree-*r* irreducible polynomials. From Theorem 3.32 in [BER84] we know that

$$\frac{1}{1-2z} = \prod_{m=1}^{\infty} \left(\frac{1}{1-z^m}\right)^{\psi(m)} \tag{10}$$

Note that for r>1 all irreducible polynomials are feedback polynomials. However, for r=1, $\psi(1)=2$, there are two irreducible polynomials X and $X+1$. We can rewrite the above equation as

$$\frac{(1-z)^2}{1-2z} = \prod_{m=2}^{\infty} \left(\frac{1}{1-z^m}\right)^{\psi(m)} \tag{11}$$

The generating function for all square-free feedback polynomials is

$$f(z) = (1+z)\prod_{m=2}^{\infty}(1+z^m)^{\psi(m)} \tag{12}$$

From equations (11) and (12) we have

$$\frac{f(z)(1-2z)}{(1-z)^2(1+z)} = \prod_{m=2}^{\infty}(1+z^m)^{\psi(m)}\prod_{m=2}^{\infty}(1-z^m)^{\psi(m)} = \prod_{m=2}^{\infty}(1-(z^2)^m)^{\psi(m)} \tag{13}$$

The right-hand side has the same structure as equation (11). Therefore,

$$\frac{f(z)(1-2z)}{(1-z)^2(1+z)} = \prod_{m=2}^{\infty}(1-(z^2)^m)^{\psi(m)} = \frac{(1-2z^2)}{(1-z^2)^2} \tag{14}$$

Solving for $f(z)$

$$f(z) = \frac{(1-2z^2)}{(1-2z)(1+z)} = 1 + \frac{1}{3(1-2z)} - \frac{1}{3(1+z)} \tag{15}$$

The coeffecient of z^r in $f(z)$ will be the number of degree-r square-free feedback polynomials, $\eta(r)$. Taking the coefficient of z^r in $f(z)$ (for r>0)

$$\eta(r) = \frac{1}{3}2^r - \frac{1}{3}(-1)^r = \left\lceil \frac{2^r-1}{3} \right\rceil \tag{Q.E.D}$$

3.2.2 Irreducible and Primitive Polynomial Tests

In Step 4, irreducibility of degree-r polynomial is tested using Berlekamp's [BER84] factorization algorithm. The complexity of verifying irreducibility of a degree-r polynomial $P_j(X)$ is $O(r^3)$ [KNU97]. Step 4 is visited by all square-free polynomials. From Theorem 2 we know that the

number of times Step 4 is visited is $O(2^r)$. Step 5 is O(1) complexity because with a known factorization of 2^r-1 it is trivial to determine the primality. Step 5 is visited by all degree-r irreducible polynomials. In Step 6, verifying that $P_j(X)$ does not divide X^t+1 is $O(r^4)$. The approach used by FactorPower algorithm to show that $P_j(X)$ does not divide X^t+1 is as follows:

- Construct a cononical matrix **A** representing $P_j(X)$.
- Compute $\mathbf{A}^t$.
- If $\mathbf{A}^t$ is not an identity matrix then $P_j(X)$ does not divide X^t+1.

Computation of $\mathbf{A}^t$ can be done by successive squaring and additions of matrix **A.** The term t can be represented by a r-bit number. Matrix multiplication (for squaring) takes $O(r^3)$ steps and for $\mathbf{A}^t$, $O(r)$ squaring operations are needed. Therefore the computation complexity of $\mathbf{A}^t$ is $O(r^4)$. Since this has to done for all distinct factors of 2^r-1, the complexity for Step 5 is $k \times O(r^4) = O(kr^4)$. Step 6 is visited by all feedback polynomials that are irreducible. The total computation time to generate all primitive polynomials is $2^{r-1} \times O(r^2) + \eta(r) \times O(r^3) + \psi(r) \times O(k\, r^4)$. $\psi(r)$ is the number of degree-r irreducible polynomials. The computation time for every primitive polynomial would then be $(2^{r-1} \times O(r^2) + \eta(r) \times O(r^3) + \psi(r) \times O(k\, r^4))/\lambda(r)$. $\lambda(r)$ is the number of degree-r primitive polynomials. From the results in Section 2.0 it follows that $\lambda(r)$ is at least $O((2^r\text{-}1)/(r \ln r))$. $\psi(r)$ is $O((2^r\text{-}1)/r)$ because:

$$\psi(r) = \frac{1}{r}\sum_{d|r} 2^d \mu\left(\frac{r}{d}\right) \tag{16}$$

It follows that $2^{r-1}/\lambda(r)$, $\eta(r)/\lambda(r)$, and $\psi(r)/\lambda(r)$ are $O(r \ln r)$, $O(r \ln r)$, and $O(\ln r)$ respectively. Therefore, the computation time to generate every new primitive polynomial using FactorPower is $O(r \ln r) \times O(r^2) + O(r \ln r) \times O(r^3) + O(1) \times O(k\, r^4) = O(r^4 \ln r) \sim O(k\, r^4 \ln r)$. This is corroborated by plots of actual measurements in Figure 5.

When 2^r-1 is prime, Step 5 is not needed because all irreducible polynomials are primitive. Therefore, the computational time to generate every new primitive polynomial when 2^r-1 is prime is $O(r^4 \ln r)$. It was for this reason that Figure 5 excluded the Mersenne prime points. Table 5 has secs/poly data for the listed Mersenne prime polynomials. For r=61, the measured secs/poly is for 100 primitive polynomials. For other degrees (r=13, 17, and 19) all primitive polynomials were generated by FactorPower. The term $\ln r$ in the complexity analysis can be practically dropped

for practical range of r as is suggested by data presented in Section 2.0. The second column in Table 5 shows that the measured values approximately track $O(r^4)$.

TABLE 5. FactorPower Computational Complexity when 2^r-1 is Prime

r	**Empirical Formula** $9.153\text{x}10^{-8}\, r^3+5.788\text{x}10^{-11} r^4$	**Measured on FactorPower secs/poly**
13	0.00020274	0.000253968
17	0.00045452	0.000453956
19	0.00063535	0.000634558
61	0.02157697	0.025400000

Readers should note that the coefficients for r^3 and r^4 in the empirical formula could vary (and in fact will increase) for large values of r because of cache and memory system effects.

3.3 Primitive Root Power and Matrix Power Isomorphism: MatrixPower Algorithm

Theorem 3 is the basis for the MatrixPower algorithm.

Theorem 3: If $\boldsymbol{A}$ is a r x r $GF(2)$ matrix representation of a degree-r primitive polynomial $P(X)$ then the set of characteristic polynomials in $GF(2)$ of $\boldsymbol{A}^t$ for all t relatively prime to 2^r-1 is identical to the set of all degree-r primitive polynomials.

Proof: Let α be the primitive element in $GF(2^r)$ generated by the primitive polynomial $P(X)$. Since $\boldsymbol{A}$ is a matrix representation of $P(X)$ there is isomorphism between the matrix powers $\boldsymbol{I}$, $\boldsymbol{A}$, $\boldsymbol{A}^2$, ... and the $GF(2^r)$ elements 1, α, α^2, ...

The following facts are true:

- Since A represents the primitive polynomial $P(X)$ the order of $\boldsymbol{A}$ is 2^r-1.
- For every t relatively prime to 2^r-1 the order of $\boldsymbol{A}^t$ is 2^r-1. Also, the characteristic polynomial $H(X)$ of $\boldsymbol{A}^t$ is primitive.
- α^t corresponds to $\boldsymbol{A}^t$ and is a primitive root of $H(X)$ in $GF(2^r)$.

By selecting all t relatively prime to 2^r-1, all primitive elements of $GF(2^r)$ are exhausted. Therefore all primitive polynomials in $GF(2)$ are generated by taking the characteristic polynomials of A^t. Q.E.D

Note that Theorem 3 generates r repetitions of each primitive polynomial in the set of all degree-r primitive polynomials. This is because by selecting all t relatively prime to 2^r-1, all conjugate

roots [5] are also considered. For example,

$$\alpha^t, \alpha^{2t}, \ldots, \alpha^{2^{r-1}t}$$

are r primitive roots of the same primitive polynomial. By selecting only one member from the set of r conjugate roots (all exponents of α are mod 2^r-1) repetitions can be avoided.

The number of t that are relatively prime to 2^r-1 is the Euler's totient function $\varphi(2^r\text{-}1)$.

By selecting only one out of the r conjugate roots, $\varphi(2^r\text{-}1)/r$ distinct degree-r primitive polynomials are generated. Another important observation is that by Theorem 3 we can generate all primitive polynomials from any known primitive polynomial.

MatrixPower program implements the following algorithm:

1. Pick a known degree-r primitive polynomial $P(X)$.
2. Construct canonical matrix $\boldsymbol{A}$ representing $P(X)$. Compute $\boldsymbol{A}^2$ and store in memory.
3. Pick $\boldsymbol{t}$ from an indexed set. Compute $\boldsymbol{A}^{\boldsymbol{t}}$.
4. Check if conjugates of $\boldsymbol{t}$ already selected. If conjugate already selected then go to Step 7 else go to Step 5.
5. Verify if $\boldsymbol{t}$ is relatively prime to 2^r-1. If $\boldsymbol{t}$ relatively prime go to Step 6 else go to Step 7.
6. Compute and print the characteristic polynomial of $\boldsymbol{A}^{\boldsymbol{t}}$
7. Continue while the indexed set lasts.

3.3.1 Complexity Analysis of MatrixPower

Step 1 is the input to the MatrixPower algorithm. The canonical matrix A for Step 2 requires $O(r^2)$ storage. The index t is selected from the set of odd numbers, $\{2m+1: 0 \le m < 2^{r-1}\}$, in increasing order. This automatically eliminates all conjugate roots that have even powers of α. Also, selecting t in increasing order allows accumulating the powers of matrix A so that the new matrix power A^t is obtained by multiplying previously stored A^{t-2} and A^2. The implies that the Step 3 computation complexity is $O(r^3)$. In Step 4, MatrixPower uses a simple procedure to discard conjugate roots by selecting the smallest exponent of α. This is done as follows: for any selected t from Step 3, new exponents $2t$ mod $(2^r\text{-}1)$, 2^2t mod $(2^r\text{-}1)$, ..., $2^{r-1}t$ mod $(2^r\text{-}1)$ are generated. If any of these exponents are less than t then t is discarded from selection in Step 4. Note that these exponents are derived from successive left cyclic shifts of an r-bit representation of t. This process ensures only one conjugate root is selected. The complexity of Step 4 therefore is $O(r)$. The complexity to verify if t is relatively prime to 2^r-1 in Step 5 is no worse than $O(r \ln r)$.

Computation of characteristic polynomial of the precomputed matrix A^t in Step 6 is no worse than $O(r^3)$ [FAD63]. Steps 3 and 4 are visited 2^{r-1} times (for all odd t). Step 5 is visited $O(2^r/r)$ times. Step 6 is visited exactly $\lambda(r)$ times. Total computation time to generate all primitive polynomials is of order

$$2^{r-1}O(r^3) + 2^{r-1}O(r) + O\left(\frac{2^r}{r}\right)O(r\ln r) + \lambda(r)O(r^3)$$

Follows that the computation time per primitive polynomial is

$$\frac{2^{r-1}O(r^3) + 2^{r-1}O(r) + O\left(\frac{2^r}{r}\right)O(r\ln r) + \lambda(r)O(r^3)}{\lambda(r)} = O(r^4\ln r)$$

The performance results presented in Figure 4 corroborate the $O(r^4 \ln r)$ complexity of MatrixPower. Table 6 illustrates the generation of all degree 5 primitive polynomials from the primitive polynomial $x^5+x^4+x^3+x^2+1$ by using MatrixPower.

TABLE 6. Matrix Power Generation of Degree 5 Primitive Polynomials

t	1	3	5	7	11	15
A^t	$\begin{bmatrix}0&0&0&0&1\\1&0&0&0&0\\0&1&0&0&1\\0&0&1&0&1\\0&0&0&1&1\end{bmatrix}$	$\begin{bmatrix}0&0&1&1&0\\0&0&0&1&1\\0&0&1&1&1\\1&0&1&0&1\\0&1&1&0&0\end{bmatrix}$	$\begin{bmatrix}1&1&0&0&1\\0&1&1&0&0\\1&1&1&1&1\\1&0&1&1&0\\1&0&0&1&0\end{bmatrix}$	$\begin{bmatrix}0&0&1&0&0\\1&0&0&1&0\\1&1&1&0&1\\1&1&0&1&0\\0&1&0&0&1\end{bmatrix}$	$\begin{bmatrix}0&1&1&1&1\\0&0&1&1&1\\1&1&1&0&0\\0&0&0&0&1\\1&1&1&1&1\end{bmatrix}$	$\begin{bmatrix}1&1&0&1&1\\1&1&1&0&1\\0&0&1&0&1\\1&1&0&0&1\\1&0&1&1&1\end{bmatrix}$
Characteristic Polynomial	$x^5+x^4+x^3+x^2+1$	$x^5+x^4+x^2+x+1$	x^5+x^3+1	$x^5+x^4+x^3+x+1$	x^5+x^2+1	$x^5+x^3+x^2+x+1$

Note that if repetition of primitive polynomials is allowed then the complexity of generating primitive polynomials using MatrixPower would be $O(r^3 \ln r)$. This is particularly advantageous when r is very large.

4.0 PeriodS, FactorPower, and MatrixPower Applications

There are situations where the choice of having many primitive polynomials enhances the design's functionality or performance. In these situations, primitive polynomial generation algorithms find application. Some of the design cases previously mentioned in Section 1.0 are:

- Error Detection and Correction (EDAC) code designs.
- Key generation in cryptography.
- Design for Testability

4.1 EDAC Designs- Using PeriodS

EDAC designs that are capable of correcting more than single bit errors or detecting more than five bit errors are often based on BCH polynomials [5-7]. In general, a BCH polynomial is formed by using a primitive polynomial as one of its factors. BCH code designs are, generally, implemented using sequential encoders and decoders. The complexity of serial encoders and decoders is largely independent of the type of primitive polynomial used. However, if the information bits a bounded by some practical value it is possible to implement parallel EDAC encoder and decoder designs. Parallel encoders and decoders allow single cycle encoding and decoding functionality. The hardware complexity, as measured in terms of area and cycle time, of parallel EDAC encoders and decoders depends on the type of primitive polynomial used. Parallel encoders and decoders for double-error correcting EDAC designs have been used in [SAX95][SAX96]. By using an appropriate primitive primitive polynomial double-error correcting EDAC designs can be optimized.

In a related research [SAX98], the authors have developed synthesis techniques that use PeriodS algorithm to optimize parallel EDAC designs. For example, Figure 11 shows the percentage improvement in logic gate count for the various double-error correcting EDAC code designs. Results show reduction up to 14% in gate counts. Each of these designs is obtained in less than 2 minutes. The value of r up to 9 are sufficient to support information bits up to length 256. PeriodS algorithm is used because it is the fastest algorithm for this range of values of r. For larger values of r either MatrixPower or FactorPower could be used; however, as the code length increases parallel encoder and decoder designs become less practical.

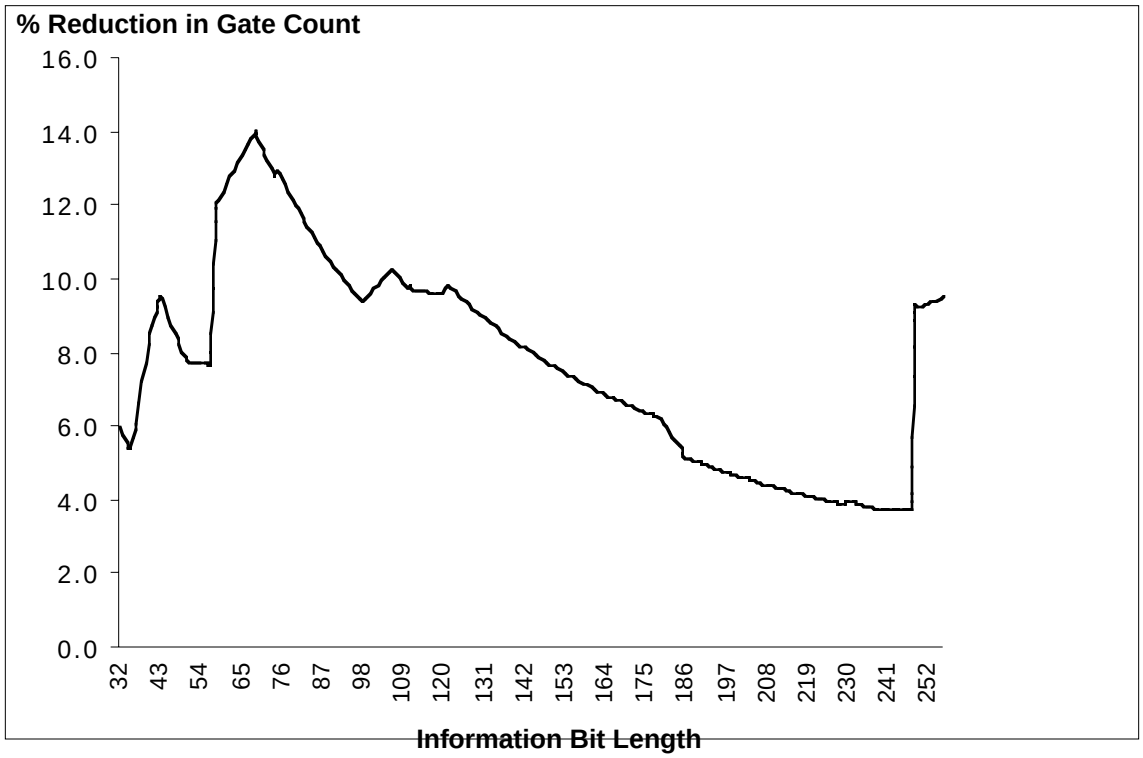

Figure 11. Double Error Correcting EDAC Design Performance Improvement

4.2 Key Generation in Stream Ciphers- Using MatrixPower

In cryptography, stream ciphers [BEK82][SCH96][RSA94] have been used to encrypt and decrypt messages. Most practical stream cipher designs use LFSRs or their variants. In general, LFSRs in stream-ciphers are maximal-length and therfore are implmentations of primitive polynomials.

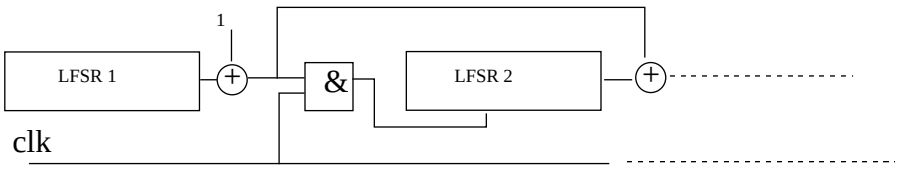

Figure 12. Gollmann Cascade

Some stream ciphers like shift register cascade (Gollmann Cascade [SCH96][RSA94], see Figure 12) appear to have good security properties. As was discussed in Section 1.0 the parameters that define the behavior of these stream ciphers are the initial contents and the defining primitive polynomials of the LFSRs. For a given stream cipher architecture, we can think of initial contents and the defining polynomials of the LFSRs as keys. Having algorithms that pick different primitive polynomials for the LFSRs could contribute to increased security of the stream ciphers. For the Hughes XPD/KPD stream cipher algorithm [SCH96] that uses a 61-bit LFSR, both MatrixPower and Factor Power can generate degree-61 primitive polynomials every 0.025 secs (see Table 5). In general order to search for more secure stream cipher implementations a greater choice in the selection of primitive polynomials is clearly a requirement.

Importance of MatrixPower (for large values of r and 2^r-1 is not prime) Key generation registry.

Following is a small list of degree 300 primitive polynomials generated by *MatrixPower* using the primitive polynomial $x^{300}+x^7+1$ listed in [BAR87]:

$X^{300}+X^{126}+X^{110}+X^{71}+X^{55}+X^{39}+X^{23}+X^7+1$
$X^{300}+X^{253}+X^{206}+X^{130}+X^{83}+X^{36}+X^{18}+X^7+1$
$X^{300}+X^{236}+X^{209}+X^{172}+X^{145}+X^{118}+X^{108}+X^{81}+X^{54}+X^{44}+X^{17}+X^7+1$
$X^{300}+X^{198}+X^{147}+X^{128}+X^{83}+X^{45}+X^{32}+X^{26}+X^{13}+X^7+1$
$X^{300}+X^{244}+X^{211}+X^{155}+X^{132}+X^{109}+X^{99}+X^{86}+X^{76}+X^{43}+X^{33}+X^{30}+X^{10}+X^7+1$
$X^{300}+X^{245}+X^{196}+X^{190}+X^{184}+X^{141}+X^{129}+X^{74}+X^{68}+X^{49}+X^{31}+X^{25}+X^{13}+X^7+1$
$X^{300}+X^{281}+X^{262}+X^{224}+X^{205}+X^{148}+X^{144}+X^{129}+X^{110}+X^{106}+X^{11}+X^7+1$
$X^{300}+X^{238}+X^{228}+X^{119}+X^{114}+X^{104}+X^{94}+X^{27}+X^{12}+X^7+1$
$X^{300}+X^{265}+X^{219}+X^{195}+X^{184}+X^{173}+X^{149}+X^{136}+X^{103}+X^{92}+X^{90}+X^{68}+X^{55}+X^{46}+X^{44}+X^{20}+X^9+X^7+1$
$X^{300}+X^{199}+X^{194}+X^{189}+X^{184}+X^{126}+X^{121}+X^{98}+X^{88}+X^{78}+X^{73}+X^{55}+X^{40}+X^{35}+X^{20}+X^{15}+X^7+X^5+1$
$X^{300}+X^{238}+X^{187}+X^{176}+X^{170}+X^{125}+X^{102}+X^{74}+X^{68}+X^{63}+X^{57}+X^{52}+X^{51}+X^{40}+X^{35}+X^{34}+X^{17}+X^{12}+X^7+X^6+1$
$X^{300}+X^{259}+X^{218}+X^{203}+X^{190}+X^{175}+X^{149}+X^{147}+X^{136}+X^{134}+X^{119}+X^{91}+X^{67}+X^{63}+X^{52}+X^{50}+X^{39}+X^{35}+X^{22}+X^{11}+X^9+X^7+1$

5.0 Summary

Table summarizes the computation complexity of PeriodS, FactorPower, and MatrixPower algorithms.

TABLE 7. Summary of Complexity

Algorithm	Computation	Storage	Requirements
PeriodS	$O(r \ln r\, 2^r)$	$O(r)$	None.
MatrixPower	$O(r^4) \sim O(r^4 \ln r)$	$O(r^2)$	Needs a known degree-r primitive polynomial.
FactorPower	$O(r^4) \sim O(k\, r^4 \ln r)$	$O(r^2)$	Needs factorization of 2^r-1.

An extremely fast algorithm called *MatrixPower* is proposed that generates all degree-*r* primitive polynomials from a known degree-*r* primitive polynomial. Such a generation technique can be gainfully employed in applications like: random pattern generation, cryptography, error correcting and detecting codes, and built-in testing for VLSI circuits.

References

[GOL67] Solomon W. Golomb, *Shift Register Sequence*, Holden-Day, 1967.

[BEK82] Henry Beker & Fred Piper, *Cipher Systems- The Protection of Communications*, John Wiley and Sons, 1982.

[SCH96] Bruce Schneier, Applied Cryptography- Protocols, Algorithms, and Source Code in C, John Wiley & Sons, 1996.

[RSA94] *Frequently Asked Questions About Today's Cryptography*, Version 3, RSA Laboratories, 1994. Web: http://www.rsa.com/rsalabs/

[PET84] W. W. Peterson and E. J. Weldon, Jr., "Error-Correcting Codes", Second Edition, The MIT Press, 1984.

[BER84] Berlekamp, E.R., *Algebraic Coding Theory*, Revised Edition, Aegean Park Press, 1984.

[WIC95] Stephen B. Wicker, *Error Control Systems for Digital Communications and Storage*, Prentice Hall, Englewood Cliffs, NJ, 1995. (Program written by Haluk Aydinoglu in Appendix A, pp. 444-445)

[BAR87] P. H. Bardell, W.H. McAnney, and J. Savir, *Built-In-Test for VLSI*, New York: John Wiley and Sons, 1987.

[SAX92] N. R. Saxena, P. Franco, and E.J. McCluskey, "Simple Bounds on Serial Signature Analysis Aliasing for Random Testing," *Special Issue on Fault Tolerant Computing, IEEE Transactions on Computers*, Vol. C-41, N0. 5, pp. 638-645, May 1992.

[SAX97] N. R. Saxena and E.J. McCluskey, "Parallel Signature Analysis Design with Bounds on Aliasing," *IEEE Transactions on Computers*, Vol. C-46, N0. 4, pp. 425-438, Apr 1997.

[ZIE68] N. Zierler & J. Brillhart, " On Primitive Trinomials (Mod 2)," Information and Control 13, pp. 541-554, 1968.

[ZIE69] N. Zierler & J. Brillhart, " On Primitive Trinomials (Mod 2), II," Information and Control 14, pp. 566-569, 1969.

[ALA64] J.D. Alanen & D.E. Knuth, "Tables of Finite Fields," Indian Journal of Statistics, Series A, Vol. 26, pp. 305-328, Dec 1964.

[BAR90] P. H. Bardell, "Design Considerations for Parallel Pseudorandom Pattern Generators," JETTA, Vol. 1, No. 1, pp. 73-87, Feb. 1990.

[BAR92] P. H. Bardell, "Primitive Polynomials of Degree 301 through 500," JETTA, Vol. 3, pp 175-176, 1992.

[BRI88] J. Brillhart , D.H. Lehmer, J.L. Selfridge, B. Tuckerman and S.S. Wagstaff, Jr., *Factorizations of* $b^n \pm 1$ b = 2, 3, 5, 6, 10, 11, 12 *up to high powers*, Contemporary Mathematics, Volume 22, Second Edition, American Mathematical Society, 1988.

[APS84] Tom M. Apostol, *Introduction to Analytic Number Theory*, Springer-Verlag, 1984.

[HAR85] G. H. Hardy & E. M. Wright, *An Introduction to Theory of Numbers*, 5th ed., Oxford 1985.

[FAD63] D. K. Faddeev & V. N. Faddeeva, *Computational Methods of Linear Algebra*, Translated by R. C. Williams, W. H. Freeman and Company, 1963.

[SWA62] Swan, R. G., "Factorization of Polynomials Over Finite Fields," Pacific J. Math., Vol. 12, pp. 1099-1106, 1962.

[KNU97] D. E. Knuth, *The Art of Computer Programming*: *Fundamental Algorithms*, Volume 1, Third Edition, Addison Wesley, 1997.

[WOL92] Stephen Wolfram, *Mathematica*: *A System of Doing Mathematics by Computer*, Second Edition, Addison Wesley, 1992.

[SAX95] N. R. Saxena et. al., "Fault-Tolerant Features in HaL's Memory Management Unit", IEEE Transactions on Computers, Feb. 1995.

[SAX96] N. R. Saxena & C-W. D. Chang, "Error Detection and Correction Method and Apparatus," **US Patent 5,533,035**, Jul. 2, 1996.

[SAX98] N. R. Saxena, L. Avra, and E. J. McCluskey, "Synthesis of Error Detection and Correction Logic," 1998 Intl. Test Synthesis Workshop, Santa Barbara, Mar 1998.

Appendix

```
/*
  This program prints out all primitive polynomials of a given degree r.
  Usage is: period <degree_of_poly>
  */

  #include <stdio.h>
  unsigned int poly;
  unsigned int mask;

  main(argc,argv)
  int argc;
  char **argv;
  {
   unsigned int r,i,j,limit,Lc;
   extern unsigned int mask,poly;
   if (argc<2)
   {
    printf("Usage:period <degree_of_poly>\n");
    exit(1);
   }
   print_primitive=1;
   r = (unsigned int) atoi(argv [1]);
   mask = 1;
   for (i=1; i < r; i++) {
     mask = 2*mask;
    }
    limit = 2*mask;
  for (i=1; i < limit; i=i+2) {
     poly = i;
     j = pred(1,0);
     Lc = 1;
    while (pred(1,j) != 0 ) {
     Lc++;
     j = pred(0,j);
    }
    if (Lc == (limit-1))
     printf("%u\n", i);
   }
  }
  pred(x,y)
  unsigned int x,y;
  {
  extern unsigned int poly,mask;
    if ( (y & mask) > 0 )
    {
     if (x == 1)
      return((y << 1) & (2*mask-1));
     else
      return(((y << 1) ^ poly) & (2*mask-1));
    }
    else
    {
     if (x == 0)
      return((y << 1) & (2*mask-1));
     else
      return(((y << 1) ^ poly) & (2*mask-1));
    }
  }
```

Figure 13. Listing of PeriodS Algorithm

```
/*
  The program prints all primitive polynomials of a given degree. The usage is:
  program_name <degree_of_poly>
*/

#define N 32
#include <stdio.h>
int i,n,xx, degree; char d[N];
void see () {
  for (i=n; i>=0; i--) printf ("%c",d[i]+'0');
  printf("\n");
}
char s[N],t,j,f; unsigned long c,max;

void visit () {
  for (i=0;i<n;i++) s[i]=1; c=0;
  do
    {c++;
    for (i=t=0;i<n;i++)
      t = (t^(s[i]&d[i]));
    for (i=0;i<n-1;i++)
      s[i]=s[i+1];
    s[n-1]=t;
    for (i=f=0;i<n;i++)
      if (!s[i]) {f=1; break;}
    }
  while (f);
  if (c==max) see ();
}

void gp (l) char l; {
   if (!l) visit();
   else {
     d[l] = 0; gp (l-1);
     d[l] = 1; gp (l-1);
   }
}

void gc (l,rw) char l, rw; {
  char q;
  if (rw==2) { visit(); return;}
  for (q=1;q>=rw-2;q--) {
    d[q]=1; gc(q-1,rw-1);d[q]=0;
  }
}

void main(int argc, char *argv[]) {
  printf("\n");
  degree = atoi(argv[1]);
  for (n=degree;n<=degree;n++) {printf("%d\n",n);
  for (xx=max=1;xx<=n;xx++) max=2*max; max--;
  d[n] = d[0] = 1;
  gp(n-1); /* use gp(n-1) if all prim polys are desired */
  printf("\n");
  }
}
```

Figure 14. Listing of PeriodA Algorithm