Product Registers: Linear Complexity, de Bruijn Sequences, and Cryptanalysis

Nitya Arora, Allen Chang, Frank Qiang, David Gordon, Vincent Mooney

Georgia Institute of Technology
VIP Secure Hardware Fall 2023

Contents

1 Introduction and Motivation

The purpose of this research is to introduce a novel cryptographic primitive called Composite Mersenne Product Registers (CMPRs) with several key properties and applications:

- **Lightweight.** With a small footprint, CMPRs can be implemented only a register and feedback/chaining functions.
- **Nonlinear.** The rigid structure of other structures related to CMPRs in prior work can allow for easy exploit. Chaining provides the property of nonlinearity.
- **Flexible.** The primitive proposed can be used in a variety of contexts, such as pseudo-random number generation.

The rest of this paper will explain the background needed to understand and conceptualize CMPRs, as well as tests of their security and comparisons to alternate constructions based off of de Bruijn sequences.

2 Notation and Terminology

We begin by briefly discussing some of the mathematics notation used in this report. For a set S, we write $x \in S$ to signify that some element x belongs to the set S. For sets A and B, their *Cartesian product* is the set

$$A \times B = \{(a, b) : a \in A, b \in B\}.$$

We write $f : A \to B$ to mean that f is a function with domain A and codomain B. We write $\deg P$ to denote the *degree* of a polynomial $P(x)$, i.e. the largest power of x that appears in the expansion of $P(x)$. We write $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$ to denote the real numbers, rational numbers, and integers, respectively. By arithmetic modulo some integer n, we mean to take the result of any usual arithmetic operation and reduce it to its remainder after dividing by n. More precisely, for $a, b \in \mathbb{Z}$, if the operations $a + b$ or ab would usually result in some number $c \in \mathbb{Z}$, we instead define its result *modulo* n, written $c \pmod{n}$, to be the unique integer $0 \leq r < n$ satisfying

$$c = qn + r$$

for some quotient $q \in \mathbb{Z}$. Now we proceed with defining some terminology freely used in the remainder of the report. Note that some definitions are intentionally introduced informally here and will be redefined more formally in its appropriate section if necessary.

Mersenne primes: Primes that are of the form $2^n - 1$ [MOV01].

Feedback shift registers: Register made of connected flip flops, where the state of the next flip flop depends on the previous one (connected through XOR or just a wire).

Update polynomial: A polynomial which is multiplied with the polynomial form of the current state of an FSR to produce the next state's polynomial representation [Gor+21].

Primitive polynomial: Polynomial used as a modulus to create Galois fields for LFSRs and MPRs; Such a polynomial generates the multiplicative group of $\mathrm{GF}(2^n)$ [MOV01].

Linear feedback shift register: Feedback shift register where linear function applied to current state leads to next state. We will usually abbreviate this via "LFSR."

Nonlinear feedback shift register: Feedback Shift Register where nonlinear function applied to current state leads to next state. We will usually abbreviate this via "NLFSR."

Product registers: Full-period LFSRs that implement finite field multiplication [GM23]. We will sometimes abbreviate this via "PR."

Linear complexity: The minimum length of an LFSR that can produce an input shift register sequence; used as a way to measure the security of a FSR [Gor+21].

Feedback functions A feedback function is a Boolean function which outputs for some input of n bits from some initial state s_0 an output s_n [Gor+21].

3 Background

3.1 Groups and Fields

Definition 1 (Group)**.** A *group* $(G, *)$ is a set G with a binary operation $* : G \times G \to G$ such that

(i) $*$ is associative, i.e. $f * (g * h) = (f * g) * h$ for any $f, g, h \in G$,

(ii) G has an identity element e such that $e * g = g * e = g$ for any $g \in G$,

(iii) and any element $g \in G$ has an inverse $g^{-1} \in G$ such that $g^{-1} * g = g * g^{-1} = e$. [Gal20]

Note that we often just write G to refer to $(G, *)$ when the operation $*$ is clear from context.

For example, the integers $\mathbb{Z}$ form a group under addition: We know that addition is associative, 0 is the identity element, and the negative of an integer is its (additive) inverse. In fact, we can even refine this to say that the integers *modulo* n form a group under (modular) addition. In both of these cases, we remark that the group operation is commutative, since we know that $a + b = b + a$ for any integers a, b. When the operation of a general group G satisfies this commutative property, we say that G is *abelian*. An example of a nonabelian group is the group of invertible matrices under matrix multiplication (called the *general linear group*).

As a side note, mathematicians like to think of a group as some collection of symmetries of some object. For example, you can imagine of the rotational symmetries of a square, which is somehow related to the integers modulo 4. However, though nice, this perspective is not very readily apparent in our setting and will not be mentioned again in this report.

A *subgroup* of a group G is some subset $H \subseteq G$ which is a group in its own right, under the operation inherited from G. Furthermore, each $g \in G$ *generates* its own subgroup

$$\langle g \rangle = \{g^n : n \in \mathbb{Z}\} = \{\dots, g^{-2}, g^{-1}, e, g, g^2, \dots\},$$

where $\langle g \rangle$ is to be read as "the subgroup generated by g." A nice result from group theory describing some properties of subgroups is the following:

Theorem 1 (Lagrange's theorem)**.** *For any finite group G and a subgroup $H \subseteq G$, $|H|$ divides $|G|$. [Gal20]*

In particular, Lagrange's theorem tells us that the size of the generated subgroup $\langle g \rangle$ must divide $|G|$ for any fixed $g \in G$. This will become relevant later in justifying the use of Mersenne primes in our product registers.

Definition 2 (Field)**.** A *field* $(\mathbb{F}, +, \times)$ is a set $\mathbb{F}$ with two binary operations $+, \times : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ such that

(i) $\mathbb{F}^+ := (\mathbb{F}, +)$ is an abelian group,

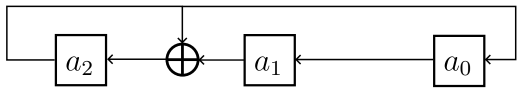
(ii) $\mathbb{F}^\times := (\mathbb{F} \setminus \{0\}, \times)$ is an abelian group,

(iii) and $\times$ distributes over $+$, i.e. $a \times (b + c) = (a \times b) + (a \times c)$ for all $a, b, c \in \mathbb{F}$. [Gal20]

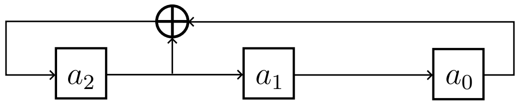
Similarly we often just write $\mathbb{F}$ to refer to $(\mathbb{F}, +, \times)$ when $+$ and $\times$ are clear from context.

An example of a familiar field is the field of rational numbers $\mathbb{Q}$, where $+$ and $\times$ are simply the usual addition and multiplication of numbers we are familiar with. Similarly the real numbers $\mathbb{R}$ form a field (notably the integers $\mathbb{Z}$ do not form a field since the reciprocal of an integer is in general not an integer). However, these are all examples of *infinite* fields. In the setting of feedback shift registers, we will use *Galois fields*, or *finite* fields.

The classical example of a finite field is the integers modulo a prime p, with the operations of modular addition and multiplication. We will call this field $\mathrm{GF}(p)$. The requirement of the



(a) LFSR in Galois form



(b) LFSR in Fibonacci form

Figure 1: Examples of Galois and Fibonacci LFSRs, respectively.

modulus being prime guarantees the existence of multiplicative inverses, which is a result from number theory. From $\mathrm{GF}(p)$, we can construct larger finite fields of the form $\mathrm{GF}(p^n)$, which we will interpret as a field containing polynomials of degree bounded by $n-1$. In particular, any $f \in \mathrm{GF}(p^n)$ can be written in the form

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$$

for some constants $a_0, \ldots, a_{n-1} \in \mathrm{GF}(p)$.

We define addition for $f, g \in \mathrm{GF}(p^n)$ simply via the usual polynomial addition $f+g$, where we note that we must be careful to reduce each coefficient of $f+g$ modulo p since we are working over the field $\mathrm{GF}(p)$. After fixing a primitive polynomial $P \in \mathrm{GF}(p)$, we can define the product of $f, g \in \mathrm{GF}(p^n)$ to be their product as polynomials, reduced modulo the polynomial P. Doing so requires the use of the *division algorithm*, which states that for any polynomial $h \in \mathrm{GF}(p^n)$, we can express it as

$$h = qP + r,$$

for some $q, r \in \mathrm{GF}(p^n)$ and where P is the previously chosen primitive polynomial. Demanding that $\deg r < \deg P$ makes the choice of q and r unique [Gal20]. In this manner, writing

$$fg = qP + r$$

uniquely defines r to be the product of f and g in the field $\mathrm{GF}(p^n)$. More explicit examples of this field addition and multiplication will be presented when we introduce the Galois LFSR in the next section.

3.2 LFSRs

Definition 3 (Linear feedback shift register (LFSR))**.** A feedback shift register where a linear function is applied to the current state to reach the next state [Gor+21]

Among the constructions of LFSRs, there are two main forms that exist: **Galois** and **Fibonacci**. We say that an LFSR is in **Galois form** when the next state is always a field multiplication with the previous state, and all 2^n states lie in a Galois field (as illustrated in Figure 1a). On the other hand, we say that an LFSR is in **Fibonacci form** when the next state is a linear combination of the previous state. These are the same types of LFSRs used by Berlekamp-Massey (as illustrated in Figure 1b).

3.2.1 Worked Example for Galois LFSR

We now provide an example of how a Galois LFSR updates on every clock cycle, defined by a primitive polynomial and the update function $U(x) = x$. Let

$$\begin{aligned} P &= x^3 + x^2 + 1 && \text{(Primitive Polynomial)} \\ U &= x && \text{(Update Polynomial)} \\ S_0 = 110 &= x^2 + x && \text{(Initial State)} \end{aligned}$$

which describes the LFSR from Figure 1a. Here we let S_i denote the ith state of the LFSR. Then the following describes the math to clock the LFSR once:

$$\begin{aligned} S_1 &= S_0 \times U && \text{(Field Multiplication)} \\ &= (x^2 + x) \times x = x^3 + x^2 && \text{(Multiply by U)} \\ &= (x^3 + x^2) \pmod{P} && \text{(Modulus on P)} \\ S_1 &= 1 \rightarrow \boxed{001} && \text{(Next State)} \end{aligned}$$

Repeatedly clocking the LFSR yields a cycle of length 7, which it then repeats for after the 7th clock update. See Table 1 to see how the full state cycle evolves.

More about Galois LFSRs [SGM22]: LFSR hardware for Galois form can be derived from field multiplication. We can identify an n-bit vector in the state of an LFSR with a polynomial of degree $n - 1$. A small example of this one-to-one mapping between the space of $n - 1$ degree polynomials and n-bit binary numbers is of 1011 corresponding to $x^3 + x + 1$. For the Galois form, the update polynomial is restricted to $U(x) = x$.

3.3 PRs, MPRs, and CMPRs

Definition 4 (Product register)**.** A *product register* A is a generalization of a Galois LFSR where we may use any update polynomial $U \in \mathrm{GF}(2^n)^\times$, in addition to just $U(x) = x$. The state update of A is given by:

$$A[t+1] = A[t] \times U \pmod{P}.$$

$\text{State}_{\text{curr}}$	Multiplication	Result	Mod P	$\text{State}_{\text{next}}$
110	$(x^2 + x) \times x$	$x^3 + x^2$	1	001
001	$1 \times x$	x	x	010
010	$x \times x$	x^2	x^2	100
100	$x^2 \times x$	x^3	$x^2 + 1$	101
101	$(x^2 + 1) \times x$	$x^3 + x$	$x^2 + x + 1$	111
111	$(x^2 + x + 1) \times x$	$x^3 + x^2 + x$	$x + 1$	011
011	$(x + 1) \times x$	$x^2 + x$	$x^2 + x$	110

Table 1: State cycle of the Galois LFSR from Figure 1a.

Note that $U = 0$ is automatically excluded since it is not part of the multiplicative group, and we usually exclude $U = 1$ because that trivially leads to an always-constant output [GM23]. But a general product register is not quite good enough; we would like some more nice properties to work with. This leads to the construction of a *Mersenne* product register:

Definition 5 (Mersenne product register)**.** A *Mersenne product register* is a product register of size n, where n is a Mersenne exponent. [GM23]

The crucial property of the Mersenne prime we need is for $|\mathrm{GF}(2^n)^\times| = 2^n - 1$ to be prime. By Lagrange's theorem, this ensures that any subgroup of $\mathrm{GF}(2^n)^\times$ has size either 1 or $2^n - 1$. Since we required $U \neq 1$ for the update polynomial of a product register, this yields $|\langle U \rangle| = 2^n - 1$ for any choice of $U \in \mathrm{GF}(2^n)^\times$. In particular, this means that $\langle U \rangle = \mathrm{GF}(2^n)^\times$, i.e. U generates all the nonzero elements of $\mathrm{GF}(2^n)$. Translating this back into the language of feedback shift registers, this means that any arbitrary choice of $U \neq 0, 1$ (with nonzero initial state) results in a full-period Mersenne product register.

Definition 6 (Composite Mersenne product register)**.** A *composite Mersenne product register* is a collection of $n + 1$ MPRs $M_0, \ldots, M_n$ and n chaining functions $\mathcal{C}_1, \ldots, \mathcal{C}_n$ that link the MPRs together. The state updates are given by:

$$M_i[t + 1] = (M_i[t] \times U_i) \oplus \mathcal{C}_i(M_0[t], \ldots, M_{i-1}[t]) \pmod{P_i}$$

for $1 \leq i \leq n$, where U_i and P_i are the update and primitive polynomials corresponding to M_i, respectively. [GM23]

In hardware, these chaining functions are implemented using logic gates (typically AND) connecting the outputs of $M_0, \ldots, M_{i-1}$ to the input of M_i for each $1 \leq i \leq n$. The upshot here is that although each individual MPR is linear, the AND gates introduced via chaining functions make the CMPR nonlinear as a whole, while still maintaining enough structure that we can meaningfully analyze.

3.4 De Bruijn Sequences

In this section, we introduce an alternative way of studying LFSRs, namely the analysis of Fibonacci-style LFSRs and their relation to *de Bruijn sequences*. Recall that a Fibonacci LFSR expresses its next state as a linear combination of its previous states. Mathematically, this is something like

$$s_n = \sum_{i=1}^{k} c_k s_{n-k}$$

for a k-bit Fibonacci LFSR, where s_i is the ith output bit and $c_1, \ldots, c_k$ are constants describing the recurrence.

Definition 7 (De Bruijn sequence)**.** A *de Bruijn sequence* of order n on an alphabet $\mathcal{A}$ is a sequence of length $|\mathcal{A}|^n$ such that each n-tuple of letters in $\mathcal{A}$ appears exactly once in one period of the sequence. [Li+15]

For example, a de Bruijn sequence of order 3 on $\{0,1\}$ might look like

$$\begin{aligned}00010111 &= [000]10111 = 0[001]0111 = 00[010]111\\ &= 0001[011]1 = 00]01011[1 = 000[101]11\\ &= 0]00101[11 = 00010[111],\end{aligned}$$

where the occurrence of each 3-bit binary number has been marked in lexicographical order. Note that the n-tuples are allowed to wrap around, since we implicitly take de Bruijn sequences to be periodic. Additionally, de Bruijn sequences are minimal in the sense that we need at least a sequence of length $|\mathcal{A}|^n$ to reach all $|\mathcal{A}|^n$ possible n-tuples. Finally, note that for applications to feedback shift registers, we usually consider *binary* de Bruijn sequences, i.e. we take $\mathcal{A} = \{0,1\}$.

Now here are some reasons we might care about these sequences. First, de Bruijn sequences are by definition full period since they contain every possible n-bit state of an FSR. They also have guaranteed high linear complexity. It can be shown that the linear complexity of a de Bruijn sequence is at least half of its length, which is exponential in n [Li+15]. Furthermore, there are many ($2^{2^{n-1}-n}$, to be precise) de Bruijn sequences of order n [Li+15], so theoretically it is possible to generate lots of such sequences. The idea then is to try to construct a de Bruijn sequence from the output of an FSR, which will ensure that the FSR satisfies these aforementioned nice properties. But the difficulty is in this last point: Currently we only know how to construct a small fraction of the $2^{2^{n-1}-n}$ possible de Bruijn sequences of order n via various ways of constructing FSRs [Li+15].

Here we discuss some known methods of generating de Bruijn sequences. We have already seen that we can generate linear sequences of length $2^n - 1$ via a full-period LFSR. In particular, a such LFSR in the Fibonacci configuration yields a cyclic sequence of length $2^n - 1$, just 1 shy of the desired de Bruijn sequence. One way to resolve this is to note that there exists some chain of $n-1$ zeros in the LFSR sequence. So we can just artificially add a single zero to the end of that chain (filling in the all-zero state) to complete the de Bruijn sequence.

But this is still mostly linear! We were looking for nonlinear sequences, and this naive construction does not quite suffice. This is where various ideas of constructing NLFSRs from existing LFSRs come in. We'll mainly discuss two such methods: cross-join pairs and cascade products.

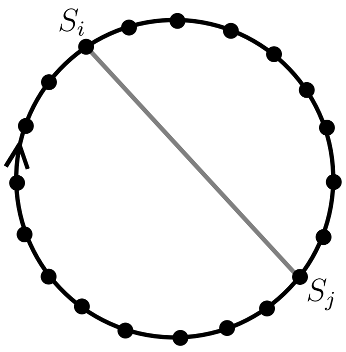
3.4.1 Cross-Join Pairs

Definition 8 (Conjugate)**.** The *conjugate* of an n-bit state $S = (s_0, s_1, \ldots, s_{n-1})$ is

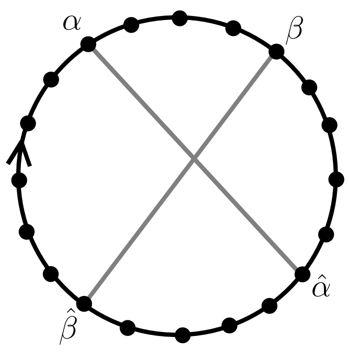
$$\hat{S} = (s_0 \oplus 1, s_1, \ldots, s_{n-1}),$$

i.e. S with the s_0 bit flipped.

We also define a *chord* on a cycle C of n states $S_1, S_2, S_3, \ldots, S_n$ to be a pair of states (S_i, S_j) on the cycle. The *successor* of a state S_i is S_{i+1}, the state directly following it [HK91; Dub13].

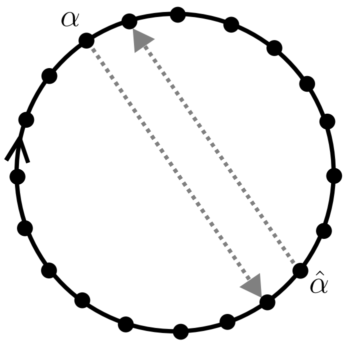


(a) Chord (S_i, S_j) on a cycle graph.

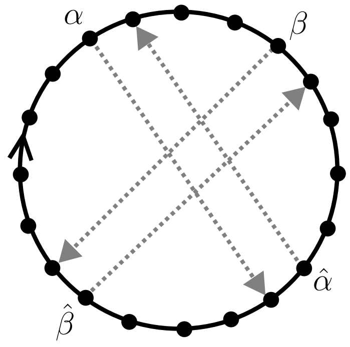


(b) A cross-join pair $(\alpha, \hat{\alpha}), (\beta, \hat{\beta})$.

Figure 2: Chords and cross-join pairs.



(a) Exchanging the successors of α and $\hat{\alpha}$.



(b) Exchanging the successors of β and $\hat{\beta}$.

Figure 3: Process of cross-joining on $(\alpha, \hat{\alpha}), (\beta, \hat{\beta})$.

We can visualize a chord as a chord of the circle (see Figure 2a) if we think of a cycle via its cycle graph. (Imagine that each dot is a state.) Then a *cross-join pair* is a pair of two chords $(\alpha, \hat{\alpha})$ and $(\beta, \hat{\beta})$ that intersect each other. See Figure 2b for an example.

To use a cross-join pair, we first take $\alpha, \hat{\alpha}$ and exchange their successors. This splits the original cycle into two disjoint cycles. Next, we take β and $\hat{\beta}$ and also exchange their successors. This joins the two disjoint cycles back into a single (nonlinear!) one. See Figures 3a and 3b for an example of this procedure.

The point of such an operation is to preserve the de Bruijn property. Recall that a de Bruijn sequence contains every 2^n possible n-tuple in a cycle of length 2^n, which requires every adjacent pair of states to have $n-1$ bits in common. In particular, if a state S_i looks like

$$S_i = (x_0, x_1, \ldots, x_{n-2}, x_{n-1}),$$

then its successor state S_{i+1} must look like

$$S_{i+1} = (x_1, x_2, \dots, x_{n-1}, x_n)$$

for some new bit x_n. The key observation here is that bits $x_1, \dots, x_{n-1}$ occur in both states. Now suppose we have a conjugate pair S_i and $\hat{S}_i$ and they look like

$$S_i = (x_0, x_1, \dots, x_{n-1}), \quad \hat{S}_i = (x_0 \oplus 1, x_1, \dots, x_{n-1}).$$

Then their successors must look like

$$S_{i+1} = (x_1, \dots, x_{n-1}, x_n), \quad \hat{S}_{i+1} = (x_1, \dots, x_{n-1}, x_n \oplus 1).$$

The last bit of $\hat{S}_{i+1}$ must be the negation of the last bit of S_{i+1} since each n-bit state appears uniquely in a de Bruijn sequence and clearly $S_{i+1} \neq \hat{S}_{i+1}$. Now we can observe that swapping the successors S_{i+1} and $\hat{S}_{i+1}$ still results in a de Bruijn sequence, since the $n-1$ bits in common property is preserved.

Note that in some sense, this new sequence obtained after cross-joining is “more nonlinear” than it was before. Furthermore, we can go on and repeat this procedure many times (there are $(2^{n-1}-1)(2^{n-1}-2)/6$ cross-join pairs for any given maximum-length sequence produced by an n-bit Fibonacci LFSR [HK91]) until we reach the desired level of nonlinearity.

To demonstrate the power of cross-join pairs, it has been shown that we can in fact get to any de Bruijn sequence of order n by repeatedly performing cross-join operations on a known de Bruijn sequence of order n [MS13]. However, the proof given by the authors is not constructive, meaning that although we know such a sequence of cross-join operations exists, we do not currently have a way of explicitly finding it. So this result is more theoretically interesting than it is practical.

3.4.2 Cascade Products

Here we discuss another form of constructing NLFSRs using LFSR structures that output de Brujin sequences—using Cascade Products. We desire de Brujin sequences due to their property of providing full period, which in turn results in a high linear complexity for every LFSR used in our NLFSR construction of cascade products. In the context of this construction, several terms must be defined:

Definition 9 (Feedback function for cascade products)**.** Given a new output bit s_n for a shift register based on previous inputs an initial state S defined with n bits $(s_0, s_1, ..., s_{n-1})$, a feedback function $f_1(x_0, x_1, ..., x_{n-1})$ outputs the next bit s_n given the n bits in the input state S.

Definition 10 (Characteristic function)**.** A characteristic function, then, for some sequence of states s outputted by a feedback shift register that works recursively on the bits of an input state s, as mentioned definition 3 above, is some $f(x_0, x_1, ..., x_n)$ such that

$$f(x_0, x_1, ..., x_n) = x_n + f_1(x_0, x_1, ..., x_{n-1}).$$

When working with linear feedback shift registers, one of the two key properties that ensure ease of analysis and construction is the periodicity they provide, each of which are ensured by the definition of the feedback functions used above. However, another key property of these update feedback functions is that they are non-singular, which depends on the periodicity of the LFSR itself [WTQ22].

Definition 11 (Non-singular feedback function)**.** A non-singular feedback function is one such that for some feedback function f_1, the function f_1 can be represented as

$$f_1(x_0, x_1, ..., x_{n-1}) = x_m + h_1(x_0, x_1, ..., x_{n-1})$$

where h_1 is a boolean function. Here, h_1 must have domain $\mathbb{F}_2^{n-1}$. A feedback function is considered non-singular if and only if the outputting sequence (of n states) for every register is periodic.

We define a *sequence* in this discussion of cascade products to be a *set of consecutive states* of an LFSR. In the case of LFSRs, unique initial states used produce unique sequences of n bits upon n clocks or updates of the LFSR. We denote by $\Omega(f)$ the set of all of these sequences, starting at the 2^n initial states that exist. It is important to note that these sequences of states can share the same cycles, but they are still noted to be distinct because of the different initial states they start at.

As different sequences in the set $\Omega(f)$ are a part of the same cycle when operated on by a left shift to align the two sequences, we denote by $[s_i]$ the set of sequences corresponding to the ith distinct cycle of an LFSR. With this, we can define what the *cycle structure* of such an LFSR is:

Definition 12 (Cycle structure)**.** If the set $\Omega(f)$ has k distinct cycles, then $\Omega(f)$ has a cycle structure that can be expressed as the union of these k cycles: $\Omega(f) = [s_1] \cup [s_2] \cup ... \cup [s_k]$.

As an example, in the case of $k = 1$ above, we would obtain just one cycle that would be of full period, and hence be of order n in the size of the LFSR.

By definition, then, for $k = 1$, the corresponding cycle that exists (the only one in the case of $k = 1$) forms a de Brujin sequence. This is because only one cycle exists, which means the resulting sequence $[s_0]$ corresponding to this cycle would contain all 2^n states, and hence contain 2^n n-tuples that form a full period as they each appear once along the chosen sequence $[s_0]$.

In the context of considering the linear complexity of such an LFSR, this means that a high linear complexity of 2^n results. Intuitively, this means that as the number of distinct cycles k increases in the cycle structure of the LFSR, the linear complexity of the LFSR decreases, as, on average, each cycle is likely to be smaller in size. By the analysis of linear complexity based on the Berlekamp Massey algorithm, it follows then that any sequence of output bits would, on average, tend to produce a linear recurrence dependent on fewer bits in the previous state. Since linear complexity is directly proportional to the number of bits of the previous state the next state depends on, a construction with more cycles will therefore be likely have lower linear complexity on average.

As such, in order to achieve a higher linear complexity for each LFSR, the goal is to combine existing cycles and produce a new cycle structure for the LFSR such that $k = 1$ and the resulting LFSR is characterized by a singular set of sequences $[s_0]$ such that any chosen representative sequence of the set $[s_0]$, where each sequence is shift equivalent to every other, is a de Bruijn sequence.

Now that we have a concrete way of denoting the cycle structure of an LFSR, we can find ways to make the outputted sequences of bits less linear by obfuscating the sequence of states outputted through interlacing cycles with one another, retaining the sequence of states within each cycle but combining cycles together in a reordered fashion.

We do this through the process of *cycle joining*. However, we first revisit the definition of predecessors and successors in the context of sequences of states in $\Omega(f)$ for some characteristic function f defining the update of an LFSR. For some state

$$s_i = (s_1, s_2, \dots, s_n)$$

The predecessor must look like:

$$s_{i-1} = (s_0, s_1, \dots, s_{n-1})$$

And the successor state would look like

$$s_{i+1} = (s_2, s_3, \dots, s_{n+1})$$

Definition 13 (Cycle joining)**.** Given two distinct cycles, $[s_j]$ and $[s_k]$ in $\Omega(f)$ such that $[s_j]$ starts with the some state $a_j = [a_0, a_1, \dots, a_{n-1} + 1)$, and $[s_i]$ starts with some successor state to a_j such that $a_i = [a_0, a_1, \dots, a_{n-1})$, interchanging the *predecessors* of a_i and a_j can allow these two cycles to be joined.

This process of cycle joining is similar to the "joining" process illustrated in Figure 3b above for cross-join pairs. The difference is that cross-join pairs have a "splitting" step, where an existing sequence of states needs to be split into two separate cycles. This step is the following by the joining step described here as cycle joining.

Example of Cycle Joining. As an example of the process of cycle joining, we can consider a 5-bit LFSR with two hypothetical distinct cycles. Given two cycles of size 5, $[s_0]$ and $[s_1]$, we must choose some state in $[s_1]$ that is a conjugate of a state in $[s_0]$ (recall the earlier definition of a conjugate). Let $[s_0]$ be denoted with the starting state $v_0 = (1, 0, 1, 1, 0)$ and $[s_1]$ is denoted with the starting state $v_1 = (1, 0, 1, 1, 1)$ for some 5-bit LFSR F, where v_1 is indeed a conjugate of v_0.

- Let the predecessor of state v_0 be $p_0 = (0, 1, 1, 0, 0)$

- Let the predecessor of state v_1 be $p_1 = (0, 1, 1, 1, 0)$
- Then the cycles with initial states v_0 and v_1 can be joined in the order:

$$v_0, p_1, \ldots, (\text{complete cycle of } p_1 \text{ back to } v_1), \ldots, v_1, p_0.$$

For k cycles, then, repeating this process of cycle joining $k-1$ times would result in a final cycle that produces a de Bruijn sequence of order n (as mentioned above in the motivation for cycle joining). The resulting FSR created after the joining of cycles of some LFSR defined initially by a characteristic function f can then be defined by the updated characteristic function:

$$f' = f(x_0, x_1, ..., x_n) + \prod_{i=1}^{n}(x_i + v_{i-1} + 1)$$

These changes to the characteristic function of the newly generated FSR after cycle joining are due to combining cycles of two states that are conjugates, and that each exist between two different cycles. Note that the arrangement above is only for the combination of two cycles. While we could go further into an analysis of what this would look like mathematically for $k-1$ joins, we refrain to as the basic linear complexity properties resulting from cycle joining (a higher linear complexity achieved) and the resulting de Bruijn sequence from cycle joining is the focus of our analysis and comparison to CMPRs.

Note that while it is very likely that the newly generated FSR is non-linear, there is a chance that it is still linear after cycle joining. Hence, we refrain from specifying whether it is an NLFSR or an LFSR. Instead, we refer to it as simply an FSR. To construct an NLFSR for certain, we proceed with creating the *cascading product* of two or more such FSRs.

Cascading Product Construction. We define the cascading product between an FSR of n bits defined by the characteristic function f and another FSR of m bits defined by the characteristic function g to be represented by the operation:

$$f * g = f(g(x_0, x_1, ..., x_m), g(x_1, x_2, ..., x_{m+1}), ..., g(x_n, x_{n+1}, ..., x_{n+m}))$$

Below is an illustration of how these two FSRs are cascaded together.

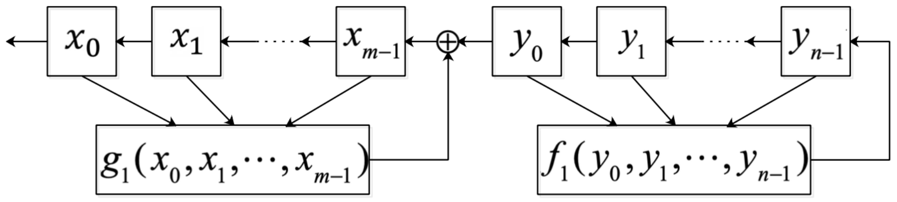

x_0
x_1
x_{m-1}
y_0
y_1
y_{n-1}
$g_1(x_0, x_1, \cdots, x_{m-1})$
$f_1(y_0, y_1, \cdots, y_{n-1})$

Figure 4: A cascade product of FSR defined by characteristic function f and FSR defined by characteristic function g obtained from [CGW20].

The registers $y_0, y_1, ..., y_{n-1}$ can be thought of as analogous to $x_m, x_{m+1}, ..., x_{m+n-1}$, aligning with the above previously defined notation of the cascading product. The construction above operated by shifting the values in registers $x_0, x_1, ...x_{m-1}$ forward on every update, such that after the first update, the value of x_0 would be the previous value in x_1, that is x_1 would be the previous value in x_2 and so on while that in x_{m-1} would be the value obtained using the characteristic function g and operating with register y_0 noted as being equivalent to x_m.

3.4.3 Note on Product-Feedback Shift Registers

There is existing literature on so-called *product-feedback shift registers* from the 1960-1970s by Green and Dimond [GD70], which can cause confusion as our product register has a very similar name.

However, our construction is noticeably different: We define a product register as a register that updates using a hardware implementation of *Galois field multiplication.* Their construction of a product-feedback shift register takes two characteristic polynomials $f(x)$ and $h(x)$ and takes their *modulo-2 product* to get a resulting characteristic polynomial

$$h(x) = f(x) * g(x) = \sum_{i=0}^{n} (f_i \wedge g_i)x^i,$$

which corresponds to a new NLFSR in hardware that looks something like a cascade product.

There is actually a striking resemblance between the cascade product and the CMPR, at least with how they look in hardware diagrams (both make use of some sort of “chaining”). However, in any case, any similarity is with the CMPR and the product register is clearly a completely different construction from previous work.

3.5 Algebraic Attacks/Prior Cryptanalysis Work

In prior work, the team working with CMPRs have implemented a range of techniques and attacks to determine and fortify the security of the CMPR. Before each attack, the team was provided with the internal structure of the CMPR in question, and N output keystream bits after clocking the CMPR a certain number of times.

Among the attacks that have been conducted include been brute force, or an exhaustive key search. In this attack, the team would try all possible keys, which in this case, are the initial internal states of the MPRs composing the CMPRs [BAS12]. This attack is, of course, computationally expensive and serves to set a benchmark for what is considered an attack: any attacks with greater computational complexity than brute force are not attacks. In the case of CMPRs, it depends on what the key is, but the complexity is likely $O(\text{period})$.

Another branch of attacks investigated extensively are algebraic attacks. The algebraic attack and modifications of it (fast algebraic attacks, for instance) model the CMPR system in terms of algebraic equations [BAS12]. This is done by modelling keystream bits in terms of initial state of the CMPR. These algebraic equations are expressed in *algebraic normal form* (ANF).

Definition 14 (Algebraic normal form)**.** A method of writing propositional logic where clauses are a set of terms operated on with ANDs between each other, after which each of the clauses is operated with an XOR between one another. Negations are not allowed in this notation. Note that any boolean function can be uniquely represented by ANF.

Given more variables than equations, it is possible to over-constrain the options for variables that appropriately solve the entire system of equations, as the set of system then becomes extensively linearly independent. Given enough of these equations then, we are able solve for initial state using Gaussian elimination, which takes $O(n^3)$ time in n as the number of equations. A variation of the Algebraic Attack includes the Fast Algebraic Attack, which involves a recombination step to reduce the number of equations in the system before solving to reduce the time complexity of the entire attack [Cou03].

3.5.1 Example Equations for Algebraic Attack

Below are the first two of a set of 17 equations used by the previous attack team on the 17-bit CMPR:

$$\begin{aligned}
2c_0[1] &= 1 \oplus c_1[0] \oplus c_6[0] \oplus c_0[0] \oplus (c_2[0]c_3[0]c_7[0]c_{13}[0]),\\
c_0[2] &= 1 \oplus c_1[1] \oplus c_6[1] \oplus c_0[1] \oplus (c_2[1]c_3[1]c_7[1]c_{13}[1])\\
&= c_3[0] \oplus c_2[0] \oplus c_5[0] \oplus c_7[0] \oplus c_1[0] \oplus (c_4[0]c_9[0]c_{11}[0]c_{14}[0])\\
&\quad \oplus (c_2[0]c_3[0]c_7[0]c_{13}[0]) \oplus (c_2[1]c_3[1]c_7[1]c_{13}[1]).
\end{aligned}$$

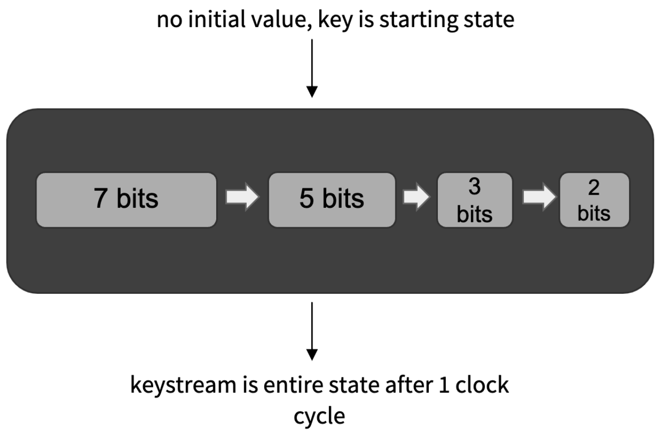
The second equation leaves four bits in terms of the first clock cycle. After multiplying the composed polynomials, the second equation has 157 terms.

In order to solve equations like those above, we make use of linearization, which involves converting the original ANF such that only linear operators remain, and any clauses that are ANDed and produce nonlinear terms are combined into variables. Linearizing the above equations would lead to the following final set:

$$\begin{aligned}
c_0[1] &= 1 \oplus c_1[0] \oplus c_6[0] \oplus c_0[0] \oplus (c_{2,3,7,13}[0]),\\
c_0[2] &= 1 \oplus c_1[1] \oplus c_6[1] \oplus c_0[1] \oplus (c_{2,3,7,13}[1])\\
&= c_3[0] \oplus c_2[0] \oplus c_5[0] \oplus c_7[0] \oplus c_1[0] \oplus (c_{4,9,11,14}[0])\\
&\quad \oplus (c_{2,3,7,13}[0]) \oplus (c_{2,3,7,13}[1]).
\end{aligned}$$

3.5.2 Attack Set Up

In the equations above and as further reminder of notation, the value of the CMPR C at bit i and time t is denoted $c_i[t]$. The illustration below demonstrates ther conditions and configuration of the CMPR used for the attacks throughout this paper. For the purposes of the algebraic attack conducted, no initialization value was given for the algebraic attacks used, and the key provided was the initial state of the CMPR. The output keystream utilized represented the entire state after *1 clock cycle*.



no initial value, key is starting state
7 bits
5 bits
3 bits
2 bits
keystream is entire state after 1 clock cycle

Figure 5: CMPR set up used for attacks

A key limitation of linearization attacks (a subset of algebraic attacks) is the number of variables one is solving for. If n is the number of variables present in nonlinear terms, we have that the number of new variables grows exponentially in terms of n.

This emphasizes the importance of bounding the number of nonlinear terms. Past results suggest this is not possible. [Gor+21]

3.6 Berlekamp-Massey

Proposed by James Massey as a simplification of an existing algorithm by Elwyn Berlekamp, the Berlekamp-Massey algorithm solves for a linear recurrence given a set of output keystream bits that corresponds to the shortest Fibonacci LFSR that can produce that sequence of output keystream bits. Recall that a general recurrence may look like

$$s_{n+1} = \sum_{i=n-k}^{n} c_i s_i.$$

On a high level, the Berlekamp-Massey algorithm updates previous iterations of "guesses" of what the shortest linear recurrence may be based on the next keystream bit observed by the algorithm, correcting errors in the recurrence by using a linear combination of the current linear recurrence and the last failed linear recurrence. The following steps roughly describe the algorithm:

1. Initialize an error $d_{\text{prev}} = 1$, a current guess $s_{n+1} = 1$ at the recurrence, and a current guess $L(s) = 0$ for the linear complexity.

1	0	1	2	4	11	19	d	$L(s)$
1							1	0
	1						0	0
		1					1	0
-1	0	1					0	2
	-1	0	1				2	2
	-1	-2	1				0	2
		-1	-2	1			-1	2
	-0.5	-1	-1.5	1			0	3
		-0.5	-1	-1.5	1		2.5	3
		-3	-6	1	1		0	3
			-3	-6	1	1	0	3

Table 2: The Berlekamp-Massey algorithm on the sequence $(0, 1, 2, 4, 11, 19)$.

2. Shift the guess at each step and compute $d = a_{n+1} - \sum_{i=n-L(s)}^{n} c_i a_i$, where a_i are the bits in the input sequence.
3. If $d \neq 0$, then correct the current guess by subtracting d/d_{prev} times the last failed guess. Update $L(s)$ if necessary.

An interesting property of the Berlekamp-Massey algorithm is that it can recover a linear recurrence of order n using only roughly $2n$ terms [Ber66].

Although we will primarily use Berlekamp-Massey on binary sequences coming from a LFSR, the algorithm works equally as well on sequences from any field. We will provide an example of Berlekamp-Massey on a sequence of rational numbers, since we feel that most readers will find the addition and multiplication of rational numbers more familiar than the modular arithmetic of finite fields.

Suppose we have the order 3 sequence defined by

$$a_{n+1} = -a_n + 6a_{n-1} + 3a_{n-2}$$

and initial state $(a_0, a_1, a_2) = (0, 1, 2)$. Then the first few terms are

$$0, 1, 2, 4, 11, 19, \ldots.$$

The Berlekamp-Massey algorithm says that these first 6 terms are enough to recover our original recurrence. Refer to Table 2 for a summary of the example.

We will walk through the first few rows of the table here. First in the first (non-header) row we initialize the algorithm with $d_{\text{prev}} = 1$, $L(s) = 0$, and an initial guess of $s_{n+1} = 1$. Sliding this single 1 one to the right brings us to our first calculation for d. To do this, we can line up our current row (which is the second row currently) with the header, and compute the sum of the pairwise products:

$$d = 1(0) = 0.$$

Here $d = 0$, so there is no need to correct anything. Sliding the 1 one more to the right gives an error of

$$d = 1(1) = 1,$$

so we must correct. We subtract by $d/d_{\text{prev}} = 1/1 = 1$ times the previous error row to get

$$(0, 0, 1) - 1(1, 0, 0) = (-1, 0, 1)$$

Note that $L(s)$ has increased in this case, though this will not necessarily always happen at every correction. Also note that the linear complexity $L(s)$ is one less than the length of our guess. We can proceed similarly for the remaining iterations, keeping in mind that calculations for d now will become a sum of multiple products (unlike the single product we have had up until now). By the end, we can see that we recover the recurrence

$$a_n + a_{n-1} - 6a_{n-2} - 3a_{n-3} = 0,$$

reading the coefficients from right to left. We can see that this result indeed matches the initial recurrence we started with.

4 Linear Complexity

4.1 Linear Complexity Motivation

Now that we have developed an intuition for what previous constructions of product registers are, we have the background to explain the motivation behind pursuing CMPRs for their high periodicity. While LFSRs are less secure compared to NLFSRs [Qi07], they can be reconstructed in $O(n^2)$ time via Berlekamp-Massey. NLFSRs, however, are typically harder to build compared to LFSRs [Qi07]. Since a main goal of this research as mentioned previously was to construct a lighweight but secure cryptographic primitive, this research approaches this problem by trying to achieving properties of NLFSRs with LFSRs through CMPRs.

To define a metric for the security of a product register construction, we rely on the idea of *linear complexity*.

Definition 15 (Linear complexity)**.** Given some random infinite binary sequence, the linear complexity (denoted $L(s)$) is the length of the shortest LFSR that can produce that sequence as a contiguous set of output bits.

CMPRs help achieve the security from the nonlinearity of NLFSRs (CMPRs have a full period) [Rue86]. High linear complexity of an input register sequence indicates how secure the cipher is [Ros09] by indicating how improbable it is to get the sequence fed in as input (using LFSRs).

4.2 Root Counting

While the Berlekamp Massey Algorithm provides a reliable framework for estimating the linear complexity of LFSRs and MPRs as they have linear complexities that are just the length of shortest LFSR that leads to the same periodically repeating output, CMPRs are expected to have larger linear complexities due to obfuscations that make their output sequences act in ways similar to NLFSRs. The key question that arises based on this is:

How do we estimate the linear complexity of a CMPR, given chaining with logic gates between MPRs and LFSRs?

On a high level, the linear complexity of an LFSR is equal to the length of the shortest LFSR that has the same outputs, which is equivalent to the degree of the primitive polynomial P that defines the equivalent LFSR which is equivalent to the number of roots of that primitive polynomial P.

Our approach to estimating the total linear complexity of the CMPR relies on the structure of the CMPR, and how it chains component MPRs, for which we are able to find the linear complexity. Each MPR is chained with either and XOR or an AND gate (indicated in red in the diagram below). Hence, we approach this problem by finding the linear complexity of individual MPRs and derive method of root counting wire by wire for chaining functions applied between MPRs (red wires in Figure 6).

4.2.1 Root Counting – Terminology

To proceed with root counting, we need a larger background in group theory, and an understanding of how chaining with logic gates affects roots of different MPRs' primitive polynomials as they are chained across different fields (and the same ones).

Definition 16 (Polynomial ring)**.** The polynomial ring GF$(p)[x]$ is the set of all polynomials over the field GF(p), with operations of addition and multiplication in polynomials.

The characteristic polynomials of any given LFSR in GF(2) also exist in a polynomial ring. Further note that a polynomial $P(x)$ in GF$(p)[x]$ is referred to as *irreducible* if $P(x)$ has no non-constant polynomial divisors. This means that for all integers k between 1 and $n-1$ inclusive, no polynomials of order k exist such that they divide $P(x)$.

This background regarding *irreducible* polynomials is key to answering the question of when

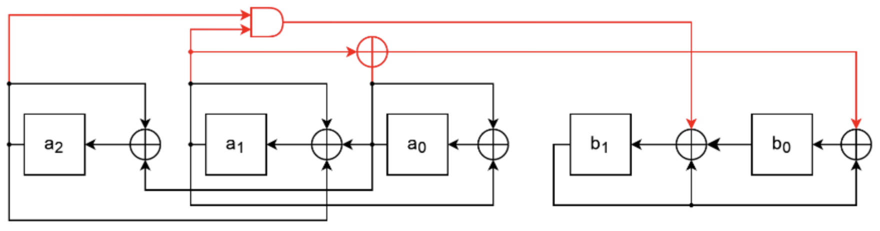


Figure 6: 5-bit CMPR with a 3-bit MPR and 2-bit MPR chained with AND and XOR functions

a characteristic polynomial for an LFSR yields a full period? Recall that the property of a full period is key to ensuring a higher linear complexity of the individual MPRs used in our CMPR construction, and as explained in the background about MPRs, their construction guarantees full period. One condition that such MPRs satisfy, then is that their characteristic polynomials are irreducible and yield a long cycle by achieving a high period.

The maximum size of any periodic state sequence for an MPR is $2^n - 1$, since at any point, the 0 state will simply lead to itself, hence it is excluded. Any characteristic polynomial $P(x)$ has a period of the smallest k such that $P(x)$ evenly divides $x^k + 1$. As such, the characteristic polynomial for a full period LFSR (or MPR in this case) must divide $x^k + 1$ where $k = 2^n - 1$.

Knowing now of at least one polynomial which must divide the characteristic polynomial of some n-bit MPR (namely $x^{2^n-1} + 1$), we may proceed with further group theory background about extension fields and splitting fields for root counting.

Extension Fields Given that GF(p) is a field contained in GF(p^n), that means GF(p^n) is an extension field of GF(p) [Rot95]. We provide an example of an extension field:

> Let $p = 2$, $n = 3$. Then GF(2) is the field with elements: $\{0, 1\}$ and GF(2^3) is a field with 8 elements: $\{0, 1, x, x + 1, x^2, x^2 + 1, x^2 + x, x^2 + x + 1\}$.

Furthermore, a splitting field K for some polynomial $f(x)$ over a field F is the smallest extension field in which $f(x)$ can be factored as a set of linear factors present in that extension field. An example of this can be illustrated if we let $\mathbb{F}$ be the field of rational numbers $\mathbb{Q}$.

Consider the polynomial $h(x) = x^3 - 2$ over $\mathbb{Q}$. The roots of this polynomial are the cube roots of 2, namely, $\sqrt[3]{2}$, $\sqrt[3]{2}\omega$ and $\sqrt[3]{2}\omega^2$, and where ω is a complex cube root of unity (a primitive cube root of unity). As a result we can conclude that the field $\mathbb{Q}(\sqrt[3]{2}, \omega)$ is the splitting field for $h(x)$ over $\mathbb{Q}$.

Definition 17 (Cyclotomic coset)**.** The set of integers of the form $\{k2^i\}, i \geq 0$, k a positive integer, when reduced modulo $2^n - 1$.

The motivation behind using cyclotomic sets is that they provide groups of roots can be seen by noticing that in GF(2), that for any polynomial P, $P(x^2) = P(x)^2$. The result of this is if $P(\beta) = 0$, then $P(\beta)^2 = 0$, which by the property discussed means $P(\beta^2) = 0$. In this way, we are able to partition the roots $\alpha^i \in 2^n - 1$ belonging to every primitive polynomial P in a certain polynomial ring of GF(p)[x] into sets of roots. This enables grouping of the roots into sets so that, based on the primitive polynomials of each of the MPRs, we can determine the number of roots corresponding to each MPR.

The procedure for determining the cyclotomic cosets based on the primitive polynomials in a certain polynomial ring GF(a)[x] is described as follows:

1. Take $A(x)$ as an irreducible polynomial in GF(a)[x]. As such, it does not have a root in GF(a), so we proceed by looking for an extension field for GF(a) that may contain a

root of $A(x)$.

2. Create a splitting field E of $\mathrm{GF}(a)$, such that E is the smallest field containing every root of $A(x)$.

   If all roots in $A(x)$ are generated in such a splitting field E, then $A(x)$ is a primitive polynomial, and the roots generated create a cyclotomic set.

This allows us to form a cyclotomic coset of the form:

$$\{C_k(n) = a^{ik \bmod n} \mid \gcd(i, n) = 1\}$$

For the case of operating in $GF(2)$ and with MPRs for which we know, by virtue of having n bits and having full period, have a period at a maximum of 2^n in length, this implies that i in the above equation only need be considered for $i \in (0, 2^n - 1)$:

$$\{a^{k \bmod 2^n-1}, a^{2k \bmod 2^n-1}, a^{4k \bmod 2^n-1}, \ldots, a^{2^{n-1}k \bmod 2^n-1}\}.$$

4.2.2 Example of Cyclotomic Coset for a 3-bit MPR

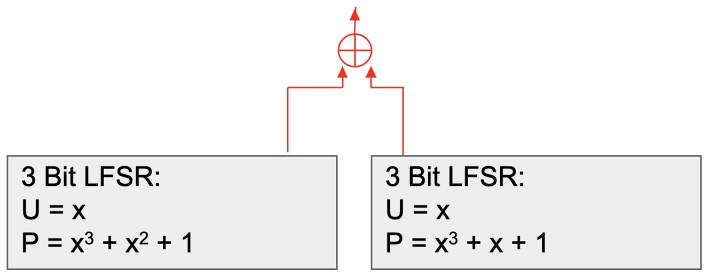
Working backwards, we can obtain the primitive polynomials and their corresponding cosets using the following formula for how roots are partitioned into cyclotomic cosets (only for GF(2)) for an irreducible polynomial. If we use an LFSR of length 3 such that $n = 3$, where k is the nth root we are considering, then only two cosets exist, each corresponding to one primitive polynomial for the field $\mathrm{GF}(2^3)$. Only two cosets for $\mathrm{GF}(2^3)$, corresponding to $x^3 + x + 1$ and $x^3 + x^2 + 1$ respectively in order of appearance in the below sequence:

$$\begin{aligned}
\text{When } k = 1: &\quad \{\alpha^{1 \bmod 2^3-1}, \alpha^{2 \bmod 2^3-1}, \alpha^{4 \bmod 2^3-1}, \ldots, \alpha^{4 \bmod 2^3-1}\} = \{\alpha, \alpha^2, \alpha^4\},\\
\text{When } k = 2: &\quad \{\alpha^{2 \bmod 2^3-1}, \alpha^{4 \bmod 2^3-1}, \alpha^{8 \bmod 2^3-1}, \ldots, \alpha^{8 \bmod 2^3-1}\} = \{\alpha^2, \alpha^4, \alpha\},\\
\text{When } k = 3: &\quad \{\alpha^{3 \bmod 2^3-1}, \alpha^{6 \bmod 2^3-1}, \alpha^{12 \bmod 2^3-1}, \ldots, \alpha^{12 \bmod 2^3-1}\} = \{\alpha^3, \alpha^6, \alpha^5\},\\
\text{When } k = 4: &\quad \{\alpha^{4 \bmod 2^3-1}, \alpha^{8 \bmod 2^3-1}, \alpha^{16 \bmod 2^3-1}, \ldots, \alpha^{16 \bmod 2^3-1}\} = \{\alpha^4, \alpha, \alpha^2\},\\
\text{When } k = 5: &\quad \{\alpha^{5 \bmod 2^3-1}, \alpha^{10 \bmod 2^3-1}, \alpha^{20 \bmod 2^3-1}, \ldots, \alpha^{20 \bmod 2^3-1}\} = \{\alpha^5, \alpha^3, \alpha^6\},\\
\text{When } k = 6: &\quad \{\alpha^{6 \bmod 2^3-1}, \alpha^{12 \bmod 2^3-1}, \alpha^{24 \bmod 2^3-1}, \ldots, \alpha^{24 \bmod 2^3-1}\} = \{\alpha^6, \alpha^5, \alpha^3\}.
\end{aligned}$$

This leads to the sets of roots to consider for both possible primitive polynomials of of a 3-bit MPR being $\{\alpha, \alpha^2, \alpha^4\}$ and $\{\alpha^6, \alpha^5, \alpha^3\}$.

Definition 18 (Coset weight)**.** The coset weight of a set of roots $\{\alpha^k, \alpha^{2k}, \alpha^{4k}, ...\}$ to be the hamming weight of any exponent $\{k, 2k, 4k, ...\}$ when written in binary. It is the number of 1s in the binary representation of the exponents of the elements in each set.

This definition of a coset weight is always the same for each exponent in the same set. The concept of coset weights becomes particularly important when we consider chaining MPRs with AND gates when they exist in the same field. However, we have now covered the background to be able to use the roots in these cyclotomic cosets to consider every case of chaining between MPRs:



$$c[t] = a[t] \oplus b[t] = \sum_{i=1}^{2^n-1} A_i(\alpha^i)^t \oplus \sum_{i=1}^{2^n-1} B_i(\alpha^i)^t = \sum_{i=1}^{2^n-1} (A_i \oplus B_i)(\alpha^i)^t$$

Figure 7: CMPR with 2 3-bit MPRs being XORed

1. *XOR Gates operating between two MPRs.* Combining sequences with an XOR gate leads to the resulting sequence containing the union of the roots of the initial sequences. This means, for root counting, we add the number of roots in the primitive polynomial of one MPR with that of another to produce the new linear complexity. An example of this can be constructed for the 3-bit MPRs we considered earlier in our example of finding cylotomic cosets for 3-bit MPRs. We know the linear complexity of two 3-bit MPRs chained together is the number of roots of each MPR added together. The union of the two different cosets of size 3 (corresponding to the two different degree 3 primitive polynomials), then, yields that the linear complexity of the output sequence is of size $3 + 3 = 6$.

2. *AND Gates operating between 2 MPRs in different fields.* In the case of chaining a 2-bit MPR A, say to a 3-bit MPR B, we are chaining across two different Galois fields of GF(2^2) and GF(2^3) respectively. The resulting sequence from combining two such sequences as above is that the roots of the new sequence are contained in the field with the exponent of the product of the exponents of the fields.

   In Figure 8, this would be GF$(2^{2 \cdot 3})$, or GF(2^6) would contain the roots for A AND B, and hence, A AND B would have linear complexity 6.

3. *AND Gates operating between 2 MPRs in the same field.* In the case of chaining a 2-bit MPR A to another 2-bit MPR B, we are chaining across two identical Galois fields, GF(2^2). As such, the resulting keystream or sequence of output bits from this CMPR cannot be represented as an LFSR with a primitive polynomial that has a number of roots (linear complexity) equal to the product of the exponents $(2 \cdot 2)$ of each respective Galois field. This is because roots combined in the same coset can exist in a different coset in the same field, which would lead to overcounting the number of roots in the output root representation of the construction (in this case a CMPR). One example of this is observed if some 3-bit MPR has root α and an MPR with the same primitive polynomial contains root α^2, both of which are in C_1, then the sequence formed by

their AND has root $\alpha \cdot \alpha^2 = \alpha^3$ which is in C_2.

The solution here is to make use of coset weights. After ANDing k sequences, the number of roots in GF(2^n) is $\leq \sum_{i=1}^{k} \binom{n}{i}$ according to analysis by Key in his paper regarding finding an upper bound for the linear complexity of filter and combination generators (two constructions using compositions of LFSRs to produce more non-linear output keystreams). [Key76]

After taking the termwise product as in the case of ANDing MPRs sequences in different fields, new exponents (and new cosets) may be generated, but none of these cosets have a weight greater than the sum of the weights of the two sequences added [Key76]. Key uses this concept of weight to tighten the upper bound on linear complexity for a filter and arrive at an upper bound for the linear complexity of a filter generator. This upper bound is the basis of our solution for estimating the linear complexity of two MPRs that have been ANDed together when they are in the same field.

The intuition behind this upper bound can be observed by realizing that multiplying two roots results in the sum of the exponents ($\alpha^a \cdot \alpha^b = \alpha^{a+b}$). Then, considering coset weight of the original exponents a and b, the coset weight of $a + b$ is bounded by the sum of the coset weights of a and b. As such, the output sequence must have a linear complexity that is upper bounded by the number of binary strings of length n with a hamming weight less than k, where k is the number of sequences being ANDed.

A small example of the way linear complexity is bounded for an AND between two output sequences of MPRs in the same field of GF(2^3) is shown below with a CMPR composed of outputs from the same 3-bit MPR.

In this example we are in GF(2^3) and are ANDing two wires—one which taps a_1 for output, and the other which taps a_0 for output. We can apply the Key bound on linear complexity to get

$$\binom{3}{1} + \binom{3}{2} = 6.$$

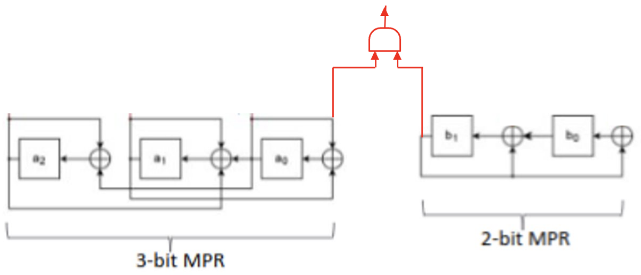


Figure 8: CMPR with a 2-bit and 3-bit MPR being ANDed

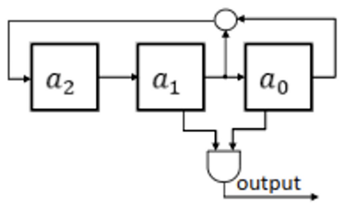


Figure 9: 2 3-bit MPRs used to create a CMPR (chained with an AND gate)

5 CMPR vs. de Bruijn Math

5.1 Basic Comparison of Frameworks

Recall that CMPRs (as described) work with Galois LFSRs whereas the de Bruijn-style constructions work with Fibonacci LFSRs. One key difference here is the lack of a clear group structure in Fibonacci LFSRs. This means that our proofs on the period and cycle structure of CMPRs will likely not apply to de Bruijn-style constructions, at least without significant modifications. We expect that the degree of modification required would be sufficiently large to clearly highlight the merits of studying each approach individually.

As for linear complexity, our analysis in this area is actually largely based on de Bruijn-style math (the use of root counting, for instance). As a result, most of these results will likely carry over to de Bruijn constructions, especially for the cascade product construction due to its seemingly high physical similarity in hardware to the CMPR. However, this is really not anything new for the de Bruijn constructions, since they already had guaranteed high linear complexity (as a result of being a de Bruijn sequence) anyway.

5.2 Discussion of Linear Complexity

Though bad examples exist, CMPRs generally have exponential linear complexity with high probability [GM23]. On the other hand, recall that a de Bruijn sequence provably guarantees a linear complexity of at least half of its period (which is exponential in its order). This result is not a probabilistic, so it is slightly stronger than the analogous result for CMPRs.

However, as previously discussed, there are constructions of de Bruijn sequences that are, in some sense, still largely linear. Take, for instance, the example where we appended a zero to the end of a chain of $n-1$ zeros from a full-period LFSR sequence. In this case, for its entire period other than the all-zero state, the de Bruijn sequence behaves completely linearly and thus exhibits very low linear complexity. The linear complexity then spikes to be exponentially high upon reaching this all-zero state.

This is highly problematic. For example, if we consider an modified LFSR (with the additional zero appended) that produces this type of length 2^n de Bruijn sequence, an attacker could determine the linear recurrence for the first 2^n-1 outputs by simply observing the first $2n$ outputs (via Berlekamp-Massey, for instance). Supposing that the all-zero state occurs at the very end, the attack will now be able to precisely predict each of the next $(2^n - 1) - 2n$ terms up until that all-zero state at the end. For a reasonable size like $n = 128$, this is the ability to predict the next

$$(2^{128} - 1) - 2(128) \approx 3 \cdot 10^{38}$$

output bits given only 256 bits initially. The general case is obviously not quite so bad and methods like cross-joining attempt to remedy exactly this problem. However, this is still definitely a potential flaw of de Bruijn constructions that we should be aware of.

In light of this, it might be more insightful to also look at an alternate metric, rather than just the raw linear complexity. For example, Berlekamp-Massey gives us the current linear complexity of a sequence at each step, so maybe graphing that would be helpful: A steady increase corresponds to "high" nonlinearity, whereas large spikes correspond to "low" or "artificial" nonlinearity. Recall that we expect roughly $i/2$ as the linear complexity at each step i (recall that Berlekamp-Massey needs $2n$ terms to recover a degree n recurrence). So maybe something like the variance of the linear complexity of a sequence s from $n/2$ could be a good metric to consider:

$$\mathrm{Var}_{n/2}[s] = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{i}{2} - L(s_{1,\ldots,i}) \right)^2 ,$$

where a low overall variance would be consider "better nonlinearity." Other statistical measures of spread could also be opportunities for further exploration.

6 Cryptanalysis

6.1 Cube Attacks

The cube attack is an algebraic attack developed by Dinur and Shamir to attack tweakable polynomials in GF(2) [DS08]. On a CMPR, the cube attack treats the register as a black box polynomial p; specifically, since the output bit in a CMPR is defined to be the 0th bit, we can take that bit as the black box polynomial. The initial state of the register is represented as the bits $x(0)[0], \ldots, x(n-1)[0]$. If we can tweak variables in the initial state (i.e., an IV), then we can setup a system of equations to solve for the initial state. Clocking the CMPR involves simply rewriting each bit as a polynomial of the previous bits, i.e. $x(i) = p(x(0)[t-1], \ldots, x(n)[t-1])$. To show how this can be applied to CMPRs, we first outline the attack using a general black box polynomial.

6.1.1 Cube Attacks Description

To do this, define

$$p \in \mathbb{P}_d^n$$

where $\mathbb{P}_d^n$ is a polynomial in GF(2) with n variables and degree bounded by d.

As an output, if (v_m), (x_n) are "private" (key) and "public" (tweakable, plaintext/IV) variables, given sufficient outputs of $p(v_1, \dots, v_m, x_1, \dots, x_n)$, recover $v_1, \dots, v_m$. Note for simplicity in notation we will indiscriminately define private and public variables as (x_n) in future examples.

Choose p. We know that p is a sum of products of variables $x(i)$. Since we are working in GF(2), the coefficients of each term must be 1. Thus, we denote an *index subset* $I \subseteq \{1, \dots, n\}$ such that t_I indexes the corresponding term in p (e.g., $t_{\{1,2\}} = x(1)x(2)$). This allows us to express any p as the sum of t_Is.

Arbitrarily fix I. Then factor t_I from p:

$$p(x_1, x_2 \dots, x_n) = t_I \cdot p_{S(I)} + q(x_1, x_2, \dots, x_n)$$

This defines two things: $p_{S(I)}$ is the *superpoly* of I in p, and q is the *remainder*.

Then it must be that $p_{S(I)}$ does not contain any common variable with t_I, since the degree of any variable in a term is bounded by 1. It must also be that q contains at least one variable x_i such that $i \notin I$. Furthermore, we define a *maxterm* of p to be a choice of t_I such that $deg(p_{S(I)}) = 1$. It turns out that if we can find sufficiently many maxterms, we can recover a system of linear equations that are solvable efficiently.

Recall that after factoring out t_I, we have

$$p(x_1, x_2 \dots, x_n) = t_I \cdot p_{S(I)} + q(x_1, x_2, \dots, x_n)$$

For an index subset I of size k, define the *cube* C_I as the set of 2^k vectors denoting the boolean assignments of variables in I. Choose $v \in C_I$. Restricting $p_{|v}$, we obtain a polynomial with at most $n - k$ variables. Now, let

$$p_I = \sum_{v \in C_I} p_{|v}$$

so that p_I is the sum of all p over all boolean combinations in C_I.

Theorem 2 (Key idea of cube attacks)**.** *For any polynomial p and subset of variables I, $p_I \equiv p_{S(I)}$.*

Proof. Let t_J be an arbitrary term in q. At least one variable in t_J is not in t_I; let this be x_i. x_i is summed an even number of times in p_I. All $v \in C_I$ zero t_I except for $v = \{1, \dots, 1\}$. This assigns 1 to all $p_{S(I)}$. $\square$

As a result, if we choose tweakable I such that t_I is a maxterm in p, then applying the cube attack allows us to recover the initial state by solving for a system of linear equations.

6.1.2 Application to CMPRs

To find index subsets I, a random subset of bits are chosen; this ensures that a uniform subset of the search space is searched. The 17-bit CMPR was treated as a blackbox polynomial p, and a random initial state was used as the secret bits $x[0], \ldots, x[16]$. From this, we extract a list of index subsets as well as their associated superpolys to solve linearly.

For each test on a potential I with k variables, p_I is computed by summing over the cube of 2^k vectors; the p_I equations are then solved as a system of linear equations. In our analysis, however, the set of monomials in $x[0], \ldots, x[16]$ do not sufficiently saturate the polynomial representing the output bit such that solving the superpolys recovers the entire initial state.

This problem is easy to see when the CMPR is insufficiently clocked. For instance, Appendix A describes SageMath code that implements the cube attack as described in [DS08], but does not clock the CMPR before running the cube attack. In this example, only bits $x[4], x[7], x[10], x[12], x[13], x[14], x[15], x[16]$ appear in any of the superpolys, which is insufficient to recover the entire state.

6.2 Correlation Attacks

Correlation attacks, as described in [Mei11], were classically made to target filter generators and combiner generators, which as previously noted are constructions by Key that make use of LFSRs to generate keystreams with non-linear outputs from the linear outputs of LFSRs used to construct the filter generator. The approach treats the system as a whitebox with similar known parts as with algebraic attacks, wherein the feedback connections of LFSRs (their primitive polynomials) are known, as well as the structure of the way they are combined to produce the final keystream. For any s LFSRs that are used as inputs to some boolean function generating an output keystream bit $k_i = f(x_1, x_2, ...x_s)$, correlation attacks exploit those keystream bits related to registers in an LFSR that harbor higher conditional probability of a certain output bit of the keystream being expected given the state of an output bit of a specific LFSR used to generate an output bit in the keystream. That is,

> $\Pr(k_i = z | x_1 = 1) = n$ for some $k_i = f(x_1, x_2, ...x_s)$, where n is some probability better than random chance ($n > 0.5$ in our case) for a useful result.

Generalizing these results to bits of the output keystream being dependent on the outputs of multiple LFSRs, correlation attacks propose a divide-and-conquer approach to solving for the initial states (and thereby full sequences) of each LFSR output that the output keystream bits show high correlation probabilities with, and then solving for all other LFSRs initial states using this information. The exhaustive search approach of trying different initial states for the LFSR incurs a time complexity of $N \cdot 2^n$ for 2^n initial states to each be checked for a high correlation (bias) factor by being XORed with every one of the N bits of the output keystream.

Instead of using an exhaustive search with such a high time complexity across all initial states of the LFSRs to find those that incur the highest probability of outputting every bit in the keystream, fast correlation attacks as proposed by Meier make use of parity checks through first developing parity check equations. [Mei11]

Definition 19 (Parity-check equations)**.** Relations developed for a given FSR construction (either composed of many FSRs or not) that connect output keystream bits to the internal state of the FSRs and are used to determine a correlation value by determining the number of satisfied relations with an expected output keystream bit for hypothesized internal states.

We start by finding one relation relating bits of the LFSR to one another based on the feedback polynomials of the LFSRs (as given in the attack). By squaring these relations, shifting them to the left or right, then even more relations can be formed between other bits in the LFSR. The more relations we are able to find between different input bits and the output keystream bit, the more criteria is present to determine how high the correlation between certain input bits and output keystream bits are. Although not all relations are expected to hold with initial input states used, if more relations that hold true for some keystream bit z_i, the conditional probability that z_i is equal to the guessed keystream bit a_i is greater.

6.2.1 Example of Correlation Attack Setup

Recall the 5-bit CMPR from Figure 6, now illustrated below. We can generate relations relating the bits in the individual 2-bit and 3-bit MPR it utilizes and how those relate to the final output bit.

For the 3-bit register a:

$$\begin{aligned} a_2[t+1] &= a_1[t] \\ a_1[t+1] &= a_0[t] \\ a_0[t+1] &= a_2[t] \oplus b_1[t] \end{aligned}$$

For the 2-bit register b:

$$\begin{aligned} b_1[t+1] &= b_0[t] \\ b_0[t+1] &= b_1[t] \oplus a_2[t] \end{aligned}$$

And the derived equations incorporating the feedback and recurrence relations of the 3-bit and 2-bit MPR substituted would be, assuming a keystream length of 4 to operate with that is known:

$$\begin{aligned} z_1 &= a_0 \\ z_2 &= a_2 \oplus b_1 \\ z_3 &= a_1 \oplus (b_1 \oplus a_2) \\ z_4 &= (a_0 \oplus a_2 \oplus b_1) \oplus (b_1 \oplus a_2) \end{aligned}$$

Given any set of keystream bits z_1, z_2, z_3, z_4 generated by the CMPR, these parity-check equations can then be used to solve for certain configurations of initial states that could be valid for the MPRs using a divide-and-conquer approach after correlation values between output keystream bits and internal states of the CMPR are tested and determined. Then, those internal states that are found to have higher correlation values are used to explore other internal states that may propose even higher correlation values (this is the divide and conquer approach on a high level). Those which propose the highest correlation values are then used to test other internal potential initial internals states for the MPRs that may be valid. [Mei11]

Applied to the case of the 17-bit CMPR, 2-bit, 3-bit, 5-bit, and 7-bit MPRs are chained together such that the output bit of one MPR leads into that of the other MPRs to finally produce an output keystream bit. When the attack team applied the fast correlation attack described above to the 17-bit CMPR set up, n relations were formed for every n bit MPR ensuring that for each relation, the output bit of every MPR included an operator with the bit of the next MPR it was chained to. This approach generated 17 distinct relations. Despite the inclusion of parity-check equations, the approach ran near brute-force complexity, similar to other algebraic attacks tested. A downside to the approach used is the large variance in the number of variables used for initial output keystream bits compared to later ones. While there are a reduced number of equations and a reduced number of terms compared to when using algebraic attacks (where the number of terms reached up to 157 for some equations), the correlation attacks run are suspected to have not been able to over-constrain to find a solution for the initial states of MPRs fast enough, suffering from the opposite problem of the previously proposed algebraic attacks.

7 Future Work

7.1 On Linear Complexity

We believe that most of the necessary work on linear complexity has already been completed. We already have decent methods of estimating the linear complexity of an arbitrary CMPR via root counting methods and the estimation algorithm [GM23]. A possible remaining area of future work is further testing of the estimation algorithm on large CMPRs and getting experimental lower bounds on the linear complexity of a CMPR (the lack of an explicit lower bound was a criticism of the previous June 2023 paper submission).

7.2 On de Bruijn Sequences

The exploration of de Bruijn sequences primarily came from a response to a past reviewer's comments from mentioning them, so it is up for debate whether further work in this area is necessary. Regardless, here we present three further research paths, listed in what we consider to be decreasing order of importance:

1. *Linear complexity simulations on cross-joined NLFSRs v. cascade products vs. CMPRs.* We expect the results to be fairly close between all three competitors, since the de Bruijn constructions have guaranteed high linear complexity and we have probabilitistic results on the linear complexity of CMPRs. However, we believe it is still worth the time to investigate this in practice, since it would serve as a good sanity check. Also, the software implementations of the de Bruijn constructions needed for this path would also be highly beneficial for any future tests we decide to work on.

2. *Exploration of alternative metrics for assessing nonlinearity of a NLFSR.* As previously mentioned, we currently hold some doubt as to how well we can understand the "nonlinearity" of a sequence solely by looking at its overall linear complexity. One clear way to proceed is to run some tests based on the aforementioned variance approach (or some other statistical measure of spread) and compare the de Bruijn constructions to the CMPR in this setting. We expect that the CMPR would win out in this comparison, but of course experimental results are needed to confirm this hypothesis. Another idea for comparison in this area can found in [ZHG20].

3. *Exploration of other de Bruijn constructions.* Due to the limited time we had to research de Bruijn sequences, we chose to focus primarily on studying the cross-join and cascade product methods. Of course, there are several other methods of doing so that are worth studying. One suggestion here is to explore the recursive construction of higher order de Bruijn sequences from lower order ones. This is done via something called the D-homomorphism, which is discussed in [Lem70].

7.3 On Cryptanalysis

7.3.1 Cube Attacks

Further work is needed to understand the underlying reason behind the problem of the set of monomials in $x[0], \ldots, x[16]$ not sufficiently saturating the polynomial representing the output bit such that solving the superpolys recovers the entire initial state, and to determine if this issue persists for all CMPRs. There is also work left to determine whether or not clocking the CMPR sufficiently many times will increase the number of linearly dependent superpoly equations that appear, as well as the exact relationship between time t and the dimension of the solution space.

In our analysis, the set of superpoly equations that appear after clocking the CMPR several times do not sufficiently decrease the dimension of the solution space, but it is unclear why this is the case. However, the hypothesis is that this could be related to the number of monomials in the ANF equation. As described in [Gor+21], only approximately half the monomials are present in any given equation, and thus it is possible that certain subsets of monomials may be impossible to appear together in an equation, reducing the possible superpolys that can be used.

7.3.2 Correlation Attacks

A likely future work to reduce the time complexity in exchange for a tradeoff with space complexity is to approach multivariate correlation attacks, which have been used on Grain and require less keystream bits on average, compared to the fast correlation attack, to recover initial states. This approach uses relations more general than parity checks, which retain a dependence on the linear recurrence relations of feedback functions for every LFSR in the CMPR being attacked. [CS23]

7.3.3 Deep Learning

Time has been invested into testing and using weights of prior neural networks that can differentiate between true random binary numbers generated and the keystream of stream ciphers. The intention with this future work would be to train more accurate deep learning models specifically on CMPRs. Although very basic architectures have been implemented, no significant results to report have been found and more work needs to be done on finetuning these models to explore their potential for being able to predict the next keystream bit, or classify a batch of keystream bits as coming from the keystream of a CMPR or from a true random number generator. The goal behind this is as another form of cryptanalysis leveraging greater compute power to test the how vulnerable the CMPR is—inability to produce a sufficiently random sequence could indicate flaws in its design. In addition, extensions may be made to investigating whether the internal structure of the CMPR can be estimated by such networks. However, large amounts of data would be needed for this, and hence, this branch of exploration is particularly low priority.

References

[BAS12] M. Bokhari, S. Alam, and F. Syeed Masoodi. “Cryptanalysis techniques for stream cipher: A survey”. In: (2012).

[Ber66] E. Berlekamp. “Non-binary BCH decoding”. In: (1966).

[CGW20] Z. Chang, G. Gong, and Q. Wang. “Cycle Structures of a Class of Cascaded FSRs”. In: (2020).

[Cou03] N. Courtois. “Fast algebraic attack on stream ciphers with linear feedback”. In: (2003).

[CS23] I. Canales-Martinez and I. Semaev. “Multivariate Correlation Attacks and the Cryptanalysis of LFSR-based Stream Ciphers”. In: (2023).

[DS08] I. Dinur and A. Shamir. “Cube Attacks on Tweakable Black Box Polynomials”. In: (2008).

[Dub13] E. Dubrova. “A Scalable Method for Constructing Galois NLFSRs with Period $2^n - 1$ Using Cross-Join Pairs”. In: (2013).

[Gal20] J. Gallian. *Contemporary Abstract Algebra*. 2020.

[GD70] D. Green and K. Dimond. “Nonlinear product-feedback shift registers”. In: (1970).

[GM23] D. Gordon and V. Mooney. “Scalable Nonlinear Sequence Generation Using Composite Mersenne Product Registers”. In: (2023).

[Gor+21] D. Gordon et al. “Product Register Design and Applications (final report)”. In: (2021).

[HK91] T. Helleseth and T. Klove. “The Number of Cross-Join Pairs in Maximum Length Linear Sequences”. In: (1991).

[Key76] E. Key. “An Analysis of the Structure and Complexity of Nonlinear Binary Sequence Generators”. In: (1976).

[Lem70] A. Lempel. “On a Homomorphism of the de Bruijn Graph and Its Applications to the Design of Feedback Shift Registers”. In: (1970).

[Li+15] M. Li et al. “De Bruijn Sequences from Joining Cycles of Nonlinear Feedback Shift Registers”. In: (2015).

[Mei11] W. Meier. “Fast Correlation Attacks: Methods and Countermeasures”. In: (2011).

[MOV01] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. 2001.

[MS13] J. Mykkeltveit and J. Szmidt. “On cross joining de Bruijn sequences”. In: (2013).

[Qi07] H. Qi. “Stream Ciphers and Linear Complexity”. In: (2007).

[Ros09] B. Rosenberg. “Linear Feedback Shift Registers”. In: (2009).

[Rot95] J. Rotman. *An Introduction to the Theory of Groups*. 1995.

[Rue86] R. Rueppel. *Analysis and Design of Stream Ciphers*. 1986.

[SGM22] I. Smith, D. Gordon, and V. Mooney. “Linear Complexity Final Report”. In: (2022).

[WTQ22] X. Wang, T. Tian, and W. Qi. “A Generic Method for Investigating Nonsingular Galois NFSRs”. In: (2022).

[ZHG20] C. Zhou, B. Hu, and J. Guan. “On the Maximum Nonlinearity of de Bruijn Sequence Feedback Function”. In: (2020).

A Cube Attacks Code

```
import random
from itertools import chain, combinations

n = 17 # Define variables
for i in range(n):
    var(f"x{i}")
```

```
R = PolynomialRing(GF(2), n, 'x')

p = R(1 + (x1) + (x0) + (x13) + (x8*x15) + (x9*x15) + (x4*x15) + \
    (x8*x4*x15) + (x9*x4*x15) + (x8*x9*x15) + (x4*x14*x15) + (x9*x14*x15) + \
    (x8*x9*x4*x15) + (x16*x10*x12) + (x8*x9*x14*x15) + (x8*x4*x14*x15) + \
    (x9*x4*x14*x15) + (x10*x13*x14*x15) + (x16*x10*x12*x15) + \
    (x4*x8*x9*x14*x15) + (x8*x10*x13*x14*x15) + (x10*x11*x13*x14*x15) + \
    (x16*x10*x12*x13*x14*x15)) # ANF of primitive polynomial

I = set([10, 12, 13, 15, 16]) # Known index subset to find maxterm

print(f"Index subset chosen: {I}")
print(f"Polynomial chosen: {p}")

def subterm(I):
    t = 1
    for i in I:
        t *= var(f"x{i}")
    return R(t)

def superpoly(p, t_I):
    r = p.reduce(Ideal([t_I]))
    x = (p-r)//(t_I)
    return r, (p-r)//t_I

def is_maxterm(sp):
    return sp.degree() == 1

q, sp = superpoly(p, subterm(I))

print(f"Superpoly: {sp}")
print(f"Remainder: {q}")

mt = is_maxterm(sp)

print(f"The superpoly is a maxterm: {mt}")

def get_rand_state(n):
    return [random.randint(0,1) for i in range(n)]

def cube_atk(I, init_state):
    state_list = list(init_state)
    index_list = list(I)
    ret = 0
    for i in range(2^len(I)): # Vary over index bits
        guess = [0 if j=="0" else 1 for j in bin(i)[2:].zfill(len(I))]
        for j in range(len(I)):
            state_list[index_list[j]] = guess[j]
        ret+=p(state_list)
    return ret%2

init = get_rand_state(n)

print(f"Initial state: {init}")
print(sp, "=", cube_atk(I,init))

print("-"*80)

def test_maxterm(I):
    q, sp = superpoly(p, subterm(I))
    return is_maxterm(sp)

def powerset(iterable):
    s = list(iterable)
    x = chain.from_iterable(combinations(s, r) for r in range(len(s)+1))
    for i in x:
        print(i)
    return x

def random_set(n):
    ret = set()
    for i in range(n):
        if random.random()<0.5:
            ret.add(i)
    return ret

def find_maxterm(n): # Find random maxterm
    while True:
        x = random_set(n)
        if test_maxterm(x):
            print(x)
            print(superpoly(p, subterm(x))[1], "=", cube_atk(x, init))

find_maxterm(n)
```