Product Registers (Math)

Linear Complexity, Cryptanalysis, de Bruijn Sequences

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

VIP Secure Hardware, Georgia Institute of Technology

December 4, 2023

Outline

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 presentation will explain the background needed to understand and conceptualize CMPRs, as well as tests of their security.

Outline

Definitions

Group: A set with an associative binary operation. Has an identity and inverse elements. [8]

Field: A set with an addition and multiplication operations with identities and inverses for both. The multiplication must distribute over the addition. [8]

Feedback shift register (FSR): 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) [10]

Linear feedback shift register (LFSR): A feedback shift register where a linear function is applied to the current state to reach the next state [10]

Nonlinear feedback shift register (NLFSR): Feedback shift register where a *non*linear function is applied to the current state to reach the next state [10]

Definitions

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 [10]

Mersenne primes: Primes that are of the form $2^n - 1$ [17]

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 [10]

Primitive polynomial: Polynomial used as a modulus to create Galois fields for LFSRs and MPRs; Such a polynomial generates the multiplicative group of GF(2^n) [17]

Product register (PR): A feedback shift register that implements Galois field multiplication [9]

Mersenne product register (MPR): A product register of size n, where n is a Mersenne exponent [9]

Definitions

Composite Mersenne product register (CMPR): A single FSR composed of several MPRs combined via chaining. [9]

Conjugate: The conjugate of an n-bit state is the same state but with the first bit flipped. [12]

Successor: A state's successor is the state directly following it. [12]

Chord: A pair of points on the cycle graph of a sequence. [12]

Cross-join pair: A pair of chords that intersect. [12]

Characteristic Function A characteristic function defines the operations done on with an existing feedback function and other inputs to produce the next output bit. [10]

Definitions

Feedback Functions A feedback function is a boolean function which outputs for some input of n bits s_0 an output s_n. [10]

Non-singular feedback functions A feedback function is considered non-singular if it does not produce a zero-state for an extended period of time. [10]

Cube Attacks An algebraic cryptanalysis technique based on tweaking variables in the initial state (i.e., an IV), and subsequently setting up a system of equations to solve for the initial state. [6]

Correlation Attacks An divide-and-conquer based cryptanalysis technique that tries to recover the initial state of every component FSR/MPR/LFSR by exploiting observed relationships between these components with knowledge of some keystream bits. [16, 4]

Outline

Linear Complexity Motivation

- LFSRs are less secure compared to NLFSRs [19], they can be reconstructed in $O(n^2)$ time via Berlekamp-Massey
- NLFSRs are typically harder to build compared to LFSRs [19]
- Thus, this research into achieving properties of NLFSRs with LFSRs through CMPRs:
  - CMPRs help achieve the security from the nonlinearity of NLFSRs (CMPRs have a full period) [22]
- High linear complexity of an input register sequence indicates how secure the cipher is [20]
  - Indicates how improbable it is to get the sequence fed in as input (using LFSRs)

Outline

Outline

Groups

Definition (Group)

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

1. $*$ is associative,
2. G has an identity element e such that $e * g = g * e = g$ for any $g \in G$,
3. and any element $g \in G$ has an inverse $g^{-1} \in G$ such that $g^{-1} * g = g * g^{-1} = e$. [8]

Example: The integers form a group under addition.

We call a group *abelian* if the binary operation $*$ is commutative.

Example: The $n \times n$ invertible matrices form a non-abelian group under matrix multiplication. This is called the *general linear group*.

Fields

Definition (Field)

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

- $\mathbb{F}^+ := (\mathbb{F}, +)$ is an abelian group,
- $\mathbb{F}^\times := (\mathbb{F} \setminus \{0\}, \times)$ is an abelian group,
- and $\times$ distributes over $+$, i.e. we have $a \times (b + c) = (a \times b) + (a \times c)$ for all $a, b, c \in \mathbb{F}$. [8]

Example: The rational numbers $\mathbb{Q}$ form a field under the usual addition and multiplication of rational numbers.

Fields – Galois Fields

The field $\mathrm{GF}(p^n)$ is defined as the set of polynomials of degree $n-1$ with coefficients in $\{0, 1, \ldots, p-1\}$. To define operations on $\mathrm{GF}(p^n)$, we need to pick a primitive polynomial P of degree n.

Addition in a Galois Field

For $a, b \in \mathrm{GF}(p^n)$, we define $a+b$ to be their sum as polynomials, taking each coefficient modulo p.

Example: In $\mathrm{GF}(2^3)$, let $a = x^2 + x$, $b = x + 1$. Then

$$a + b = (x^2 + x) + (x + 1) = x^2 + 2x + 1 \equiv x^2 + 1.$$

Multiplication in a Galois Field

For $a, b \in \mathrm{GF}(p^n)$, we define $a \times b$ to be their product as polynomials, taking the result modulo P.

Example: In $\mathrm{GF}(2^3)$, let $a = x^2 + x$, $b = x + 1$, $P = x^3 + 1$. Then

$$a \times b = (x^2+x)(x+1) = x^3+2x^2+x \equiv 2x^2+x-1 \equiv x+1 \mod P.$$

Outline

LFSRs

There are two main forms of LFSRs: **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.

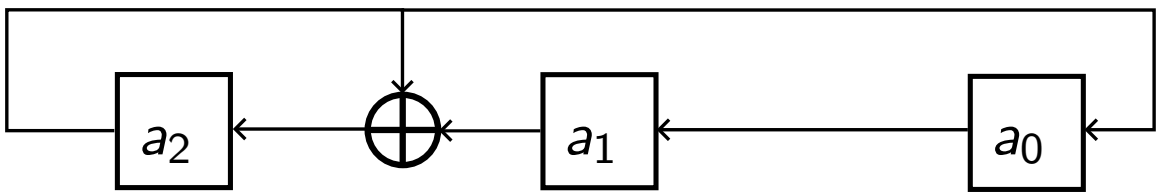


Figure: LFSR in Galois form

LFSRs

There are two main forms of LFSRs: **Galois** and **Fibonacci**.

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.

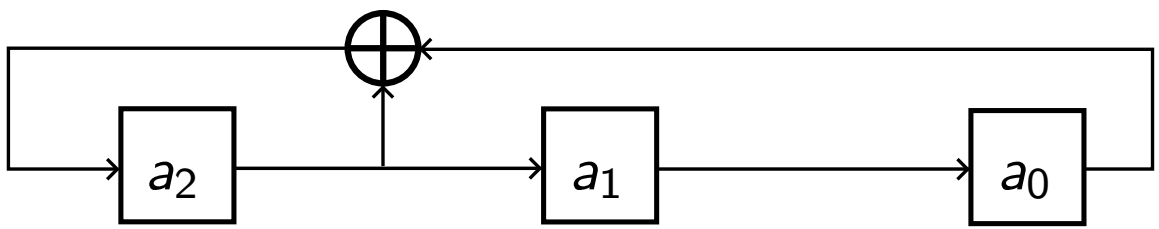


Figure: LFSR in Fibonacci form

More about Galois LFSRs [23]:

- 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 $n-1$

    Example: 1011 corresponds to $x^3 + x + 1$
- Update polynomial is restricted to $U(x) = x$

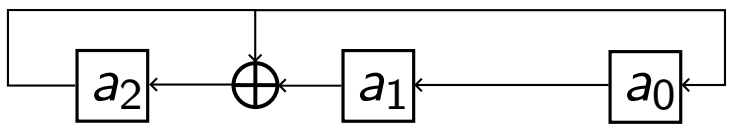
LFSRs: Worked Example for Galois LFSR

Let

$$P = x^3 + x^2 + 1 \qquad \text{(Primitive Polynomial)}$$

$$U = x \qquad \text{(Update Polynomial)}$$

$$S_0 = 110 = x^2 + x \qquad \text{(Initial State)}$$



$$S_1 = S_0 \times U \qquad \text{(Field Multiplication)}$$

$$= (x^2 + x) \times x = x^3 + x^2 \qquad \text{(Multiply by U)}$$

$$= (x^3 + x^2) \pmod{P} \qquad \text{(Modulus on P)}$$

$$S_1 = 1 \rightarrow \boxed{001} \qquad \text{(Next State)}$$

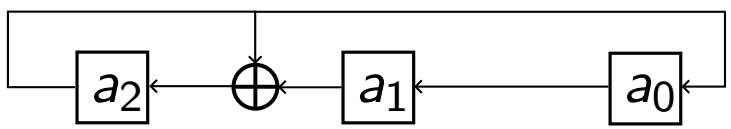
LFSRs: Worked Example for Galois LFSR

Let

$$P = x^3 + x^2 + 1 \qquad \text{(Primitive Polynomial)}$$

$$U = x \qquad \text{(Update Polynomial)}$$

$$S_0 = 110 = x^2 + x \qquad \text{(Initial State)}$$



State_{curr}	Multiplication	Result	Mod P	State_{next}
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

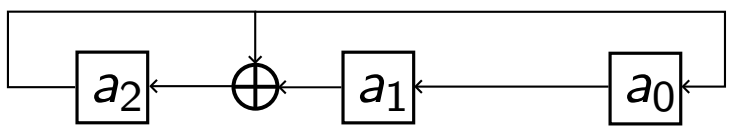
LFSRs: Worked Example for Galois LFSR

Let

$$P = x^3 + x^2 + 1 \qquad \text{(Primitive Polynomial)}$$

$$U = x \qquad \text{(Update Polynomial)}$$

$$S_0 = 110 = x^2 + x \qquad \text{(Initial State)}$$

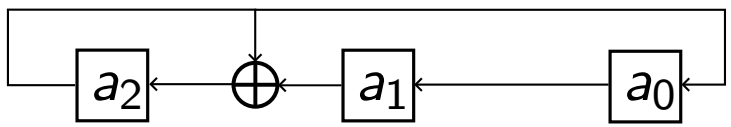


State_{curr}	Multiplication	Result	Mod P	State_{next}
010	$x \times x$	x^2	x^2	100
100	$x^2 \times x$	x^3	$x^2 + 1$	101
101	$(x^2 + 1) \times x$	$x^3 + x$	$x^2 + x + 1$	111

LFSRs: Worked Example for Galois LFSR

Let

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



State_{curr}	Multiplication	Result	Mod P	State_{next}
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

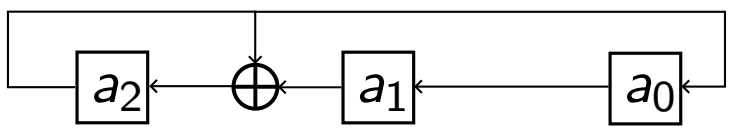
LFSRs: Worked Example for Galois LFSR

Let

$$P = x^3 + x^2 + 1 \qquad \text{(Primitive Polynomial)}$$

$$U = x \qquad \text{(Update Polynomial)}$$

$$S_0 = 110 = x^2 + x \qquad \text{(Initial State)}$$

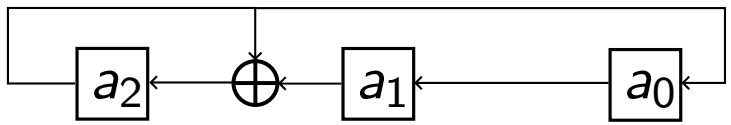


State_{curr}	Multiplication	Result	Mod P	State_{next}
101	$(x^2 + 1) \times x$	$x^3 + x$	$x^2 + x + 1$	111
111	$(x^2 + x + 1) \times x$	$x^3 + x^2 + x$	$x + 1$	011
011	$(x + 1) \times x$	$x^2 + x$	$x^2 + x$	110

LFSRs: Worked Example for Galois LFSR

Let

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



State_{curr}	Multiplication	Result	Mod P	State_{next}
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
110	$(x^2 + x) \times x$	$x^3 + x^2$	1	001

The LFSR has completed a cycle of length 7.

Outline

Product Registers

Definition (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 is given by

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

Note that $U = 0$ is not 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. [9]

Mersenne Product Registers

Definition (Mersenne product register)

A *Mersenne product register* is a product register of size n, where n is a Mersenne exponent. [9]

So $|\mathrm{GF}(2^n)^\times| = 2^n - 1$ will be prime. This is important because:

Theorem (Lagrange's theorem)

For any finite group G and a subgroup $H \subseteq G$, $|H|$ divides $|G|$. [8]

Since $2^n - 1$ is prime, this essentially means that any nontrivial ($U \neq 1$) update polynomial U, it will always generate the entire multiplicative group, i.e. it will be full period.

Composite Mersenne Product Registers (CMPRs)

Definition

A *composite Mersenne product register* is a group 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

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

where U_i and P_i are the update and primitive polynomials corresponding to M_i, respectively. [9]

- The chaining functions are implemented using logic gates (typically AND) connecting the outputs of $M_0, \ldots, M_i$ to the input of M_{i+1}.
- Even though MPRs are linear, the AND gates introduced by the chaining functions make the CMPR nonlinear as a whole.

Outline

Fibonacci LFSRs

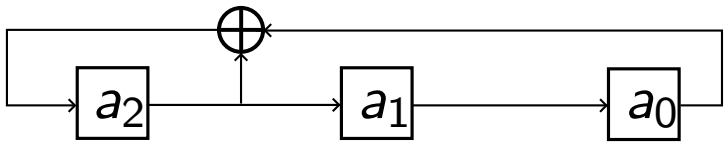


Figure: A Fibonacci-style LFSR.

Recall that a Fibonacci LFSR expresses the next state as a linear combination of the previous states. Mathematically this is something like

$$a_{n+1} = \sum_{i=1}^{n} c_i a_i,$$

where $c_1, \ldots, c_n \in \{0, 1\}$ are constants describing the recurrence.

Note that on each iteration, the values of a_2, a_1 (in the top diagram) will flow into the a_1, a_0 blocks uninterrupted.

De Bruijn Sequences

Definition

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. [14]

Example: A de Bruijn sequence of order 3 on $\{0, 1\}$ might look like

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

Note that the n-tuples are allowed to wrap around, since de Bruijn sequences are periodic. De Bruijn sequences are also 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.

Significance

For applications to feedback shift registers, we usually consider *binary* de Bruijn sequences, i.e. $\mathcal{A} = \{0, 1\}$.

- De Bruijn sequences are by definition full period: They contain every possible n-bit state of an FSR
- They have high linear complexity: It can be shown that the linear complexity of a de Bruijn sequence is at least half of its length (so exponential in n) [14]
- There are many $(2^{2^{n-1}-n})$ de Bruijn sequences of order n [14]
- The idea is to try to construct a de Bruijn sequence by using FSRs, then the FSR will will satisfy these nice properties

However, the difficulty lies in the 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 in this manner. [14]

Creating 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, an LFSR in the Fibonacci configuration yields a cyclic sequence of length $2^n - 1$, just 1 shy of the desired de Bruijn sequence.

Solution: We can just add a 0 to the end to complete the de Bruijn sequence.

But this is still mostly linear! We were looking for nonlinear sequences. 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.

Cross-Join Pairs

Definition (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.

A *chord* on a cycle C of n states

$$S_1, S_2, S_3, \ldots, S_n$$

is 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. [12, 7]

Cross-Join Pairs

We can visualize a chord as a chord of the circle, if we think of a cycle via its cycle graph. (Imagine that each dot is a state.)

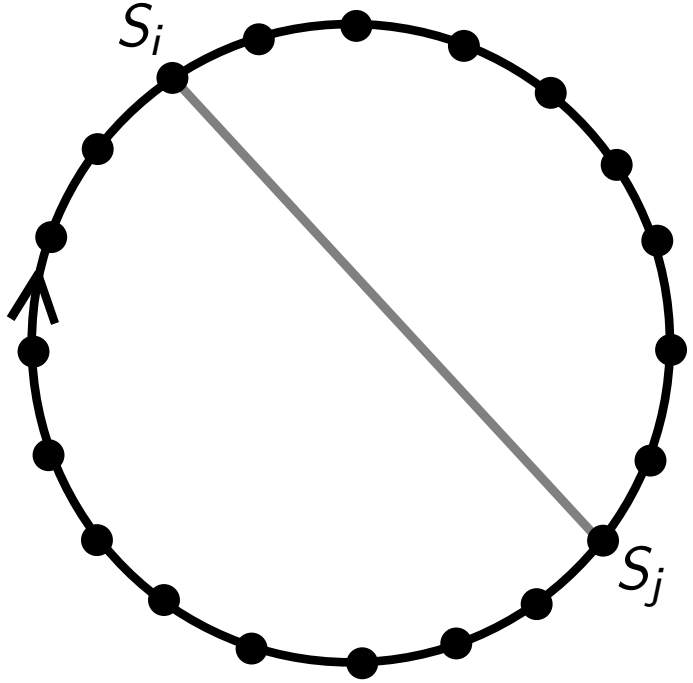


Figure: Chord (S_i, S_j) on a cycle graph.

Cross-Join Pairs

Then a *cross-join pair* is a pair of two chords $(\alpha, \hat{\alpha})$ and $(\beta, \hat{\beta})$ that intersect each other. For example, we may have:

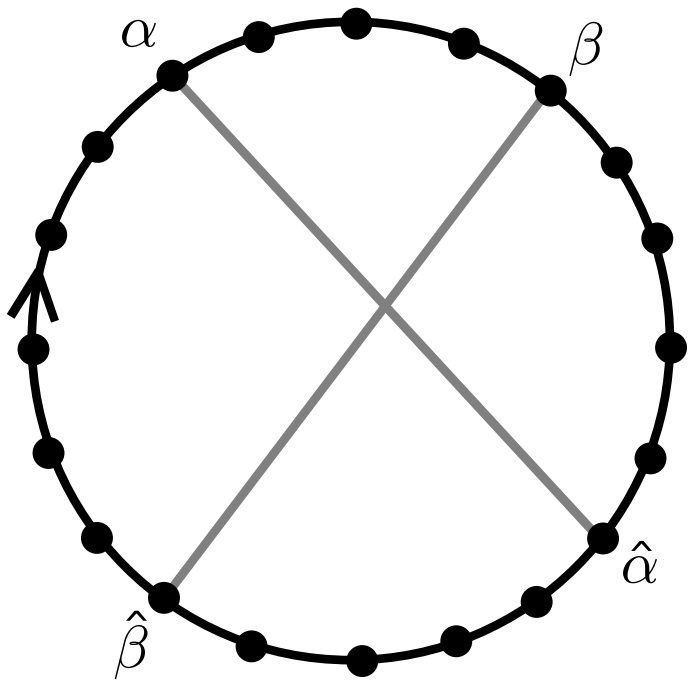


Figure: A cross-join pair $(\alpha, \hat{\alpha}), (\beta, \hat{\beta})$.

Cross-Join Pairs

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.

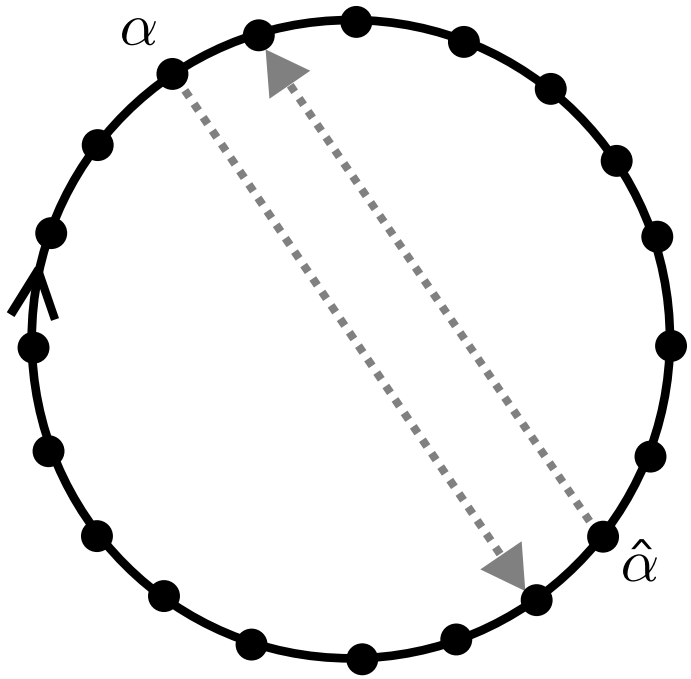


Figure: Exchanging the successors of α and $\hat{\alpha}$.

Cross-Join Pairs

Next, we take β and $\hat{\beta}$ and also exchange their successors. This joins the two disjoint cycles back into a single (nonlinear!) one.

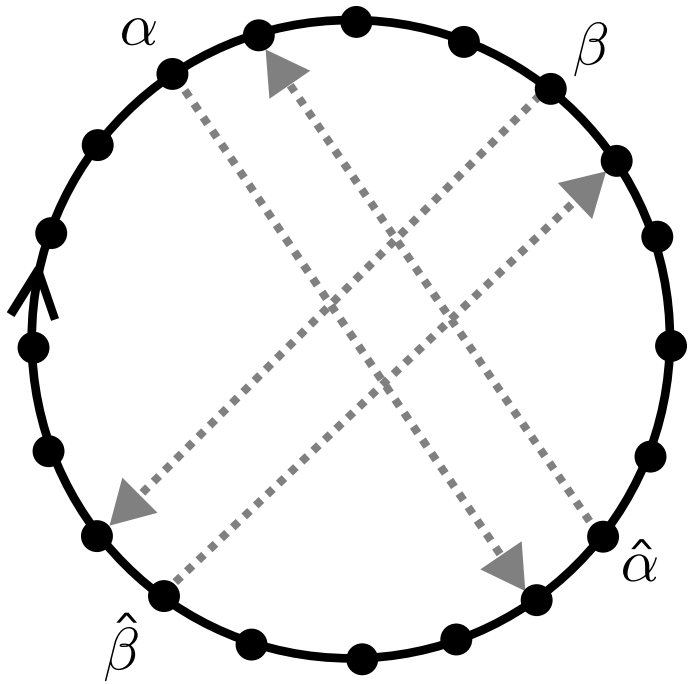


Figure: Exchanging the successors of β and $\hat{\beta}$.

Why Conjugate?

Point: Preserve de Bruijn property.

Recall that a de Bruijn sequence contains every 2^n possible n-tuple in a cycle of length 2^n. This means that every adjacent pair of states need to have $2n-1$ bits in common. In particular, if 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

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

for some new bit x_n. But bits $x_1, \ldots, x_{n-1}$ must occur in both states.

Why Conjugate?

Now suppose we have a conjugate pair S_i and $\hat{S}_i$ and they look like

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

Then their successors must look like

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

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.

Key Takeaways from Cross-Join

- We can take an existing de Bruijn sequence (e.g. by from a 0 at the end of an Fibonacci LFSR sequence) and cross-join to make a "more nonlinear" de Bruijn sequence
- We can repeat the cross-join process many times to get better nonlinearity
- There are $(2^{n-1}-1)(2^{n-1}-2)/6$ cross-join pairs for any given maximum-length size-n Fibonacci LFSR sequence [12]
- 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. [18]
    - However, the proof of this is not constructive.

Cascading Products

Definition (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 $f_1(x_0, x_1, ..., x_{m-1}) = x_m + h_1(x_0, x_1, ..., x_{m-1})$ where h_1 is a boolean function. Here, h_1 must have domain $\mathbb{F}_2^{m-1}$. We assume that the feedback functions f_1 and g_1 are strictly non-singular.

- A trivial example for a non-singular feedback function would be $h_1(x_0, x_1, ..., x_m) = 0$.
- Given two FSRs F_1 and F_2, let f be the characteristic function that defines how F_1 is updated, and similarly, let g be the characteristic function defining updates of F_2.
- This means F_1 is updated as $f(x_0, x_1, ..., x_m) = f(x_0, x_1, ..., x_{m-1}) + x_m$, and means F_2 is updated as $g(x_0, x_1, ..., x_m) = g(x_0, x_1, ..., x_{m-1}) + x_m$

Cascading Products

- For FSR_1 with characteristic function f_1, we define $\Omega(f)$ to be the set of 2^n initial states for all sequences. We know there are 2^n such sequences if and only if f_1 is a **non-singular** feedback function.

Definition (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: $\Omega(f) = s[1] \cup s[2] \cup ...s[k]$.

- For $k = 1$ above, we would obtain just one cycle that would be of full period, and hence be of order n.
- By definition, then, for $k = 1$, the corresponding cycle that exists (the only one) forms a de Brujin sequence.

Cycle Joining

- While an LFSR, can provide linear state sequences, we wish to produce non-linearity in the eventual registers we use to obfuscate the predictability of the next output in a typical LFSR or FSR. We can make use of **cycle joining** to do this.

Definition (Cycle joining)

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

- Done $k - 1$ times to combine the k cycles, eventually, there will be only one cycle left with full period which forms a de Brujin sequence.
- This provides a full-period LFSR state sequence, which emulates the behavior of MPRs.

Cycle Joining (Example)

Given two cycles of size 5, $s[1]$ and $s[2]$, where $s[1]$ is denoted with the starting state $s[1] = (1, 0, 1, 1, 0)$ and $s[2] = (0, 1, 0, 0, 1)$ for some 5-bit LFSR F:

- These cycles match the relationship that they should be successors of one another.
- If the predecessor of the initial state of $s[1]$ in the state sequences produced by LFSR F is some $p[1] = (0, 0, 0, 1, 0)$ and the predecessor of the initial state of $s[2]$ is some $p[2] = (1, 0, 0, 1, 0)$, then some the cycles can be joined in the order:
  s[1], p[2] ...(complete cycle of p[2] back to s[2]) ... s[2], p[1].

Cascading Products

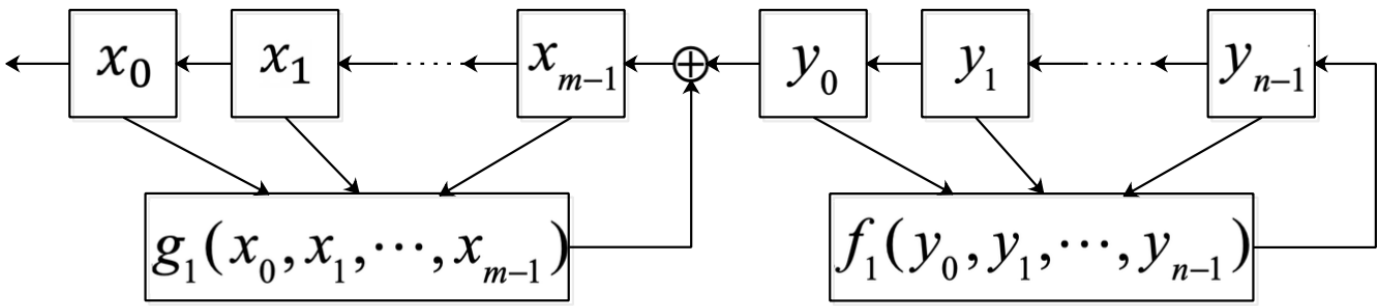
Definition (Predecessor)

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

- Done $k - 1$ times to combine the k cycles, eventually, there will be only one cycle left with full period which forms a de Bruijn sequence.
- This provides a full-period LFSR state sequence, which emulates the behavior of MPRs (which also have full period).

Cascading Products

- The hardware construction for a cascading product where f is the characteristic function for an n-bit LFSR that has an irreducible primitive polynomial and g is the characteristic function for an m-bit LFSR is as follows:



- This hardware representation of the LFSR with function f being *cascaded* with g is expressed as:

$$f * g = f(g(x_0, x_1, x_2, ..., x_m), g(x_1, x_2, x_3, ...x_{m+1}, ..., g(x_n, x_{n+1}, ...x_{n+m}))$$

Note on Product Registers (the Name)

There is existing literature on so-called *product-feedback shift registers* from the 1960-1970s by Green and Dimond. [11]

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.

Outline

Cryptanalysis

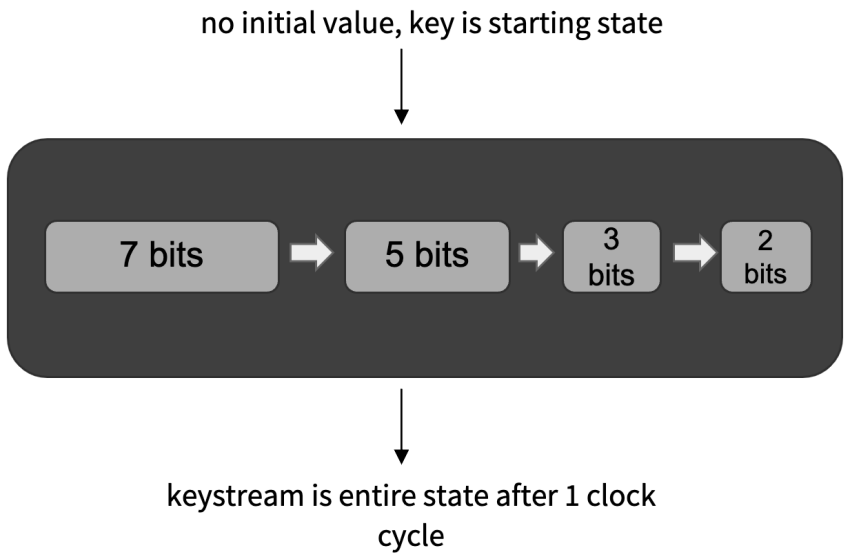
- Brute Force/Exhaustive Key Search: try all possible keys [3]. Attacks with greater computational complexity than brute force are not attacks. In the case of CMPRs, it depends on what the key is, but the complexity is likely $O(\text{period})$.
- Algebraic Attacks: model system in terms of algebraic equations [3]. Example equation: keystream in terms of initial state. Given enough of these equations (more variables than equations) solve for initial state. Solving system of equations using Gaussian elimination takes $O(n^3)$.
- Linear Cryptanalysis: Statistical attack that relates input and output bits of a cipher. [15].

Cryptanalysis - Prior Attacks & Definitions

- Guess and Determine: guess part of the internal state, try to determine the rest of the state. Validate by comparing keystream generated with given keystream [3].
- Berlekamp-Massey: given bit sequence, algorithm finds smallest LFSR that could generate it [13]. The complexity is $O((\text{length of bit sequence})^2)$. If the linear complexity is $< \sqrt{\text{period}}$ (often is) this would be considered an attack.
- Differential Cryptanalysis: analyze the effect of changes in plaintext on changes in cipher text used to assign probabilities to possible keys. [2]
- Fast Algebraic Attack: linearly combine equations before solving to increase speed of attack. [5]

Cryptanalysis - Linearization Example

- The value of the CMPR C at bit i and time t is denoted $c_i[t]$. No initialization value was given for this attack, and the key was the starting state. Keystream represented the entire state after **1 clock cycle**.



Cryptanalysis - Linearization Example

- Equations used in ANF form—Any boolean function can be uniquely represented by ANF and reduced for Gaussian elimination/linearization. Consider the following example:

$$c_0[t+1] = (c_0[t] \wedge c_1[t] + c_2[t]$$
$$c_1[t+1] = (c_0[t] \wedge c_1[t] + (c_1[t] \wedge c_2[t]$$
$$c_2[t+1] = (c_0[t] \wedge c_1[t] \wedge c_2[t] + c_0[t]$$

Replacing nonlinear terms with variables, we can see the following:

$$c_0[t+1] = c_{\{0,1\}}[t] + c_2[t]$$
$$c_1[t+1] = c_{\{0,1\}}[t] + c_{\{1,2\}}[t]$$
$$c_2[t+1] = c_{\{0,1,2\}}[t] + c_0[t]$$

Which results in a solvable system of linear equations, computable in polynomial time.

Cryptanalysis - Linearization Example

One key limitation of linearization 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. [10]

Outline

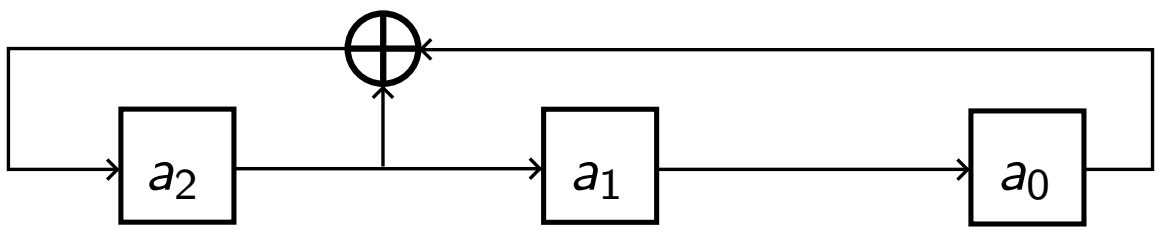
Berlekamp-Massey

- Any recurrence can be written as a sum of a very large number of variables [19]
- **Key Idea**: How to solve for the *shortest* linear recurrence that describes a sequence. [19] We call the order of this linear recurrence the *linear complexity* of the sequence, which we denote $L(s)$.
- Output of CMPRs are periodic, so their behavior can be emulated via an (potentially much longer) LFSR [23]:
  - Berlekamp-Massey finds the minimum length linear recurrence, and thereby LFSR, that models a CMPR to achieve a target cycle between states described by an input register sequence
- Output of algorithm: Linear recurrence describing CMPR that can be used to get the input sequence [23]
- Worst case time complexity [1]: $O(n^2)$, where n is the bit sequence length.

Berlekamp-Massey

At the bottom is a Fibonacci LFSR, as defined previously.

- In the example, the hardware has the relationship $a_2 = a_1 + a_0$.
- Idea: Write this as the recurrence relation $s_{n+1} = s_n + s_{n-1}$, where s_i is the ith output bit.



Overview of the Berlekamp-Massey Algorithm

Recall that a general recurrence may look like

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

Rough idea: Guess and correct the error as we go.

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.
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.

Berlekamp-Massey on a Rational Sequence

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

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

Berlekamp-Massey says that these first 6 terms are enough to recover our original recurrence.

Berlekamp-Massey on a Rational Sequence

We first put an implicit 1 at the beginning of the sequence and start with a guess of simply 1 with error $d = 1$.

1	0	1	2	4	11	19	d	$L(s)$
1							1	0

This is essentially just initialization for the algorithm.

Berlekamp-Massey on a Rational Sequence

Next we slide our guess one to the right:

1	0	1	2	4	11	19	d	$L(s)$
1							1	0
	1						0	0

Here we can calculate the error as

$$d = 1(0) = 0,$$

so no adjustment is necessary.

Berlekamp-Massey on a Rational Sequence

Sliding our guess one more to the right gives:

1	0	1	2	4	11	19	d	$L(s)$
1							1	0
	1						0	0
		1					1	0

Here we can calculate the error as

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

so we need to correct! Here we subtract $d/d_{\text{prev}} = 1/1 = 1$ times the previous error row to get

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

as our next guess.

Berlekamp-Massey on a Rational Sequence

So we end up with:

1	0	1	2	4	11	19	d	$L(s)$
1							1	0
	1						0	0
		1					1	0
-1	0	1					0	2

Here we can see that

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

so our correction has worked. Also note that $L(s)$ has jumped since we increased the length of our guess.

Berlekamp-Massey on a Rational Sequence

Now going back to shifting gives:

1	0	1	2	4	11	19	d	$L(s)$
		1					1	0
-1	0	1					0	2
	-1	0	1				2	2

Here we note that

$$d = -1(0) + 0(1) + 1(2) = 2,$$

so we need to correct again. We can subtract by $d/d_{\text{prev}} = 2/1 = 2$ times the previous error row to get

$$(-1, 0, 1) - 2(0, 1, 0) = (-1, -2, 1)$$

as the new guess. Note here that $L(s)$ did not increase.

Berlekamp-Massey on a Rational Sequence

Putting this in gives:

1	0	1	2	4	11	19	d	$L(s)$
		1					1	0
-1	0	1					0	2
	-1	0	1				2	2
	-1	-2	1				0	2

We can verify that

$$d = -1(0) - 2(1) + 1(2) = 0,$$

so all is good.

Berlekamp-Massey on a Rational Sequence

Shifting to the right now yields:

1	0	1	2	4	11	19	d	$L(s)$
	-1	0	1				2	2
	-1	-2	1				0	2
		-1	-2	1			-1	2

Here the error is

$$d = -1(1) - 2(2) + 1(4) = -1,$$

so we can correct by subtracting $d/d_{\text{prev}} = -1/2 = -0.5$ times the previous error row. This is

$$(0, -1, -2, 1) + 0.5(-1, 0, 1, 0) = (-0.5, -1, -1.5, 1).$$

Berlekamp-Massey on a Rational Sequence

Putting this in gives:

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

We can verify

$$d = -0.5(0) - 1(1) - 1.5(2) + 1(4) = 0.$$

Also note that $L(s)$ has increased.

Berlekamp-Massey on a Rational Sequence

Shifting once again gives

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

We can once again calculate

$$d = -0.5(1) - 1(2) - 1.5(4) + 1(11) = 2.5,$$

so we need to subtract by $d/d_{\text{prev}} = 2.5/(-1) = -2.5$, which gives

$$(-0.5, -1, -1.5, 1) + 2.5(-1, -2, 1, 0) = (-3, -6, 1, 1).$$

Berlekamp-Massey on a Rational Sequence

Shifting once again gives

1	0	1	2	4	11	19	d	$L(s)$
		-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

We can check

$$d = -3(1) - 6(2) + 1(4) + 1(11) = 0.$$

Note that $L(s)$ has not increased.

Berlekamp-Massey on a Rational Sequence

Finally shifting for a final time gives:

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

Here we can compute

$$d = -3(2) + -6(4) + 1(11) + 1(19) = 0.$$

No error! Since we have reached the end, this means that the correct recurrence is

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

or $a_n = -a_{n-1} + 6a_{n-2} + 3a_{n-3}$, which is what we started with.

Berlekamp-Massey on a Rational Sequence

As a recap, here's the whole table in one place:

$_1$	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

We can kind of intuitively see the $O(n^2)$ complexity of Berlekamp-Massey here.

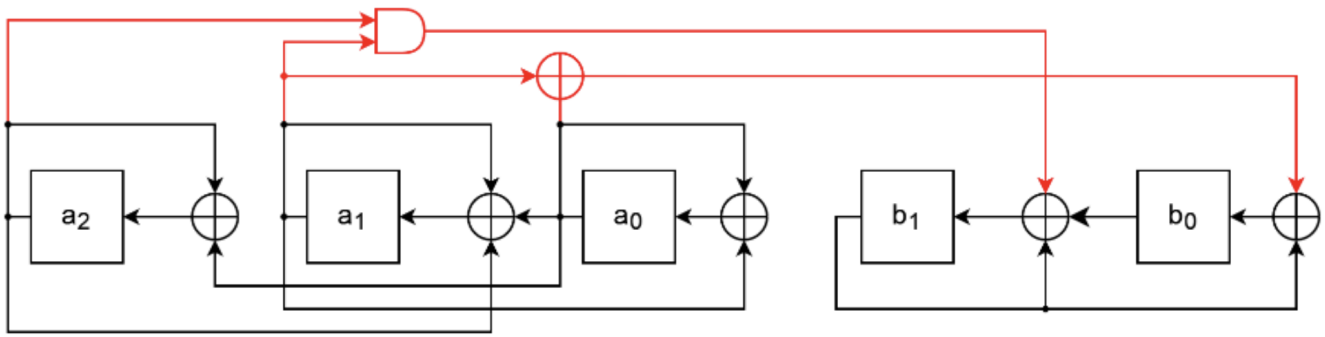
Outline

Linear Complexity Motivation

- 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.
- LFSRs and MPRs have linear complexities that are just the length, but CMPRs are expected to have larger linear complexities due to obfuscations that make their output sequences act in ways similar to NLFSRs
- Question: How do we estimate the linear complexity of a CMPR, given chaining with logic gates between MPRs and LFSRs? Loose logic:
  - **Linear Complexity** = Length of LFSR or Length of MPR
  - Length of LFSR or MPR = Degree of P = **Number of roots of** P

Linear Complexity Motivation - Diagram

- Each MPR is chained with either and XOR or an AND gate (indicated in red in the diagram below).
- Idea: find Linear Complexity of individual MPRs and derive method of root counting wire by wire for chaining functions applied between MPRs (red wires).



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

Outline

Root Counting - Terminology

- We need a few more tools to proceed with root counting for CMPRs, as well as understanding how chaining with logic gates affects them.
- 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.
- Recall that: A polynomial $P(x)$ in GF(p)[x] is irreducible if $P(x)$ has no non-constant polynomial divisors.

More Root Counting Background - Irreducible Polynomials

- A polynomial $P(x)$ is said to be irreducible if, for all integers k between 1 and $n-1$ inclusive, no polynomials of order k exist such that they divide $P(x)$.
- Key Question:
  - **When does a characteristic polynomial for an LFSR yield a full period?**
  - One condition is irreducibility - Well chosen characteristic polynomial yields a long cycle by achieving a high period
- Maximum size of a periodic state sequence is $2n-1$, since at any point, the 0 state will simply lead to itself, hence it is excluded.
- Theorem: A characteristic polynomial $P(x)$ has a period of the smallest k such that $P(x)$ evenly divides x^k+1
- Therefore, characteristic polynomial for full period LFSR must divide x^k+1 where $k=2^n-1$

Root Counting - Terminology

- Extension Field: If GF(p) is a field contained in GF(p^n), then GF(p^n) is an extension field of GF(p). [21]
  - Example: Let $p = 2$, $n = 3$.
    Then GF(2) is the field with elements: $\{0, 1\}$.
    $GF(2^3)$ is a field with 8 elements:
    $\{0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1\}$.
- Splitting Fields: An extension field K for some field F is a splitting field for some polynomial $f(x)$ if $f(x)$ can be factored as a set of linear factors present in that extension field.
  - Example: Let 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).
    - Thus, the field $\mathbb{Q}(\sqrt[3]{2}, \omega)$ is the splitting field for $h(x)$ over $\mathbb{Q}$.

a

Root Counting - Cyclotomic Cosets

- Cyclotomic coset: A set of integers that generates all primitive roots of a polynomial modulo n. [17]
- How to obtain a cyclotomic coset:
  - 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 $A(x)$.
  - Create a splitting field E of 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.
- Key idea: Allow us to move from individual root-counting to sets of roots corresponding to each primitive polynomial for the MPRs.
- Cyclotomic coset: $\{C_k(n) = a^{ik \bmod n} \mid \gcd(i, n) = 1\}$

Root Counting - Cyclotomic Cosets Continued

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

- 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 GF(2^3)
  - Only two cosets for GF(2^3), corresponding to $x^3 + x + 1$ and $x^3 + x^2 + 1$ respectively in order of appearance in the below sequence:
    - When $k = 1$: $\{\alpha^{1 \bmod 2^3-1}, \alpha^{2 \bmod 7}, \alpha^{4 \bmod 7} \ldots \alpha^{4 \bmod 7}\} = \{\alpha, \alpha^2, \alpha^4\}$
    - When $k = 2$: $\{\alpha^{2 \bmod 2^3-1}, \alpha^{4 \bmod 7}, \alpha^{8 \bmod 7} \ldots \alpha^{8 \bmod 7}\} = \{\alpha^2, \alpha^4, \alpha\}$
    - When $k = 3$: $\{\alpha^{3 \bmod 2^3-1}, \alpha^{6 \bmod 7}, \alpha^{12 \bmod 7} \ldots \alpha^{12 \bmod 7}\} = \{\alpha^3, \alpha^6, \alpha^5\}$
    - When $k = 4$: $\{\alpha^{4 \bmod 2^3-1}, \alpha^{8 \bmod 7}, \alpha^{16 \bmod 7} \ldots \alpha^{16 \bmod 7}\} = \{\alpha^4, \alpha, \alpha^2\}$
    - When $k = 5$: $\{\alpha^{5 \bmod 2^3-1}, \alpha^{10 \bmod 7}, \alpha^{20 \bmod 7} \ldots \alpha^{20 \bmod 7}\} = \{\alpha^5, \alpha^3, \alpha^6\}$
    - When $k = 6$: $\{\alpha^{6 \bmod 2^3-1}, \alpha^{12 \bmod 7}, \alpha^{24 \bmod 7} \ldots \alpha^{24 \bmod 7}\} = \{\alpha^6, \alpha^5, \alpha^3\}$

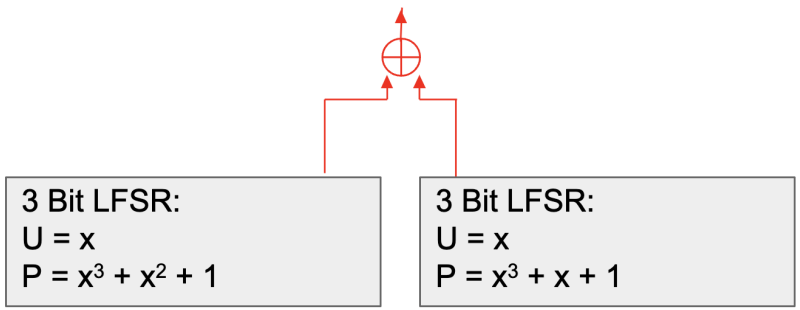
Coset Weight

- **Coset weight** for a set of roots is the number of 1s in the binary representation of the exponents of the elements in the set (always the same for each element in the set), and also known as **Hamming weight**.

Cyclotomic Coset	Corresponding Polynomial	Weight
$C_1 = \{\alpha, \alpha^2, \alpha^4\}$	$x^3 + x + 1$	1
$C_3 = \{\alpha^3, \alpha^6, \alpha^{12} = \alpha^5\}$	$x^3 + x^2 + 1$	2

XOR Gates

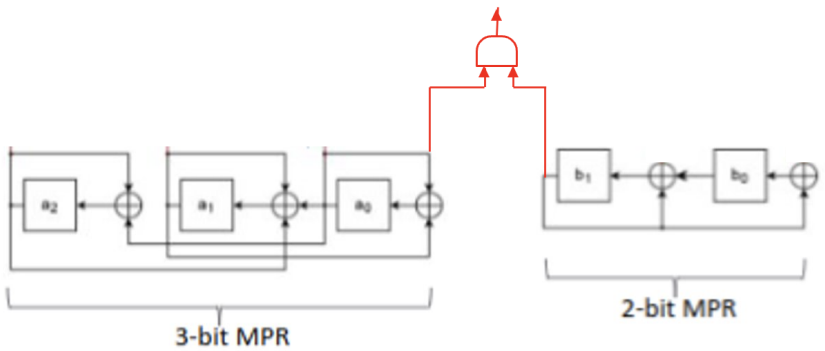
- Combining sequences with an XOR gate leads to the resulting sequence containing the union of the roots of the initial sequences.
- This means, for root counting, we add the number of roots in the primitive polynomial of one MPR with that of another to produce the new linear complexity
- Example: We know the linear complexity is the union of two different cosets of size 3 (corresponding to the two different degree 3 primitive polynomials), so the linear complexity of the output sequence is of size 6



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

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



AND Gates (Same Fields)

- 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).
- The resulting sequence 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
- Roots combined in the same coset can exist in a different coset in the same field (would lead to overcounting the number of roots)
- Example: If sequence A has root α and sequence B 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_3.

AND Gates (Same Fields)

- Solution: Use coset weights.
- Theorem: After ANDing k sequences, the number of roots in GF(2^n) is $\leq \sum_{i=1}^{k} \binom{n}{i}$. [13]
- Multiplying two roots results in the sum of the exponents ($\alpha^a \cdot \alpha^b = \alpha^{a+b}$).
- 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.
- The weight of all cosets contained in a product is less than or equal to the sum of the two maximum weights in the original sequence.

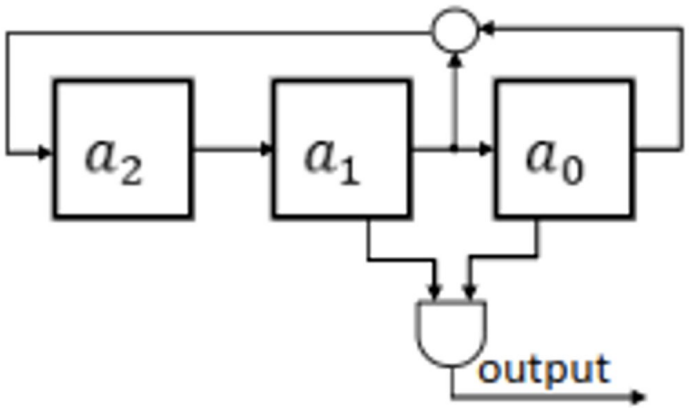
$$c[t] = a[t]b[t] = \left(\sum_{i=1}^{2^n-1} A_i(\alpha^i)^t\right)\left(\sum_{j=1}^{2^n-1} B_j(\alpha^j)^t\right) = \sum_{i=1}^{2^n-1}\sum_{j=1}^{2^n-1} A_i B_j(\alpha^{i+j})^t$$

Outline

AND Gates (Same Fields) – Example

- For example, Key applies this bound to filter generators
  - In this example we are in GF(2^3) and are ANDing two wires
  - We can apply the Key bound on linear complexity to get

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



Outline

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 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.
- However, most of our linear complexity analysis actually comes from de Bruijn-style math (e.g. root counting)
  - So most of these results will likely carry over to de Bruijn constructions, especially the cascade products

Comparison of Linear Complexity

- Though examples cases exist, CMPRs have exponential linear complexity with high probability. [9]
- Recall that a de Bruijn sequence guarantees a linear complexity of at least half of its period (which is exponential in its order). No probabilistic result here.

However, is this really a fair comparison? Take the de Bruijn sequence generated by adding a 0 to an LFSR, for instance.

For its entire period other than the 0, it exhibits very low linear complexity. Then it spikes to exponentially high linear complexity upon reaching the 0. So, in some sense, it's still *mostly linear*.

A Better Metric?

In this case, it might be more insightful to look at some 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.

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 the variance of the linear complexity of a sequence s from $n/2$ could be a good metric to test:

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

Outline

Outline

Cube Attacks Overview

- Algebraic attack developed by **Dinur and Shamir** to attack "tweakable polynomials" in GF(2). [6]
- We can represent the output bit as a polynomial in GF(2), where the variables are the initial state of the CMPR. 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.

Why is this important?

Cube Attacks Overview

- Algebraic attack developed by **Dinur and Shamir** to attack "tweakable polynomials" in GF(2). [6]
- We can represent the output bit as a polynomial in GF(2), where the variables are the initial state of the CMPR. 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.

Why is this important? This potentially presents an attack better than any currently known attack.

Cube Attack Algorithm

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. [6]

Output: If $(v_m), (x_n)$ are "private" (key) and "public" (tweakable, plaintext/IV) variables, given sufficient outputs of $p(v_1, \ldots, v_m, x_1, \ldots, x_n)$, recover $v_1, \ldots, v_m$.

Note for simplicity in notation we will indiscriminately define private and public variables as (x_n) in future examples.

Cube Attack Algorithm, contd.

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** $I \subseteq \{1, \ldots, n\}$ 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. [6]

Cube Attack Algorithm, contd.

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** $I \subseteq \{1, \ldots, n\}$ 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. [6]
Arbitrarily fix I. Then factor t_I from p:

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

This defines two things:

Cube Attack Algorithm, contd.

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** $I \subseteq \{1, \ldots, n\}$ 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. [6]
Arbitrarily fix I. Then factor t_I from p:

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

This defines two things:

- $p_{S(I)}$ is the **superpoly** of I in p.

Cube Attack Algorithm, contd.

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** $I \subseteq \{1, \ldots, n\}$ 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. [6]
Arbitrarily fix I. Then factor t_I from p:

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

This defines two things:

- $p_{S(I)}$ is the **superpoly** of I in p.
- q is the **remainder**.

Cube Attack Algorithm, contd.

Recall that after factoring out t_I, we have

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

We remark several things:

Cube Attack Algorithm, contd.

Recall that after factoring out t_I, we have

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

We remark several things:

- $p_{S(I)}$ does not contain any common variable with t_I, since the degree of any variable in a term is bounded by 1;

Cube Attack Algorithm, contd.

Recall that after factoring out t_I, we have

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

We remark several things:

- $p_{S(I)}$ does not contain any common variable with t_I, since the degree of any variable in a term is bounded by 1;
- q contains at least one variable x_i such that $i \notin I$.

Cube Attack Algorithm, contd.

Recall that after factoring out t_I, we have

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

We remark several things:

- $p_{S(I)}$ does not contain any common variable with t_I, since the degree of any variable in a term is bounded by 1;
- q contains at least one variable x_i such that $i \notin I$.

We define a **maxterm** of p to be a choice of t_I such that $deg(p_{S(I)}) = 1$.

Cube Attack Algorithm, contd.

Let

$$p = x_1x_2x_3 + x_1x_2x_4 + x_2x_4x_5 + x_1x_2 + x_2 + x_3x_5 + x_5 + 1$$

$$I = \{1, 2\} \text{ and } t_I = x_1x_2$$

Then

$$p = x_1x_2(x_3 + x_4 + 1) + (x_2x_4x_5 + x_3x_5 + x_2 + x_5 + 1)$$

where $p_{S(I)} = x_3 + x_4 + 1$ and $q = x_2x_4x_5 + x_3x_5 + x_2 + x_5 + 1$

Cube Attack Algorithm, contd.

Recall that after factoring out t_I, we have

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

- 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.

Cube Attack Algorithm, contd.

Theorem (Key Idea of Cube Attacks)

For any polynomial p and subset of variables I, $p_I \equiv p_{S(I)}$. [6]

Proof Sketch.

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

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.

Cube Attack Algorithm, contd.

Theorem (Key Idea of Cube Attacks)

For any polynomial p and subset of variables I, $p_I \equiv p_{S(I)}$. [6]

Proof Sketch.

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

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 $v = \{1, \ldots, 1\}$. This assigns 1 to all $p_{S(I)}$. □

Cube Attack Algorithm, contd.

Theorem (Key Idea of Cube Attacks)

For any polynomial p and subset of variables I, $p_I \equiv p_{S(I)}$. [6]

Proof Sketch.

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

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 $v = \{1, \ldots, 1\}$. This assigns 1 to all $p_{S(I)}$. □

Observation: 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.

Applying the Cube Attack to CMPRs

- Initialize an n-bit CMPR by seeding the state and clocking it during the initialization round.
- Now, the CMPR can be treated as a polynomial p.
- If we can tweak a portion of the initial state, then we can choose subsets of I such that it indexes tweakable bits.
- Limitation: Initial state needs to be tweakable! This limits the utility of such an attack.

Cube Attack Results

- Does not seem to work very successfully. Insufficient maxterms were found to successfully replicate cube attack as described in paper. In the $\mathrm{C17}$ CMPR example, only 4 maxterms were found.
- Previously, one idea of future work was to restrict maxterms to be nonlinear (i.e. $deg(p_{S(I)} \leq 2)$, then use linearization to solve for the initial state. Upon closer examination, it seems that this does not fundamentally change the difficulty of the attack.

Outline

Correlation Attacks Overview

- An divide-and-conquer based cryptanalysis technique that tries to recover the initial state of every component FSR/MPR/LFSR by exploiting observed relationships between these components with knowledge of some keystream bits. [16, 4]
- Siegenthaler first introduces a binary correlation attack as a published-key recovery attack on LFSRs, upon which Fast Correlation attacks are a significant improvement.
- We explore a generalization of the binary correlation attack and fast correlation attack using the multivariate correlation attack to improve on the typical order of 2^n time complexity of such attacks.

Correlation Attacks - Fast Correlation Attacks

Instead of exhaustive search, fast correlation attacks target relations more intelligently by using a two-phased approach:

1. Find a set of suitable parity-check recurrence relations.
2. Use said relations to decode initial state of LSFR. Intuitively, the more relations that are satisfied for a certain bit, the higher the probability that the bit is correct. [16]

Why is this important?

- It presents us with a better understanding behind the intuition of correlation attacks.
- Acts as a setup to multivariate correlation attacks.
- General concept is promising, and this form of statistical attacks have not yet been fully explored.

Limitations.

- Only semi-applicable to LSFRs because of univariate approach.

Correlation Attacks - Multivariate Correlation Attacks Implementation

- Goal: Find particular bits which when modified or given maximize the conditional likelihood of certain parity relationships holding in the rest of the recurrence (for an LFSR) and system (generally). [4]
- Building off of the intuition of fast correlation attacks, multivariate correlation attacks generalize by introducing metrics more non-specific than parity checks for linear recurrences (which may not exist for an only N-bit long keystream for systems that extend beyond just LFSRs).
    - Current implementation introduces further modifications such as using FFT as key statistic metric to reduce time complexity and trade it off for increase in space complexity of running attack.

Outline

Future Work on Linear Complexity

Work on investigating the linear complexity of CMPRs is mostly complete (have results from root counting and an estimation algorithm based on it).

However, some remaining future work can be done on running more simulations on larger-size CMPRs on HPC.

In particular, getting some experimental lower bounds on linear complexity would be nice.

Future Work on de Bruijn Sequences

Lots of simulation and testing.

1. Run raw linear complexity tests on cross-joined NLFSRs vs. cascade products vs. CMPRs
    - We expect this to be fairly even between all 3 since the de Bruijn constructions have guaranteed high linear complexity and we have probabilistic results on the linear complexity of CMPRs
2. Try other metrics of comparing the *growth* of linear complexity (e.g. the previously discussed variance idea, another possible idea is given in [24])
    - We are expecting this to be better for the CMPRs, due to the potential weaknesses of de Bruijn constructions discussed earlier (specifically in the cross-join method) that we think are not present in CMPRs
3. Explore the recursive construction of higher-order de Bruijn sequences from existing ones

Future Work on Cryptanalysis

- Understand the key property of CMPRs that result in the fact that few maxterms are found in cube attacks.
- Running multi-variate correlation attacks on HPC (for larger CMPR inputs) to further explore relationships and vulnerabilities between knowing increasing numbers of bits of input MPRs.
- Explore feasibility of fast correlation attacks as well as other statistical attacks.
- Determine which multivariate correlation attacks are most promising; implement variety of correlation attacks.
- Reading papers regarding Deep Learning with stream ciphers and implementing similar networks for the CMPR (recent re-exploration).

Outline

References I

[1] E. Berlekamp. "Non-binary BCH decoding". In: (1966).

[2] E. Biham and A. Shamir. "Differential cryptanalysis of DES-like cryptosystems". In: (1991).

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

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

[5] N. Courtois. "Fast algebraic attack on stream ciphers with linear feedback". In: (2003).

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

References II

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

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

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

[10] D. Gordon et al. "Product Register Design and Applications (final report)". In: (2021).

[11] D. Green and K. Dimond. "Nonlinear product-feedback shift registers". In: (1970).

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

References III

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

[14] M. Li et al. "De Bruijn Sequences from Joining Cycles of Nonlinear Feedback Shift Registers". In: (2015).

[15] M Matsui. "Linear cryptanalysis method for DES cipher". In: (1993).

[16] W. Meier. "Fast Correlation Attacks: Methods and Countermeasures". In: (2011).

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

[18] J. Mykkeltveit and J. Szmidt. "On cross joining de Bruijn sequences". In: (2013).

[19] H. Qi. "Stream Ciphers and Linear Complexity". In: (2007).

[20] B. Rosenberg. "Linear Feedback Shift Registers". In: (2009).

References IV

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

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

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

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