Application of LFSRs for Parallel Sequence Generation in Cryptologic Algorithms

Sourav Mukhopadhyay and Palash Sarkar

Cryptology Research Group
Applied Statistics Unit
Indian Statistical Institute
203, B.T. Road, Kolkata
India 700108
e-mail:{sourav_t,palash}@isical.ac.in

Abstract We consider the problem of efficiently generating sequences in hardware for use in certain cryptographic algorithms. The conventional method of doing this is to use a counter. We show that sequences generated by linear feedback shift registers (LFSRs) can be tailored to suit the appropriate algorithms. For hardware implementation, this reduces both time and chip area. As a result, we are able to suggest improvements to the design of DES Cracker built by the Electronic Frontier Foundation in 1998; provide an efficient strategy for generating start points in time-memory trade/off attacks; and present an improved parallel hardware implementation of a variant of the counter mode of operation of a block cipher.
Keywords: DES Cracker, TMTO, Counter Mode of Operation, LFSR.

1 Introduction

Consider the following cryptologic algorithms which require the generation of a sequence of s-bit vectors.

Exhaustive Search: In this case, the search space consists of all elements of $\{0,1\}^s$ and the algorithm must consider each element of this set. Exhaustive search algorithms like the DES Cracker [1] employ a high degree of parallelism. Hence, the requirement is to generate in parallel a set of pairwise disjoint sequences of s-bit vectors whose union is the set $\{0,1\}^s$.

Time-Memory Trade-Off (TMTO): This is also a generic search method. The pre-computation phase of such algorithms require the generation of parallel independent (pseudo)-random sequences of s-bit values.

Counter Mode of Operation: This is a mode of operation of a block cipher, which converts the block cipher into an additive stream cipher. In this mode of operation, one requires to generate a long non-repeating sequence of s-bit values.

The first two are cryptanalytic algorithms, while the third one is a cryptographic algorithm. Implementations of the above three algorithms use a counter to generate the required sequences. While this is intuitively simple, it is not the best possible option for hardware implementation.

In this paper, we explore the possibility of using sequences obtained from linear feedback shift registers (LFSRs) for the hardware implementation of the above algorithms. In each case, we show how LFSR sequences can be tailored for use in the respective algorithm. Replacing counters by LFSRs in hardware implementation has the following two advantages.

Time: The next state of an LFSR can be obtained in one clock. For a counter, we need to add one to the current value. Addition requires much more time. We also show that parallel generation of pairwise disjoint subsequences can be done very efficiently.

Chip Area: Implementing the next state function of an LFSR in hardware requires only a few XOR gates. In contrast, sophisticated carry-look-ahead adders require significantly more circuitry. Consequently, replacing adders by LFSRs will reduce the required chip area.

A combination of the above two effects can lead to a significant improvement in price-performance ratio. This leads us to suggest changes to the DES Cracker [1] which simplify the design as well as reduce the time; to provide an efficient strategy for generating start points in hardware implementation of TMTO algorithms; and finally, to present a new variant of the classical counter mode of operation of a block cipher. This new variant has a more efficient parallel hardware implementation.

Related Work: The DES Cracker [1] is the most famous implementation of special purpose cryptanalytic hardware. Descriptions of hardware implementations of TMTO algorithms can be found in [8, 11]. The work in [9] also suggest the use of LFSRs in TMTO algorithm. However, the suggestion in [9] is to use LFSR for function generation, whereas, our work considers the use of LFSRs for start point generation.

2 LFSR Preliminaries

A binary linear feedback shift register (LFSR) of length s is an s-bit register. Let at time $t \geq 0$, the content of stage i be $a_i^t \in \{0, 1\}$ for $0 \leq i \leq s-1$. Then the state at time t is given by the vector $S_t = (a_{s-1}^{(t)}, a_{s-2}^{(t)}, \ldots, a_0^{(t)})$. The state at time $t+1$ is given by the vector $S_{t+1} = (a_{s-1}^{(t+1)}, a_{s-2}^{(t+1)}, \ldots, a_0^{(t+1)})$, where $a_i^{(t+1)} = a_{i+1}^{(t)}$ for $0 \leq i \leq s-2$; and $a_{s-1}^{(t+1)} = c_0 a_{s-1}^{(t)} c_1 a_{s-2}^{(t)} \oplus \cdots \oplus c_{s-1} a_0^{(t)}$. The values $c_0, \ldots, c_{s-1}$ are constant bits and the polynomial $p(x) = x^s \oplus c_{s-1}x^{s-1} \oplus c_{s-2}x^{s-2} \oplus \cdots \oplus c_1 x \oplus c_0$ over $GF(2)$ is called the connection polynomial of the LFSR. The behaviour of an LFSR is described by the sequence $S_0, S_1, \ldots$ of s-bit vectors and is completely determined by the state S_0 and the polynomial $p(x)$.

Below we highlight some features of LFSRs which are relevant to our work. See [6, 7] for more details and theory of LFSR sequences, including non-binary LFSRs. Additionally, we would like to point out that even though we consider only LFSRs in this paper, our ideas carry over in a straightforward manner to other linear finite state machines like cellular automata.

Maximal length LFSR: It is well known that if $p(x)$ is a primitive polynomial, then for any non-zero s-bit vector S_0, the sequence $S_0, S_1, S_2, \ldots, S_{2^s-2}$ consists of all the 2^s-1 non-zero s-bit vectors. An LFSR which has this property is called a maximal length LFSR. The number of primitive polynomials of degree s over $GF(2)$ is given by the expression $\phi(2^s-1)/s$, where $\phi(i)$ is the Euler totient and is defined to be the number of positive integers less than i and co-prime to i. The expression $\phi(2^s-1)/s$ is almost as large as 2^s and hence there are a large number of maximal length LFSRs of a certain degree. Further, maximal length LFSR sequences satisfy certain well defined pseudorandomness properties and hence such sequences are used in generating test vectors.

Matrix representation: There is another way to view an LFSR sequence, which will be useful to us later. The next state S_{t+1} is obtained from state S_t by a linear transformation and hence we can write $S_{t+1} = S_t M$, where M is an $s \times s$ matrix whose characteristic polynomial is $p(x)$. Extending this, we can write $S_t = S_0 M^t$. Thus, knowing M^t, we can directly jump from S_0 to S_t without going through the intermediate states. For any fixed value of $t < 2^s - 1$, computing the matrix exponentiation M^t can be done using the usual square and multiply method and requires at most $2 \log t \leq 2s$ matrix multiplications. Appropriate addition chain heuristics can speed up the computation. Later we will apply this idea for parallel generation of subsequences of the sequence $S_0, S_1, \ldots, S_{2^s-2}$.

Implementation: Implementing an LFSR in hardware is particularly efficient. Such an implementation requires s flip-flops and $\mathsf{wt}(p(x)) - 1$ many 2-input XOR gates, where $\mathsf{wt}(p(x))$ is the number of non-zero coefficients in $p(x)$. With this hardware cost, the next s-bit state is obtained in one clock. For maximal length LFSR, one requires $p(x)$ to be primitive. It is usually possible to choose $p(x)$ to be of very low weight, either a trinomial or a pentanomial. Thus, an s-bit maximal length LFSR provides a fast and low cost hardware based method for generating the set of all non-zero s-bit vectors. Software generation of an LFSR sequence is in general not as efficient as in hardware. On a machine which supports w-bit words, the next s-bit state of an LFSR can be obtained using $(\mathsf{wt}(p(x)) - 1)s/w$ XOR operations (see [3]).

2.1 LFSRs versus Counters

The set of all s-bit vectors can be identified with the set of non-negative integers less than 2^s. In certain cryptologic algorithms, the requirement is to generate a sequence of non-negative integers with the only condition that no value should repeat. One simple way of doing this is to generate integers $0, 1, 2, \ldots$ using

a counter. While intuitively simple, this is not the only method of generating non-repeating sequence. One can use an s-bit maximal length LFSR to generate the sequence $S_0, S_1, \ldots$, which is also non-repeating. We next discuss the relative advantages of LFSR and counter sequences with respect to hardware implementation.

Implementing a counter which can count from 0 upto $2^s - 1$ requires an s-bit register and an adder. The task required of the adder is to add one to the current state of the register. Due to carry propagation, the simplest adder implementation will require s clocks in the worst case (and $s/2$ clocks in the average case) to generate the next value. More sophisticated carry-look-ahead adders can reduce the number of clocks but the circuitry becomes significantly more complicated and costlier. In contrast, for LFSR sequences, apart from the s-bit register, we require only $\mathsf{wt}(p(x)) - 1$ many 2-input XOR gates and the next s-bit state is obtained in one clock cycle.

Another advantage is that of scalability. The main cost of implementing an LFSR is the register and the interconnections. The number of XOR gates can usually be taken to be either two or four and can be assumed to be less than ten for all values of s. Thus, the cost of implementing an LFSR scales linearly with the value of s. On the other hand, the cost of implementing an adder circuit scales quadratically with the value of s.

Hence, using an LFSR in place of a counter leads to significantly lower hardware cost and also provides a faster method of generating a non-repeating sequence. Additionally, for certain applications, the requirement is to generate a pseudorandom sequence of non-negative integers. In such cases, the only option is to use an LFSR sequence.

3 Parallel Sequence Generation

Consider the following problem.

- Generate n parallel and pairwise disjoint sequences of s-bit strings such that the union of these n sequences is the set of all (non-zero) s-bit strings.

We provide a simple LFSR based strategy for solving the above problem. Let $L = (s, p(x))$ be an s-bit LFSR where $p(x)$ is a primitive polynomial of degree s over GF(2). Let $2^s - 1 = \tau \times n + r = (\tau + 1)r + \tau(n - r)$ where $0 \leq r < n$. Let $n_1 = n - r$, $n_2 = r$ and note that $\tau = \lfloor \frac{2^s-1}{n} \rfloor$. Let S_0 be any nonzero s-bit string and for $t \geq 1$, we define $S_t = S_0 M^t$, where M is the state transition matrix of L. Further, let $T_0 = S_{n_1\tau}$ and for $t \geq 1$, $T_t = T_0 M^t = T_{t-1} M$. Also let $\tau' = \lceil \frac{2^s-1}{n} \rceil$. Define n sequences as follows.

$$\begin{array}{llll} \mathcal{S}_0: & S_0, S_1, \ldots, S_{\tau-1}; & \mathcal{T}_0: & T_0, T_1, \ldots, T_{\tau'-1}; \\ \mathcal{S}_1: & S_\tau, S_{\tau+1}, \ldots, S_{2\tau-1}; & \mathcal{T}_1: & T_{\tau'}, T_{\tau'+1}, \ldots, T_{2\tau'-1}; \\ \vdots & & \vdots & \\ \mathcal{S}_{n_1-1}: & S_{(n_1-1)\tau}, \ldots, S_{n_1\tau-1}; & \mathcal{T}_{n_2-1}: & T_{(n_2-1)\tau'}, \ldots, T_{n_2\tau'-1}. \end{array} \tag{1}$$

The $\mathcal{S}$ sequences are of length τ, while the $\mathcal{T}$ sequences are of length $\tau' \geq \tau$. Note that, $T_{n_2\tau-1} = T_0 M^{n_2\tau'-1} = S_0 M^{n_1\tau} M^{n_2\tau'-1} = S_0 M^{n_1\tau+n_2\tau'-1} = S_0 M^{2^s-2} = S_{2^s-2}$. Since $p(x)$ is primitive, the sequence $S_0, S_1, \ldots, S_{n_1\tau-1}, T_0, T_1, \ldots, T_{n_2\tau'-1}$ consists of all non-zero s-bit vectors. This ensures that the sequences $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{n_1-1}, \mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_{n_2-1}$ are pairwise disjoint. Thus, we obtain a solution to the problem mentioned above. We now consider the problem of actually generating the sequences in hardware.

Implementation: Let $L_0, \ldots, L_{n-1}$ be n implementations of the LFSR L. Hence, each L_i has $p(x)$ as its connection polynomial. The initial conditions for $L_0, \ldots, L_{n_1-1}$ are $S_0, S_\tau, \ldots, S_{(n_1-1)\tau}$ respectively and the initial conditions for $L_{n_1}, \ldots, L_{n-1}$ are $T_0, T_{\tau'}, \ldots, T_{(n_2-1)\tau'}$ respectively. At any point of time, the current states of the L_i's provide the current values of the $\mathcal{S}$ and the $\mathcal{T}$ sequences. All the L_i's operate in parallel, i.e., they are all clocked together and hence the next states of the $\mathcal{S}$ and the $\mathcal{T}$ sequences are generated in parallel. The total hardware cost for implementing the n LFSRs consists of $n \times s$ flip-flops and $n \times (\mathsf{wt}(p(x) - 1)$ many 2-input XOR gates. With this minimal hardware cost, the parallel generation of the $\mathcal{S}$ and the $\mathcal{T}$ sequences become possible.

Obtaining the initial conditions: We explain how to obtain the initial condition for the n LFSRs. Let $M_1 = M^\tau$ and $M_2 = M^{\tau+1} = M \times M_1$. Then $S_{i\tau} = S_0 M^{i\tau} = S_{(i-1)\tau} \times M^\tau = S_{(i-1)\tau} \times M_1$. Now $T_0 = S_{(n_1-1)\tau} \times M_1$ and $T_{j\tau'} = T_{(j-1)\tau'} \times M_2$. Once we know M_1 and M_2 it is easy to find all the $S_{i\tau}$'s and $T_{j\tau'}$'s. Computing M_1 requires a matrix exponentiation which as mentioned before requires $2 \log \tau \leq 2s$ many matrix multiplications. Obtaining M_2 from M_1 requires one matrix multiplication. After M_1 and M_2 have been obtained, computing the initial conditions require a total of n many vector-matrix multiplications. These initial conditions are obtained once for all in an offline phase. These are then pre-loaded into the LFSRs and do not need to re-computed during the actual generation of the parallel sequences.

4 Application 1: The DES Cracker

The data encryption standard (DES) was the industry standard block cipher from the mid-seventies to around the end of the nineties. The best attack on DES was implemented by a civil liberties group called the Electronics Frontier Foundation (EFF) in the year 1998 at a cost of around 200,000 USD (80,000 USD for man power + 120,000 USD for production). EFF built a special purpose machine called DES Cracker [1] for performing a ciphertext only *exhaustive search* attack on DES. The DES Cracker is given two ciphertexts which are encryptions of two English plaintext messages using the same secret key. The goal is to find the secret key. The DES cracker was able to find the secret key in about three and a half days.

In the design of DES cracker, a computer drives 2^{16} *search units*. The search units are parallel hardware units while the computer provides a central control

software. The key space is divided into segments and each search unit searches through one segment. For each candidate key, a search unit does the following. Let k be the current candidate key. A search unit decrypts the first ciphertext using k and checks whether the resulting plaintext is "interesting". If yes, then it decrypts the second ciphertext using k and checks if it is also interesting. (The search unit considers a plaintext to be interesting if all its 8 bytes are ASCII.) If the both plaintexts are found to be interesting then the (key, plaintext) pair is passed to a computer to take the final decision. The search unit then adds one to k to obtain the next candidate key.

Recall that in DES, the message and cipher block size is 64 bits while the key size is 56 bits. In each search unit, a counter (and an adder) generates the candidate keys. A 32-bit counter is used to count through the bottom 32 bits of the key. The reason for using a 32-bit adder is that it is cheaper to implement than a 56-bit adder. The top 24 bits of the key are loaded onto the search unit by the computer. After completing 2^{32} keys with a fixed value of the 24 bits, a search unit sends a signal to the computer. The computer stops the chip; resets the key counter; puts a new value in the top 24 bits; and the search starts once more with this new 24-bit value.

4.1 LFSR Based Solution

We describe an alternative LFSR based solution for candidate key generation in the DES cracker. This solution is based on the parallel sequence generation described in Section 3. The number of parallel search units $n = 2^{16}$, while $s = 56$. Thus, $\tau = 2^{40} - 1$, $\tau' = 2^{40}$, $n_1 = 1$ and $n_2 = 2^{16} - 1$.

Choose the LFSR L such that $p(x)$ is the primitive pentanomial $(x^{56} + x^{22} + x^{21} + x + 1)$. Choose S_0 to be an arbitrary non-zero 56-bit value and compute the values $T_0, \ldots, T_{n_2-1}$ using the method of Section 3. The total number of 56×56 binary matrix multiplications required is at most $2 \times s + 1 = 113$. Additionally, one has to compute a total of 2^{16} many multiplications of a 56-bit vector with a 56×56 binary matrix. Even with a straightforward software implementation, the entire computation can be completed within a few hours. The initial condition of the LFSR in the first search unit is set to S_0, while the initial conditions for the LFSRs in the other search units are set to $T_0, T_1, \ldots, T_{n_2-1}$. Computing the initial conditions can be considered to be part of design stage activity.

In our design, each search unit of the DES cracker has its own implementation of L. This implementation requires n flip-flops and only four 2-input XOR gates. Each search unit now generates the candidate keys independently of the computer and also independently of each other. To obtain the next candidate key, it simply clocks its local LFSR once and uses the state of the LFSR as the candidate key. The first search unit does this for $\tau = 2^{40} - 1$ steps while the other search units do this for $\tau' = 2^{40}$ steps. This ensures that all non-zero keys are considered, with the all-zero key being considered separately.

4.2 Comparison to the Counter Based Solution

There are two ways in which the LFSR based solution improves over the counter based solution.

- There are 2^{16} search units. In the counter based solution, each search unit sends an interrupt signal to the computer after completing an assigned key segment. Thus, the computer needs to handle a total of 2^{24} interrupts from all the search units. In the LFSR based solution, candidate key generation is done solely by the search unit without any involvement from the computer.
- In the counter method, each search unit requires a 32-bit adder for a total of 2^{16} such adders. In contrast, in the LFSR based solution, the circuitry for generating the next candidate key consists of only 4 XOR gates per search unit. Thus, the adders of the counter based method take up significantly more chip area which could be utilised otherwise. One could either build more parallel search units at the same cost, or build the same number of search units at a lesser cost.

The above two factors can lead to a substantial improvement in the price-performance ratio of the DES cracker.

4.3 General Exhaustive Search

The LFSR based candidate key generation algorithm described above for DES cracker can easily be generalized to generate candidate keys for exhaustive search on any cryptographic algorithm. We need to choose the appropriate value of s (for example, for AES, $s = 128$) and a suitable primitive polynomial of degree s over $GF(2)$. Now given n, the number of parallel search units, we can apply the method of Section 3 to obtain the initial conditions of the local LFSR implementations of all the search units. This in effect divides the entire key space into disjoint subspaces, with each search unit searching through its alotted subspace.

5 Application 2: TMTO Pre-Computation

In 1980, Hellman [5] described a chosen plaintext time/memory trade-off attack for block ciphers. For a fixed chosen plaintext msg, define the function $f(k) = E_k(\mathsf{msg})$, where E is the encryption function of a block cipher and k is an s-bit key. The function f maps keys to the ciphertexts. In the attack stage, a target cpr is provided which is an encryption of msg under an unknown key. The goal of the attack is to find the unknown key.

More generally, TMTO can be considered to be a generic method for inverting a one-way function. This consists of two phases: pre-computation phase and online attack phase. A set of table(s) is constructed during the pre-computation phase. The tables store keys in an off-line phase. In the online phase, an image $y = f(x)$ under an unknown key x is received. The goal is to find the unknown key x by making use of the precomputed tables. The main idea is to store only a

part of the tables. This incurs a cost in the online phase and leads to a trade-off between the memory and time requirements. In the following, we describe the pre-computation of three TMTO methods. We will not propose any modification to the online phase and hence we do not discuss this phase.

Hellman Pre-Computation: The pre-computation phase in Hellman's method consists in preparing several tables. Hellman derives r functions $f_0, \ldots, f_{r-1}$ from f by minor modifications (permuting the output bits of f). Each function is used to prepare one table leading to a total of r tables $\mathcal{M}_0, \ldots, \mathcal{M}_{r-1}$ where each $\mathcal{M}_i$ is an $m \times (t+1)$ table. We describe the construction of $\mathcal{M}_i$. Let $x^{(i)}_{0,0}, x^{(i)}_{1,0}, \ldots, x^{(i)}_{m-1,0}$ be a set of points chosen independently and uniformly at random from $\{0,1\}^s$. These m points form the first column of $\mathcal{M}_i$. For $0 \leq j_1 \leq m-1$ and $0 \leq j_2 \leq t-1$, the other entries of $\mathcal{M}_i$ are obtained using the rule $x^{(i)}_{j_1,j_2+1} = f_i(x^{(i)}_{j_1,j_2})$. Thus, each row is a chain of the form: $x^{(i)}_{j_1,0};\ x^{(i)}_{j_1,1} = f_i(x^{(i)}_{j_1,0});\ x^{(i)}_{j_1,2} = f_i(x^{(i)}_{j_1,0});\ \cdots;\ x_{j_1,t} = f_i(x^{(i)}_{i,t-1})$. For the table $\mathcal{M}_i$, the pairs of points $(x^{(i)}_{j_1,0}, x^{(i)}_{j_1,t})$ are stored sorted on the second components. The first component is called a start-point and the second component is called an end-point.

Distinguished Point Method: Rivest improved Hellman's technique by incorporating the distinguished point (DP) property. We can define a DP property on the key space K as follows: A key k satisfies the DP property if its first p bits are zero. We choose a maximum chain length t. For each table, we randomly choose m distinct start points from the key space. For each start point, we generate a chain as usual until we reach a DP or until the length of the chain is t. If a DP is encountered in the chain, then we store the tuple (start point, DP point, length of the chain), otherwise the chain is discarded. The tables are sorted in the increasing order of the endpoints. If the same DP occurs in two different tuples, then the tuple with the maximum chain length will be stored.

Rainbow Method: In 2003, Oechslin [10] described a different construction method. In Oechslin's method, a single $m \times (t+1)$ table covers N points in the following manner. This method uses t functions $f_0, \ldots, f_{t-1}$ obtained from f by output modifications as in Hellman's method. The first column of the table is chosen to be a set of random points $x_{0,0}, \ldots, x_{m-1,0}$. The ith row of the table is formed as follows: $x_{i,0};\ x_{i,1} = f_0(x_{i,0});\ x_{i,2} = f_1(x_{i,1});\ \cdots;\ x_{i,t} = f_{t-1}(x_{i,t-1})$. Each such row is called a rainbow chain and the method is called the rainbow method. The set of pairs $(x_{i,0}, x_{i,t})$ is stored sorted on the second component.

5.1 Parallel Implementation

The pre-computation phase is essentially an exhaustive search which is required to be done only once. Practical implementations of TMTO attack will use parallel f-invocation units to perform the pre-computation. The problem that we consider is of generating the start points on chip. We show an LFSR based

method for doing this. But before that, we consider the counter based method (and its disadvantage) proposed in the literature.

Counter Based Start Point Generation: Quisquater and Standaert [11] describe a generic architecture for the hardware implementation of Hellman + DP method. Nele Mentens et al [8] propose a hardware architecture for key search based on rainbow method. A global s-bit counter is used [8] as a start point generator which is connected to each of the processor. This approach has at least the following problems.

- In the analysis of success probability for TMTO, the start points are assumed to be randomly chosen. Using a counter to generate start points violates this assumption.
- Using a global s-bit counter (adder) to generate start points for n processors has the following disadvantage. Some (or all) of the n processors may ask for a start point at the same time. Then there will be a delay since there is only one global counter to generate the start points.
- On the other hand, using n counters will require n adders which can be quite expensive.

LFSR Based Start Point Generation: To generate r tables with size $m \times t$, we require a total of $m \times r$ many s-bit start points. Suppose we have n many processors $P_1, P_2, \ldots, P_n$ available for the pre-computation phase. We may assume $n|m$, since both are usually powers of two and $n < m$.

We choose n distinct primitive polynomials $p_1(x), \ldots, p_n(x)$ and set-up a local start point generator (SPG) for processor P_i as follows. The local SPG is an implementation of a maximal length LFSR L_i with connection polynomial $p_i(x)$. The intial condition S_i for L_i is chosen randomly and loaded into L_i during the set-up procedure. For preparing a single table all the n processors run in parallel. For each table m chains need to be computed. This is done by requiring each processor to compute m/n chains. The description of P_i is as follows.

$\mathbf{P}_i$: U_i denotes the current state of L_i;

1. $U_i \leftarrow S_i; j \leftarrow 1;$
2. do while $(j \leq \frac{m}{n})$
3. generate the *chain* with start point U_i;
4. if the *chain reaches an end point* T_i
5. store (S_i, T_i) into Tab_i;
6. $j \leftarrow j + 1;$
7. end if;
8. $U_i = \mathsf{next}_i(U_i);$
9. end do

end.

The function $\mathsf{next}_i()$ refers to clocking LFSR L_i once. In this design, each processor P_i has its own SPG as opposed to a global SPG for all the P_i's. This

simplifies the design considerably while retaining the pseudo-random characteristic of start points. Further, as discussed earlier, implementing the LFSRs is significantly more cost effective and faster than implementing counters in hardware.

6 Application 3: Counter Mode of Operation

In 1979, Diffie and Hellman [4] introduced the counter mode (CTR mode) of operation for a block cipher. This mode actually turns a block cipher into an additive stream cipher. Let $E_k()$ be a $2s$-bit block cipher. The pseudorandom sequence is produced as follows:

$$E_k(\mathsf{nonce}||S_0)||E_k(\mathsf{nonce}||S_1)||E_k(\mathsf{nonce}||S_2)||\ldots,$$

where nonce is an s-bit value and $S_0, S_1, \ldots$ is a sequence of s-bit values. The security requirements are the following.

1. The nonce is changed with each message such that the same (key,nonce) pair is never repeated.
2. The sequence $S_0, S_1, S_2, \ldots$ is a non-repeating sequence.

Usual implementations define $S_i = \mathsf{bin}_s(i)$, where $\mathsf{bin}_s(i)$ is the s-bit representation of the integer i. With this definition, the sequence S_i can be implemented using a counter.

Hardware implementation of CTR mode can incorporate a high degree of parallel processing. The inherent parallelism is that each $2s$-bit block of pseudorandom bits can be produced in parallel. Suppose we have n many processors $P_0, P_1, \ldots, P_{n-1}$ where each processor is capable of one block cipher encryption. Processor P_i encrypts the values $\mathsf{nonce}||S_i$, $\mathsf{nonce}||S_{n+i}$, $\mathsf{nonce}||S_{2n+i}, \ldots$. If S_i is defined to be $\mathsf{bin}_s(i)$, then there are two ways of generating the sequence.

Single adder: With a single adder, the algorithm proceeds as follows. At the start of the jth round ($j \geq 1$), the adder generates the values $S_{n(j-1)}, \ldots, S_{nj-1}$. Then all the processors operate in parallel and processor P_i encrypts $\mathsf{nonce}||S_{n(j-1)+i}$.
Problem: The single adder introduces delay which affects the overall performance of the parallel implementation.

n **adders:** In this case, each P_i has its own adder. Its local counter is initialized to S_i and after each block cipher invocation, the adder adds n to the local counter.
Problem: In this implementation, the cost of implementing n adders can take up chip area which is better utilised otherwise.

6.1 LFSR Based Solution

Note that the only restriction on the sequence $S_0, S_1, \ldots$ is that it is non-repeating. Thus, one can use a maximal length LFSR with a primitive connection

polynomial to generate the sequence. Again there are two aproaches to the design both of which are better than the corresponding approach based on using adders.

Single LFSR: In this case, a single LFSR is used which is initialised with a non-zero s-bit value. For $j \geq 1$, before the start of the jth round, the LFSR is clocked n times to produce the values $S_{n(j-1)}, \ldots, S_{nj-1}$. P_i then encrypts $\mathsf{nonce}||S_{n(j-1)+i}$ as before. Clocking the LFSR n times introduces a delay of only n clocks into the system. This is significantly less than the time required for n increments using an adder.

n **LFSRs:** We can avoid the delay of n clocks by using n different implementations of the same LFSR initialised by suitable s-bit values to ensure that the sequences generated by the implementations are pairwise disjoint. The description of how this can be done is given in Section 3. As discussed earlier, the cost of n separate implementations of the same LFSR scales linearly with the value of n and does not consume too much chip area.

Using the LFSR based method to generate the sequence $S_0, S_1, \ldots$ will lead to an improved price-performance ratio compared to the counter based method. The design must specify the actual LFSR being used, and the required initial condition(s). Since there are many maximal length LFSRs to choose from, this provides additional flexibility to the designer.

6.2 Salsa20 Stream Cipher

Salsa20 [2] is an additive stream cipher which has been proposed as a candidate for the recent Ecrypt call for stream cipher primitives. The core design of Salsa20 consists of a hash function which is used in the counter mode to obtain a stream cipher. Denote by $\mathsf{Salsa20}_k()$ the Salsa20 hash function. Then the pseudorandom stream is defined as follows.

$$\mathsf{Salsa20}_k(v, S_0), \mathsf{Salsa20}_k(v, S_1), \mathsf{Salsa20}_k(v, S_2), \ldots$$

where v is a 64-bit nonce and $S_i = \mathsf{bin}_{64}(i)$. For hardware implementation, we can possibly generate the sequence $S_0, S_1, \ldots$ using an LFSR as described above. This defines a variant of the Salsa20 stream cipher algorithm. We believe that this modification does not diminish the security of Salsa20.

6.3 Discussion

For certain algorithms replacing counters by LFSRs will not provide substantial improvements. For example, hardware implementation of $\mathsf{Salsa20}_k()$ will require an adder since addition operation is required by the Salsa20 algorithm itself. Hence, avoiding the adder for generating the sequence $S_0, S_1, S_2, \ldots$ might not provide substantial improvements. On the other hand, let us consider AES. No adder is required for hardware implementation of AES. Hence, using LFSR(s) to produce the sequence $S_0, S_1, S_2, \ldots$ will ensure that no adder is required for hardware implementation of the counter mode of operation. In this case, the benefits of using LFSRs will be more pronounced.

7 Conclusion

In this paper, we have shown that it is more efficient to use LFSRs instead of counters to generate sequences for use in hardware implementation of certain cryptologic algorithms. Two of the algorithms are cryptanalytic generic search algorithms, while the third algorithm is the counter mode of operation of a block cipher. Since LFSR based counter mode of operation is more efficient than conventional counter mode of operation, future designs of cryptographic co-processors should incorporate the facility of LFSR sequence generation.

References

[1] Electronics Frontier Foundation, Cracking DES, O'Reilly and Associates, 1998.

[2] D. J. Bernstein. Salsa20 specification, ecrypt submission 2005. http://www.ecrypt.eu.org/

[3] S. Burman and P. Sarkar. An Efficient Algorithm for Software Generation of Binary Linear Recurrences, Appl. Algebra Eng. Commun. Comput. 15(3-4): 201-203 (2004)

[4] W. Diffie and M. Hellman. Privacy and Authentication: An Introduction to Cryptography, Proceedings of the IEEE, 67, pp. 397-427, 1979.

[5] M. Hellman. A cryptanalytic Time-Memory Trade-off, IEEE Transactions on Information Theory, vol 26, pp 401-406, 1980.

[6] R. Lidl and H. Niederreriter. Introduction to Finite Fields and their applications, Cambridge University Press, Cambridge, pp 189-249, 1994 (revised edition).

[7] A.J. Menezes, P.C. van Oorschot and S.A. Vanstone. Handbook of Applied Cryptography, pp 195-201. CRC, Boca Raton, 2001.

[8] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. Cracking Unix passwords using FPGA platforms, Presented at SHARCS'05, 2005.

[9] S. Mukhopadhyay and P. Sarkar. Application of LFSRs in Time/Memory Trade-Off Cryptanalysis, in the proceedings of WISA 2005, LNCS, to appear.

[10] P. Oechslin. Making a faster Cryptanalytic Time-Memory Trade-Off, in the proceedings of CRYPTO 2003, LNCS, vol 2729, pp 617-630, 2003.

[11] J.J. Quisquater and F.X. Standaert. Exhaustive Key Search of the DES: Updates and Refinements, Presented at SHARCS'05, 2005.