# Random Number Generator Based on Chained Random Tree for Cryptographic Applications

Hyung-Chul Moon[1], Jong-Koo Park[2]

[1] Electrical and Computer Engineering, Sungkyunkwan University,
300 Cheoncheon-dong, Jangan-gu, Suwon, 440-746 S. Korea
hcmoon@skku.edu
[2] School of Information and Communication, Sungkyunkwan University,
300 Cheoncheon-dong, Jangan-gu, Suwon, 440-746 S. Korea
pjk@yurim.skku.ac.kr

**Abstract.** 우리는 기존의 암호 공격법들로는 분석이 불가능하도록 비선형 자료구조인 트리를 이용하여 구조적으로 안전성을 갖도록 설계한 새로운 의사 난수 생성기인 CRTG 를 제안한다. CRTG 는 하나의 난수 비트를 생성하기 위해 네 번의 XOR 연산만을 수행하나 완전하게 비선형으로 동작하는 알고리즘의 수행으로 인해 내부 상태 변수들은 높은 엔트로피를 유지하며 예측 불가능하게 변한다. 그로 인해 깊이 8 의 트리를 사용해 32 비트의 워드 크기를 갖는 시스템에서 소프트웨어로 구현 시 1.5 Gbps 에 달하는 성능을 낼 수 있고 난수성 입증을 위한 모든 통계 테스트에서도 0.97 이상의 엔트로피 수치를 보여준다. CRTG 는 알고리즘이 매우 짧고 하드웨어 구현 시 병렬처리가 가능하며 다양한 길이를 갖는 시드의 사용이 가능해 초경량 하드웨어 환경에서부터 초고속 네트워크 환경까지 모든 응용에서 사용 가능하다. 또한 우리는 CRTG 의 기본 알고리즘을 이용하여 암호학적으로 안전하게 사용될 수 있도록 CRTG 의 암호 응용을 위한 구현 모델을 제시한다.

## 1 Introduction

난수(random number)는 암호 알고리즘에서의 기본 요소 중 하나로 난수 생성기(Random Number Generator, RNG)는 암호 시스템의 기초 단위가 된다. RNG 는 기본 암호 알고리즘의 키 생성기 뿐만 아니라 각종 암호 프로토콜의 초기벡터(initial vector), 도전(challenge), 논스(nonce), 소금(salt) 등을 만들기 위해 사용되고 스트림 암호(stream cipher)에서의 핵심 모듈인 키 수열 생성기로도 사용된다 [1][2].

참다운 의미의 난수를 생성하기 위해서는 전자적 노이즈나 방사능 붕괴 시간 등의 자연 현상의 랜덤성을 이용한 하드웨어 기반의 RNG 를 사용한다. 그러나 하드웨어 기반의 RNG 는 암호 시스템에서 요구되어지는 처리량을 만족하지 못하는 경우가 많다. 또한 그 출력 수열이 고유의 편차를 가지므로 적절한

보정을 통해 랜덤한 수열로 만들어줘야 한다 [3][4][5]. 이러한 이유로 일반적인 시스템에서는 시드(seed)라 불리는 짧은 길이의 초기 난수를 이용해 매우 효율적으로 난수열을 생성하는 의사 난수 생성기(Pseudo Random Number Generator, PRNG)를 사용한다. 그러나 암호 시스템에서의 PRNG 는 블록 암호나 공개키 암호 알고리즘과 마찬가지로 중요한 안전성적 요소의 하나로 인식되므로 암호학적으로 안전한 PRNG(Cryptographically Secure Pseudo Random Number Generator, CSPRNG)를 사용하여야 한다 [1].

현재 암호 시스템에서의 난수 생성을 위해 표준으로 제정되거나 제품으로 구현되어 사용되고 있는 PRNG 로는 FIPS 186, ANSI X9.17, RSA 사의 BSAFE 와 Counterpane 사의 Yarrow-160 등이 있으며, 이외에도 NIST 에서 규격화시킨 해쉬 함수 기반 DRBG, 블록 암호 기반 DRBG 와 공개키 암호 기반 DRBG 등이 있다. FIPS 186 PRNG 는 해쉬 함수를 기본 알고리즘으로 사용하고, BSAFE 도 MD5 를 기본 알고리즘으로 사용하나, ANSI X9.17 PRNG 는 E-D-E 모드의 삼중 DES 를 기본 알고리즘으로 사용한다. 이처럼 현재 사용되고 있는 대부분의 PRNG 들은 해쉬 함수, 블록 암호, 공개키 암호 등의 이미 기존에 만들어져 사용되고 있는 암호 알고리즘들을 그 기본 알고리즘으로 사용하여 각각 설계되어 있다. 기존에 이미 검증되어 사용되고 있는 암호 알고리즘들을 재사용함으로 인해 각 PRNG 들의 안전성은 그 PRNG 에서 사용되고 있는 기본 알고리즘의 안전성에 근간을 두게 된다. 또한 기존의 암호 알고리즘들은 실제 암호 시스템들에서 하드웨어로 구현되어 있는 경우가 많아 각 PRNG 들은 그 암호 시스템의 하드웨어 모듈을 그대로 이용할 수가 있다 [6][7][8][9][10].

이처럼 현재의 일반적인 웹 환경이나 금융권 등에서 사용되고 있는 각 PRNG 들은 그 기능이 효과적으로 잘 수행되고 있다 [11]. 그러나 다양한 환경에서 정보와 기술의 융합이 이루어지는 유비쿼터스 네트워크(ubiquitous network)나 인터넷의 규모가 점점 더 방대해짐으로 인해 더 높은 성능과 보안 수준이 요구되어지는 광역통신망(Wide Area Network, WAN)의 라우터 등으로 인해 기존의 대부분의 암호 기술들은 성능이 만족스럽지 못하거나 구현 자체가 불가능한 경우도 발생하고 있다 [12][13]. 이러한 초경량 하드웨어 환경이나 초고속 네트워크 환경을 위한 새로운 암호 기술들이 요구되고 있고, 실제로 초기에 연구되기 시작한 무선 센서 네트워크(Wireless Sensor Network, WSN)의 경우에는 다양한 새로운 암호 기술들이 보고되었다 [14][15]. 그러나 아직까지 제대로 검증 받고 학계로부터 인정 받은 연구 결과가 발표되지는 못하고 있고, PRNG 의 경우는 인증이나 암호화 알고리즘 등과는 다르게 아직까지 새로운 연구 결과에 대한 발표도 나오지 못하고 있다. 그래서 본 논문에서는 지금까지의 PRNG 설계 방법과는 전혀 다른 비선형 자료구조를 이용한 새로운 형태의 PRNG 를 제안한다. 우리는 이것을 CRTG(Chained Random Tree Generator)라 부른다.

CRTG 의 기본 알고리즘은 하나의 랜덤 비트(random bit)를 만들기 위해 네 번의 XOR(eXclusive-OR) 연산만을 수행하므로 알고리즘이 매우 단순하다. 그러나 트리 구조의 비선형성을 바탕으로 전체 내부 상태 변수들 중에서 극히 일부분만이 임의의 위치에서 선택되어 연산이 수행된다. 이를 위해 세 개의 체인으로 이루어지는 체인 조합을 운영한다. 결과적으로 한가지의 선형

연산만을 사용하나 내부 상태는 비선형성을 유지하며 변하고 높은 엔트로피(entropy)를 유지한다. 비선형 자료구조를 이용한 설계 방법으로 인해 CRTG 의 기본 알고리즘은 구조적으로 일방향성(one-wayness)을 갖게 되어 알고리즘 분석을 통한 공격이 불가능하다. 이러한 CRTG 의 기본 알고리즘을 이용하여 암호 시스템을 위한 PRNG 로 사용될 수 있도록 보안 강도를 높인 CRTG 의 실제 설계 모델을 제시한다.

본 논문의 CRTG 의 기본 알고리즘은 하드웨어로의 구현을 가정하여 비트별 연산을 수행하는 알고리즘으로 설계된다. 설계된 기본 알고리즘을 바탕으로 소프트웨어 시뮬레이션을 통해 통계적으로 높은 난수성과 빠른 수행 성능을 가짐을 보인다. 또한 CRTG 는 사용되는 트리의 깊이에 따라 시드의 길이와 병렬처리의 단계수가 달라지게 된다. 이러한 특성으로 인해 초경량 하드웨어 환경에서 초고속 네트워크 환경까지 모든 조건에 맞추어 선택적으로 사용 가능하다. 또한 CRTG 의 비선형 자료 구조는 빠른 수행 성능과 높은 엔트로피를 요구하는 모든 암호학 관련 알고리즘들에서 사용 가능하다.

2 장에서는 CRTG 의 기본 알고리즘의 내부 상태가 되는 트리의 구조와 랜덤 비트를 생성하는 알고리즘에 대해서 설명하고 3 장에서는 NIST SP800-22 Statistical Test Suite 을 이용해 통계적으로 훌륭한 난수성을 가짐을 보인다. 4 장에서는 소프트웨어 시뮬레이션을 통해 매우 높은 수행 성능을 가짐을 보이고 5 장에서는 하드웨어 구현 시 갖는 장점과 소프트웨어로의 확장 구현이 가능함을 설명하여 다양한 환경에서 사용될 수 있음을 보인다. 6 장에서는 CSPRNG 의 암호학적 공격들에 안전하도록 CRTG 의 기본 알고리즘을 응용한 실제 설계 모델을 제시하고 그 안전성에 대하여 논한다. 마지막으로 7 장에서 결론을 맺는다.

## 2 The PRNG Design

우리는 다음의 네 가지 조건을 만족하도록 PRNG 를 설계한다.

첫째, 생성된 랜덤 비트는 통계적으로 균등분포(uniform distribution)를 유지해야 한다. 이것은 RNG 가 가져야 할 기본 조건으로 CRTG 는 하나의 데이터 집합을 선형과 비선형의 두 개의 자료구조로 구성하여 정보를 확산시키므로 높은 엔트로피를 얻는다.

둘째, 빠른 성능을 위해 효율적이어야 한다. CRTG 는 하나의 랜덤 비트를 만들기 위해 네 번의 XOR 연산만을 수행한다.

셋째, 암호학적 응용에 사용하기 위한 것이므로 암호학적으로 안전해야 한다. CRTG 는 전체 알고리즘이 트리 구조 자체가 갖는 비선형성과 인위적으로 만들어낸 비선형성들이 서로 상호 작용하여 동작하므로 구조적으로 안전하다.

넷째, 다양한 환경에서 융통성 있게 동작해야 한다. CRTG 는 시스템 성능, 보안 강도, 그리고 사용 목적 등에 맞게 시드의 길이와 수행 속도를 조절할 수 있다.

2.1 장에서는 CRTG 에서 사용되는 트리의 구조를 설명하고, 2.2 장에서 이 트리를 이용해 랜덤 비트를 만들기 위한 알고리즘을 설명한다. 비선형으로

동작하는 것을 확인하기 위해 2.3 장에서는 실제로 만들어진 트리 구조 위에서 알고리즘이 동작하여 랜덤 비트가 만들어지는 예를 보인다.

### 2.1 The Structure of the Entropy Tree

우리는 트리 구조를 기반으로 하여 PRNG 를 설계한다. 기존에 PRNG 에서의 시드는 단순히 연속된 비트들의 나열이었으나 CRTG 는 시드를 포화 이진 트리(full binary tree)의 형태로 재배열하여 이용한다. 우리는 이것을 엔트로피 트리(entropy tree)라 부르며 우리의 PRNG 에서의 엔트로피 풀(entropy pool)이 된다. 생성기의 내부 상태(internal state)를 유지하는 시스템 내부의 레지스터(register)들도 논리적으로 포화 이진 트리의 구조적 형태를 유지한다.

일반적으로 트리의 깊이는 루트 노드(root node)가 있는 최상위 레벨에서 리프 노드(leaf node)가 있는 최하위 레벨까지의 모든 레벨들의 수이다 [16]. 그러나 우리의 엔트로피 트리에서의 루트 노드는 실제 데이터를 갖지 않으므로 루트 노드가 있는 최상위 레벨을 제외한 다른 모든 레벨들의 수를 트리의 깊이로 정의한다. 엔트로피 트리에서 루트 노드는 메모리와 같은 저장 공간을 차지하지 않으나 그 외의 모든 노드들은 각각 독립적으로 사용되는 두 개의 비트로 구성된다. 루트 노드는 가상의 노드로서 논리적으로 포화 이진 트리의 형태를 구성하고 트리의 다른 모든 노드들로 접근하기 위해 그 이름만이 만들어지고 사용된다. 그 외의 다른 모든 노드들은 각각 1 비트의 크기를 갖는 키 비트(key bit)와 노드 비트(node bit)로 구성된다. 트리를 구성하는 모든 키 비트들과 노드 비트들은 랜덤 비트 생성을 위한 시드로 사용된다. Fig. 1 은 우리의 엔트로피 트리의 구조를 보여준다.

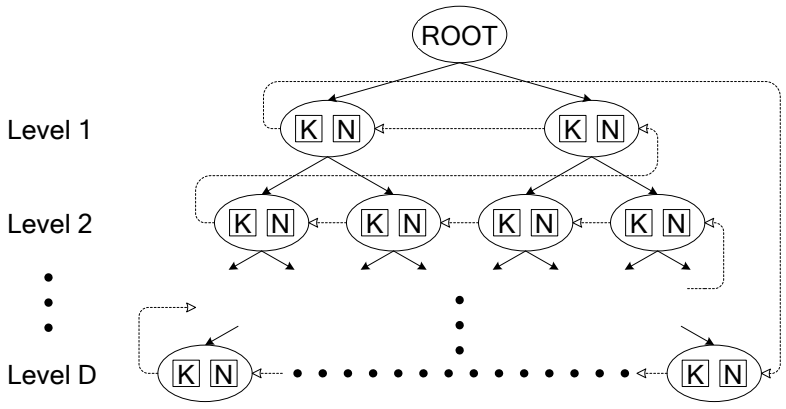


**Fig. 1.** The structure of entropy tree where the *K* is key bit, the *N* is node bit, the *ROOT* is the root node, and the *D* is the depth of the tree. Many nodes of the lower middle part are omitted

CRTG 는 트리 형태의 시드를 사용하므로 트리의 깊이에 의해 시드의 길이가 정해진다. 우리의 엔트로피 트리는 루트 노드를 제외한 포화 이진 트리의 형태이므로 깊이가 $D$ 인 트리의 실제 노드들의 수는 $D \geq 0$ 일 때 $2(2^D - 1)$ 개다.

각 노드들은 2 비트의 크기를 가지므로 실제 시드의 길이는 $D \geq 0$ 일 때 $4(2^D - 1)$ 비트이다. 엔트로피 트리의 깊이에 따른 노드들의 수와 시드의 길이는 Table 1 에서 보여준다.

**Table 1.** The number of nodes and the length of seed by tree depth

| Depth | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|
| Number of nodes | 30 | 62 | 126 | 254 | 510 | 1022 | 2046 | 4094 | 8190 |
| Length of seed (bits) | 60 | 124 | 252 | 508 | 1020 | 2044 | 4092 | 8188 | 16380 |

우리의 엔트로피 트리는 포화 이진 트리 외에 또 하나의 자료구조인 원형 리스트(circular list)를 갖는다. Fig. 1 에서 보면 모든 노드들을 실선으로 연결된 링크를 통해 포화 이진 트리를 구성하고 있다. 그러나 동일한 노드 집합에 대해 점선으로 연결된 또 하나의 링크가 있다. 이것은 이진 트리 순회 방법 중 하나인 레벨 순서 순회를 변형시킨 역 레벨 순서 순회(reverse level order traversal)이다. 우리는 이 순회 방법을 이용하여 단순 연결 원형 리스트(singly linked circular list)를 만든다. 결과적으로 하나의 데이터 집합이 이진 트리와 원형 리스트의 두 개의 자료구조로 동시에 정의되어 사용된다.

PRNG 의 전체 알고리즘이 한 번 수행되는 과정을 한 사이클(cycle)이라 한다. CRTG 는 전체 내부 상태 변수(internal state variable) 중에서 일부분만을 이용해 해당 사이클의 랜덤 비트를 생성한다. 전체 트리에서 세 개의 부분 조각이 선택되어 연산이 이루어지고 각 부분 조각은 트리의 깊이에 해당하는 개수만큼 연속된 노드들로 구성된 체인(chain)이다. 이 세 개의 체인 중 두 개는 이진 트리로부터 세로 방향으로 연속된 노드들로 구성된 하나의 체인이 만들어지고 나머지 하나는 원형 리스트로부터 가로 방향으로 연속된 노드들로 구성된 하나의 체인이 만들어진다. CRTG 의 엔트로피 트리로의 접근은 이 세 개의 체인에 의해서만 가능해진다. 이들 세 개의 체인의 각 노드들의 데이터를 이용하여 트리의 깊이와 동일한 길이를 갖는 스트림(stream) 단위로 각 사이클의 랜덤 비트를 생성한다.

매 사이클마다 각 체인의 위치는 이전 사이클의 체인들의 정보를 이용하여 바뀌게 된다. 원형 리스트에서는 각 사이클의 체인들이 리스트를 따라 머리와 꼬리가 연결된 형태로 순서대로 나열되어 있다. 그래서 각 사이클의 체인은 이전 사이클의 체인의 바로 앞에 위치한다. 포화 이진 트리에서는 첫 번째 레벨에서부터 마지막 레벨에 걸쳐 각 레벨마다 부모 노드(parent node)와 자식 노드(child node)로 연결되는 하나의 노드가 선택되어 세로 방향의 하나의 체인이 만들어진다. 포화 이진 트리에서의 각 사이클의 체인은 이전 사이클의 포화 이진 트리의 체인에서 각 노드들의 노드 비트 값을 이용해 동일한 방법으로 부모 노드와 자식 노드의 관계를 이용해 새로운 하나의 체인이 만들어진다.

CRTG 가 동작하는 전체 시간 동안 각 노드들에 노드 비트의 값들은 높은 엔트로피를 유지한다. 그로 인해 원형 리스트에서와는 다르게 포화 이진 트리에서의 각 사이클을 위한 체인은 트리의 임의의 위치에서 만들어지게 된다. 이것은 CRTG 가 갖는 또 하나의 비선형성으로 우리는 이것을 랜덤 웍스(random

walks)라 부른다. 트리 구조에 의한 비선형성은 알고리즘 동작 시에 자연적으로 적용되나 랜덤 웍스에 의한 비선형성은 인위적인 조작에 의해 만들어진 것이다. 이러한 비선형성들로 인해 알고리즘을 역으로 분석해 PRNG 의 내부 상태 변수들의 값을 찾아내는 것은 매우 어렵게 된다 [17] [18].

### 2.2 The CRTG Algorithm

이번 장의 전반부에서는 CRTG 의 동작을 수식을 통해 정형화하여 정의하고 후반부에서는 프로그래밍 언어에 가까운 형태의 알고리즘으로 기술한다.

우리의 엔트로피 트리는 다음과 같이 두 개의 삼각 행렬(triangular matrix)의 형태로 정의된다.

$$K = \begin{pmatrix} k_{1,1} & k_{1,2} & & & & & \\ \vdots & & \ddots & & & & \\ \vdots & & \vdots & \ddots & & & \\ k_{i,1} & \cdots & k_{i,j} & \cdots & k_{i,2^i} & & \\ \vdots & & \vdots & & & \ddots & \\ k_{D,1} & \cdots & \cdots & \cdots & \cdots & \cdots & k_{D,2^D} \end{pmatrix}$$

$$N = \begin{pmatrix} n_{1,1} & n_{1,2} & & & & & \\ \vdots & & \ddots & & & & \\ \vdots & & \vdots & \ddots & & & \\ n_{i,1} & \cdots & n_{i,j} & \cdots & n_{i,2^i} & & \\ \vdots & & \vdots & & & \ddots & \\ n_{D,1} & \cdots & \cdots & \cdots & \cdots & \cdots & n_{D,2^D} \end{pmatrix} \quad (1)$$

$K$ 행렬은 엔트로피 트리의 모든 노드들에서 키 비트들의 집합이고 $N$ 행렬은 노드 비트들의 집합이다. 행렬의 각 행은 트리의 각 레벨에 해당되므로 인덱스 $i$ 는 트리의 최상위 레벨에서부터 시작되는 각 레벨의 순서 번호이다. $D$ 는 트리의 깊이이므로 마지막 레벨의 번호가 된다. 행렬의 각 행의 원소들은 트리의 각 레벨의 노드들에 해당되므로 인덱스 $j$ 는 트리의 각 레벨 별로 가장 왼쪽 노드에서부터 시작하는 각 노드의 순서 번호이다. 트리에서와 마찬가지로 행렬에서도 $i$ 행의 원소의 개수는 $2^i$ 개고 각 행렬의 전체 원소의 수는 $2(2^D - 1)$ 개다. $k_{i,j}$ 는 트리의 어느 한 노드의 키 비트이고 $n_{i,j}$ 는 같은 노드의 노드 비트이므로 2 원소 유한체(two-element finite field) $\mathbf{F}_2 = \{0,1\}$ 의 범위 내에서 값을 갖는다 [19].

엔트로피 트리에서와 마찬가지로 동작을 위한 연산은 $K$ 행렬과 $N$ 행렬의 전체가 아닌 일부의 원소들을 이용하여 이루어진다. 이를 위해 포화 이진 트리와 원형 리스트로부터 각각 체인이 만들어져 사용된다는 것을 2.1 장의 후반부에서 언급하였다. 포화 이진 트리로부터 만들어지는 체인의 경우에는 $K$ 행렬과 $N$ 행렬의 각 행마다 하나의 원소가 사용되고 두 행렬에서 동일한 위치의 원소들이 사용된다. 이 원소들의 해당 행에서의 열의 위치는 $D \times 1$의 크기를 갖는 ${}_cP$ 행렬에 의해서 참조된다.

$$ {}_cP = \begin{pmatrix} {}_cp_1 \\ \vdots \\ {}_cp_i \\ \vdots \\ {}_cp_D \end{pmatrix} \qquad \text{where} \quad 1 \le {}_cp_i \le 2^i \tag{2} $$

$c$ 는 현재의 사이클 번호이고 ${}_cp_i$ 는 $c$ 사이클에서 두 행렬에 대하여 모든 $i$ 행마다 유일하게 정해지는 각 원소의 위치인 $j$ 열을 값으로 갖는다.

다음 사이클인 $c+1$ 사이클의 체인인 ${}_{c+1}P$ 는 2.1 장에서 설명한 것과 동일하게 노드 비트들의 집합인 $N$ 행렬로부터 구한다.

$$ {}_{c+1}p_i = 2 \cdot {}_{c+1}p_{i-1} - 1 + n_{i,{}_cp_i} \qquad \text{where} \quad {}_{c+1}p_0 = 1 \tag{3} $$

${}_{c+1}p_{i-1}$ 은 트리에서 $i-1$ 레벨의 노드 번호이므로 $2 \cdot {}_{c+1}p_{i-1} - 1$ 은 트리에서 ${}_{c+1}p_{i-1}$ 의 왼쪽 자식(left child)의 노드 번호가 된다. 따라서 $n_{i,{}_cp_i}$ 의 값이 0 이면 ${}_{c+1}p_i$ 는 ${}_{c+1}p_{i-1}$ 의 왼쪽 자식의 노드 번호가 되고, 1 이면 오른쪽 자식(right child)의 노드 번호가 된다.

$$ {}_{c+1}p_i = \sum_{z=1}^{i} \left( 2^{i-z} \cdot n_{z,{}_cp_z} \right) + 1 \tag{4} $$

수식 (4)는 $n_{z,{}_cp_z}$ 만을 변수로 갖는 수식 (3)의 일반화된 식이다.

2.1 장의 후반부에서 언급하였듯이 한 사이클의 실행 동안 포화 이진 트리로부터 만들어진 두 개의 체인이 사용된다. 이는 엄밀히 말하면 한 사이클 동안에 사용되는 체인은 두 개인데 만들어지는 체인은 하나이다. 현재 사이클에서 만들어지는 체인은 다음 사이클의 ${}_{c+1}P$ 이고 사용되는 체인은 ${}_cP$ 와 ${}_{c+1}P$ 이다. 다시 말해 포화 이진 트리로부터 만들어진 하나의 체인은 연속한 두 개의 사이클에 걸쳐서 사용된다.

원형 리스트로부터 만들어지는 체인은 각 사이클의 체인들이 리스트를 따라 머리와 꼬리가 연결된 형태로 이어진다. $K$ 와 $N$ 행렬에서의 원형 리스트는 각 행의 $k_{i,1}$ 과 $n_{i,1}$ 을 $k_{i-1,2^{i-1}}$ 과 $n_{i-1,2^{i-1}}$ 로 연결하고 $k_{1,1}$ 과 $n_{1,1}$ 을 $k_{D,2^D}$ 와

$n_{D,2^D}$ 로 연결함으로써 얻어진다. 이 원소들의 위치는 $D \times 2$ 의 크기를 갖는 ${}_cF$ 행렬에 의해서 참조된다.

$$ {}_cF = \begin{pmatrix} {}_cf_{1,1} & {}_cf_{1,2} \\ \vdots & \vdots \\ {}_cf_{i,1} & {}_cf_{i,2} \\ \vdots & \vdots \\ {}_cf_{D,1} & {}_cf_{D,2} \end{pmatrix} \qquad \text{where} \quad \begin{matrix} 1 \le {}_cf_{i,1} \le D \ , \\ 1 \le {}_cf_{i,2} \le 2^{{}_cf_{i,1}} \end{matrix} \tag{5} $$

${}_cP$ 행렬에서는 각 행마다 하나의 원소만이 사용되므로 각 원소에 대한 열의 위치 값만을 가졌으나 ${}_cF$ 행렬의 경우에는 각 원소에 대한 행의 위치 값과 열의 위치 값을 갖는다. ${}_cf_{i,1}$ 은 현재 사이클에서 원형 리스트에 의한 체인의 $i$ 번째 원소의 행 값이고 ${}_cf_{i,2}$ 는 열 값이다.

다음 사이클인 $c+1$ 사이클의 체인인 ${}_{c+1}F$ 는 행렬의 원소들을 연결해 만들어진 원형 리스트를 따라 ${}_cF$ 체인에서 시작하여 오른쪽에서 왼쪽으로, 아래쪽에서 위쪽으로의 방향으로 이동하며 순서대로 인접한 새로운 $D$ 개의 노드들을 묶어 하나의 체인이 만들어진다.

$$ \begin{aligned} {}_{c+1}f_{i,1} &= {}_cf_{D,1} + \left\lfloor \frac{1}{{}_cf_{D,2}} \right\rfloor \cdot \left( \left\lfloor \frac{1}{{}_cf_{D,1}} \right\rfloor \cdot D - 1 \right) && \text{for} \quad i = 1 \\ {}_{c+1}f_{i,1} &= {}_{c+1}f_{i-1,1} + \left\lfloor \frac{1}{{}_{c+1}f_{i-1,2}} \right\rfloor \cdot \left( \left\lfloor \frac{1}{{}_{c+1}f_{i-1,1}} \right\rfloor \cdot D - 1 \right) && \text{for} \quad i \ge 2 \end{aligned} \tag{6} $$

수식 (6)은 ${}_{c+1}F$ 체인에서 각 $i$ 번째 노드들의 $K$ 와 $N$ 행렬에서의 새로운 행 값을 구한다.

$$ \begin{aligned} {}_{c+1}f_{i,2} &= {}_cf_{D,2} + \left\lfloor \frac{1}{{}_cf_{D,2}} \right\rfloor \cdot 2^{{}_{c+1}f_{i,1}} - 1 && \text{for} \quad i = 1 \\ {}_{c+1}f_{i,2} &= {}_{c+1}f_{i-1,2} + \left\lfloor \frac{1}{{}_{c+1}f_{i-1,2}} \right\rfloor \cdot 2^{{}_{c+1}f_{i,1}} - 1 && \text{for} \quad i \ge 2 \end{aligned} \tag{7} $$

수식 (7)은 ${}_{c+1}F$ 체인에서 각 $i$ 번째 노드들의 $K$ 와 $N$ 행렬에서의 새로운 열 값을 구한다.

$c$ 사이클을 위한 ${}_{c}P$ , ${}_{c+1}P$ , ${}_{c}F$ 의 세 개의 체인은 이전 사이클의 CRTG 알고리즘의 마지막 단계인 다섯 번째 단계에서 구해진다. CRTG 동작의 첫 번째와 두 번째 단계는 이미 구해진 ${}_{c}P$ 와 ${}_{c+1}P$ 체인을 이용해 $K$ 와 $N$ 행렬에서 해당 원소들의 정보를 확산시킨다.

$$\begin{aligned}
n_{1,{}_{c}p_1} &= n_{1,{}_{c}p_1} + k_{2,{}_{c}p_2} \bmod 2 \\
&\vdots \\
n_{i,{}_{c}p_i} &= n_{i,{}_{c}p_i} + k_{i+1,{}_{c}p_{i+1}} \bmod 2 \\
&\vdots \\
n_{D-1,{}_{c}p_{D-1}} &= n_{D-1,{}_{c}p_{D-1}} + k_{D,{}_{c}p_D} \bmod 2 \\
n_{D,{}_{c}p_D} &= n_{D,{}_{c}p_D} + k_{1,{}_{c}p_1} \bmod 2
\end{aligned} \tag{8}$$

수식 (8)은 $N$ 행렬에서 ${}_{c}P$ 체인의 원소들에 값을 변경시킨다. $N$ 행렬의 ${}_{c}P$ 체인의 원소들은 $K$ 행렬의 ${}_{c}P$ 체인에서 한 행 아래의 원소들과 XOR 연산이 이루어진다. 서로 다른 행의 원소들에 대해 연산을 수행함으로써 순환 자리 이동(cyclic shift)연산의 효과를 얻는다. 이것은 $K$ 와 $N$ 행렬의 각 행의 원소들이 0 으로 수렴하는 것을 막아준다.

$$\begin{aligned}
k_{1,{}_{c}p_1} &= k_{1,{}_{c}p_1} + n_{3,{}_{c+1}p_3} \bmod 2 \\
&\vdots \\
k_{i,{}_{c}p_i} &= k_{i,{}_{c}p_i} + n_{i+2,{}_{c+1}p_{i+2}} \bmod 2 \\
&\vdots \\
k_{D-2,{}_{c}p_{D-2}} &= k_{D-2,{}_{c}p_{D-2}} + n_{D,{}_{c+1}p_D} \bmod 2 \\
k_{D-1,{}_{c}p_{D-1}} &= k_{D-1,{}_{c}p_{D-1}} + n_{1,{}_{c+1}p_1} \bmod 2 \\
k_{D,{}_{c}p_D} &= k_{D,{}_{c}p_D} + n_{2,{}_{c+1}p_2} \bmod 2
\end{aligned} \tag{9}$$

수식 (9)는 $K$ 행렬에서 ${}_{c}P$ 체인의 원소들에 값을 변경시킨다. $K$ 행렬의 ${}_{c}P$ 체인의 원소들은 $N$ 행렬의 ${}_{c+1}P$ 체인에서 두 행 아래의 원소들과 XOR 연산이 이루어진다.

PRNG 의 주 목적인 난수의 생성은 첫 번째와 두 번째 단계의 확산 과정을 거친 후 $K$ 행렬로부터 ${}_{c}P$ 체인의 원소들의 값을 이용해 구한다.

$$r = \sum_{i=1}^{D} \left( 2^{D-i} \cdot k_{i,{}_{c}p_i} \right) \tag{10}$$

$K$ 행렬의 $i$ 행의 원소는 $2^{D-i}$ 자릿수에 해당하는 이진수로서 ${}_{c}p_i$ 를 이용해 $K$ 행렬로부터 $2^D$ 의 자리를 갖는 2 진수가 만들어진다. $r$ 은 이것을 10 진 정수로

변환한 값으로 우리가 얻고자 하는 난수가 된다. 그러나 실제 응용에서는 ${}_cP$ 체인의 원소에 해당하는 노드들의 키 비트들이 스트림의 형태로 그대로 출력되어 사용된다.

난수를 생성하여 출력하는 것은 비밀로 유지되어야 하는 엔트로피 트리의 내부 상태 변수들 중 몇 개의 값이 외부로 내보내지는 것이다. 이것은 공격에 이용될 수도 있다. CRTG 는 이것에 대응하기 위해 한 번에 확산 과정을 더 거친다. 확산의 범위와 횟수가 증가함으로 인해 내부 상태의 엔트로피도 더 높아진다. 다음은 CRTG 동작의 세 번째와 네 번째 단계로 ${}_cP$ , ${}_{c+1}P$ , ${}_cF$ 의 체인을 이용해 $K$ 와 $N$ 행렬에서 해당 원소들의 정보를 확산시킨다.

$$k_{i,{}_c p_i} = k_{i,{}_c p_i} + n_{{}_c f_{i,1},{}_c f_{i,2}} \mod 2 \quad (11)$$

수식 (11)은 $K$ 행렬에서 ${}_cP$ 체인의 원소들에 값을 변경시킨다. $K$ 행렬의 ${}_cP$ 체인의 원소들은 $N$ 행렬의 ${}_cF$ 체인의 원소들과 XOR 연산이 이루어진다.

$$n_{i,{}_{c+1} p_i} = n_{i,{}_{c+1} p_i} + k_{{}_c f_{i,1},{}_c f_{i,2}} \mod 2 \quad (12)$$

수식 (12)는 $N$ 행렬에서 ${}_{c+1}P$ 체인의 원소들에 값을 변경시킨다. $N$ 행렬의 ${}_{c+1}P$ 체인의 원소들은 $K$ 행렬의 ${}_cF$ 체인의 원소들과 XOR 연산이 이루어진다.

지금까지는 CRTG 의 동작을 수식을 통해 정형화하였고 다음부터는 프로그래밍 언어에 가까운 형태의 알고리즘으로 기술한다. 다음부터 기술되는 알고리즘은 Appendix A 에서 깊이가 4 인 CRTG 에 대한 C 프로그램으로 구현하였다.

CRTG 를 소프트웨어로 구현할 때 우리의 엔트로피 트리는 각 노드를 링크 표현(linked representation)으로 나타냄으로써 간결하게 구현된다. 각 노드는 키 비트, 노드 비트의 데이터 영역과 왼쪽 자식, 오른쪽 자식, 그리고 역 레벨 순서 순회시의 다음 노드로의 포인터 영역(pointer field)을 가지고 자체참조 구조(self-referential structure)의 형태로 구성된다. 다음은 C 언어의 문법을 이용하여 노드의 구조를 정의하고 CRTG 의 알고리즘을 기술하기 위해 사용될 세 개의 체인을 생성하는 프로그램 코드이다 [20].

**Prog. 1.** The global variable declaration of CRTG in C language

```
#define DEPTH ( depth_of_tree )
#define CYCLE ( total_length_of_random_bits / DEPTH )

typedef struct s_node {
  int K, N ;
  struct s_node *C[2], *F ;
} NODE ;

NODE *CP[DEPTH+1], *NP[DEPTH+1], *FP[DEPTH+1] ;
```

Prog. 1 에서 K 와 N 은 각각 키 비트와 노드 비트이고 C[2] 선언에 의해 생성되는 C[0]와 C[1]은 각각 트리에서의 왼쪽 자식과 오른쪽 자식으로의 포인터이다. F 는 트리의 노드들을 따라 역 레벨 순서 순회 방법으로 트리를 순회 할 때의 인접한 다음 노드로의 포인터이다. CP, NP, FP 는 노드들로의 포인터로 엔트로피 트리에서 동작하는 세 개의 체인에 해당한다. CP 와 NP 는 포화 이진 트리에서 동작하는 체인들이고 FP 는 원형 리스트에서 동작하는 체인이다. CP, NP, FP 는 포인터들의 배열(pointers' array)로서 CP[0]와 NP[0]는 루트 노드의 메모리 주소를 값으로 갖고 FP[0]는 이전 사이클의 FP 에서 마지막 노드의 주소를 값으로 갖는다. CP[1], NP[1], FP[1]부터 CP[D], NP[D], FP[D]까지는 각각의 체인에 속한 노드들의 메모리 주소를 값으로 갖는다. 프로그램의 동작 시에 시스템 내부의 메모리에 생성된 엔트로피 트리로의 접근은 오직 CP, NP, FP 를 통해서만이 가능하다.

프로그램 최초 실행 시 위에서 정의한 노드 구조를 이용해 시스템 메모리에 엔트로피 트리를 생성하고 실제 키 비트와 노드 비트의 데이터가 될 시드를 이용해 초기화시킨다. Appendix A 의 CRTG 에서 generate_tre() 함수가 이 작업을 수행한다. generate_tre() 함수에서는 C 언어의 rand() 함수를 이용해 키 비트와 노드 비트들을 초기화시키고 있다. 최초의 CP 는 엔트로피 트리의 각 행에서 첫 번째 노드들을 연결한 체인으로 초기화되고 NP 는 이 CP 의 노드 비트들을 이용해 초기화된다. 최초의 FP 는 엔트로피 트리의 마지막 행의 마지막 번째 노드에서 시작하는 체인으로 초기화된다.

다음은 위에서 생성한 세 개의 체인과 엔트로피 트리를 이용해 한 사이클의 랜덤 비트 스트림(random bit stream)을 생성하기 위한 CRTG 의 주요 알고리즘이다.

**Prog. 2.** The core algorithm of CRTG in general algorithm convention

```
Step 1.   CP[1].N ← CP[1].N ⊕ CP[2].K
          CP[2].N ← CP[2].N ⊕ CP[3].K
                            ⋮
          CP[DEPTH-1].N ← CP[DEPTH-1].N ⊕ CP[DEPTH].K
          CP[DEPTH].N ← CP[DEPTH].N ⊕ CP[1].K

Step 2.   CP[1].K ← CP[1].K ⊕ NP[3].N
          CP[2].K ← CP[2].K ⊕ NP[4].N
                            ⋮
          CP[DEPTH-2].K ← CP[DEPTH-2].K ⊕ NP[DEPTH].N
          CP[DEPTH-1].K ← CP[DEPTH-1].K ⊕ NP[1].N
          CP[DEPTH].K ← CP[DEPTH].K ⊕ NP[2].N

Step 3.   for level ← 1 to DEPTH do
            begin
              CP[level].K ← CP[level].K ⊕ FP.N
            end
```

```
Step 4.   for level ← 1 to DEPTH do
            begin
              NP[level].N ← NP[level].N ⊕ FP.K
            end

Step 5.   FP[0] ← FP[DEPTH]
          for level ← 1 to DEPTH do
            begin
              FP[level] ← FP[level-1].F
              CP[level] ← NP[level]
              if CP[level].N = 0
                 then NP[level] ← NP[level-1].C[0]
                 else NP[level] ← NP[level-1].C[1]
            end
```

Prog. 2 의 다섯 단계로 구성된 알고리즘은 앞서서 수식을 통해 정형화시켜 보인 다섯 단계의 CRTG 동작과 일치한다. 첫 번째 단계는 CP 의 노드 비트들을 CP 의 한 레벨 아래의 키 비트들과 XOR 연산을 수행한 값으로 변경한다. 두 번째 단계는 CP 의 키 비트들을 NP 의 두 레벨 아래의 노드 비트들과 XOR 연산을 수행한 값으로 변경한다. 첫 번째와 두 번째 단계가 끝난 후 프로그램의 출력으로 트리의 깊이에 해당하는 만큼의 길이를 갖는 랜덤 비트들을 내보낸다. CP[1].K 에서 CP[D].K 까지가 현재 사이클의 출력으로 사용되는 랜덤 비트 스트림이다. 한 사이클의 동작에서만 보면 CP 의 키 비트들의 출력은 두 번째 단계로 인해 CP 의 키 비트들뿐만 아니라 NP 의 노드 비트들의 정보도 프로그램 외부로 내보내지는 것이라 볼 수 있다. 이것은 공격에 이용될 가능성을 준다. 이것을 방지하기 위해 세 번째와 네 번째 단계를 통해 CP 의 키 비트들과 NP 의 노드 비트들의 값을 한 번 더 변경하여 준다. 세 번째 단계는 CP 의 키 비트들을 FP 의 노드 비트들과 XOR 연산을 수행한 값으로 변경하고 네 번째 단계는 NP 의 노드 비트들을 FP 의 키 비트들과 XOR 연산을 수행한 값으로 변경한다. 마지막 단계는 다음 사이클의 수행을 위한 준비 작업으로 CP, NP, FP 의 위치를 각각 변경한다. 현재 사이클의 FP 는 원형 리스트를 따라 다음 위치로 이동하여 다음 사이클을 위한 FP 가 된다. 현재 사이클의 NP 는 다음 사이클을 위한 CP 가 된다. 다음 사이클을 위한 NP 는 새로 준비된 CP 의 노드 비트들을 이용해 새 위치가 정해진다.

Prog. 2 의 알고리즘은 Appendix A 의 CRTG 에서 generate_prn() 함수로 구현하였다. 실제 구현을 하게 되면 Prog. 2 에서 기술된 알고리즘을 통해 예상되는 것과는 다르게 매우 짧은 길이의 프로그램 코드를 갖게 된다. 이것은 우리의 알고리즘을 단계별로 설명하기 쉽도록 풀어서 기술하였기 때문이다. 또 다른 이유는 이해하기 쉽고 효율적인 프로그램 코드를 만들기 위해 구현 시에 C 언어가 갖는 문법적 특성들을 충분히 이용하였기 때문이다. 그 한 예로 마지막 단계의 if-then-else 로 이어지는 세 줄의 알고리즘은 generate_prn() 함수에서는 조건문을 사용하지 않으면서 한 줄의 간결한 코드로 구현되었다.

우리에 알고리즘은 각 단계 별로 트리의 전체 레벨에 대해 동일한 연산을 수행한다. 이는 하드웨어로 구현 시 각 단계 별로 병렬처리를 가능하게 해준다.

이를 이용하여 세 번째부터 마지막 번째 단계까지는 같은 형식을 갖는 for 문을 사용하여 간결하게 표현되었다. 그러나 첫 번째와 두 번째 단계는 for 문을 사용할 경우 배열의 사용을 위해 추가적인 인덱스 연산이 필요해 for 문 없이 직관적으로 표현하였다. 또한 세 번째부터 마지막 번째 단계까지는 트리의 각 레벨 별로 해당 레벨에 대해서만 연산이 이루어지므로 generate_prn() 함수에서와 같이 하나의 for 문 안에서 구현될 수 있다. 그러나 첫 번째와 두 번째 단계는 서로 다른 두 개의 레벨에 대해 연산이 이루어지므로 다른 단계들과 병행하여 구현될 수 없다.

### 2.3 The Generation of the Random Bits

우리에 CRTG 는 트리가 갖는 비선형성을 기반으로 하여 동작한다. 트리의 비선형성은 포화 이진 트리에서 체인의 운영 시에 랜덤 웍스라는 또 하나의 비선형성을 가능하게 한다. 트리와 랜덤 웍스에 의한 비선형성은 하나의 데이터 집합을 포화 이진 트리와 원형 리스트의 두 개의 자료구조로 운영함으로 인해 또 다른 비선형성을 낳는다. CRTG 에서 실제 사용되는 단위 연산은 XOR 이나 이러한 많은 비선형성들로 인해 내부 상태 변수들은 예측하기가 매우 어렵고 높은 엔트로피를 유지한다. 지금까지는 수식이나 알고리즘을 통해 CRTG 의 동작 방식에 대해 살펴보았다. 그러나 수식이나 알고리즘만으로는 이러한 비선형성들을 이해하기가 힘들다. CRTG 가 비선형으로 동작하는 것을 눈으로 확인하기 위해 이번 장에서는 실제 예를 통해 살펴보도록 한다.

깊이가 4 인 CRTG 에 대하여 한 사이클 동안의 알고리즘 수행을 단계별로 나누어 살펴보도록 한다. 예를 보이기 위한 모든 그림들은 전체 엔트로피 트리에서 현재 사이클 동안에 사용될 노드들만을 보여주도록 한다.

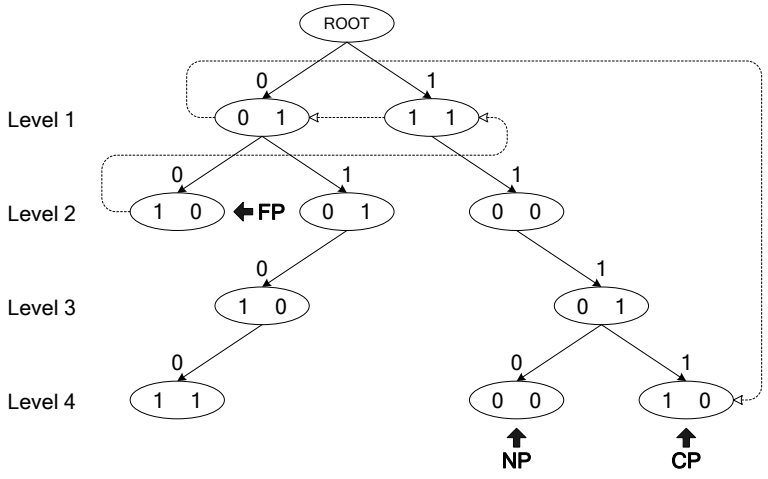

**Fig. 2.** The current state of the entropy tree where the *CP* and *NP* are chains of full binary tree, the *FP* is a chain of circular list, and the *ROOT* is the root node

Fig. 2 는 현재 사이클에서 알고리즘의 첫 번째 단계를 시작하기 바로 전의 엔트로피 트리를 보여준다. 각 노드의 바로 위에 표시된 0 과 1 은 부모 노드에 대해 각 노드가 왼쪽 자식인지 오른쪽 자식인지를 나타내주는 인덱스이다. 이 인덱스는 포화 이진 트리에서 생성되는 CP 와 NP 체인의 위치를 확인하기 위해 사용된다. CP 는 최상위 레벨에서부터 인덱스 1111 을 갖는 체인이고 CP 의 키 비트와 노드 비트는 각각 1001 과 1010 이다. NP 는 인덱스 1110 을 갖는 체인이고 NP 의 키 비트와 노드 비트는 각각 1000 과 1010 이다. 원형 리스트로부터 생성되는 FP 는 Fig. 2 에서 점선의 화살표로 연결된 노드들로 구성되는 체인이고 FP 의 키 비트와 노드 비트는 각각 1101 과 0110 이다.

엔트로피 트리에서 확인하지 않고 알고리즘만을 보고 생각할 때는 각각 네 개의 노드들로 구성되는 세 개의 체인에 의해 모두 12 개의 노드들이 참조되고 있을 것으로 예상할 수 있다. 그러나 Fig. 2 에서 보는 것처럼 서로 중복으로 참조되는 몇 개의 노드들로 인해 실제 참조되고 있는 전체 노드의 수는 7 개이다. 이것은 우리의 엔트로피 트리가 하나의 데이터 집합에 대해 두 가지의 자료구조로 각각 재구성되어 사용되기 때문이다. 또한 포화 이진 트리에서는 랜덤 웍스에 의해 CP 와 NP 가 겹칠 수도 있다.

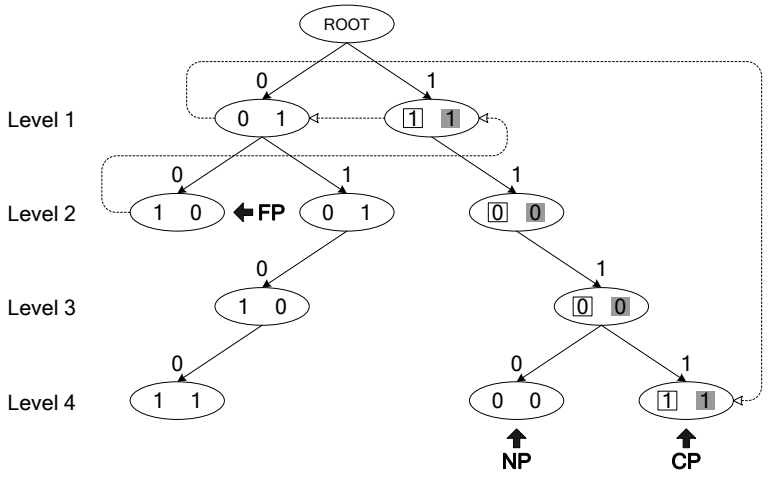


**Fig. 3.** The modified state of the entropy tree by the step 1 of our algorithm where the *CP*'s node bits are updated by the *CP*'s key bits

Fig. 3 은 알고리즘의 첫 번째 단계를 수행한 직후의 엔트로피 트리를 보여준다. CP 의 변경 전 노드 비트인 1010 이 CP 의 키 비트인 1001 을 왼쪽으로 1 비트만큼 순환 자리 이동시킨 0011 과 XOR 연산되어 1001 로 변경되었다. 알고리즘에서는 CP 의 노드 비트만을 변경시키나 실제로는 NP 와 FP 의 노드 비트도 동시에 변경되고 있다.

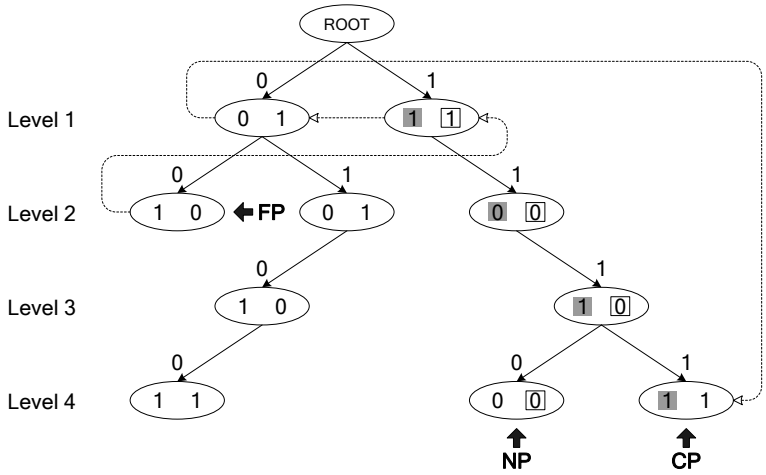


**Fig. 4.** The modified state of the entropy tree by the step 2 of our algorithm where the *CP*'s key bits are updated by the *NP*'s node bits

Fig. 4 는 알고리즘의 두 번째 단계를 수행한 직후의 엔트로피 트리를 보여준다. CP 의 변경 전 키 비트인 1001 이 NP 의 노드 비트인 1000 을 왼쪽으로 2 비트만큼 순환 자리 이동시킨 0010 과 XOR 연산되어 1011 로 변경되었다. 알고리즘에서는 CP 의 키 비트만을 변경시키나 실제로는 NP 와 FP 의 키 비트도 동시에 변경되고 있다. 알고리즘의 두 번째 단계를 수행한 직후 CRTG 의 현재 사이클의 출력으로 CP 의 키 비트인 1011 이 스트림의 형태로 출력된다.

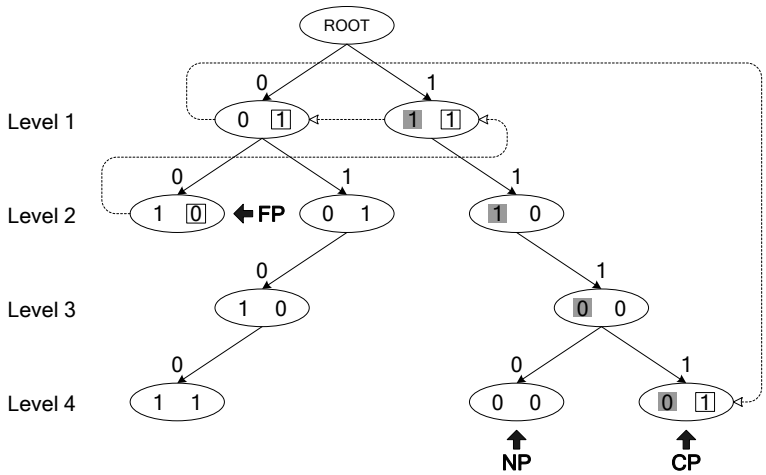

**Fig. 5.** The modified state of the entropy tree by the step 3 of our algorithm where the *CP*'s key bits are updated by the *FP*'s node bits

Fig. 5 는 알고리즘의 세 번째 단계를 수행한 직후의 엔트로피 트리를 보여준다. CP 의 변경 전 키 비트인 1011 이 FP 의 노드 비트인 0111 과 XOR 연산되어 1100 으로 변경되었다. 알고리즘에서는 CP 의 키 비트만을 변경시키나 실제로는 NP 와 FP 의 키 비트도 동시에 변경되고 있다.

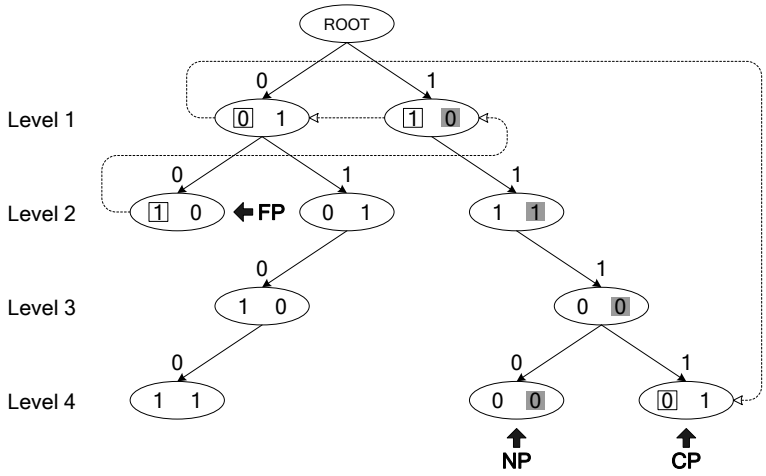


**Fig. 6.** The modified state of the entropy tree by the step 4 of our algorithm where the *NP*'s node bits are updated by the *FP*'s key bits

Fig. 6 은 알고리즘의 네 번째 단계를 수행한 직후의 엔트로피 트리를 보여준다. NP 의 변경 전 노드 비트인 1000 이 FP 의 키 비트인 1100 과 XOR 연산되어 0100 으로 변경되었다. 알고리즘에서는 NP 의 노드 비트만을 변경시키나 실제로는 CP 와 FP 의 노드 비트도 동시에 변경되고 있다.

**Table 2.** The transition of the internal state variables referenced by the *CP*, *NP*, and *FP*

| Variables | Step 0 | Step 1 | Step 2 | Step 3 | Step 4 |
|---|---|---|---|---|---|
| CP.K | 1001 | 1001 | **1011** | **1100** | 1100 |
| CP.N | 1010 | **1001** | 1001 | 1001 | **0101** |
| NP.K | 1000 | 1000 | **1010** | **1100** | 1100 |
| NP.N | 1010 | **1000** | 1000 | 1000 | **0100** |
| FP.K | 1101 | 1101 | 1101 | **1100** | 1100 |
| FP.N | 0110 | **0111** | 0111 | 0111 | **0011** |

알고리즘의 네 번째 단계까지의 수행을 마치면 현재 사이클에서 참조되는 내부 상태 변수들에 대한 실제 데이터의 변경은 끝나게 된다. Table 2 는 알고리즘의 각 단계의 수행 결과인 Fig. 2 에서부터 Fig. 6 까지의 각각의 내부 상태 변수들의 값을 표로 나타낸 것이다. 각 단계에서 변수들의 값이 변경된 경우에는 굵은 글씨체로 표시되었다. 알고리즘에서는 각 단계마다 하나의

변수의 값만을 변경하나 실제 결과는 Table 2 에서 보는 것처럼 둘 또는 세 개의 변수들의 값이 동시에 변하고 있다. 이는 현재 사이클에서 세 개의 체인의 위치적 관계로 인해 일어난다. 다음 사이클의 실행을 위해 알고리즘의 다섯 번째 단계를 수행하고 나면 세 개의 체인의 위치적 관계는 새로운 형태로 구성된다. 그로 인해 내부 상태 변수들의 변화는 현재 사이클의 Table 2 와는 또 다른 변화 형태를 띠게 된다. 이는 우리의 엔트로피 트리가 갖는 여러 비선형적 특징들로 인한 결과로 하나의 데이터 집단이 복잡하게 얽힌 많은 위치적 관계들로 구성되기 때문이다 [21].

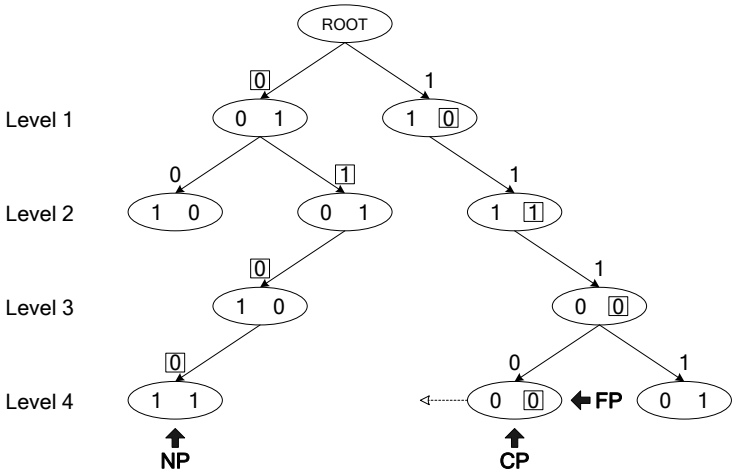


**Fig. 7.** The altered references of the *CP*, *NP*, and *FP* where the current *NP* is changed to new *CP*, new *NP* is traced by the new *CP*'s node bits, and new *FP* is traced on circular list

Fig. 7 은 알고리즘의 다섯 번째 단계를 수행한 직후의 엔트로피 트리를 보여주나 실제 내부 상태 변수들의 값에는 변경이 없다. 알고리즘의 다섯 번째 단계는 다음 사이클의 알고리즘 수행을 위한 준비 단계로 CP, NP, FP 세 개의 체인의 위치만을 변경한다. CP 와 NP 는 포화 이진 트리의 구조를 이용해 각각 다음 위치로 이동한다. 현재 사이클의 NP 의 위치는 다음 사이클을 위한 CP 의 위치가 된다. 다음 사이클을 위한 NP 는 다음 사이클을 위해 준비된 CP 의 노드 비트의 값과 일치하는 인덱스 번호의 위치로 이동한다. FP 는 원형 리스트의 구조를 이용해 다음 위치로 이동한다. 원형 리스트를 따라 이동할 때 현재 사이클의 FP 체인의 마지막 번째 노드의 바로 다음에 인접해 위치한 노드가 다음 사이클을 위한 FP 체인의 첫 번째 노드가 된다. 알고리즘의 다섯 번째 단계를 수행하고 나면 세 개의 체인 서로간에 새로운 위치적 관계가 만들어진다.

원형 리스트를 따라 순차적으로 이동하는 FP 는 내부 상태 변수들의 값으로부터 영향을 받지 않는다. 그러나 랜덤 웍스에 의해 이동하는 NP 는 내부 상태 변수들의 값으로부터 위치가 결정된다. 우리의 엔트로피 트리는 프로그램이 동작하는 동안 항상 높은 엔트로피를 유지하고 있다. 그로 인해 엔트로피 트리의 내부 상태로부터 영향을 받는 NP 의 위치는 항상 무작위적으로 결정된다.

## 3 Statistics

좋은 RNG 는 통계적으로 훌륭한 난수성을 가져야 하고 이는 암호학과 관련된 알고리즘들에도 적용된다. 안전성이 입증된 대부분의 암호학 관련 알고리즘들에 의해 생성된 암호문은 훌륭한 난수성을 보인다. 난수성을 입증하기 위해 우리는 NIST SP800-22 Statistical Test Suite 을 사용한다 [22]. SP800-22 는 15 가지 항목으로 나뉘어 진행되고 각 테스트의 통과기준도 0.01 의 유의수준(significant level)으로 높은 신뢰도를 갖는다. 테스트를 위한 매개변수(parameter)들은 테스트 스위트(test suite)에 기본으로 설정되어 있는 값과 같고 우리는 이것을 바꾸지 않고 사용하여 테스트를 진행한다. 테스트를 위한 시퀀스(sequence)들의 수는 300 개이고 각 시퀀스의 길이는 1 메가 비트(mega bits)이다. 총 300 메가 비트의 테스트 데이터를 가지고 15 가지의 항목에 대하여 반복적으로 테스트가 진행된다. 이번 장의 모든 테스트 결과는 그래프를 통해 보이고 원본 데이터는 Appendix B 에서 보인다. 테스트의 재확인을 위하여 이번 장의 모든 CRTG 의 시드는 SP800-22 프로젝트 웹사이트에서 제공되는 샘플 데이터인 sts.data.tar 파일 내의 BBS.dat 파일을 사용한다 [23].

첫 번째 테스트는 XORG, CRT8G, BBSG 의 세 개의 PRNG 에 대하여 진행한다. XORG(Exclusive OR Generator)는 127 비트의 시드를 사용하고 수식 (13)에 의하여 다음 비트를 생성한다.

$$x_i = x_{i-1} \oplus x_{i-127}, \qquad \text{for } i \geq 128 \tag{13}$$

XORG 는 CRTG 와 연산량의 차이에 따른 성능 비교를 위해 선정한 것으로 둘 다 XOR 를 기본 연산으로 사용한다는 점 이외에는 다른 연관성이 없다. CRTG 는 1 비트를 만들기 위해 네 번의 XOR 연산을 수행하나 XORG 는 한 번의 XOR 연산만을 수행한다. BBSG(Blum-Blum-Shub Generator)는 대표적인 CSPRNG 로 소인수 분해 문제에 기반한 $x^2 \bmod n$ 을 이용한다 [24]. XORG 와 BBSG 는 모두 테스트 스위트 내에 기본 PRNG 로 구현되어 있으며 각 항목을 선택하는 것만으로 각 PRNG 의 랜덤 비트를 생성하여 바로 테스트할 수 있다. XORG 의 시드는 테스트 스위트의 코드 상에 미리 정해져 있으나 BBSG 의 시드인 입력 매개변수들은 시간에 따라 다양한 값으로 매번 다르게 정해진다. CRT8G 는 본 논문에서 제안하는 CRTG 로 깊이가 8 인 엔트로피 트리를 사용한다. CRT8G 는 시드의 길이가 1020 비트로 충분한 보안 강도를 갖고 지금까지 제안된 다른 PRNG 들과 비교하여 초기 엔트로피 풀의 크기에서 큰 차이를 보이지 않는다. 스트림의 길이도 8 비트 단위로 생성되어 스트림 암호로 사용시 시스템에 효율적인 데이터 길이이므로 비교 대상으로 선정하였다.

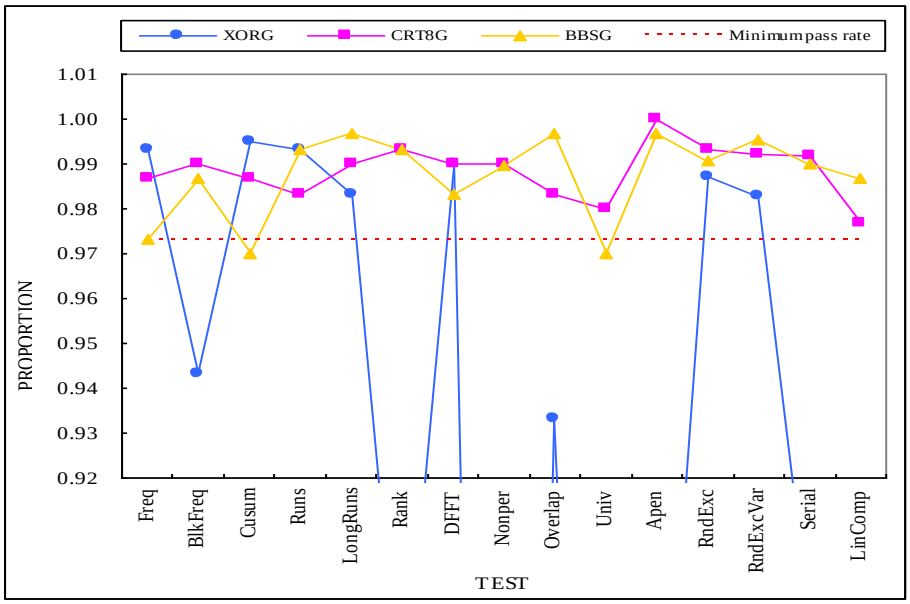


**Fig. 8.** NIST SP800-22 test scores of XORG, CRT8G, and BBSG

Fig. 8 의 그래프에서 X 축은 왼쪽에서부터 Frequency (Monobit), Frequency within a Block, Cumulative Sums, Runs, Longest Run of Ones in a Block, Binary Matrix Rank, Discrete Fourier Transform (Spectral), Non-overlapping Template Matching, Overlapping Template Matching, Maurer's "Universal Statistical", Approximate Entropy, Random Excursions, Random Excursions Variant, Serial, Linear Complexity 테스트의 항목들로 구성되어있다. Y 축은 300 개의 시퀀스들 중에서 각 테스트를 통과한 시퀀스들의 비율로 0 에서 1 까지의 값을 갖는다. 그래프 중에서 최소 통과 비율(minimum pass rate)은 0.01 의 유의수준에 해당하는 값인 0.972766 으로 각 테스트 결과가 이 값보다 크면 해당 테스트를 통과한 것으로 본다. Fig. 8 의 그래프에서 CRT8G 는 모든 테스트 항목에서 최소 통과 비율을 만족한다. 위의 테스트에서 CRT8G 는 시드로 BBS.dat 파일을 사용하고 있으나 BBSG 처럼 매번 다른 시드를 사용한 여러 차례의 반복 테스트에서도 모두 훌륭한 난수성을 보여주었다. BBSG 는 두 테스트 항목에서 최소 통과 비율에 조금 미달되는 결과를 보여주고 있으나 여러 차례의 반복 테스트에서 대체적으로 훌륭한 난수성을 보여주었다. XORG 는 여러 테스트 항목에서 기준에 만족하지 못하였다.

다음 테스트는 CRT4G, CRT9G, CRT16G 의 세 개의 PRNG 에 대하여 진행한다. 세 개의 PRNG 는 모두 본 논문에서 제안하는 CRTG 로 각각 깊이가 4, 9, 16 인 엔트로피 트리를 사용한다.

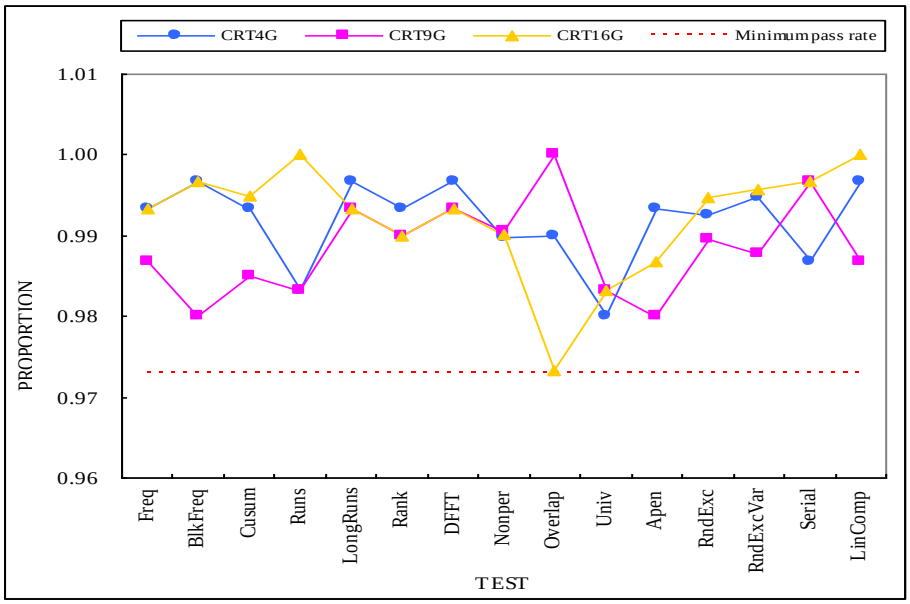


**Fig. 9.** NIST SP800-22 test scores of CRT4G, CRT9G, and CRT16G

Fig. 9 는 CRTG 가 모든 트리 깊이에 대하여 통계적으로 훌륭한 난수성을 가짐을 보이기 위한 것으로 세 개의 PRNG 모두 모든 테스트 항목에서 최소 통과 비율을 만족한다. CRT8G 와 마찬가지로 Fig. 9 의 세 개의 PRNG 들도 매번 다른 시드를 사용한 여러 차례의 반복 테스트에서 기준을 만족한다. 트리 깊이가 4 이상인 모든 CRTG 는 Fig. 9 와 동등한 수준의 난수성을 보인다.

우리의 PRNG 는 암호학적 응용에 어느 한 모듈로 사용하기 위해 설계되었으므로 충분히 높은 엔트로피를 갖는 예측 불가능한 시드를 사용해야 한다. 이번 장의 모든 CRTG 는 BBS.dat 파일을 시드로 사용하였다. BBS.dat 파일은 BBSG 에 의해 생성된 난수열의 일부분으로 충분히 높은 엔트로피를 갖는다. 그러나 우리의 PRNG 는 모든 비트가 0 인 시드를 제외한 그 외의 모든 가능한 시드에서도 Fig. 9 와 같은 좋은 난수성을 보인다. 모든 비트가 1 을 갖는 엔트로피가 0 인 시드의 경우에도 Fig. 9 와 동등한 수준의 난수성을 보인다.

## 4 Performance

수행시간 측정은 프로그램 코드 상에서 ANSI C 의 표준 라이브러리에 포함되어 있는 clock() 함수를 사용한다. clock() 함수는 운영체제가 지원하는 함수로 이것으로부터 CPU 의 클럭 틱(clock tick)을 구해 초단위로 환산하여 측정할 수 있다. 시뮬레이션은 Pentium4 2.4GHz, Windows XP, 2GB Main Memory 의 환경에서 수행하고 3 장에서와 동일하게 총 300 메가 비트의 데이터에 대하여

측정한다. 비교를 위한 PRNG 들의 프로그램 코드는 기본적으로 NIST SP800-22 Statistical Test Suite 에 구현되어 있는 프로그램 코드를 참조한다.

첫 번째 시뮬레이션은 소프트웨어로 구현된 XORG, CRT8G, BBSG 의 세 개의 PRNG 의 랜덤 비트 생성 시간을 측정한다. 정확한 비교 테스트를 위해 세 개의 PRNG 에서 시드를 포함한 입력 매개변수들의 설정과 파일 입출력을 제외한 랜덤 비트를 생성하는 함수의 시간만을 측정한다. XORG 의 프로그램 코드는 테스트 스위트에 있는 것을 기본으로 하여 동일한 알고리즘으로 동작하면서 최상의 성능을 낼 수 있도록 수정하였다. 테스트 스위트에 있는 BBSG 의 프로그램 코드는 ANSI C 를 이용한 Pate Williams 의 프로그램 코드로 우리도 이것을 그대로 이용하였다 [25].

**Table 3.** The CPU time of XORG, CRT8G, and BBSG

| Generators | XORG | CRT8G | BBSG |
|---|---|---|---|
| CPU time (second) | 1.406 | 6.671 | 549.703 |

Table 3 의 시뮬레이션 결과에서 XORG 는 한 번의 XOR 연산의 $300\times10^6$ 번의 수행시간이고 CRT8G 는 네 번의 XOR 연산과 그에 따른 메모리 참조 연산의 $300\times10^6$ 번의 수행시간이다. CRT8G 의 수행시간은 XORG 의 네 배 정도의 수치로 두 알고리즘에서 유일하게 사용되는 연산인 XOR 의 사용빈도에 비례한다. BBSG 는 매우 큰 수 n 을 이용해 법(modulus) 연산을 수행함으로 인해 XOR 연산만을 사용하는 알고리즘들과는 비교할 수 없을 정도로 긴 시간이 소요된다 [26]. Table 3 의 시뮬레이션 결과를 통해 CRTG 의 성능을 가늠할 수 있다. 실제로 CRTG 는 기존의 PRNG 들과 비교하였을 때 최상위의 성능을 보여준다.

다음 시뮬레이션은 CRTG 를 소프트웨어로 구현했을 시 엔트로피 트리의 깊이에 따른 성능의 변화를 보여준다. 시뮬레이션 결과는 그래프를 통해 보이고 원본 데이터는 Appendix B 에서 보인다. 시스템 메인 메모리의 제약으로 4 부터 23 까지의 트리 깊이에 대하여 시뮬레이션을 수행한다.

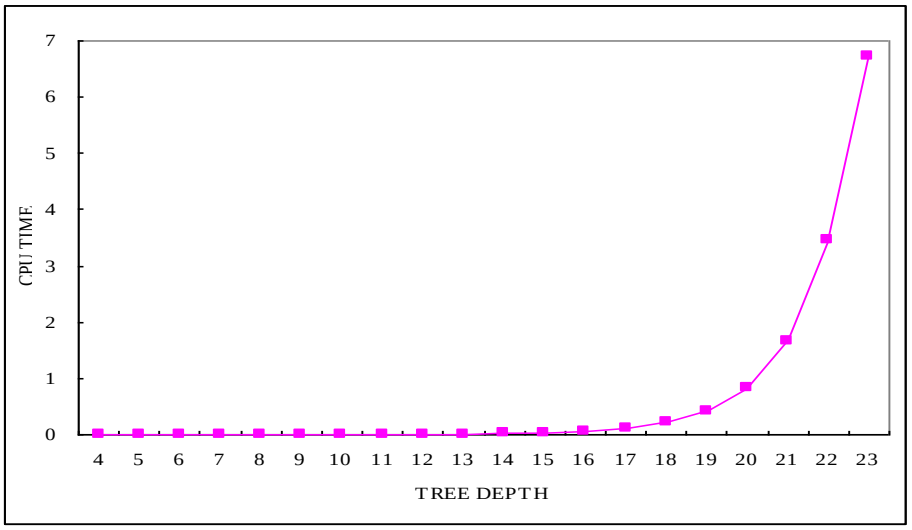


**Fig. 10.** The CPU time of CRTG's tree generation and initialization

Fig. 10 은 CRTG 에서 각 트리 깊이 별로 최초의 내부 상태가 되는 엔트로피 트리를 시스템 메모리에 생성하고 시드를 이용해 초기화하는 동안의 수행시간이다. Fig. 10 에서 각 트리 깊이에 따른 시간의 증가는 두 배 정도의 크기로 지수 증가하고 있다. 이는 시스템 메모리 사용량의 증가에 따른 것으로 Table 1 에서 트리 깊이에 따른 노드의 수가 두 배 정도로 지수 증가하는 것과 일치한다.

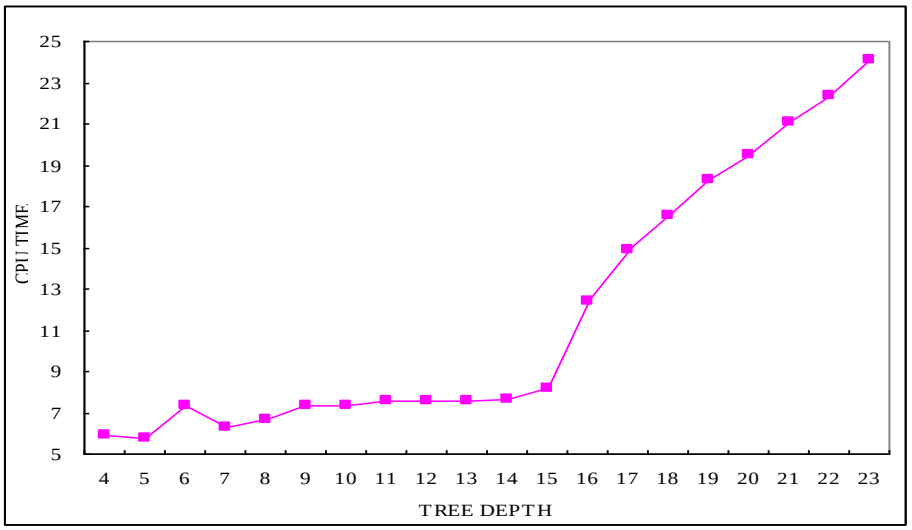

**Fig. 11.** The CPU time of CRTG's pseudo random bits generation

Fig. 11 은 CRTG 에서 각 엔트로피 트리의 깊이 별로 300 메가 비트의 랜덤 비트를 생성하는 동안의 수행시간이다. 이는 네 번의 XOR 연산과 그에 따른 메모리 참조 연산의 $300\times10^6$ 번의 수행시간이다. 네 번의 XOR 연산에 의한 시간은 모든 트리 깊이에 대하여 동일하다. 그러나 트리 깊이가 증가함에 따라 시스템 메모리의 사용량이 증가해 전체적으로 메모리 참조 연산에 많은 부하를 가져온다. 전체 수행시간의 대부분은 XOR 연산에 의한 것이나 트리 깊이가 증가함에 따라 메모리 참조 연산의 시간이 길어져 Fig. 11 과 같은 형태의 그래프가 나타난다. 이는 정확하게 지수 증가하고 있지는 않지만 시스템 메모리 사용량의 증가에 따른 것으로 Fig. 10 과 유사한 형태를 보인다.

Prog. 1 에서는 논리적으로 트리를 구현하기 위해 자체참조 구조를 사용하여 노드를 정의하고 있다. 이는 C 언어의 포인터 연산을 이용하는 것으로 내부적으로 메모리 주소의 참조를 통해 데이터에 접근한다. 이로 인해 메모리의 사용량이 커지고 그 구조가 복잡해 질수록 프로그램 실행시의 성능이 떨어져 Fig. 11 과 같은 결과를 보여준다. 이는 시스템의 구조적인 특성에 따른 것으로 FPGA 와 같은 IC 칩 상에 배선에 의한(hardwired) 트리로 구현할 경우 메모리 참조 연산으로 인한 성능의 저하를 막을 수 있다. 그로 인해 CRTG 를 하드웨어 칩으로 구현했을 시의 수행시간은 트리 깊이에 관계없이 같아진다. 이는 우리의 알고리즘이 동일한 양의 랜덤 비트들을 생성할 시에 모든 트리 깊이에 있어서 동일한 복잡도를 갖기 때문이다 [27].

## 5 Implementation

CRTG 에서 실제 하드웨어 칩 상에 구현되는 부분은 배선에 의한 트리 구조와 매 사이클 마다 이를 반복 조작하는 Prog. 2 의 알고리즘이다. Prog. 2 알고리즘의 소프트웨어 코드는 Appendix A 의 generate_prn() 함수로 단 몇 줄에 해당하는 크기를 갖는다. 그로 인해 알고리즘 구현을 위한 회로 소자(circuit element)의 수가 많지 않아 사용되는 칩의 면적은 매우 작아지게 된다. 또한 하나의 랜덤 비트를 만들기 위해 단 네 번의 XOR 연산이 사용된다. 이는 하드웨어 칩으로 구현 시 매우 낮은 전력소모를 가져온다. CRTG 는 작은 크기의 칩과 낮은 양의 전력소모로 인해 하드웨어 제약조건이 열악한 환경에서 사용될 수 있다. 깊이 4 에서 8 까지의 CRTG 는 WSN 의 터미널 등에서 스트림 암호기로 사용 가능하다 [28] [29].

칩으로 구현 시 트리 깊이와 같은 수의 XOR 연산기와 데이터 버스를 둘 경우 Prog. 2 의 각 단계의 내부에서 병렬처리가 가능해져 각 단계 별로 하나의 CPU 사이클에서 동작시킬 수 있다. 병렬 처리된 CRTG 는 그렇지 않은 것과 비교해 트리 깊이의 배수 배만큼 더 높은 성능을 낼 수 있다. 예를 들어 병렬 처리된 CRT8G 는 병렬 처리되지 않은 CRT8G 와 비교해 여덟 배의 더 높은 성능을 낼 수 있다. 그로 인해 트리 깊이가 더 깊어질수록 더 높은 성능을 낼 수 있다. 모두 병렬 처리된 경우에 CRT8G 는 CRT4G 와 비교해 두 배의 더 높은 성능을 갖게

된다. 병렬처리를 이용한 향상된 성능을 통해 깊이 8 이상의 CRTG 는 초고속 네트워크의 라우터 등에서 스트림 암호기로 사용 가능하다 [30] [31].

3 장과 4 장의 모든 테스트는 일반 PC 상에서 소프트웨어 코드로 구현된 프로그램을 사용하여 시뮬레이션하고 있다. 이 논문에서의 CRTG 는 한 비트를 기본 연산 단위로 가정하고 있으나 실제로는 시스템의 워드 크기 만큼을 사용하여 소프트웨어 코드로 구현되었다. 이는 CRTG 가 하드웨어 칩으로 구현될 것을 가정하여 설계되었기 때문이다. 시스템 내에서 소프트웨어로 구현하여 사용할 경우에는 노드를 구성하는 키 비트와 노드 비트의 크기를 시스템의 워드 크기만큼 확장하여야 실제의 성능을 낼 수 있다 [32]. 그럴 경우 성능은 위 시뮬레이션 결과와 큰 차이가 나지 않으면서 출력은 시스템의 워드 크기에 트리 깊이의 배수 배만큼의 블록단위가 된다 [33]. 그러나 이 논문의 성능 평가의 목적은 하드웨어 구현을 위해 소개된 CRTG 알고리즘에 대하여 소프트웨어 시뮬레이션을 수행하고 그 성능을 가늠하는 것이다.

## 6 Security Analysis

지금까지 우리는 CRTG 의 기본 알고리즘을 설명하고 그것이 갖는 훌륭한 난수성과 수행시간의 성능을 보였다. 앞서 살펴본 바와 같이 CRTG 는 기본 알고리즘에 대한 성능 테스트에서 대략 40~50 Mbps 의 처리량을 갖는다. 그러나 이는 어디까지나 CRTG 의 성능을 대략적으로 가늠하기 위한 것으로서 실제 일반적인 시스템인 32bit 의 워드 크기에 맞게 구현할 경우에는 그것의 32 배인 1.5 Gbps 에 달하는 성능을 낼 수 있다. 이는 암호 응용을 위한 것이 아닌 일반적인 PRNG 들과 비교하여도 최상위 수준에 속하는 성능이다. 그러나 우리의 본래의 설계 목적은 암호 응용에서 안전하게 사용할 수 있는 PRNG 를 만드는 것이다. 이를 위해 이번 장에서는 CRTG 의 암호학적 안전성에 대하여 논한다.

일반적인 PRNG 는 출력 수열에 대하여 확률론적인 관점에서의 난수성으로 평가한다. 그러나 암호 시스템에서의 PRNG 는 블록 암호나 공개키 암호 알고리즘과 마찬가지로 중요한 안전성적 요소의 하나이므로 그 안전성에 대한 평가가 이루어져야 한다. 이러한 PRNG 의 안전성을 수학적으로 정의하기 위해 구별 불가능 개념(indistinguishability)과 예측 불가능 개념(unpredictability)이 사용된다. 먼저 구별 불가능 개념은 다항식 시간 내에서 결정적 다항식 알고리즘(deterministic polynomial algorithm)에 의해 생성된 출력 수열이 랜덤 특성들(random properties)을 유지하여야 함을 의미한다. 다음으로 예측 불가능 개념은 이전에 생성된 출력 수열을 이용해 다음에 생성될 출력 수열에 대한 어떠한 정보도 얻을 수 없어야 함을 의미한다. 또한 이것은 다음 비트 검정(next bit test)을 통과함을 의미한다 [34][35][36]. 그러나 구별 불가능 개념과 예측 불가능 개념은 어디까지나 이론적인 안정성인데 반해 J. Kelsey, B. Schneier, D. Wagner 와 C. Hall 은 암호 시스템에서 사용되는 PRNG 에 대한 몇 가지의 실제적인 가능한 공격법들을 제시하였다 [37].

앞서 언급한 것처럼 일반적으로 CSPRNG 가 되기 위해서는 “next-bit test”와 “state compromise extensions”의 두 가지 조건을 만족하여야 한다 [38]. 이 두 가지 조건을 만족하는 대표적인 CSPRNG 로 BBS(Blum-Blum-Shub)를 예로 들 수 있다 [39]. BBS 는 복잡도 이론의 NP 문제에 해당하는 소인수 분해 문제를 이용한 알고리즘으로 이것의 증명은 “next-bit test”의 입증과 동치 관계에 있다는 것이 증명되었다 [40]. 그러나 ANSI X9.17 과 같은 경우에는 이러한 이론적 증명 없이 표준으로 제정되어 암호 응용에 사용되고 있다. 일반적으로 이런 종류의 PRNG 는 암호 응용에는 사용되나 CSPRNG 로는 부르지 않는다 [1]. 우리가 제안하는 CRTG 또한 아직까지는 “next-bit test”에 대한 효과적인 증명 방법을 찾지 못하였다. 그러나 블록 암호나 해쉬 함수 기반의 PRNG 들과 유사한 방법으로 “state compromise extensions”을 만족하며 암호 알고리즘으로서 충분한 보안 강도를 가질 수 있도록 새로운 PRNG 의 구현 모델을 제시한다.

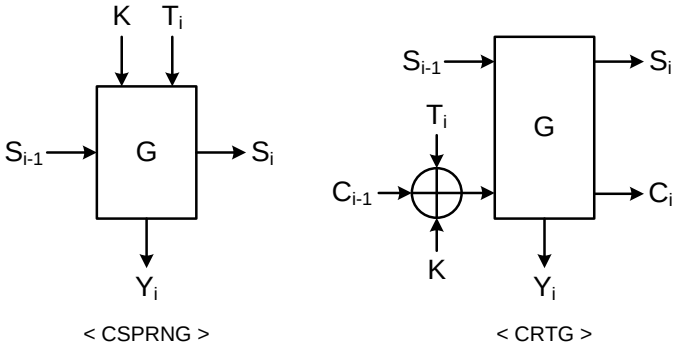


**Fig. 12.** The PRNG model for the cryptographic applications

Fig. 12 에서는 암호 응용을 위한 일반적인 PRNG 의 구현 모델과 우리가 제안하는 CRTG 의 구현 모델을 보이고 있다. 기존의 모델과 다른 점은 CRTG 는 PRNG 의 상태 정보인 S 와 함께 CRTG 만이 갖는 체인의 위치 정보인 C 를 갖는다. 좀더 세부적인 CRTG 의 내부 구현 모델은 Fig. 13 과 같다.

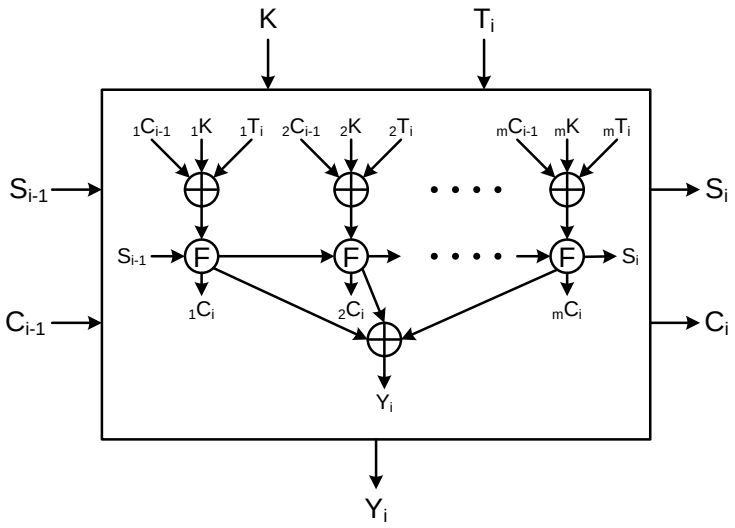


**Fig. 13.** The inner parts of the CRTG model

Fig. 13 에서 F 함수는 2 장에서 살펴본 CRTG 의 기본 알고리즘으로 이것을 기본으로 하여 실제 PRNG 를 구현한다. $_mC_{i-1}$ 은 기본 알고리즘에서의 CP, NP 와 FP 체인의 위치 값으로서 이 세 체인이 하나의 체인 조합을 형성하게 된다. Fig. 13 에서 보는 바와 같이 CRTG 의 실제 구현 모델에서는 m 개의 체인 조합을 사용하여 동일한 F 함수에 대해 m 번의 단계를 거친다. 그러나 PRNG 의 시드인 $S_{i-1}$ 은 그 값 전체가 다음 단계의 F 함수로 전달되며 값이 변한다. PRNG 의 출력인 $Y_i$ 는 각 단계의 출력들에 대해 논리합 연산과정을 거쳐 생성된다. K 는 암호 응용을 위한 PRNG 들에서 주로 사용되는 키로서 PRNG 의 매 반복 수행에 대해 변하지 않는 값이다. $T_i$ 는 타임스탬프나 카운터로서 CRTG 가 사용되는 환경에 따라 사용 여부를 선택 가능하다. CRTG 의 입력인 $C_{i-1}$, K, $T_i$ 는 각각 $C_{i-1} = {}_1C_{i-1} \| {}_2C_{i-1} \| \cdots \| {}_mC_{i-1}$ , $K = {}_1K \| {}_2K \| \cdots \| {}_mK$ , $T_i = {}_1T_i \| {}_2T_i \| \cdots \| {}_mT_i$ 로 각 단계의 연산을 위해 m 개의 조각들로 나누어져 사용된다. 그리고 각 단계의 $_mC_{i-1}$ , $_mK$ , $_mT_i$ 의 세 값이 논리합 연산되어 F 함수의 입력인 체인의 위치 값으로 사용된다. PRNG 의 다음 반복을 위한 체인의 위치 값인 $C_i$ 는 $C_i = {}_1C_i \| {}_2C_i \| \cdots \| {}_mC_i$ 로 각 단계의 F 함수에서 새로 만들어지는 체인의 위치 값들을 조합하여 만들어진다. 다음은 Fig. 13 의 모델을 알고리즘으로 기술한 것으로 트리 깊이 8 인 CRTG 에 대한 알고리즘은 Prog. 3 과 같다.

**Prog. 3.** CRT8G pseudorandom number generator

For PRNG's current cycle *i* :

INPUT: a random (and secret) 1020-bit seed $s_{i-1}$, 130-bit chain $c_{i-1}$, 130-bit key *k*, 130-bit timestamp $t_i$
OUTPUT: a pseudorandom 8-bit string $y_i$, 1020-bit seed $s_i$, 130-bit chain $c_i$

1. Divide the $c_{i-1}$, $k$, $t_i$ into 5 pieces :
   1.1 $c_{i-1} = {}_1c_{i-1} \| {}_2c_{i-1} \| \cdots \| {}_5c_{i-1}$
   1.2 $k = {}_1k \| {}_2k \| \cdots \| {}_5k$
   1.3 $t_i = {}_1t_i \| {}_2t_i \| \cdots \| {}_5t_i$

2. For $n$ from 1 to 5 do the following :
   2.1 $x_n \leftarrow {}_nc_{i-1} \oplus {}_nk \oplus {}_nt_i$
   2.2 $( {}_ny_i , {}_nc_i , s_{i-1} ) \leftarrow F ( s_{i-1} , x_n )$
   2.3 $y_i \leftarrow y_i \oplus {}_ny_i$
   2.4 $c_i \leftarrow c_i \| {}_nc_i$

3. $s_i \leftarrow s_{i-1}$

4. Return the $y_i$, $s_i$, $c_i$

CRTG 의 실제 구현 모델에서 암호학적 안전성을 제공하는 방법은 Fig. 13 에서 보는 바와 같이 하나의 시드에 대해 서로 상호 연관되어있는 다수의 체인 조합을 운영하여 F 함수를 반복 수행하는 것이다. 이로 인해 공격자는 시드와 체인의 각각에 대한 소모적 탐색(brute-force search)을 성공하여야만 한다. 트리 깊이 5 이상의 CRTG 는 모두 120 비트 이상의 시드 크기를 가지므로 소모적 공격에 안전하다. 그러나 2 장에서 살펴본 CRTG 의 기본 알고리즘의 세 체인은 트리 깊이 40 이상의 CRTG 만이 안전하다. 그 이하의 트리 깊이의 CRTG 는 120 비트 이상의 안전도를 갖도록 다수의 체인 조합을 사용하여야 한다. 깊이 d 인 엔트로피 트리에서 이진 트리로부터 생성될 수 있는 CP 와 NP 는 각각 $2^d$ 개이고 원형 리스트로부터 생성될 수 있는 FP 는 $2(2^d-1)$ 개이므로 모두 $2^{2d+1}(2^d-1)$ 개의 체인 조합을 가질 수 있다. 그러므로 m 개의 체인 조합을 운영할 경우 소모적 공격을 막기 위해서는 $2^{2md+1}(2^{md}-1) > 2^{120}$ 가 되도록 m 의 값을 선택하여야 한다. 이는 대략 $md > 40$ 로 각 트리 깊이 d 에 대한 체인 조합의 수인 m 의 값은 Table 4 에 나와있다.

**Table 4.** The number of chains' combinations by tree depth

| Tree depth | 5 | 6 | 7 | 8 | 9 | 10 | 12 | 16 | 20 | 40 |
|---|---|---|---|---|---|---|---|---|---|---|
| Roughly possible chains | $2^{15}$ | $2^{18}$ | $2^{21}$ | $2^{24}$ | $2^{27}$ | $2^{30}$ | $2^{36}$ | $2^{48}$ | $2^{60}$ | $2^{120}$ |
| Required combinations | 8 | 7 | 6 | 5 | 5 | 4 | 4 | 3 | 2 | 1 |

Fig. 13 의 F 함수의 반복 사용 수는 Table 4 의 m 값으로 정해진다. 그래서 Table 4 에서 확인할 수 있듯이 Prog. 3 의 트리 깊이 8 인 CRT8G 는 내부에서 5 개의 체인 조합을 사용하고 5 단계의 F 함수 동작 과정을 갖는다. 이는 하나의 엔트로피 트리에 대해 한 사이클 동안 15 개의 체인이 사용되는 것이다.

다음은 지금까지 우리가 보인 CRTG 모델이 J. Kelsey, B. Schneier, D. Wagner 와 C. Hall 에 의해 제안된 PRNG 의 공격법들에 대해 안전함을 보인다.

### 6.1 Direct Cryptanalytic Attack

현재 사이클 이전에 생성된 모든 난수열과 PRNG 의 알고리즘을 이용하여 앞으로 생성될 난수열을 예측할 수 없음을 입증하는 "next-bit test"에 대한 증명 방법을 우리는 아직 찾지 못하였다. 그러나 반대로 앞으로 생성될 난수열을 예측할 수 있는 방법 또한 찾지 못하였다. 이는 우리의 알고리즘도 DES 와 마찬가지로 수학적 논리에 의해서가 아니라 구조상으로 분석 불가능함을 이용해 안전성을 제공하기 때문이다.

DES 의 경우는 전치와 환자의 반복 수행을 통해 분석이 불가능하게 만든다. 우리의 알고리즘도 전치와 환자의 반복 수행과 유사한 과정을 갖는다. 그러나 CRTG 는 난수들로 채워진 트리와 그 트리 구조의 비선형성을 이용해 알고리즘 내부에서 무작위로 생성되어 비밀 값으로 유지되는 체인들에 의해 더 큰 안전성을 얻는다 [41].

Fig. 13 의 CRTG 모델은 그 구조와 기능에서 블록암호인 DES 와 비교될 수 있다. CRTG 의 엔트로피 트리는 DES 의 F 함수의 S-Box 와 유사하고 CRTG 의 시드는 DES 의 S-Box 의 환자표와 유사하다. CRTG 의 체인은 DES 에서의 키와 같은 기능을 수행한다. 또한 DES 가 내부에서 동일한 반복 과정을 여러 단계 수행하는 것은 CRTG 가 Table 4 의 m 값에 해당하는 내부 단계를 갖는 것과 유사하다. 그러나 DES 의 F 함수에서 사용되는 S-Box 와 키는 고정된 값이나 CRTG 의 F 함수에서 사용되는 시드와 체인은 매 사이클 마다 난수성을 유지하며 예측 불가능하게 값이 변한다. 또한 소모적 탐색이 불가능한 크기를 가지므로 CRTG 는 높은 난수성과 보안 강도를 갖게 된다.

### 6.2 Input-Based Attacks

Prog. 3 의 알고리즘에서 보면 타임스탬프인 $t_i$ 는 그 길이가 체인이나 키와 동일하게 120 비트 이상의 길이를 가지므로 소모적 탐색으로 예측되어질 수는 없다. 그러나 $t_i$ 는 PRNG 의 매 반복 수행 마다 시스템이나 사용자로부터 수집되어지는 값이므로 기지 입력(known-input), 반복 입력(replayed-input)과 선택 입력(chosen-input)의 공격들을 받을 수 있다. CRTG 의 $t_i$ 는 $k$, $c_{i-1}$ 과 함께 하나의 $x$ 로 논리합 연산되어 F 함수의 체인 위치 값으로 사용된다. 그런데 $c_{i-1}$ 은 매 반복 수행 마다 PRNG 내부에서 무작위로 생성되는 값이므로 $t_i$ 에 상관없이 F 함수에는 항상 무작위 값이 사용된다. 따라서 공격자는 $t_i$ 에 대한 선택 입력 공격을 수행할지라도 PRNG 의 출력을 실제 난수 값과 구별할 수 없다.

### 6.3 State Compromise Extension Attacks

Prog. 3 의 알고리즘에서 보면 CRTG 모델의 동작에는 120 비트 이상의 길이를 갖는 $c_{i-1}$, $k$, $t_i$ 와 $s_{i-1}$ 의 변수들이 사용되고 비밀 값으로 유지되고 있다. 그러나 공격자는 $c_{i-1}$, $k$ 와 $t_i$ 의 세 값을 모두 알아야만 $x$ 를 구하여 공격에 이용할 수 있으므로 실제로는 $k$ 와 $t_i$ 의 값을 알더라도 $c_{i-1}$ 과 $s_{i-1}$ 의 값을 모르면 공격에

성공할 수 없다. $c_{i-1}$과 $s_{i-1}$은 모두 CRTG의 상태 변수들로서 PRNG의 내부에서 스스로 값이 변하고 외부로는 보여지지 않는 비밀 값들이다. 그러나 숙련된 공격자는 일시적으로 시스템의 일부 자원에 대한 접근 권한을 얻어 어느 한 순간의 $c_{i-1}$과 $s_{i-1}$에 대한 부분적인 정보들을 알아낼 수 있다. 그 결과로서 어느 한 순간의 $s_{i-1}$의 전체 값을 알아낸 경우와 $s_{i-1}$의 일부분의 값과 $c_{i-1}$의 전체 값을 알아낸 경우의 두 상황을 가정할 수 있다. 첫 번째의 경우는 시드 전체의 값은 알아냈으나 체인들의 위치는 알 수 없으므로 그 시드에서 어느 값들이 어떤 순서로 사용되는지를 알 수가 없다. 이를 위해 공격자는 120 비트 이상의 길이를 갖는 $c_{i-1}$의 값을 추측해야만 한다. 두 번째는 전체 체인들의 위치는 알아냈으나 시드는 일부분의 값만을 알아낸 경우로서 만약 공격자가 알아내지 못한 시드의 길이가 120 비트라면 PRNG가 여러 사이클을 진행하는 동안 공격자는 결국 부족한 120 비트를 추측해야만 한다. 어느 한 사이클의 시드에서 참조되는 위치에 있는 어느 한 개의 비트만을 모르더라도 다음 사이클을 위한 정확한 체인의 위치를 얻을 수가 없다. 그러므로 첫 번째와 마찬가지로 두 번째의 경우도 부족한 비트 수만큼 공격이 어려워진다. 따라서 CRTG의 공격은 $c_{i-1}$, $k$, $t_i$와 $s_{i-1}$의 변수들 중에서 어느 것을 먼저 알아내더라도 결국은 알아내지 못한 부분들의 길이와 동일한 복잡도를 갖는다. 이것은 CRTG가 체인을 이용해 시드의 임의의 위치에서 일부분의 정보만을 사용하는 것과 PRNG 외부로부터 입력되는 키와 타임스탬프를 시드가 아닌 체인의 위치 정보에 확산시키기 때문이다.

만약 어느 한 순간에 $c_{i-1}$, $k$, $t_i$와 $s_{i-1}$의 네 변수들의 모든 값을 알아냈을 경우에는 그 이후의 모든 출력을 알 수가 있고 이것은 어떤 PRNG도 마찬가지이다. 그러나 그러한 경우라도 CRTG에서 그 이전에 생성되었던 모든 출력을 알아내는 것은 불가능하다. 2 장에서의 CRTG의 기본 알고리즘에서 보면 매 사이클 마다 포화 이진 트리에서 새로운 NP의 위치를 탐색한다. 포화 이진 트리에서 바로 이전 사이클의 체인은 현재 사이클의 체인이 참조하는 노드들의 인덱스를 포화 이진 트리에서 만들어질 수 있는 가능한 모든 체인들의 노드 비트와 비교하여 역으로 찾을 수 있다. 그러나 역으로 탐색한 이전 사이클의 체인의 개수는 하나 이상으로서 예측할 수 없다. 이것은 우리의 엔트로피 트리의 실제 데이터인 시드가 항상 난수성을 유지하기 때문이다. 여러 사이클 이전의 체인을 찾으려는 경우에는 각 사이클마다 찾은 체인의 개수가 지수적으로 증가하게 되므로 실질적으로 이전 사이클의 체인을 찾는 것은 확률적으로 불가능하다. 그러므로 CRTG 에서는 역탐색 공격(backtracking attack)도 불가능하다.

## 7 Conclusion

이 논문에서 우리는 트리 구조에 기반한 새로운 의사 난수 생성 알고리즘을 제안하였다. 트리 구조의 비선형성을 이용해 암호학적 공격에도 안전하도록 설계하였다. NIST SP800-22 Statistical Test Suite 을 이용해 통계적으로 훌륭한

난수 생성 알고리즘임을 확인하였고, 소프트웨어로 구현된 다른 난수 생성 알고리즘들과의 비교를 통해 높은 수행 성능을 가짐을 확인하였다.

트리의 깊이를 사용하려는 환경에 맞게 선택할 수 있고 낮은 복잡도를 갖는 알고리즘으로 인해 하드웨어 제약조건이 열악한 유비쿼터스 환경 등에서 사용할 수 있다. 또한 병렬처리를 통해 더 높은 성능을 낼 수 있어 Giga-Bps 급의 초고속 네트워크 환경에서도 사용할 수 있다. 우리의 알고리즘은 소프트웨어 프로그램으로 구현하여 사용하여도 기존 알고리즘들에 뒤지지 않는 성능을 가지나 하드웨어 칩으로 구현하여 사용할 경우 더 높은 성능을 낼 수 있다.

## References


1. A. Menezes, P. Oorschot, S. Vanstone: Handbook of Applied Cryptography. CRC Press (1997)
2. D. Eastlake, J. Schiller, S. Crocker: Randomness requirements for security. IETF RFC 4086 (2005)
3. R. Davis: Hardware random number generators. Statistics Research Associates Limited, http://statsresearch.co.nz (2000)
4. B. Jun, P. Kocher: The Intel Random Number Generator. Cryptography Research Inc., White Paper (1999)
5. Hotbits Hardware. http://www.fourmilab.ch/hotbits/hardware.html
6. Digital Signature Standard. NIST FIPS 186 (1994)
7. American National Standard - Financial institution message authentication (wholesale). ANSI X9.17 (1986)
8. Using RSA BSAFE Crypto-C with the Intel Random Number Generator. RSA Data Security White Paper, http://www.rsasecurity.com/products/bsafe/intel/rsa_rng_tech.pdf
9. J. Kelsey, B. Schneier, N. Ferguson: Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator. Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag (1999)
10. E. Barker, J. Kelsey: Recommendation for random number generation using deterministic random bit generators. NIST SP 800-90 (2005)
11. S. Babbage: Stream ciphers, What does the industry want. Proceedings of SASC 2004 (2004)
12. J. Undercoffer, S. Avancha, A. Joshi, J. Pinkston: Security for Sensor Networks. http://www.cs.umbc.edu/cadip/2002Symposium/sensor-ids.pdf
13. A. Shamir: Stream ciphers, Dead or Alive. invited lecture at SASC 2004 (2004)
14. D. Wheeler, R. Needham: TEA, a Tiny Encryption Algorithm. http://www.ftp.cl.cam.ac.uk/ftp/papers/djw-rmn/djw-rmn-tea.html (1994)
15. A. Perrig, R. Szewczyk, V. Wen, D. Culler, J. D. Tygar: SPINS: Security Protocols for Sensor Networks. MOBICOM 2001 (2001)
16. D. Knuth: The Art of Computer Programming. Vol. 1. Fundamental Algorithms 3rd Ed. Addison-Wesley (1997)
17. E. Key: An analysis of the structure and complexity of nonlinear binary sequence generators. IEEE Transactions on Information Theory, 22, 732-736 (1976)
18. W. Chambers: On Random Mappings and Random Permutations. Fast Software Encryption Proceedings, LNCS 1008, 22-28, Springer-Verlag (1995)
19. L. Massey, J. Omura: Computational Method and Apparatus for Finite Field Arithmetic. US Patent No. 4, 587, 627 (1986)
20. P. Ning, Y. Yin: Efficient Software Implementation for Finite Field Multiplication in Normal Basis. ICICS 2001, LNCS 2229, 177-188, Springer-Verlag (2001)
21. J. Seberry, X. Zhang, Y. Zheng: Nonlinearly Balanced Boolean Functions and Their Propagation Characteristics. CRYPTO 93, LNCS 773, 49-60, Springer-Verlag (1994)
22. NIST Special Publication 800-22, A Statistical Test for Random and Pseudorandom Number Generators for Cryptographic Application. (2001)
23. NIST Random Number Generation and Testing Project download web page. http://csrc.nist.gov/rng/rng2.html
24. L. Blum, M. Blum, M. Schub: A Simple Unpredictable Pseudo-Random Number Generator. SIAM J. on Computing, 15, 2, 364-383 (1986)
25. P. Williams: ANSI C reference implementation of BBS. http://www.mindspring.com/~pate/crypto/chap05/blumblum.c

26. W. Stallings: Cryptography and Network Security. Principles and Practice 4rd Ed. Prentice Hall (2006)
27. C. Wang, T. Troung, H. Shao, L. Deutsch, J. Omura, I. Reed: VLSI Architecture for Computing Multiplications and Inverses in GF(2m). IEEE Transactions on Computers, 34, 8, 709-716 (1985)
28. S. Rhee, D. Seetharam, and S. Liu: i-bean network. An ultra-low power wireless sensor network. In UBICOMP Adjunct Proceedings (2003)
29. L. Batina, J. Lano, N. Mentens, S. Ors, B. Preneel, I. Verbauwhede: Energy, performance, area versus security trade-offs for stream ciphers. In State of the Art of Stream Ciphers, 302-310 (2004)
30. C. Paar: Effcient VLSI Architectures for Bit-Parallel Computation in Galois Fields. Institute for Experimental Mathematics, University of Essen, Germany (1994)
31. H. Lee, S. Moon: Parallel Stream Cipher for Secure High-Speed Communications. Signal Processing, 82, 2, 137-143 (2002)
32. P. Rogaway, D. Coppersmith: A Software Optimized Encryption Algorithm. Journal of Cryptology, 11, 4, 273-287 (1998)
33. M. Matsui, S. Fukuda: How to maximize software performance of symmetric primitives on Pentiums. FSE 2005, LNCS, Springer-Verlag (2005)
34. S. Goldwasser, S. Micali: Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information. 14th ACM STOC, pp. 165-177 (1982)
35. O. Goldreich: Foundation of Cryptography. Class Note, pp 88-95 (1989)
36. A. Desai, A. Hevia, Y. Yin: A Practice-Oriented Treatment of Pseudorandom Number Generators. EUROCRYPT, LNCS 2332, pp. 368-383 (2002)
37. J. Kelsey, B. Schneier, D. Wagner, C. Hall: Cryptanalytic Attacks on Pseudorandom Number Generators. Fast Software Encryption, Fifth International Workshop Proceedings, Springer-Verlag, pp. 168-188 (1998)
38. B. Sadeghiyan, J. Mohajeri: A new universal test for bit strings. ACISP 1996 (1996)
39. M. Blum, S. Micali: How to Generate Cryptographically Strong Sequences of Pseudorandom Bits. 23rd IEEE FOCS, pp. 112-117 (1982)
40. A. Levin: One-Way Functions and Pseudorandom Generators. Combinatorica, 7, 4, 357-363 (1987)
41. J. Kam, G. Davida: Structured Design of Substitution-Permutation Encryption Networks. IEEE Transactions on Computers, 28, 10, 747-753 (1979)

## Appendix A: CRT4G

```
/* A C-program for CRT4G                                */
/* The program's output is Bits in ASCII format         */
/* generate_tre() generates and initializes tree        */
/* generate_prn() generates four random bits            */

#include "stdlib.h"
#include "time.h"

#define DEPTH 4
#define CYCLE ( 300000000 / DEPTH )

typedef struct s_node {
  int K, N ;
  struct s_node *C[2], *F ;
} NODE ;

NODE *CP[DEPTH+1], *NP[DEPTH+1], *FP[DEPTH+1] ;

void generate_tre()
{
  int level, order, width = 1 ;
  NODE **TP[DEPTH+1] ;
  TP[0] = (NODE**)calloc(width,sizeof(NODE*)) ;
  CP[0]=NP[0]=TP[0][0] = (NODE*)malloc(sizeof(NODE)) ;
  for ( level=1 ; level <= DEPTH ; level++ ) {
    width *= 2 ;
    TP[level] = (NODE**)calloc(width,sizeof(NODE*)) ;
    for ( order=0 ; order < width ; order++ ) {
      TP[level][order]=TP[level-1][order/2]->C[order%2]
                    =(NODE*)malloc(sizeof(NODE)) ;
      TP[level][order]->F = ( order == 0 ) ?
        TP[level-1][(width/2)-1] : TP[level][order-1] ;
      TP[level][order]->K = rand() % 2 ;
      TP[level][order]->N = rand() % 2 ;
    }
  }
  TP[1][0]->F = TP[DEPTH][width-1] ;
  FP[0] = TP[1][0] ;
  for ( level=1 ; level <= DEPTH ; level++ ) {
    FP[level] = FP[level-1]->F ;
    CP[level] = CP[level-1]->C[0] ;
    NP[level] = NP[level-1]->C[CP[level]->N] ;
  }
}

void generate_prn()
{
  CP[1]->N ^= CP[2]->K ;
```

```
  CP[2]->N ^= CP[3]->K ;
  CP[3]->N ^= CP[4]->K ;
  CP[4]->N ^= CP[1]->K ;
  CP[1]->K ^= NP[3]->N ;
  CP[2]->K ^= NP[4]->N ;
  CP[3]->K ^= NP[1]->N ;
  CP[4]->K ^= NP[2]->N ;
  FP[0] = FP[DEPTH] ;
  for ( int level=1 ; level <= DEPTH ; level++ ) {
    printf( "%d", CP[level]->K ) ;
    CP[level]->K ^= FP[level]->N ;
    NP[level]->N ^= FP[level]->K ;
    FP[level] = FP[level-1]->F ;
    CP[level] = NP[level] ;
    NP[level] = NP[level-1]->C[CP[level]->N] ;
  }
}

void main()
{
  generate_tre() ;
  for ( int i=0 ; i < CYCLE ; i++ ) {
    generate_prn() ;
  }
}
```

## Appendix B: Test Results

**Table 5.** NIST SP800-22 test scores of XORG, CRT8G, and BBSG

| Test | XORG | CRT8G | BBSG |
|---|---|---|---|
| Frequency (Monobit) | 0.9933 | 0.9867 | 0.9733 |
| Frequency within a Block | 0.9433 | 0.9900 | 0.9867 |
| Cumulative Sums | 0.9950 | 0.9867 | 0.9700 |
| Runs | 0.9933 | 0.9833 | 0.9933 |
| Longest Run of Ones in a Block | 0.9833 | 0.9900 | 0.9967 |
| Binary Matrix Rank | 0.8633 | 0.9933 | 0.9933 |
| Discrete Fourier Transform (Spectral) | 0.9900 | 0.9900 | 0.9833 |
| Non-overlapping Template Matching | 0.4562 | 0.9899 | 0.9895 |
| Overlapping Template Matching | 0.9333 | 0.9833 | 0.9967 |
| Maurer's "Universal Statistical" | 0.6833 | 0.9800 | 0.9700 |
| Approximate Entropy | 0.8267 | 1.0000 | 0.9967 |
| Random Excursions | 0.9871 | 0.9934 | 0.9907 |
| Random Excursions Variant | 0.9828 | 0.9920 | 0.9953 |
| Serial | 0.8867 | 0.9917 | 0.9900 |
| Linear Complexity | 0.0000 | 0.9767 | 0.9867 |

**Table 6.** NIST SP800-22 test scores of CRT4G, CRT9G, and CRT16G

| Test | CRT4G | CRT9G | CRT16G |
|---|---|---|---|
| Frequency (Monobit) | 0.9933 | 0.9867 | 0.9933 |
| Frequency within a Block | 0.9967 | 0.9800 | 0.9967 |
| Cumulative Sums | 0.9933 | 0.9850 | 0.9950 |
| Runs | 0.9833 | 0.9833 | 1.0000 |
| Longest Run of Ones in a Block | 0.9967 | 0.9933 | 0.9933 |
| Binary Matrix Rank | 0.9933 | 0.9900 | 0.9900 |
| Discrete Fourier Transform (Spectral) | 0.9967 | 0.9933 | 0.9933 |
| Non-overlapping Template Matching | 0.9898 | 0.9906 | 0.9901 |
| Overlapping Template Matching | 0.9900 | 1.0000 | 0.9733 |
| Maurer's "Universal Statistical" | 0.9800 | 0.9833 | 0.9833 |
| Approximate Entropy | 0.9933 | 0.9800 | 0.9867 |
| Random Excursions | 0.9925 | 0.9895 | 0.9948 |
| Random Excursions Variant | 0.9948 | 0.9877 | 0.9957 |
| Serial | 0.9867 | 0.9967 | 0.9967 |
| Linear Complexity | 0.9967 | 0.9867 | 1.0000 |

**Table 7.** The CPU time of CRTG's tree generation and initialization

| Tree Depth | CPU Time (second) | Tree Depth | CPU Time (second) |
|---|---|---|---|
| 4 | 0.000 | 5 | 0.000 |
| 6 | 0.000 | 7 | 0.000 |
| 8 | 0.000 | 9 | 0.000 |
| 10 | 0.000 | 11 | 0.000 |
| 12 | 0.000 | 13 | 0.000 |
| 14 | 0.015 | 15 | 0.031 |
| 16 | 0.062 | 17 | 0.109 |
| 18 | 0.218 | 19 | 0.421 |
| 20 | 0.843 | 21 | 1.671 |
| 22 | 3.468 | 23 | 6.734 |

**Table 8.** The CPU time of CRTG's pseudo random bits generation

| Tree Depth | CPU Time (second) | Tree Depth | CPU Time (second) |
|---|---|---|---|
| 4 | 5.937 | 5 | 5.734 |
| 6 | 7.312 | 7 | 6.296 |
| 8 | 6.671 | 9 | 7.343 |
| 10 | 7.343 | 11 | 7.562 |
| 12 | 7.546 | 13 | 7.546 |
| 14 | 7.656 | 15 | 8.203 |
| 16 | 12.422 | 17 | 14.859 |
| 18 | 16.547 | 19 | 18.297 |
| 20 | 19.516 | 21 | 21.094 |
| 22 | 22.375 | 23 | 24.078 |