Da Yan Concentrator and Extender of Random Data (DYCE)

Anna M. Johnston Puru Kulkarni
Juniper Networks

October 18, 2022

Abstract

DYCE is a simple to analyze algorithm which converts raw entropy into usable cryptographic entropy using the Da Yan (commonly known as the 'Chinese Remainder Theorem'). This paper describes and analyzes DYCE, giving its detailed algorithmic description.

1 Entropy and Random Data

Random data and entropy are critical components for cryptographic systems. Exposed random data compromises security. Getting, measuring, and insuring a steady stream of secure random data, however, is difficult.

Before continuing, I'd like to clarify the difference between entropy and random data. Entropy is what is measured when we speak of randomness. For our purposes, data is a container which stores entropy, with random data containing a measurable amount of entropy. The term 'Shannon'[1] is the unit for information/entropy within data used in this document. This is a standard term[3] and avoids confusion between data and entropy. Data may be said to have an entropy rate, such as Shannons-per-bit or Shannons-per-byte, or have a total entropy in Shannons. The maximal entropy rate is one and the minimal is zero Shannons-per-bit.

Raw (or 'true') random often has low entropy rate, is time and computationally costly, and by its nature, unpredictable. Oxymoronically named "Deterministic Random Number Generators", or RNG/DRNG[2] for short, convert the raw entropy into a cheap, plentiful supply of data which has maximal entropy under various statistical tests.

These 'generator' algorithms do not create entropy. To avoid the implication of generating entropy, these algorithms will be called 'Entropy Concentrator and Extender', or simply ECE.

ECEs rely on having a random initial state with the output revealing little information about the state. This leads to a two part generation process, and (essentially) two different

[1] Shannon is an entropy unit in base 2. 'Nat' and 'hartley' are the information unit measures base e and base 10 respectively.

[2] 'Number' may be replaced by 'Bit', and 'Cryptographic' or 'Pseudorandom' may precede (ex: CRNG). The terms 'Generator' and 'Deterministic' are misleading as no deterministic system generates entropy.

types of entropy:

1. Raw entropy is collected from one or more sources and collected until 'enough' entropy has been gathered;

2. Cryptographic entropy is created by passing the raw entropy through a cryptographic-like process. In particular, it must be computationally one-way: given a bounded amount of output and complete knowledge of the algorithm, an attacker is unable to determine any part of the secret state with a given amount of computational resources and time. This produces a longer, more cryptographically robust random data stream.

As this paper describes a new cryptographic ECE, unless otherwise stated, ECE implies cryptographic ECE.

Existing cryptographic algorithms – block ciphers (AES) and hash functions – are often used in ECE design[1]. While these algorithms do a great job of statistically smoothing data and giving the output some of their cryptographic properties, they tend to waste entropy (ex: hash functions coalesce) and computation – overkill for the job at hand. Furthermore, it may be risky to use the same cryptographic processes to generate their own cryptovariables or parameters. For example, it may not be wise to use AES to generate the cryptographic keys for AES.

This paper describes DYCE (Da Yan cryptographic Concentrator and Extender), a different approach to entropy concentration and extension. Instead of using cryptographic algorithms, a mathematical approach to concentrating and extending entropy is proposed. Using very simple, well understood techniques such as Galois registers and the Da Yan (a.k.a. the Chinese remainder theorem), entropy is concentrated much more efficiently. Failure cases are easy to spot and handle, and the theoretical nature of the design also makes it easier to analyze.

1.1 Entropy Flow

DYCE (and other ECEs) extend entropy, but no system can convert finite entropy to infinite. For this reason, raw entropy entering and cryptographic entropy exiting the system should be monitored. The flow of output cryptographic entropy should be balanced with the flow of input raw entropy.

To manage this flow, DYCE uses an *entropy ratio*: Cryptographic Shannon-per-Raw Shannon (equation 1.1.1). This ratio estimates how many Shannons each raw bit contributes how much is lost in each output bit. Raw entropy entering the system is easily monitored, as the entropy rate of the incoming raw random data is assumed known. Cryptographic entropy leaving the system is harder to estimate. This depends on the strength of the cryptographic wall between the two types of entropy. In other words, what attacks are possible and how much output does it need to gain information about the state of the system?

All known cryptographic attacks against secure ECEs are are impractical (hence why they are considered secure), and allow for nearly infinite ratios. This may be why they are mistakenly called ‘generators’. Even with no known practical attacks, entropy flow should be monitored to account for unforeseen attacks. DYCE approximates the entropy flow, cryptographic random output per Shannon of raw input, at either a conservative, moderate, or generous rate.

Cryptographic Shannon per Raw Shannon

Conservative	Moderate	Generous
2^{10}	2^{16}	2^{20}

(1.1.1)

Note that these numbers are somewhat arbitrary and may be modified if needed. In the design proposed the moderate flow is used (figure 2.5).

2 Basic Design

This design, like most, consists of two different types of entropy: raw and cryptographic (section B). Unlike many designs, DYCE does not use cryptographic techniques (block ciphers or cryptographic hash functions) to process raw random. Simple, well understood mathematical tools are used instead to accumulate, mix, and extend raw random input.

2.1 Design Overview

DYCE (figure 2.1) consists of two processes and storage areas:

1. An **entropy reservoir** (section 2.2) collects, concentrates, and mixes raw random entropy. A simple Galois stepping mechanism (section 2.2.2) insures data is evenly mixed, with no entropy lost in the process. This is not a cryptographic process. Its only purpose is to evenly mix and concentrate entropy.

2. The **extraction state** (section 2.3) stores how data is extracted from the reservoir. The Quotient Ring Transform (QRT) [4] extracts data from the entropy reservoir dependent on the extraction state. Output words are stored in the **extraction register**. They can be thought of as small hash values of the entire entropy reservoir.

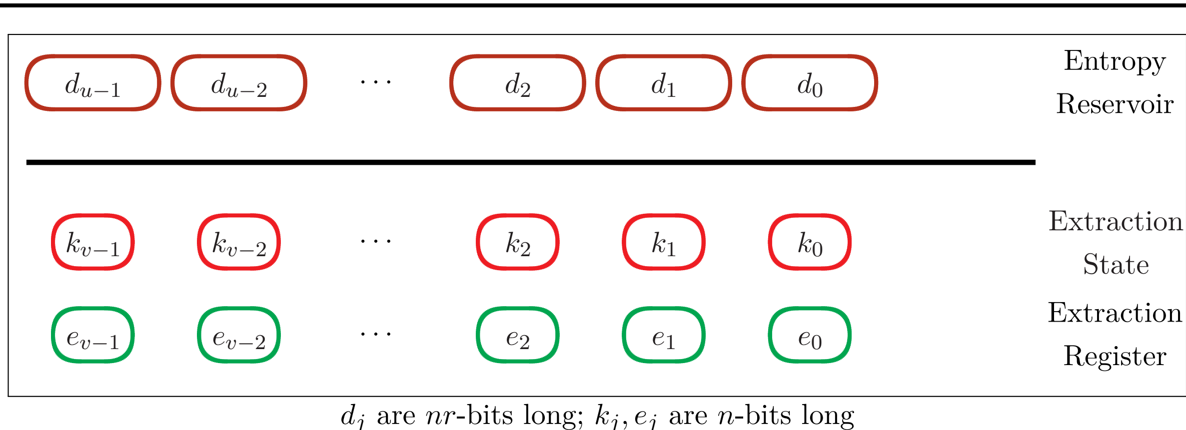


Figure 2.1: DYCE Entropy Reservoir and Extraction State

Galois stepping and QRT create a system where output (extraction register) can be shown to reveal nothing about the current data in the entropy reservoir (and thus the next state) or the extraction state. Analysis in the case where an attacker knows or has control over various portions (section 4) of the device is also simplified.

The computations and theoretical underpinnings of this design are based on finite extension fields over $\mathbb{F}_2$.

n	Smallest data unit is in $\mathbb{F}_{2^n}$. Bytes ($n = 8$) are generally chosen for efficiency.
u	This is the number words in the entropy reservoir; $u > 4$.
r	Each word in the entropy reservoir is rn-bits long.
v	This is the number of words in the extraction state. Each iteration will produce v words, each n-bits long; output at each iteration should be no more than half the bits in entropy reservoir: $v < (u \cdot r)/2$.

Figure 2.2: Chosen Register Sizes (appendix A) for the DYCE

2.2 Entropy Reservoir

The entropy reservoir consists of a set of u words, each rn-bits long (figure 2.1). These words are used and viewed in two ways. Let d_j (figure 2.3) be the data in the j-th word, with

$$d_j = d_{j,r-1} \| d_{j,r-2} \| \ldots \| d_{j,1} \| d_{j,0},$$

d_i	The data in the i-th entropy reservoir register (nr bits). This is treated as a degree $(r-1)$ polynomial over $\mathbb{F}_{2^n}$: $d_i = \sum_{j=0}^{r-1} d_{i,j} x^j$ with $d_{i,j} \in \mathbb{F}_{2^n}$ or an element in $\mathbb{F}_{2^{nr}}$.
$\mathbf{q}_i$	Constant, monic polynomials over $\mathbb{F}_{2^n}$ of degree r, with $d_i \in \mathbb{F}_{2^n}[x]/\mathbf{q}_i$.
e_j	The data in the the j-th extraction register (n bits).
$\mathbf{p}_j$	The monic degree one polynomial $\mathbf{p}_j = (x - k_j)$ over $\mathbb{F}_{2^n}$, with $e_j \in \mathbb{F}_{2^n}/\mathbf{p}_j$. Note that k_j varies over time and $k_j \neq k_i$ for all $i \neq j$.

This data is related with the da yan: If $\mathbf{D}(x) \equiv d_j \bmod \mathbf{q}_j$ and has degree less than ur, then $e_j \equiv \mathbf{D}(x) \bmod \mathbf{p}_j$, or equivalently, $e_j \equiv \mathbf{D}(k_j)$.

Figure 2.3: DYCE Data

where $d_{j,k}$ is an n-bit value. Then:

1. d_j is a polynomial of degree less than r over $\mathbb{F}_{2^n}$: $d_j = \sum_{k=0}^{r-1} d_{j,k} x^k$, with $d_{j,k} \in \mathbb{F}_{2^n}$. Note that

$$\mathbb{F}_{2^n} \cong \mathbb{F}_2[x]/\mathbf{b}$$

   where $\mathbf{b}$ (figure 2.6) is a constant, irreducible polynomial over $\mathbb{F}_2$ of degree n.

2. d_j is an element of the extension field $\mathbb{F}_{2^{rn}}$, with the field represented by

$$\mathbb{F}_{2^{rn}} \cong \mathbb{F}_2[x]/\mathbf{B}$$

   where $\mathbf{B}$ (figure 2.6) is a constant, irreducible polynomial over $\mathbb{F}_2$ of degree rn.

The first representation of d_j (as a polynomial) is used to extract random, while the second representation (as an element) is used to concentrate and extend the raw random.

2.2.1 Entropy Reservoir As A Polynomial

As a polynomial, each d_j is considered modulo a degree r polynomial over $\mathbb{F}_{2^n}$: $\mathbf{q}_j$ (figure 2.6). These u polynomials, $\{\mathbf{q}_j\}_{j=0}^{u-1}$, are distinct and irreducible (section A). The collection of u data/moduli pairs represents a large polynomial $\mathbf{D}$. If $\mathbf{Q}$ is the product of the moduli,

$$\mathbf{Q} = \prod_{j=0}^{u-1} \mathbf{q}_j,$$

then $\mathbf{D}(x) \bmod \mathbf{Q}$ is the large polynomial such that:

$$\mathbf{D}(x) \equiv d_j \bmod \mathbf{q}_j \tag{2.2.1}$$

for $0 \leq j < u$. This large polynomial is guaranteed to exist and be unique modulo $\mathbf{Q}$ due to the da yan (Chinese remainder theorem).

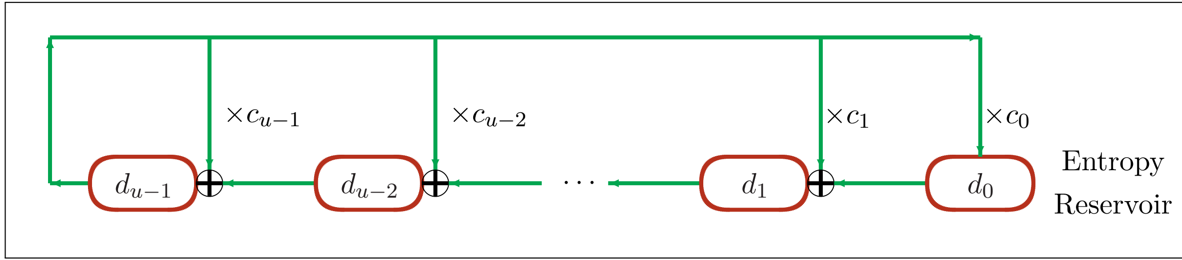
2.2.2 Entropy Reservoir As An Element

As an element of $\mathbb{F}_{2^{rn}}$, d_j is a coefficient of a polynomial modulo $\mathbf{f}$ (figure 2.6), a fixed primitive polynomial over $\mathbb{F}_{2^{rn}}$. The full entropy reservoir is treated as a large Galois register, stepped (figure 2.4) by multiplying by x and reducing modulo $\mathbf{f}$.

If $\mathbf{f}$ is $\mathbf{f} = x^u + \sum_{j=0}^{u-1} c_j x^j \bmod \mathbf{B}$, a single step of the Galois register can also be described with the following equations and in algorithmic form (algorithm 7):

$$d'_i = \begin{cases} c_0 \cdot d_{u-1} \bmod \mathbf{B} & i = 0 \\ d_i + c_i \cdot d_{u-1} \bmod \mathbf{B} & 0 < i < u \end{cases} \tag{2.2.2}$$

The updated entropy reservoir will be the result of stepping the register a t-times, where $u \leq t < 2u$ (algorithm 6)



where $\mathbf{f}(x) = x^u + \sum_{i=0}^{u-1} c_i x^i$, $\times$ is multiplication over $\mathbb{F}_{2^{nr}}$.

Figure 2.4: Galois Step of Entropy Reservoir

Primitive polynomials have the property that a root has maximal order: $(2^{urn} - 1)$. This means that the stepping process, for non-zero initial states, will step through every non-zero possible state. For the variables chosen, this implies that the entropy reservoir will loop through $(2^{urn} - 1) = \left(2^{280} - 1\right)$. Note that this does not mean that the output data (extraction register) will never repeat. One of the strengths of this system is that there are more combinations of possible entropy reservoir and extraction state which produce the same output data than there are possible output data states.

2.3 Extraction State and Extraction Register

The extraction state and register each consists of a set of v n-bit words (figure 2.1). The extraction state contains the values k_j which define the polynomials

$$\mathbf{p}_j = (x - k_j) \bmod \mathbf{b}. \tag{2.3.1}$$

These polynomials determine how data from entropy reservoir is converted to the random output in extraction register.

s	This variable stores the number of Shannon's currently in the entropy reservoir.
$s_m = 260$	This is the minimum number of Shannons (section 1.1) which must be present in entropy reservoir in order for cryptographic random data to be extracted.
$S_r = 2^{15}$	This is the ratio of output Shannons to input (cryptographic vs raw Shannons). The value given here is the moderate rate (equation 1.1.1).
w	This is the number of times cryptographic random has been extracted from the system since the last update of the Shannon level.
$w_m = 585$	This is the maximum number of times DYCE can be iterated before a Shannon of raw random has been used (section 1.1). This number corresponds to the moderate entropy flow measure – i.e., the flow rate (S_r) divided by cryptographic entropy out (vn) at each step. $585 = \left\lfloor \frac{S_r}{vn} \right\rfloor = \left\lfloor \frac{2^{15}}{56} \right\rfloor$

Figure 2.5: DYCE Shannon Tracking Variables

Data in the extraction register will be the large entropy reservoir polynomial (equation 2.2.1), reduced modulo $\mathbf{p}_j$ (equation 2.3.1).

$$e_j \equiv \mathbf{D}(x) \bmod \mathbf{p}_j \quad \text{or equivalently} \quad e_j \equiv \mathbf{D}(k_j) \bmod \mathbf{b}. \tag{2.3.2}$$

Cryptographic random output is the data in extraction register: $\{e_j\}_{j=0}^{v-1}$.

3 DYCE Algorithm

DYCE has three main, somewhat independent sections:

Initialization: The states of the DYCE must be initialized (algorithm 2) before they can be used;

Add New Entropy: New entropy can be added (algorithm 3) at at any time except when random data is being extracted. The number of Shannons in the entropy reservoir must be tracked as entropy flows in and out of the DYCE. Entropy entering and exiting during extraction can be very roughly estimated, but can never be greater than urn.

One restriction on the entropy reservoir is that they can never be all zero. This can only happen during the addition of new raw random or during an internal update of entropy reservoir (algorithm 9).

b	This is an irreducible polynomial (equation A.0.1) of degree n over $\mathbb{F}_2$. Arithmetic over $\mathbb{F}_{2^n}$ is performed modulo **b** ($\mathbb{F}_{2^n} \cong \mathbb{F}_2[x]/\mathbf{b}$).
$\mathbf{q}_j$	Each entropy reservoir word j $(0 \leq j < u)$ has a corresponding, fixed, degree r irreducible polynomial (figure A.2) over $\mathbb{F}_{2^n}$ (modulo **b**). These polynomials are relatively prime to each other. In other words, the greatest common denominator of any two polynomials is 1: $gcd(\mathbf{q}_i, \mathbf{q}_j) = 1$ for all $i \neq j$.
Q	This is the product of all $\mathbf{q}_j$ polynomials: $\mathbf{Q} = \prod_{j=0}^{u-1} \mathbf{q}_j$.
B	This is an irreducible polynomial (equation A.0.1) of degree nr over $\mathbb{F}_2$. Arithmetic over $\mathbb{F}_{2^{nr}}$ is performed modulo **B** ($\mathbb{F}_{2^{nr}} \cong \mathbb{F}_2[x]/\mathbf{B}$).
f	This is a fixed primitive polynomial (equation A.0.2) of degree u over $\mathbb{F}_{(2^{nr})}$ used to 'Galois step' the entropy reservoir, with $$\mathbf{f} = x^u - \sum_{i=0}^{u-1} c_i x^i \bmod \mathbf{B}, \qquad (2.2.3)$$ and $c_i \in \mathbb{F}_2[x]/\mathbf{B}$. A primitive polynomial is an irreducible (prime) polynomial such that its root has maximal order: $(2^{urn} - 1)$.

Figure 2.6: Fixed Polynomial Parameters (appendix A) for the DYCE

Extract cryptographic random: The Shannon level in entropy reservoir must be fully filled before the first extraction of cryptographic random data. Once fully filled, DYCE extends the raw Shannon's entering the system to a larger number of cryptographic Shannons by

1. extracting random from entropy reservoir (algorithm 1);
2. stepping the entropy reservoir for the next extraction (algorithm 6).

Output leaving the system will be accounted for. Once in use, the Shannon level should not get below a set threshold: $s \geq s_m$. Random is extracted at a rate of vn cryptographic random per iteration. Using the chosen entropy flow S_r (figure 2.5), $\lfloor \frac{S_r}{vn} \rfloor$-iterations can be performed before reducing the Shannon level (s) in entropy reservoir by one.

Besides Galois stepping the entropy reservoir after each random extraction, the extraction state and entropy reservoir are internally updated. Every time a raw Shannon is used, the extraction state will be updated (algorithm 8). If too many raw Shannon's are used without adding new random, old random from the extraction state is extracted and added

to the entropy reservoir (algorithm 9), then stepped and new extraction state is computed. This does not add new raw Shannons to the system, but the process is one-way which reduces the ability of an attacker to compute old states from a compromised current state. This update may never occur if raw random enters the entropy reservoir at a fast enough pace.

3.1 Algorithm Summary

Algorithms for DYCE are contained in appendix D. The following tables summarize these algorithms and their functions. Variables for the general size of the parameters (figure 2.2), tracking Shannons (figure 2.5), data registers (figure 2.3), and constants (figure 2.6) are used throughout the algorithms.

Main algorithms to initialize, add entropy, and extract cryptographic entropy for use

Alg	Name	Description
2	Initialize	Initializes an empty DYCE box, preparing the system for incoming entropy.
3	Add new raw entropy	Adds incoming words of raw entropy into the DYCE box, mixing the random and updating the Shannon level of the box. Called before algorithm 4 if there is not enough random for extraction, and at any time between random data extractions when random is available.
1	DYCE Generate Random	Compute cryptographic random, updating the Shannons used as needed.

Subroutines required either by the main algorithms or by other subroutines

Alg	Name	Description
4	Compute next random state	Computes internal random in DYCE box, storing the new random output in the extraction register. Called when random is requested (algorithm 1) and during update procedures (algorithms 9 and 8).
5	Adjust Entropy Level After Extraction	Adjusts the Shannon levels after an extraction of random data and updates the extraction state and entropy reservoir if needed. Called in algorithm 1.
7	Single Step of Entropy Reservoir (Galois Stepping)	This function performs a single Galois step on the entropy reservoir. Called in algorithm 6
6	Stepping the Entropy Reservoir	Computes a random stepping number `t` between $u \leq \mathtt{t} < 2u$ and single steps the register `t` steps. Called in algorithms 8, 11, 3, and 4
9	Update Entropy Reservoir	Updates the entropy reservoir in states when enough random has been extracted with out being replaced. Called in algorithm 4.
8	Update Extraction State	The extraction state values are updated every time a Shannon of raw random is is used. Called from algorithms 11, 1, 4, and 9.
11	Handling All Zeros	An all zero entropy reservoir results in a static state, which is not allowed. This routine extracts random from the extraction state and puts it back into the entropy reservoir, updating the Shannon level of the system. Note that a zero state can only occur when new random is being added or entropy reservoir is updated. The probability of an all zero state occurring at any point is close to zero: $pr(\text{zero entropy reservoir}) = \frac{1}{2^{urn}}$. Called in algorithms 3 and 9.
16	Single QRT Extraction, using $p(x) = (x - k)$	Extracts the value from the woven version of **D**, modulo a monic degree one polynomial $(x - k)$. For a more theoretical, less concrete version, see algorithm 15. Called in algorithms 10, 14.

Subroutines required either by the main algorithms or by other subroutines

Alg	Name	Description
13	Inward QRT	Converts the data in the entropy reservoir to a form which can be easily evaluated to obtain the random output in extraction register. This is called from algorithms 8, 12. Output from this algorithm is also needed as input to several other algorithms.
14	Outward QRT	Converts the transformed values (algorithm 13) into reduced (evaluated) output. This is called from algorithms 12.
12	QRT	Performs the Quotient Ring Transform on the entropy reservoir, returning the transformed version which is used to extract random or create a new extraction state. Called from algorithm 4.
15	Single QRT Extraction	This is a more general, and less concrete, extraction algorithm. For a more concrete version, see algorithm 16. Called in algorithms 10, 14.
17	Polynomial Evaluation over $\mathbb{F}_{2^h}$	Evaluates $G(x)$ at a where the a and the coefficients of G are considered modulo a degree h binary polynomial m
18	Modular Multiplication in $\mathbb{F}_{2^h}$, for small h	Performs multiplication modulo $m(x)$ over $\mathbb{F}_2$, where the degree of h is less than a single computer word size. Polynomials are stored in a single computer word. For example, the polynomial $x^3 + x + 1$ is stored as the binary value `1011`, or in hex as `0xb`. Note: for if h is sufficiently small (say $h \leq 8$), multiplication is much faster if done using look-up tables (algorithm 19).
19	Modular Multiplication in $\mathbb{F}_{2^h}$, for small h using a primitive degree h polynomial.	Performs multiplication modulo $m(x)$ (where m is a primitive polynomial) over $\mathbb{F}_2$, where the degree h is less than a single computer word size, using power and log table look ups.
20	Polynomial Modular Multiplication over $\mathbb{F}_{2^h}$	Performs multiplication modulo $q(x)$ over $\mathbb{F}_{2^h}$. This algorithm is similar to 18, except coefficients of the polynomials a, b are h-bit values (elements in $\mathbb{F}_{2^h}$) instead of bits, and algorithm 18 or 19 replaces some of the shifts.

4 Simple Analysis

The DYCE was designed to eliminate entropy loss in the collection mechanism and have an extraction mechanism which can be easily analyzed. This is accomplished using well understood mathematical processes: a very large Galois register to mix and retain entropy, and using the da yan to extract entropy.

4.1 Entropy Collection, Mixing and Retention

The entropy reservoir uses Galois registers to collect and mix entropy. Galois registers efficiently step through all possible $(2^{urn} - 1) = (2^{280} - 1)$ non-zero states. All entropy (up to the maximum urn-Shannons) entering the system remains in the system, no matter how many Galois steps are performed. In other words, stepping the entropy reservoir does not make it any more predictable. This contrasts with random functions, such as a hash, where repeatedly stepping the function does lose entropy (section 4.3).

It should be noted that Galois stepping is easily invertible. One argument against using Galois registers might be that if an attacker compromises the randomizer, then all previous states would be compromised. However, there are several actions in the system which prevents an attacker with full knowledge of the entropy reservoir at a given time to back it up. First, new random data is regularly added to the register. Without knowledge of the entropy added at each state, it is impossible to back up the register beyond the most recent addition of new entropy.

Second, entropy flow is monitored, and if cryptographic output above a given limit is produced without sufficient raw random input, the entropy reservoir will be updated using a one-way function on the current entropy reservoir and extraction state. This internal update prevents an inversion in such a catastrophic failure, but it does not create more entropy. As mentioned earlier, deterministic generation of entropy is a fallacy.

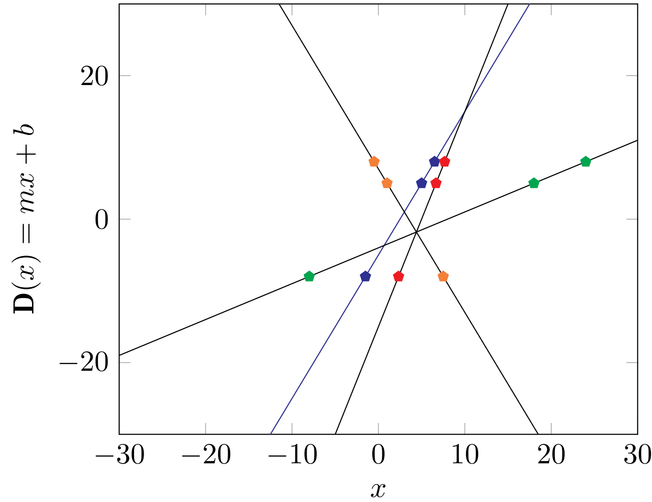
4.2 Entropy Extraction

The following properties are guaranteed by using the da yan for the extraction process and having the number of output bits (extraction register) at each round equaling only a small fraction of the entropy reservoir bits:

- Each output byte output depends on every bit in the entropy reservoir.
- Identical output values can occur with completely different data in the entropy reservoir.
- No information about the entropy reservoir or extraction state is gained from the extraction register output alone.

The major concern with entropy extraction is: what information about the entropy reservoir or the extraction state can be obtained from the output (extraction register)? The simplest model of DYCE equates the entropy reservoir to a random polynomial $\mathbf{D}(x)$ (equation 2.2.1), with degree bounded by $ur = 7 \cdot 5$. Output in the extraction register (e_j) is simply this polynomial evaluated at points determined by extraction state:

$$\mathbf{D}(x) \equiv e_j \bmod (x - k_j),$$



Simplified **D**, over the rational numbers.
If the $ur = 2$, and the output is $\{5, 8, -8\}$, the **D** could be any line except one that is vertical.
This graph shows only four possible **D** polynomials out of an infinite set.

Figure 4.1: Simplified **D**, Extraction State, and Extraction Register

for $0 \leq j < v = 7$, with k_j unknown. This is what the extraction register reveals if either the extraction state or the entropy reservoir were compromised:

1. If the extraction state(k_j) was compromised, then the polynomial $E(x) \bmod P$ (degree bounded by v) would be known and

$$\mathbf{D}(x) = E(x) + P(x)\,H(x)$$

   where $H(x)$ is an unknown polynomial of degree bounded by $ur - v$. Using the values given, there are still $(7{\cdot}5{-}7)8$ different entropy reservoir states which give the same output. Assuming the extraction state does not change over time and that no new entropy enters the system, multiple iterations could be used to reduce the possible candidates for **D**.

2. If the entropy reservoir (**D**) was compromised and the extraction state did not change, multiple iterations could be used to reduce the set of possible extraction states.

To illustrate how little information is exposed in the output, assume that instead of a high degree, **D** was a degree 1 polynomial – i.e., a line (figure 4.1). Excluding horizontal or vertical lines (and non-distinct extraction register), every line contains all possible distinct set of output values. If all combinations are possible for any **D**, no information on **D** or the constants it was evaluated at (k_j) is revealed.

4.3 A Case Against Using Hash Functions

Hash functions are a commonly used tool for generating cryptographic random data. These functions, which in theory resemble a random function, do loose entropy. Repeatedly applying a hash function coalesces to a subset (cycle) of all possible values [2]. For example, if

$$\begin{aligned} x_1 &= SHA_{256}(x_0) \\ x_2 &= SHA_{256}(SHA_{256}(x_0)) \\ x_n &= SHA_{256}^{n}(x_0) = SHA_{256}(SHA_{256}(SHA_{256}(\ldots SHA_{256}(x_0)))) \end{aligned}$$

is SHA-256 repeated n times. This function will eventually repeat: $x_a = x_b$ for some positive integers $b > a$, giving a cycle length of $\text{len} = (b - a)$. Different starting input may give different cycle lengths. The size of the cycle varies, but is on average around the square root of all possible values. For SHA-256, you'd expect a cycle length of about $n = 2^{128}$. While 2^{128} is still huge, it does indicate that hash functions may not be optimal in collecting and preserving entropy.

While cryptographic hash functions do not preserve entropy, they do prevent attacks based off compromise of internal data. These functions are designed to resist first and second pre-image attacks as well as collision attacks. If a hash is used in a deterministic random number generator, and the current state of that hash is revealed, it would be computationally infeasible for an attacker to find previous values.

Galois registers do not have this characteristic. They are trivial to back up, so given contents of the entropy reservoir at time t and assuming no new raw random has been added after time $(t - h)$, the contents at time $t - j$ can be computed, for any $j < h$. However, adding raw entropy and updating the entropy reservoir (algorithm 9) prevent the inversion process. Furthermore, the state of the cryptographic randomizer should be treated similarly to the sensitive cryptovariables. Compromise of these variables is catastrophic and indicates problems beyond algorithmic fixes.

A Fixed Parameters

Table A.1 gives the suggested sizes for the fixed parameters in DYCE.

There are a few things to note before defining these parameters:

- Coefficients and polynomials over $\mathbb{F}_2$ are represented in short hand as hexadecimal values, with bit j representing coefficient j. For example, $\mathbf{b} =$ `0x171` represents the polynomial $x^8 + x^6 + x^5 + x^4 + 1$.

1	7				1			
1	0	1	1	1	0	0	0	1
x^8	–	x^6	x^5	x^4	–	–	–	1

- The base field polynomials **b** and **B** were generated using standard techniques [5]. The 7 polynomials of degree 5 over $\mathbb{F}_{2^8}$ were generated by first generating 7 irreducible polynomials of degree $5 \cdot 8$ over $\mathbb{F}_2$. Each of these polynomials factors into 8 polynomials of degree 5 over $\mathbb{F}_2[x]/\mathbf{b}$. One of these factors is chosen for the entropy reservoir moduli, with the remaining $(8-1)$ discarded. This produces 7 independent degree 5 irreducible polynomials over $\mathbb{F}_{2^8}$.

The entropy reservoir polynomial moduli needed for the QRT and random extraction are in figure A.2. During the transform (algorithm 13), inverses of some of these polynomials are required. These can be computed using the Euclidian algorithm (not given here) either as needed or precomputed before the algorithm begins. For those not wishing to implement the Euclidian algorithm for polynomials over $\mathbb{F}_{2^n}$, the inverses of the necessary entropy reservoir moduli are given (figure A.2).

These parameters and their description follows:

n	: Size of lowest degree base field used in DYCE.
u	: Number of elements in the entropy reservoir.
r	: Degree of entropy reservoir elements over $\mathbb{F}_{2^n}$.
v	: Number of elements in extraction state/register.
b	: A degree n polynomial over $\mathbb{F}_2$. This is the base polynomial for both the entropy reservoir and extraction reservoir/states.
B	: A degree nr polynomial over $\mathbb{F}_2$. This is the base polynomial for the Galois register polynomial (equation A.0.2).
$\mathbf{q}_j$	: Each entropy reservoir word is assigned a unique degree r polynomial over $\mathbb{F}_{2^n}$. The data in the reservoir word j is considered to be in the field $\mathbb{F}_{2^n}[x]/\mathbf{q}_j$. Inverses of $\mathbf{q}_j$ mod $\mathbf{q}_i$ for $0 \leq i < j < u$, needed in converting data from the entropy reservoir to the extraction register, can be found in figure A.2.
f, c_i	: This is a fixed primitive polynomial (equation A.0.2) of degree u over $\mathbb{F}_{(2^{nr})}$ used to 'Galois step' the entropy reservoir, with $\mathbf{f} = x^u - \sum_{i=0}^{u-1} c_i x^i \bmod \mathbf{B}$ and $c_i \in \mathbb{F}_2[x]/\mathbf{B}$. A primitive polynomial is an irreducible (prime) polynomial such that its root has maximal order: $(2^{urn} - 1)$.

$n = 8$	$u = 7$	$r = 5$	$v = 7$	$\mathbf{B} = \texttt{0x1a9fa84e079}$	$\mathbf{b} = \texttt{0x171}$

(A.0.1)

Coefficients for Galois Register Polynomial $\mathbf{f}(x) = x^u + \sum_{i=0}^{u-1} c_i x^i$

i	6	5	4	3	2	1	0
c_i	0xa255a09801	0x2b364ac4d9	0x2ca010c6f3	0x228fdea59c	0xc6e97260ef	0x1dc49bff3e	0x72f6f5f1de

(A.0.2)

$\mathbf{q}_0(x) \equiv x^5 + \texttt{0x50}x^4 + \texttt{0x18}x^3 + \texttt{0x24}x^2 + \texttt{0x75}x^1 + \texttt{0x02}$
$\mathbf{q}_1(x) \equiv x^5 + \texttt{0xcd}x^4 + \texttt{0x1c}x^3 + \texttt{0x53}x^2 + \texttt{0x26}x^1 + \texttt{0xd8}$
$\mathbf{q}_2(x) \equiv x^5 + \texttt{0xe2}x^4 + \texttt{0xb7}x^3 + \texttt{0x9a}x^2 + \texttt{0xfa}x^1 + \texttt{0x9a}$
$\mathbf{q}_3(x) \equiv x^5 + \texttt{0xd5}x^4 + \texttt{0x09}x^3 + \texttt{0xde}x^2 + \texttt{0x34}x^1 + \texttt{0xe6}$
$\mathbf{q}_4(x) \equiv x^5 + \texttt{0xd5}x^4 + \texttt{0x64}x^3 + \texttt{0xec}x^2 + \texttt{0xa0}x^1 + \texttt{0x25}$
$\mathbf{q}_5(x) \equiv x^5 + \texttt{0x9b}x^4 + \texttt{0xd9}x^3 + \texttt{0xc6}x^2 + \texttt{0x39}x^1 + \texttt{0x61}$
$\mathbf{q}_6(x) \equiv x^5 + \texttt{0xd7}x^4 + \texttt{0x81}x^3 + \texttt{0x25}x^2 + \texttt{0xad}x^1 + \texttt{0xea}$

(A.0.3)

Figure A.1: Suggested Parameter Values

$\mathbf{q}_0(x) = x^5 + \mathtt{0x50}x^4 + \mathtt{0x18}x^3 + \mathtt{0x24}x^2 + \mathtt{0x75}x^1 + \mathtt{0x02}$	
	$\mathbf{q}_2^{-1}(x) \equiv \mathtt{0x65}x^4 + \mathtt{0x0f}x^3 + \mathtt{0xe2}x^2 + \mathtt{0x38}x^1 + \mathtt{0x0f}$
	$\mathbf{q}_2^{-1}(x) \equiv \mathtt{0x24}x^4 + \mathtt{0x24}x^3 + \mathtt{0xb2}x^2 + \mathtt{0xf2}x^1 + \mathtt{0x6f}$
	$\mathbf{q}_3^{-1}(x) \equiv \mathtt{0x95}x^4 + \mathtt{0xb1}x^3 + \mathtt{0x08}x^2 + \mathtt{0x82}x^1 + \mathtt{0x0d}$
	$\mathbf{q}_4^{-1}(x) \equiv \mathtt{0x88}x^4 + \mathtt{0x3a}x^3 + \mathtt{0xcd}x^2 + \mathtt{0x6f}x^1 + \mathtt{0x8f}$
	$\mathbf{q}_5^{-1}(x) \equiv \mathtt{0x36}x^4 + \mathtt{0xac}x^3 + \mathtt{0x72}x^2 + \mathtt{0x9c}x^1 + \mathtt{0xe4}$
	$\mathbf{q}_6^{-1}(x) \equiv \mathtt{0xa9}x^4 + \mathtt{0x1a}x^3 + \mathtt{0x1e}x^2 + \mathtt{0xb9}x^1 + \mathtt{0x1f}$
$\mathbf{q}_1(x) = x^5 + \mathtt{0xcd}x^4 + \mathtt{0x1c}x^3 + \mathtt{0x53}x^2 + \mathtt{0x26}x^1 + \mathtt{0xd8}$	
	$\mathbf{q}_2^{-1}(x) \equiv \mathtt{0xf5}x^4 + \mathtt{0x23}x^3 + \mathtt{0x29}x^2 + \mathtt{0x6b}x^1 + \mathtt{0x3c}$
	$\mathbf{q}_3^{-1}(x) \equiv \mathtt{0x9c}x^4 + \mathtt{0x4a}x^3 + \mathtt{0x7d}x^2 + \mathtt{0x74}x^1 + \mathtt{0x76}$
	$\mathbf{q}_4^{-1}(x) \equiv \mathtt{0xc9}x^4 + \mathtt{0xec}x^3 + \mathtt{0xf1}x^2 + \mathtt{0xb4}x^1 + \mathtt{0xd6}$
	$\mathbf{q}_5^{-1}(x) \equiv \mathtt{0x1b}x^4 + \mathtt{0xe3}x^3 + \mathtt{0xad}x^2 + \mathtt{0x4c}x^1 + \mathtt{0x3f}$
	$\mathbf{q}_6^{-1}(x) \equiv \mathtt{0x3e}x^4 + \mathtt{0x3a}x^3 + \mathtt{0xae}x^2 + \mathtt{0x2a}x^1 + \mathtt{0x4a}$
$\mathbf{q}_2(x) = x^5 + \mathtt{0xe2}x^4 + \mathtt{0xb7}x^3 + \mathtt{0x9a}x^2 + \mathtt{0xfa}x^1 + \mathtt{0x9a}$	
	$\mathbf{q}_3^{-1}(x) \equiv \mathtt{0xaa}x^4 + \mathtt{0x93}x^3 + \mathtt{0x8f}x^2 + \mathtt{0x56}x^1 + \mathtt{0x37}$
	$\mathbf{q}_4^{-1}(x) \equiv \mathtt{0x72}x^4 + \mathtt{0x8f}x^3 + \mathtt{0x5a}x^2 + \mathtt{0x51}x^1 + \mathtt{0x50}$
	$\mathbf{q}_5^{-1}(x) \equiv \mathtt{0x4e}x^4 + \mathtt{0x9a}x^3 + \mathtt{0x05}x^2 + \mathtt{0xda}x^1 + \mathtt{0x18}$
	$\mathbf{q}_6^{-1}(x) \equiv \mathtt{0x2e}x^4 + \mathtt{0xb3}x^3 + \mathtt{0x3c}x^2 + \mathtt{0x6e}x^1 + \mathtt{0xd7}$
$\mathbf{q}_3(x) = x^5 + \mathtt{0xd5}x^4 + \mathtt{0x09}x^3 + \mathtt{0xde}x^2 + \mathtt{0x34}x^1 + \mathtt{0xe6}$	
	$\mathbf{q}_4^{-1}(x) \equiv \mathtt{0xea}x^4 + \mathtt{0x74}x^3 + \mathtt{0x26}x^2 + \mathtt{0x10}x^1 + \mathtt{0x5c}$
	$\mathbf{q}_5^{-1}(x) \equiv \mathtt{0x06}x^4 + \mathtt{0x4d}x^3 + \mathtt{0x59}x^2 + \mathtt{0xe0}x^1 + \mathtt{0xb0}$
	$\mathbf{q}_6^{-1}(x) \equiv \mathtt{0x47}x^4 + \mathtt{0x2c}x^3 + \mathtt{0x32}x^2 + \mathtt{0x16}x^1 + \mathtt{0x54}$
$\mathbf{q}_4(x) = x^5 + \mathtt{0xd5}x^4 + \mathtt{0x64}x^3 + \mathtt{0xec}x^2 + \mathtt{0xa0}x^1 + \mathtt{0x25}$	
	$\mathbf{q}_5^{-1}(x) \equiv \mathtt{0xfc}x^4 + \mathtt{0x8f}x^3 + \mathtt{0x5b}x^2 + \mathtt{0xa0}x^1 + \mathtt{0xdf}$
	$\mathbf{q}_6^{-1}(x) \equiv \mathtt{0xb2}x^4 + \mathtt{0x0e}x^3 + \mathtt{0x9e}x^2 + \mathtt{0x7d}x^1 + \mathtt{0x98}$
$\mathbf{q}_5(x) = x^5 + \mathtt{0x9b}x^4 + \mathtt{0xd9}x^3 + \mathtt{0xc6}x^2 + \mathtt{0x39}x^1 + \mathtt{0x61}$	
	$\mathbf{q}_6^{-1}(x) \equiv \mathtt{0xd2}x^4 + \mathtt{0x08}x^3 + \mathtt{0xe6}x^2 + \mathtt{0x81}x^1 + \mathtt{0x52}$
$\mathbf{q}_6(x) = x^5 + \mathtt{0xd7}x^4 + \mathtt{0x81}x^3 + \mathtt{0x25}x^2 + \mathtt{0xad}x^1 + \mathtt{0xea}$	

Figure A.2: Entropy Reservoir Polynomial Moduli and Required Inverses

B Entropy Basics

As mentioned earlier, cryptographic random data is generally created using raw, or 'true', random data processed through a conditioner/extender. How do we reconcile the raw entropy

entering the system with the entropy leaving the system? To answer that question we need to briefly examine what entropy is, how it is measured, and what we expect from our systems.

C What is Entropy and How do we measure it.

In information theory, entropy in data is defined by two equivalent definitions:

1. The measure of randomness in data;
2. The amount of information contained in data.

Note that entropy is relative. It is not a solid, physical entity. Entropy depends on perspective or what is known and unknown about the data to a given entity. Once viewed, all information in the data is known to the viewer (zero entropy in the viewers perspective), but the data still contains entropy to non-viewers. The belief that entropy is something that has a classical, fixed measure is false and causes many interpretation issues.

Knowledge of underlying entropy is represented in a probability distribution, $\mathcal{P}$. Assume that there are r possible states $\{x_j \mid 0 < j \leq r\}$, with state j having probability $pr(x_j) = p_j$ of occurring, with

$$\mathcal{P} = \left\{ p_j \mid 0 < j \leq r;\ 0 \leq p_j \leq 1;\ \sum_{j=1}^{r} p_j = 1 \right\}. \tag{C.0.1}$$

Shannons[3] are the most commonly used unit[3]. A Shannon (Sh) is the maximal entropy which can be contained in a single bit. Alternatively, a Shannon is the entropy of a two state system with equally probable states. One bit can store at most one Shannon, n-bits can store at most n-Shannons, and in general, if there are k possible states, maximal entropy in the system is

$$log_2(k) = lg(k)\,.$$

Entropy of an individual event is inversely proportional to the probability of it occurring: the more unlikely an event, the higher the entropy from its occurrence. For example, in English we get more information about a hidden word if the letter Z is revealed than if the letter E is revealed. The entropy of the event j (for the given probability distribution (equation C.0.1)) is given by:

$$\mathfrak{ent}(j) = -lg(p_j)\,. \tag{C.0.2}$$

[3]The term 'bits' is often used instead of Shannons. Multitasking the word 'bit' for both an element of the set $\{0, 1\}$ and an information measure can be misleading and confusing. While use of base 2 logarithms is most common, there are other entropy measurement units based off varying bases: hartleys (bans or dits) use base 10 and nat (natural units, nit) use base e.

D Algorithms

DYCE has a number of variables (figure 2.3, 2.6), constants (figure 2.2), and computational notation. Operations are performed over finite fields and integers. Both have operations such as addition and multiplication and standard symbols for these operations are identical. To clarify, the following symbols will be used:

$a + b$	Integer addition
$a \oplus b$ $a + b \bmod 2$	Mod 2 addition, or an XOR of a and b
$a \odot b$ $ab \bmod 2$	Mod 2 multiplication, or an AND of a and b
$(f(x) \bmod p)$	When a modulus is present, this implies all operations are modulo p, and the result is in reduced form. For a polynomial p this implies that the degree of $(f(x) \bmod p)$ is less than the degree of p. In DYCE, p is a binary polynomial – i.e., addition is always XOR, multiplication depends on the modulus (algorithm 20, 18).
$a \ll b$	This symbol represents a left shift of the binary value a by the number of bits b. A left shift equates to multiplication by x (algorithm 20, 18) (the root) when working modulo polynomials over $\mathbb{F}_2$.
$a \gg b$	This symbol represents a right shift of the binary value a by the number of bits b.

The upper level algorithms for DYCE are in section D.1, with the foundation algorithms for quotient ring transform (QRT) and underlying finite field operations are in sections D.2 and D.3 respectively.

D.1 DYCE

Algorithm 1: DYCE Generate Random

Description: Compute cryptographic random, updating the Shannons used as needed.

Input: DYCE box with variables $s,\ w,\ \{d_j\}_{j=0}^{u-1},\ \{k_j, e_j\}_{j=0}^{v-1}$

Output: Updated DYCE box, with the output random: $\{e_j\}_{j=0}^{v-1}$ n-bit long words of cryptographic random data

I: Initialize (algorithm 2) DYCE box;

II: While cryptographic random data is needed:

- *A*: Until $s < s_m$, add raw random (algorithm 3); *Ensure there is enough entropy*
- *B*: If $w = 0$, update extraction state (algorithm 8); *Make sure extraction state is initialized/reset with random data*
- *C*: Compute internal random (algorithm 4) and return $\{e_j\}_{j=0}^{v-1}$
- *D*: Account for random extracted: $w = w + 1$;
- *E*: Update the Shannons used and the DYCE state as needed (algorithm 5).

End of Algorithm 1

Algorithm 2: Initialize

Description: Initializes an empty DYCE box, preparing the system for incoming entropy.

Input: Empty DYCE box

Output: Initialized DYCE box with a full random state

I: Initialize entropy reservoir to $d_j =$ `0xffffffffff` for $0 \leq j < u$;

II: Initialize extraction state to $k_j = j$ for $0 \leq j < v$;

III: Initialize the Shannon level of entropy reservoir: $s = 0$;

IV: Set $w = 0$. *set the number of times cryptographic random has been extracted to zero*

End of Algorithm 2

Algorithm 3: Add new raw entropy

Description: Adds incoming words of raw entropy into the DYCE box, mixing the random and updating the Shannon level of the box. Called before algorithm 4 if there is not enough random for extraction, and at any time between random data extractions when random is available.

Input: DYCE box
t-words (rn-bits each) of raw random, $\{a_i\}_{i=0}^{t-1}$, at p-Shannon's per bit and $t \leq u$

Output: DYCE box with updated entropy level s

I: for j=0 to $t-1$:

- $d_{u-1-j} = d_{u-1-j} \oplus a_j$;

II: Update the Shannons in entropy reservoir:

A: Set $\mathtt{ns} = \lfloor trnp \rfloor$, the number of new Shannons added;

B: Set the new level:

$$s = \begin{cases} urn & s + \mathtt{ns} > urn \\ s + \mathtt{ns} & \text{otherwise} \end{cases}$$

III: If entropy reservoir is all zero, reset entropy reservoir (algorithm 11);

IV: Step entropy reservoir (algorithm 6);

End of Algorithm 3

Algorithm 4: Compute next random state

Description: Computes internal random in DYCE box, storing the new random output in the extraction register. Called when random is requested (algorithm 1) and during update procedures (algorithms 9 and 8).

Input: DYCE box

Output: $\{e_j\}_{j=0}^{v-1}$ n-bit long words of cryptographic random data

I: Step entropy reservoir (algorithm 6);

II: Compute extraction register: $\{e_j\}$ (algorithm 12);

End of Algorithm 4

Algorithm 5: Adjust Entropy Level After Extraction

Description: Adjusts the Shannon levels after an extraction of random data and updates the extraction state and entropy reservoir if needed. Called in algorithm 1.

Input: DYCE Box, just after random was extracted

Output: Updated DYCE parameters

I: if $w \geq w_m$: *Update the count of raw Shannons used, and compute new extraction state*

- *A*: $s = s - 1$; *One Shannon has been used.*
- *B*: $w = 0$; *This triggers the computation of extraction state (algorithm 8) in algorithm 1*
- *C*: if $(s \bmod r) = 0$: *Update the entropy reservoir periodically if Shannon level drops*
  - Update entropy reservoir (algorithm 9);

End of Algorithm 5

Algorithm 6: Stepping the Entropy Reservoir

Description: Computes a random stepping number $\mathtt{t}$ between $u \leq \mathtt{t} < 2u$ and single steps the register $\mathtt{t}$ steps. Called in algorithms 8, 11, 3, and 4

Input: DYCE box

Output: DYCE with stepped entropy reservoir

I: Set $\mathtt{t} = d_{u-1}$; *Add all extraction state values to get variable for stepping entropy reservoir*

II: For $j = 0$ to $v - 1$:

- $\mathtt{t} = \mathtt{t} \oplus k_j$;

III: Set $\mathtt{t} = u + (\mathtt{t} \bmod u)$; *Stepping count should be between $u \leq t < 2u$*

IV: For $j = 0$ to $\mathtt{t} - 1$:

- Step entropy reservoir (algorithm 7);

End of Algorithm 6

Algorithm 7: Single Step of Entropy Reservoir (Galois Stepping)

Description: This function performs a single Galois step on the entropy reservoir. Called in algorithm 6

Input: DYCE box

Output: updated DYCE Box

I: Set $d = d_{u-1}$;

II: for $j = u - 1$ down to 1: *Note: this can be parallelized*

- $d_j = d_{j-1} \oplus d \cdot c_j \bmod \mathbf{B}$;

III: Set $d_0 = d \cdot c_0$;

End of Algorithm 7

Algorithm 8: Update Extraction State

Description: The extraction state values are updated every time a Shannon of raw random is is used. Called from algorithms 11, 1, 4, and 9.

Input: DYCE box variables

Output: updated extraction state

I: Compute random data (algorithm 4): $\{e_j\}_{j=0}^{v-1}$

Note: $\left\{\overline{d}_i\right\}_{i=0}^{(u-1)}$ is computed here and used later in this algorithm

II: Set $\texttt{t} = 1$; $k_0 = e_0$ — *t is the number of distinct k_i found*

III: For $j = 1$ to $v - 1$: — *Find distinct subset of $\{e_i\}$ to use as new $\{k_i\}$*

- If $e_j \notin \{k_i\}_{i=0}^{\texttt{t}-1}$ — *if the j-th random output data is not in the current set of extraction state*
  - *A*: $k_{\texttt{t}} = e_j$
  - *B*: $\texttt{t} = \texttt{t} + 1$

IV: Expand the current extraction state (algorithm 10);

V: While $\texttt{t} < v$:

- *A*: Step the entropy reservoir (algorithm 6);
- *B*: Compute the inner QRT state (algorithm 13);
- *C*: Expand the current extraction state (algorithm 10);

End of Algorithm 8

Algorithm 9: Update Entropy Reservoir

Description: Updates the entropy reservoir in states when enough random has been extracted with out being replaced. Called in algorithm 4.

Input: DYCE box variables

Output: updated DYCE box variables

I: Compute (algorithm 4) random: $\{e_i\}_{i=0}^{v}$

II: For $i = 0$ to $v - 1$:

A: $d_{u-1} = d_{u-1} \oplus e_i$; *Infuse random from extraction state back into entropy reservoir*

B: Single step the entropy reservoir (algorithm 7) *Spread the added value through out the full reservoir*

III: If $\{d_i\}_{i=0}^{u-1} = \{0\}$:

- Reset entropy reservoir (algorithm 11)

IV: Compute new extraction state (algorithm 8)

End of Algorithm 9

Algorithm 10: Expand Extraction State

Description: This routine finds an elements of $\mathbb{F}_{2^n}$, not equal to any elements in $\{d_j\}_{j=0}^{t-1}$, by evaluating the current transformed. Called from algorithm 8.

Input: $\left\{\overline{d}_j\right\}_{j=0}^{u-1}$: Inner QRT state (algorithm 13)
$\mathtt{t}$: Number of Extraction State filled

Output: $\mathtt{t}$: Updated number of extraction state filled
Updated extraction state

I: Set $j = 0$

II: While $j < \mathtt{t}$ and $\mathtt{t} < v$; *Grow the extraction state up to the maximal v values*

- *A*: Compute $e = \mathbf{D}(k_j) \bmod \mathbf{b}$ (algorithm 15, 16);
- *B*: if $e \notin \{k_i\}_{i=0}^{\mathtt{t}-1}$; *add computed value if it is not yet in extraction state*
  - *1*: $k_{\mathtt{t}} = e$;
  - *2*: $\mathtt{t} = \mathtt{t} + 1$;

III: Return $\mathtt{t}$.

End of Algorithm 10

Algorithm 11: Handling All Zeros

Description: An all zero entropy reservoir results in a static state, which is not allowed. This routine extracts random from the extraction state and puts it back into the entropy reservoir, updating the Shannon level of the system. Note that a zero state can only occur when new random is being added or entropy reservoir is updated. The probability of an all zero state occurring at any point is close to zero: $pr(\text{zero entropy reservoir}) = \frac{1}{2^{urn}}$. Called in algorithms 3 and 9.

Input: Initialized DYCE box, with all zero entropy reservoir

Output: DYCE box with non-zero entropy reservoir and adjusted Shannon level

I: Set $d_j = k_j$ for $0 \leq j < min(u, v)$.

II: Step entropy reservoir (algorithm 6)

III: Compute new extraction register: $\{k_j\}_{j=0}^{v-1}$ (algorithm 8).

IV: Set the Shannon level of entropy reservoir:

$$s = min\,(u, v)\, n.$$

End of Algorithm 11

D.2 QRT algorithms

The following algorithms perform the QRT (Quotient Ring Transform)

Algorithm 12: QRT

Description: Performs the Quotient Ring Transform on the entropy reservoir, returning the transformed version which is used to extract random or create a new extraction state. Called from algorithm 4.

Input: $\{d_i \bmod \mathbf{q}_i\}_{i=0}^{(u-1)}$, $\{\mathbf{q}_j\}_{j=0}^{(u-1)}$

Output: $\{e_j\}_{j=0}^{(u-1)}$

I: Compute $\left\{\overline{d}_i\right\}$ (algorithm 13)

II: Compute and return $\{e_j\}$ (algorithm 14)

End of Algorithm 12

Algorithm 13: Inward QRT

Description: Converts the data in the entropy reservoir to a form which can be easily evaluated to obtain the random output in extraction register. This is called from algorithms 8, 12. Output from this algorithm is also needed as input to several other algorithms.

Input: $\{d_i \bmod \mathbf{q}_i\}_{i=0}^{(u-1)}$

Output: $\{\overline{d}_i\}_{i=0}^{(u-1)}$

I: Set $\overline{d}_i = d_i$ for $0 \leq i < u$.

II: for $i = 0$ to $(u-2)$ do:

Note: This can be done in parallel

A: for $j = i+1$ to $(u-1)$:

- $\overline{d}_j = \overline{d}_j \oplus \overline{d}_i$
- $\overline{d}_j = \overline{d}_j \cdot \mathbf{q}_i^{-1} \bmod \mathbf{q}_j$ (algorithm 20)

Note: Inverses in this equation ($\mathbf{q}_i^{-1}$) can be computed using Euclid's extended GCD algorithm [] or can be found in figure **??**.

III: return $\{\overline{d}_i\}_{i=0}^{u-1}$

End of Algorithm 13

Algorithm 14: Outward QRT

Description: Converts the transformed values (algorithm 13) into reduced (evaluated) output. This is called from algorithms 12.

Input: $\{\overline{d}_i\}_{i=0}^{(u-1)}$, $\{\mathbf{q}_j\}_{j=0}^{(u-1)}$

Output: $\{e_j\}_{j=0}^{(u-1)}$

I: for $j = 0$ to $(u-1)$:

- Compute e_j (algorithm 15), or equivalently, use constant term version (algorithm 16)

II: return $\{e_j\}$

End of Algorithm 14

Algorithm 15: Single QRT Extraction

Description: This is a more general, and less concrete, extraction algorithm. For a more concrete version, see algorithm 16. Called in algorithms 10, 14.

Input: $\{\overline{d}_i\}_{i=0}^{(u-1)}, \{\mathbf{q}_i\}_{i=0}^{(u-1)}$: Derived from inner QRT (algorithm 13), represents large polynomial $\mathbf{D}(x)$
$p(x)$: Reduction polynomial

Output: $e : e = \mathbf{D}(x) \bmod p(x)$

I: Set $e = \overline{d}_{u-1} \bmod p(x)$

II: for $i = u-2$ down to 0:

- $e = \left(\overline{d}_i + \mathbf{q}_i e \bmod p(x)\right)$

III: Return e

End of Algorithm 15

Algorithm 16: Single QRT Extraction, using $p(x) = (x - k)$

Description: Extracts the value from the woven version of $\mathbf{D}$, modulo a monic degree one polynomial $(x - k)$. For a more theoretical, less concrete version, see algorithm 15. Called in algorithms 10, 14.

Input: $\{\overline{d}_i\}_{i=0}^{(u-1)}, \{\mathbf{q}_i\}_{j=0}^{(u-1)}$: Woven version of $\mathbf{D}$ (equation 2.2.1), computed using inner QRT (algorithm 13)
k : Reduction polynomial is $p(x) = (x - k)$

Output: e : $e = \mathbf{D}(x) \bmod p(x)$ or equivalently, $\mathbf{D}(k)$

I: Use polynomial view of $\overline{d}_{u-1}$ (figure 2.3):

$$\overline{d}_{u-1}(x) = \sum_{j=0}^{r-1} \overline{d}_{u-1}^{(j)} x^j$$

II: Set $e = \overline{d}_{u-1}(k)$ (algorithm 17)

III: for $i = u - 2$ down to 0 compute: $e = \overline{d}_i(k) \oplus e\mathbf{q}_i(k) \bmod \mathbf{b}$

A: $\texttt{temp} = \mathbf{q}_i(k) \bmod \mathbf{b}$ (algorithm 17)

B: $e = \texttt{temp} \cdot e \bmod \mathbf{b}$ (algorithm 18, 19)

C: $\texttt{temp} = \overline{d}_i(k) \bmod m$ (algorithm 17)

D: $e = e \oplus \texttt{temp}$.

IV: Return e

End of Algorithm 16

D.3 Finite Field Operations

Finite field arithmetic is required throughout the DYCE. This section details algorithms to perform these operations.

Algorithm 17: Polynomial Evaluation over $\mathbb{F}_{2^h}$

Description: Evaluates $G(x)$ at a where the a and the coefficients of G are considered modulo a degree h binary polynomial m

Input: h: Degree of polynomial m
m: m represents a degree h polynomial moduli $m(x) = x^h + \sum_{j=0}^{h-1} m_j x^j \bmod 2$, where m_j is the j-th bit of m
$G = [G_z, G_{z-1}, \ldots, G_0]$: A degree z polynomial over $\mathbb{F}_{2^h}$ with $G_j \in \mathbb{F}_{2^h}$
a : An element of $\mathbb{F}_{2^h}$

Output: c: $c = \sum_{j=0}^{(z-1)} B_j a^j \bmod m$

I: $c = B_{z-1}$

II: For $j = (z-1)$ down to $j = 0$ do:

A: $c = c \cdot a \bmod m$ (algorithm 18, 19)

B: $c = c \oplus B_j$

III: Return c.

End of Algorithm 17

Algorithm 18: Modular Multiplication in $\mathbb{F}_{2^h}$, for small h

Description: Performs multiplication modulo $m(x)$ over $\mathbb{F}_2$, where the degree of h is less than a single computer word size. Polynomials are stored in a single computer word. For example, the polynomial $x^3 + x + 1$ is stored as the binary value `1011`, or in hex as `0xb`. Note: for if h is sufficiently small (say $h \leq 8$), multiplication is much faster if done using look-up tables (algorithm 19).

Input: h: Degree of polynomial m
m: m represents a degree h polynomial moduli $m(x) = x^h + \sum_{j=0}^{h-1} m_j x^j \bmod 2$, where m_j is the j-th bit of m
a: a represents a polynomial in reduced form modulo $m(x)$: $a(x) = \sum_{j=0}^{h-1} a_j x^j \bmod m(x)$ where a_j is the j-th bit of a
b: b represents a polynomial in reduced form modulo $m(x)$: $b(x) = \sum_{j=0}^{h-1} b_j x^j \bmod m(x)$ where b_j is the j-th bit of b.

Output: c: $c \equiv a \cdot b \bmod m$, with $c(x) = \sum_{j=0}^{h-1} c_j x^j \bmod m(x)$ where c_j is the j-th bit of c.

I: if $a = 0$ or $b = 0$: Return 0.

II: Set $\texttt{hob} = 1 \ll (h-1)$ — *This mask determines when a polynomial reduction is needed.*

III: Set $\texttt{msk} = 1 \ll (h-1)$ — *This mask determines when polynomial b should be added to c.*

IV: Set $\texttt{t} = (h-1)$ — *Current bit of a to examine.*

V: while $a \odot \texttt{msk} = 0$ — *Find highest order, non-zero bit in a. Note: ⊙ is binary AND*

- Adjust bit check variables: $\texttt{t} = \texttt{t} - 1$, and $\texttt{msk} = \texttt{msk} \gg 1$

VI: Adjust bit check variables: $\texttt{t} = \texttt{t} - 1$, and $\texttt{msk} = \texttt{msk} \gg 1$

High order bit is dealt with by setting c = b

VII: Set $c = b$.

VIII: While $t \geq 0$:

A: if $\texttt{hob} \odot c \neq 0$: — *Multiply by x and reduce mod m*

1: $c = c \oplus \texttt{hob}$

2: $c = (c \ll 1) \oplus m$

B: else: — *Multiply by x, but no reduction is needed*

- $c = c \ll 1$

C: if $a \odot \texttt{msk} \neq 0$ — *If the current a coefficient is 1, add b to the result.*

- $c = c \oplus b$.

D: Adjust bit check variables: $\texttt{t} = \texttt{t} - 1$, and $\texttt{msk} = \texttt{msk} \gg 1$

IX: Return c.

End of Algorithm 18

Algorithm 19: Modular Multiplication in $\mathbb{F}_{2^h}$, for small h using a primitive degree h polynomial.

Description: Performs multiplication modulo $m(x)$ (where m is a primitive polynomial) over $\mathbb{F}_2$, where the degree h is less than a single computer word size, using power and log table look ups.

Input: h: Degree of polynomial m
$[pow_j]_{j=0}^{2^h-2}$: $pow_j \equiv x^j \bmod m(x)$
$[log_j]_{j=1}^{2^h-1}$: $log_j = t$ such that $x^t \equiv j \bmod m(x)$ (i.e., inverse of power table) a: a represents a polynomial in reduced form modulo $m(x)$: $a(x) = \sum_{j=0}^{h-1} a_j x^j \bmod m(x)$ where a_j is the j-th bit of a
b: b represents a polynomial in reduced form modulo $m(x)$: $b(x) = \sum_{j=0}^{h-1} b_j x^j \bmod m(x)$ where b_j is the j-th bit of b.

Output: c: $c \equiv a \cdot b \bmod m$, with $c(x) = \sum_{j=0}^{h-1} c_j x^j \bmod m(x)$ where c_j is the j-th bit of c.

I: if a or b is zero, return 0.

II: Compute $j = (log_a + log_b) \bmod 2^h - 1$;

III: Return pow_j.

End of Algorithm 19

Algorithm 20: Polynomial Modular Multiplication over $\mathbb{F}_{2^h}$

Description: Performs multiplication modulo $q(x)$ over $\mathbb{F}_{2^h}$. This algorithm is similar to 18, except coefficients of the polynomials a, b are h-bit values (elements in $\mathbb{F}_{2^h}$) instead of bits, and algorithm 18 or 19 replaces some of the shifts.

Input:
h: Degree of base polynomial moduls m
m: A degree h irreducible polynomial over $\mathbb{F}_2$
q : Represents a degree z polynomial over $\mathbb{F}_{2^h}$ (i.e., mod m) - $q = x^z + \sum_{j=0}^{z-1} q_j x^j$ where q_j is an h-bit value.
a: Represents a polynomial of degree less than z over $\mathbb{F}_{2^h}$ – $a(x) = \sum_{j=0}^{z-1} a_j x^j$ where a_j is an h-bit value.
b: Represents a polynomial of degree less than z over $\mathbb{F}_{2^h}$ – $b(x) = \sum_{j=0}^{z-1} b_j x^j$ where b_j is an h-bit value.

Output: c: $c \equiv a \cdot b \bmod q$, with $c(x) = \sum_{j=0}^{z-1} c_j x^j \bmod m(x)$ where c_j is an h-bit value.

I: if $a = 0$ or $b = 0$: Return 0.

II: Set $c = 0$.

III: Set $\mathtt{t} = (z-1)$ *Current word of a to examine.*

IV: While $a_{\mathtt{t}} = 0$ and $\mathtt{t} \geq 0$: $\mathtt{t} = \mathtt{t} - 1$ *Find highest order, non-zero word in a.*

V: While $t \geq 0$: *For each term a_t, multiply c by x and add a_t multiplied by b*

- *A*: $\mathtt{temp} = c_{z-1}$ *Copy off high order term. After multiplication by x (left shift one) this term is $c_{z-1}x^z$ which must be reduced mod q*
- *B*: for $j = (z-1)$ down to 1 do: *New c_j is the old c_{j-1} plus the j-th component of $c_{z-1}x^z$ mod q*
  - *1*: $c_j = \mathtt{temp} \cdot q_j \bmod m$ (algorithm 18, 19) *This is the j-term for $c_{z-1}x^z$*
  - *2*: $c_j = c_j \oplus c_{j-1}$. *This is the old c_{j-1} after multiplication by x*
- *C*: $c_0 = \mathtt{temp} \cdot q_0 \bmod m$ (algorithm 18, 19) *The 0-th term is just the constant term of $c_{z-1}x^z$ mod q*
- *D*: for $j = 0$ to $(z-1)$: *Add on the $a_t \cdot b$*
  - *1*: $\mathtt{temp} = a_{\mathtt{t}} \cdot b_j \bmod m$ (algorithm 18, 19);
  - *2*: $c_j = c_j \oplus \mathtt{temp}$
- *E*: $\mathtt{t} = \mathtt{t} - 1$

VI: Return c.

End of Algorithm 20

References

[1] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno, Cryptography engineering: Design principles and practical application, ch. 9, pp. 137–161, Wiley Publishing, Inc, 10475 Crosspoint Boulevard; Indianapolis, IN 46256, 2010.

[2] P. Flajolet and A. M. Odlyzko, Random mapping statistics (invited), Advances in Cryptology - EuroCrypt '89 (Berlin) (Jean-Jacques Quisquater and Joos Vandewalle, eds.), Springer-Verlag, 1989, Lecture Notes in Computer Science Volume 434, pp. 329–354.

[3] ISO, IEC 80000-13:2008: Quantities and units – part 13: Information science and technology, Standards document 13, International Organization for Standardization (ISO), Geneva, Switzerland, March 2008.

[4] Anna Johnston, Dispersed cryptography and the quotient ring transform, IACR e-Print, February 2017.

[5] Rudolf Lild and Harald Niederreiter, Finite fields, second ed., Cambridge University Press, 1997.