Some Primitive Structures in *Seqbuild*

Richard Pham

November 2025

Contents

1 Introduction

As developments in or concerning computing continue on, spotty in some academic subfields, seemingly arbitrary valuations in marketing although economics and finance are really another matter, and invested in for arguably arbitrary ethics, the computing subfield of pseudo-random number generators (PRNG) remains a staple in secure computing. The work, detailed in this paper, focuses on the software side of cryptography, implementation by which PRNGs are in fundamental demand.

The paper is split into three sections, that is, theoretical frameworks for measuring and generating numbers according to pre-defined definitions of randomness, descriptions of available primitive structures of *Seqbuild* used for number generators, and issues/improvements on the measurement of randomness. Paradoxy, of the computationally perplexing kind, is at the center of the difficulties elaborated on in the third section. The use of the descriptor "primitive" for the number generating structures is also questionable, but nevertheless selected as the most fitting adjective for them in the greater scope of infinitely many pseudo-random number generators possible through synthesis of *Seqbuild*'s primitive structures.

An all-too-common number generator class is linear congruential generators (**LCG**). This class of number generator serves as an effective norm in measurement and improvement of diversified pools of computerized methodologies for numerical generation. Three big terms are emphasized in pseudo-random number generation. They are "distinguishability", "randomness", and "secured". The technicalities in the first two come to an obvious forefront in situations where statistical tests (neutral party) and attacks (adversarial party) are almost certain to be relied on to induce "true" meaning behind the ciphertext. As of this point in time, the *Seqbuild* system contains some 10+ primitive structures. It does not abide by some constrained encryption scheme in the same structural sense that classical encryption schemes (i.e. AES,RSA) are. Although these classical encryption schemes possess the virtue of a passing difficulty of being defeated, they are essentially rigid algorithms. These procedures for mapping plaintext to ciphertext follow the same constraints in using the same structures (blocks, Lagrangian congruence, et cetera), and their strengths are entirely proportional to the brute-force requirements in finding the numbers to the formalized structure, assumed to be known by the adversary. RSA encryption, in particular, was not founded on the dogma of finiteness so associated with engineering. Instead, the central question is, in lay and imprecise terms, "what is infinity divided and modulated by two numbers, and the answer is not infinity?" A difficult question, and adversarial attacks against this category of cryptosystem would always benefit more from reduction of probable search space, thereby shrinking prioritized spans for shorter brute-force attacks. [ZM23]

Seqbuild is utilized by an API language built on top of the primitive structures of the suite. This API language is the most apparent distinguisher that sets *Seqbuild* apart from classical encryption schemes. Due to the overarching trait of fluidity found in language, the *Seqbuild* system leans heavily into the

realm of "fluid-state" encryption, a clear detour from the "solid-state" design of AES. As of this point in time, the one available encryption process of *Seqbuild* is a symmetric key that maps each character in the plaintext to one real number, and the resulting sequence of real numbers is the ciphertext. There are specific examples in this paper that show the difficulties of removing this one-layer mask from the plaintext it covers. Comparatively, the AES protocol uses 10 (128-bit), 12 (192-bit), and 14 (256-bit) rounds (layers) of encryption, each centering on a block for the block cipher scheme that it is. [Hey17] The one-layer mask that *Seqbuild*'s encryption scheme offers is from a number generator that could be any configuration of the available primitive structures, each also having its spans of initializing parameters. The Diehard Randomness Tests, authored by George Marsaglia, [MT02] and NIST Randomness Tests are comprehensive, yet incomplete, ways of measuring the quality of a PRNG. The field of machine-learning can be used to further define characteristics of PRNGs not entirely covered by these batteries of tests.

2 Defining Randomness

Defining randomness is antithetical to what randomness really is: non-expectations according to human perception and associated calculations. The premise behind this statement, questionable in the lens of empiricism, adds to the perspective that all number generators, computationally (arguably, deterministic) constructed to output values according to pre-defined processes, can only be pseudo-random. The "spottiness" of differential cryptanalysis could mislead one to believe some sequence of values is random, out of the lack of pertinent information behind those values, said pertinent information then the centerpiece of knowledge games from the source, perhaps sentient.

Statistical methodologies are important for implementing and maintaining qualities that pass pre-defined randomness metrics. The theory behind these methodologies can be more thoroughly described without the nuances and frustrations from opinion. Motives abound for why some arbitrary person or organization would create and spur half-transparent cryptosystems such as the RSA family, united solely on the premise of utilizing a relatively simple congruence formula that rests on the complexity of numerical fields for its strength. Perhaps that adds to the value of secret communication: entirely secret from the bottom up.

There are three theories of randomness. The perspective of Shannon's Information Theory views randomness in terms of uncertainty, modelled by probability distributions. Uniform distributions of ciphertext are preferred by this perspective. The second theory is credited to Solomonoff, Kolmogorov and Chaitin, and is observant of the "depths" and "lengths" of the structure. This is a more materialist approach to measuring randomness. And in practice, the "kernel" that is the source generator for the arbitrary number sequence is an objective for knowledge. The generator output is also investigated for the minimal patterned units that best characterize it, although the knowledge of the

generator, in full pertinent detail, is certainly more valuable for the temporal span of the generator's use in past, present, and future instances. The third theory of randomness is ripe for contention and authority, based on the assumption that information is valuable in and of their existences and, equally as important, that sentience values that information. There is just no way to value every piece of information equally without there being preventable failure, and as such, this third theory will collide with the underpinnings of an imagined ideal that is uniform distribution. Randomness is, according to this third theory, "an effect on the observer and thus as being relative to the observer's abilities." [Gol10] This third theory is especially intertwined with the second theory: with more knowledge on the structure of generator and generator output, the wonder (psychological state) of arbitrary phenomenon fades below a threshold that was agreed upon to be the lower bound of random. And the first theory is tied to the second during quantitative analysis, and also tied to the third for predictive capabilities. The feat of being able to predict the next thing in a chain or network of phenomena lends a convincing argument of the phenomena not being random. The source must be known enough to be at least the lower bound of confidence, a concept that also has psychological underpinnings and convolution comparable to the most sophisticated definitions of randomness, something that has already been stated as antithetical to random. But why, human beings and their meddling cannot be at full fault, only faults through their intentions or accidents. And these varieties typically demand responses in line with "there are consequences" for order, the properness of the order demanding proceeds not random from actionable intention. Note that the phrase "actionable intention" is used instead of sentient intention, due to the comparable performance of today and tomorrow's machine intelligence.

Computational indistinguishability is an essential body of concepts, used to produce appropriate comparative measure for some pre-defined metric of randomness, or alternatively but unhelpfully, the uncapturability of understanding randomness. Goldreich defines computational indistinguishability using probability ensembles.[Gol10]

Definition 2.1. Probability Ensemble. An infinite sequence of random variables $\{Z_n\}_{n\in\mathbb{N}}$ such that each Z_n ranges over strings of length polynomially related to n. There exists a polynomial p such that $\forall n$, $|Z_n| < p(n)$ and $p(|Z_n|) \geq n$.

Definition 2.2. Computational indistinguishability. The probability ensembles $\{X_n\}_{n\in\mathbb{N}}$ and $\{Y_n\}_{n\in\mathbb{N}}$ are such if for all polynomial-time ditinguisher algorithms D, any positive polynomial p, and all sufficiently large n, it holds that

$$\left|Pr[D(\{X_n\}) = 1] - Pr[D(\{Y_n\}) = 1]\right| \leq \frac{1}{p(n)}.$$

Definition 2.2 applies to single pairs (X_n, Y_n) and multiple samples

$$((X_n^{(1)}, \ldots, X_n^{(j)}), (Y_n^{(1)}, \ldots, Y_n^{(j)})).$$

The phenomenon most frequently selected to reproduce binary randomness is coin tosses. Randomness of higher single degrees oftentimes refer to the six-sided die as the physical source for generation. These practices are important to understanding discussions of event probabilities in random settings.

One-way functions are a competent measure for characterizing randomness, this competency completely tried to the incapability of third parties, especially of the adversarial variety, to understand through reproduction.

Definition 2.3 (One-way function)**.** A function f is a one-way function if $O(f(x)) < T(n)$, T an upper-bound threshold and typically polynomial for practical purposes, and is difficult to invert: for any function searcher S,

$$Pr[S(\dots) = f^{-1}] < \frac{1}{p(n)},$$

for any polynomial function p and positive real number n.

Passing effectiveness of functions designed to be one-way implies that the third party can only produce a hash table of input/output (I/O) pairs. Ideally, this hash table would infinitely grow in length with every additional I/O pair. To add more to this frustration, the function f could produce inconsistent I/O pairs. If an input at time t outputs Y_0 but outputs Y_1 at time $(t + j)$, the hash table would get complicated. I/O inconsistency is usually implemented through modulating the paramters behind function f that are second-order seeds. In number generating structures, their initialization requires a seed value (first-order). Standard implementations usually equate this seed value with some $r \in \mathbb{R}$. A second-order seed of f is thus defined as such.

Definition 2.4 (Second-order seed)**.** A number generating structure $G^{(?)}$ uses a function $G^{(?)}.f$. Structure $G^{(?)}$ has first-order seed q_0 such that

$$init_G(q_0) = G^{(?)}$$

and second-order seed q_1,

$$q_1 = parameters(G^{(?)}.f).$$

This second-order seed is mutable, unlike the finalistic signle use of q_0 in initializing $G^{(?)}$ from generator type G.

The text by Goldreich provides an analogical term, "hardcore predicate", for binary sequences.[Gol10]

In practice, the objective of strengthening or replacing current distinguishers for higher performance in use cases is a computationally expensive, relatively from the additional work involved, task. The objective is a sort of competition in knowledge games, as is the anti-objective of the encryption party. Goldreich states "the general paradigm of pseudo-randomness relies on the fact that placing computational restrictions on the observer gives rise to distributions

that are not uniform and still cannot be distinguished from uniform distributions."[Gol10] Paradoxical wisdom turns to actionable problem, as per one interpretation of his statement. The task of derandomization (technically, depseudorandomization), on the premise of there not being any actual random generator produced by sentience, will be defined and discussed in these later sections on *Seqbuild* output.

3 Description of Primitive Structures and API Language

Note: For the sake of this paper's brevity, not all of the primitive structures of *Seqbuild* are covered to an extent of detail. Some structures are omitted in these descriptions.

3.1 The LCG Class

The linear congruential generator (LCG) is one type of primitive structure in *Seqbuild*. An LCG is initialized with four values

$$(v_0, v_1, v_2, v_3) = \overrightarrow{V}.$$

The seed $\overrightarrow{V}$ is of these values

$$G(v_{t+1}) = (x_t v_1 + v_2) \ \%v_3; x_0 = v_0.$$

There are two extensions of this primitive structure, LCG variant 2 (**LCGv2**) and LCG variant 3 (**LCGv3**), that offer capabilities outside of the small confines of $\overrightarrow{V} \in \mathbb{R}$. There is at least one classification scheme for LCGs, that which observes sign changes between every contiguous pair of the output sequence (infinitely lengthed of finite set size). The modulo operator is an additional variable to the rudimentary two-dimensional form of a linear equation,

$$G(x) = xv_1 + v_2,$$

and is responsible for the finite span set by $[0, v_3)$ (positive case) or $(v_3, 0]$ (negative case) of an LCG's output values. Psychologically, an LCG can be visualized as a worm that crawls out of its hole of design (initialization or post-modulation) and re-enters this same hole in the course of LCG output up until modulation, such that every crawl until modulation may be a sequence of values in $\mathbb{R}$ different from that of the past crawl. Due to advancements and democratization of computing, in turn resulting in many seemingly random patterns becoming concretely defined enough through data mining, a significant proportion of LCGs should not be used in the implementation of trapdoor functions. This proportion of LCGs have seeds predominantly in $\mathbb{Z}^{(4)}$. Brute-forcing methodology for cases of $\mathbb{Z}^{(4)}$ is finite, and $\mathbb{Z}^{(4)}$ is the category of LCG exclusively focused on by *Seqbuild*'s solvers. Given an arbitrarily lengthed sequence $S \in \mathbb{Z}^*$,

$exec_{|S|}(G) = S$, such that at least two modulations from G have occurred, the implemented solver of *Seqbuild* considers this span of multiples,

$$(0, m_x] \ \ \& \ \ [-m_x, 0) = [-m_x, m_x] - \{0\}.$$

This span is not complete enough to accomodate all possible LCGs of $\overrightarrow{V} \in \mathbb{Z}^{(4)}$. But the search bounds guarantee for every sufficiently lengthed sequence S from an LCG G, $n = max(|S|)$, the solver runtime is $O(n^2)$. If multiple v_1 falls outside of this span $[-m_x, m_x] - \{0\}$, there is still a probability the solver solves $\overrightarrow{V}$ of G with an extraneous solution. The condition is that there exists some $m_i \in [-m_x, m_x] - \{0\}$ such that

$$(m_i x_t + v_2) \equiv (v_1 x_t + v_2) \pmod{v_3} \forall x_t \in exec_\infty(G).$$

LCGs have the statistical weakness of non-uniform distribution of its values. This weakness is especially concerning when

$$exec_\infty(G) \subset [0, v_3) \oplus (v_3, 0].$$

When $\{exec_\infty(G) = [0, v_3) \oplus (v_3, 0]\}$, there is still the constraint of that range. Granted, all numerical generators output values within a range. The density is markedly greater in LCGs. As George Marsaglia, creator of the Diehard Randomness Tests, puts it: "[t]he problem lies in the crystalline nature of multiplicative generators."[Mar68] He goes on to mathematically explain, in his paper, the similarities found in comparing n-tuples of multiplicative congruential generator output. The work in Marsaglia's paper focused on LCGs with 0 additive, in other words. The "crystalline nature" he notes as a prevalent pattern is simply skewed, by a non-zero additive, for every additional output from an LCG. Modern-day artificial intelligence, after having trained significantly on outputs from different LCGs, should have little difficulties in solving LCGs for cases of $\mathbb{Z}^{(4)}$ seeds. A hypothesis is that adequately trained A.I. would hack the runtime from $O(n^2)$ to $O(n)$.

The second version of the standard LCG has the capability of calculating all the sub-cycles (i.e. subgroups) of its output initialized from $\mathbb{Z}^{(4)}$. As of this point in time, *Seqbuild* does not make much use for this capability. The capability of cycle-detection is important in the determination of an LCG's viability. In the regular setting of demand for randomness, the more subcycles an LCG initialized by

$$\overrightarrow{V} = [?, v_1, v_2, v_3], \texttt{?} \ \ \texttt{variable,} \ v_i \ \texttt{constant},$$

has, the less viable it is in terms of uniform distribution. In an LCG, a proper subcycle can be entered from a value not in said subcycle, and the LCG output, having entered the subcycle, cannot exit out of it. In practice, LCG subcycles are trapdoors amenable to cryptanalysis.

The third version of the standard LCG resolves this problem of subcycle traps. **LCGv3** is initialized using more than $\overrightarrow{V} \in \mathbb{R}^4$. It additionally takes an

arbitrary number generator G that interminably outputs values in $\mathbb{R}$. There is a start and end modulus (pre-image range) and another pair of modulus (image range). Two booleans are also required, and are related to ternary vectors $\overrightarrow{V} \in \{-1, 0, 1\}^*$. Every sequence S from an LCG has an associated ternary vector,

$$\overrightarrow{V}_3(S) = \begin{cases} 1 & \text{if } s_{i+1} > s_i, \\ 0 & \text{if } s_{i+1} = s_i, \\ -1 & \text{otherwise.} \end{cases} \tag{1}$$

The two boolean variables are the following:

- exclusion of 0 from automated (generated) ternary delta,
- reflective modification.

Complete details of the schematics of LCGv3 can be viewed and dissected in *Seqbuild*'s source code. A rundown of the enhancements of LCGv3 over the standard LCG is about the ternary delta. LCGv3 outputs its values according to multiple v_1 and additive v_2, and the ranged modulus (pre-image range) is adjusted to satisfy a ternary vector, every x iterations, generated by its number generator parameter G. The variable x is also generated by G. If reflective modification is set to 0, x will always be 0. When the zero-exclusion is set to 1, every generated ternary delta for adjusting the output value relative to the previous does not contain 0. LCGv3 has the potential, due to this design described, to markedly differ in output quality from the "crystalline" LCG, as put by Marsaglia. LCGv3s have 2 traits, out of its design, that make it a harder generator than the standard LCG to solve. The first is LCGv3s are permutative noise adders of standard LCG output. In other words, if the two ranged modulii can be "nullified" in effect, known and accounted for by the adversary, there is the problem of

$$exec_*(G_1) = perm^{-1}(exec_*(G_3) - \overrightarrow{V}_3;$$

$$perm^-1 \texttt{ the reverse of the permutation from } G_1 \texttt{ to } G_3,$$

$$\overrightarrow{V}_3 \texttt{ the noise vector from } perm(G_1) \texttt{ to } G_3\texttt{, with length equal to } *.$$

Marsaglia's paper proves that every multiplicative congruential generator (MCG) yields output that lies in a set of parallel hyperplanes. Linear congruential generators add another dimension of sophistication, that is, the additive that cannot be accounted for with a simple shift of their MCG. Every addition from the additive is compounded in effect over the course of LCG output. As such, *LCGv2* was devised to understand its LCG kernel via a graph structure comprised of an arbitrary number of cycles C, such that for every two cycles $c_0, c_1 \in C$, c_0 and c_1 may be unconnected. If c_0 is connected to c_1 by some k unidirectional edges, all k of those edges must have source vertices from the same cycle $c_0 \oplus c_1$.

The mathematical description of the first trait of LCGv3 is not entirely fit for precisely determining the LCG kernel of an LCGv3. It does provide for a

way to correlate, after the adversary removes all the fuzz, an output sequence from an LCG3, G_3, to an output sequence from its LCG kernel, G_1. There is a probability of extraneous solutions in the equation

$$exec_*(G_1) = perm^{-1}(exec_*(G_3)) - \overrightarrow{V}_3.$$

The second trait of LCGv3 is its ability to escape sub-cycle traps that make its LCG kernel much more predictable. The adjustment process of a ranged modulii is interesting out of its pertinence to number theory. There are relations between ranged modulii and the input/output integers that result from their application. This paper does not dive into the number theory of this aspect of LCGv3. One important detail about this process is that it is runtime-intensive, the primary reason why LCGv3 loses to LCG in speed comparisons. For some cases of adjustment needed for ranged modulii, there is no simple conditional arithmetic function, but an indeterminate linear scan for candidates to replace one value of the ranged modulii. Output from LCGv3 "explode" in value after the ranged modulii adjustment for the ternary fit. In some cases, where the constriants are the entire permissible range of Python Numpy float values, this search process explodes the original ranged modulii to millions of times in absolute value. So this design of LCGv3 does appear foolish. In defense of this design, however, is a relatively difficult mathematical relation for any cryptanalytical adversary to solve. LCGv3 is also not a bijective structure, unlike the standard LCG.

In short, here is the central problem that LCGv3 solves to satisfy its ternary vector. Adjust ranged modulo R to R' to satisfy

$$(v_a \mod R'[1]) + R'[0] = v_w,$$

when given the present condition

$$(v_a \mod R[1]) + R[0] = v_c.$$

The variable v_a is the actual value from $mx + a$ (no mod), v_c is the current value $(mx + a) \mod R[1] + R[0]$, and v_w is the wanted value that satisfies the demanded ternary relation with previous value x.

3.2 The MDR Class

Description now shifts to a structure used to fit any arbitrary sequence $S \in \mathbb{Z}^*$. The structure is called Modular Decomposed Representative (**MDR**), and uses a methodology built on top of that described previously on solving for $\overrightarrow{V} \in \mathbb{Z}^4$ of an LCG. MDR works especially well for producing sequences, differing from those of the reference LCG, but with the same starting integer. This variable, the starting integer, is used as a seed to generate a sequence S', $|S'| = |S|$ and S is the reference sequence, thus making MDR a practical $i'th$ order seed for more complex number generators.
[1]

[1]The algorithm was implemented to prioritize positive integers for modulii, so it does make mistakes in representing (replicating) its reference sequence.

Structure MDR beigns the search by splitting sequence S using its ternary vector $\overrightarrow{V}_T$: S is partitionedby contiguous subsequence, each j'th subsequence ($j \neq 0$) with starting element index i corresponding to $\overrightarrow{V}_T[i-1] = -1$. This partitioning of S, call it C, is then the focus of a pattern-mining algorithm. For every chunk $c \in C$, algorithm finds the most common (multiple, additive) pair. There is only one for linear sequences. But in the arbitrary case, algorithm has to calculate more than one (multiple,additive) pair to fit just one chunk of C. Suppose an arbitrary chunk c has the most common pair (m, a) in prefix $c[:q]$, but (m, a) cannot be used to fit $c[:q]$ so that $c[:q]$ instead has the most common pair (m_0, a_0), $m_0 \neq m$ and $a_0 \neq 0$. For the contiguous chunk c_{i+1} after c_i, the last (m, a) pair from c_i is applied to the last element of c_i:

$$s = c_i[-1] \times m + a.$$

Algorithm calculates the modulus d for the result of $s \equiv c_{i+1}[0] \pmod{d}$. At the end of this algorithm's forward search, from left to right, a map M is produced,

$$M : \texttt{index range of the chunk} \rightarrow \texttt{index subrange} \rightarrow (m, a).$$

Accomodating this map M is a sequence of integers, $\overrightarrow{I}$, of length $|M| - 1$ that is the modulii connecting every chunk to its immediate successor. Algorithm's backward search comes after, optimizing the solution of M. Optimizing entails shrink the size of M through merging every two contiguous elements (chunk info) into one. For a pair of chunks (c_i, c_{i+1}), algorithm usually merges the two chunks into one if the prefix of c_{i+1} shares the last (m, a) pair of c_i. Algorithm performs a frequency count of (m, a) pairs for the merged chunks $c_i \oplus c_{i+1}$. The backward search proceeds in the same direction, left to right, as the forward search. The runtime of MDR is polynomial with respect to the maximum absolute value of its reference sequence. Be aware that in the code implementation, the nuances posed by number sequences with negative values result in a non-null probability of the MDR not able to find a solution (M, $\overrightarrow{I}$) for the sequence. Sequences with a lot of -1 ternary relations between contiguous elements cause solution M to be of greater size. The implementation of MDR in *Seqbuild* is not conclusive to accomodate all number sequences of $\mathbb{Z}^*$. The runtime of MDR finding a solution (M, $\overrightarrow{I}$) for an arbitrary sequence S, of maximum absolute value m_x, is

$$O(|S|^2 m_x).$$

As mentioned before, MDR can serve as a seed after being initialized with an integer sequence. Every time the seed is reloaded with another integer or float, it outputs another sequence of equal length, calculated using its (M, $\overrightarrow{I}$) solution.

These two classes of structures, LCG and MDR, are the most basic, in terms of complexity, of *Seqbuild* primitive structures. As of this point in time, LCGv3 is the simplest number generator with a second-order seed, the ternary vectors (generated from its parameter generator) used to refit the LCG kernel's original

output. And these ternary vectors are generated by LCGv3's abstract generator parameter.

In regards to automation, MDR is not an inputless number generator since it requires an external force (human user) or additional code. MDR can only generate numbers when loaded with an integer or float and only in batches of length equal to its reference sequence. For clarification on the term "inputless number generator", see below definition.

Definition 3.1 (Inputless number generator)**.** A number generator G is an inputless number generator if, after its instantiation by an arbitrary seed fit for its type, it can generate numbers without any input inserted into it, such as a data structure external to it or a human user's effects, when called.

Structure **MDRGen** takes these variables as input: a reference MDR, an arbitrary number generator G, a boolean specifying whether to exclude negatives for MDR's (m, a) search, generative type $1 \oplus 2$, and five booleans (to be clarified) on the indexing scheme used by the structure in the case of generative type 1.

MDRGen requires default constraints to be able to generate numbers for timely results. Measures on average runtimes for every PRNG primitive structure are found in a table later in this paper. The default constraints are the following:

- maximum absolute multiple for MDR's (m, a) search.
- maximum absolute value in MDR's reference sequence S.
- size range for integer seed cycle.
- iteration range for every intger in the seed cycle.

The first generative scheme is relatively straightforward. MDRGen has an ordered queue that contains elements it will output. When this queue is empty, MDRGen calls G to output an integer. MDRGen loads this integer k, as a seed, into its reference MDR. The MDR then outputs a sequence of integers, using the same solution $(M, \overrightarrow{I})$, but with k as the first integer.

The second generative scheme uses these five boolean values, due to the scheme's use of a secondary container, C, before loading the finalized real numbers into the primary container, Q.

- R/C switch: can switch between row and column of matrix $C[i]$.
- index selector: can switch index of row and column in selecting a sequence from $C[i]$.
- seed selector: for two integer seeds $i_1, i_2 \in keys(C)$, can switch from i_1 to i_2 before $C[i]$ is exhausted.
- sequential seed selector (w.r.t. integer seed cycle): has to iterate through the integer seed cycle in sequential order.

- include integer seed: includes integer seed, w.r.t. integer seed cycle, in output.

C is a seed-to-sequence map,

`integer seed` $\rightarrow$ `matrix of sequences generated from reference MDR`.

The reference MDR is mutable in this scheme.

Note: The third and fourth boolean partially overlap with each other in effect of sequence selection.

Scheme 2 calculates a new C every time $|C| = 0$. To do so, an integer seed cycle, of some arbitrary length l_0 that satisfies the default size range, is generated from parameter G. Then scheme 2 sets a maximum sequence size (number of sequences) of $l_0 \times l_1$, l_1 an integer in the iteration range. Scheme then cyclically iterates through the integer seed cycle. For every integer i in this iteration, reference MDR is loaded with it and outputs a sequence S'. Sequence S' is stored into $C[i]$'s matrix as an additional row. Then MDRGen instantiates a secondary MDR, denoted D_2. D_2 finds a solution $(M_2, \overrightarrow{I}_2)$ such that there is a probability of $(M_2, \overrightarrow{I}_2) \neq (M, \overrightarrow{I})$. The principles for why this occurs revolve around the ternary vector associated with S', the multiplicative relations between elements of S', and the absolute (minumum,maximum) values of S. After the new map of C has been calculated, MDRGen draws out sequences from C into Q every time Q goes empty. This drawing is done according to the mentioned five boolean variables.

The concept of "exploding values" is important in certain cryptographic algorithms. Algorithms that iterate through some indeterminate span, such as for certain cases of LCGv3's ranged modulii adjustment, take a markedly longer runtime for anomalies where termination condition is not met.

Exploding values may result from algorithmic iteration through indeterminate spans, via a number found in an indeterminate span and that satisfies some arbitrary condition. This phenomenon, in itself, is problematic for runtime as well. In design of cryptographic data structures, exploding values can come about through unconstrained auxiliary generators, as in the case of MDRGen's parameter G. Exploding values could also come about through applying a function f, reserved for a range from $\mathbb{R}$, onto a value outside of the range. The $(M, \overrightarrow{I})$ solution of MDR is susceptible to the problem of exploding values during the course of re-fitting arbitrary sequence S' with a new MDR instance. Here is one attempt to define what an exploding value is.

Definition 3.2 (Exploding value)**.** A value from a number generator is "exploded" if it falls outside the range of (minumum,maximum) of the generator's prior output or alternatively (in the higher-order cases), the expectations defined by a human being on the generator output.

Exploding values, depending on the algorithm, could be set as the "new norm", such as a variable weight for a second-order seed, of the generative

process, thereby compounding the "explosion" of values until computing infrastructure is forced to terminate or crash. An example of this is found in the case of an MDR's unconstrained solution (M, $\overrightarrow{I}$) update.

3.3 The Operation Triangle Generator

The Operation Triangle Generator (**OTG**) is similar to MDRGen on the attribute of drawing output values from matrices. Instead of using a fitter structure, such as MDR, on a reference sequence, OTG uses a pair of operators, one for forward and the other for backward computation, such as (subtract,add) on a reference sequence S to produce an upper-right triangle matrix of dimension $(|S| - 1)^2$. OTG derives values from this upper-right triangle matrix's upper-right triangular half, and this derivation produces new upper-right triangle matrices for feedback to repeat the derivation process.

$$\begin{bmatrix} 00 & 01 & 02 & 03 & 04 \\ \emptyset & 11 & 12 & 13 & 14 \\ \emptyset & \emptyset & 22 & 23 & 24 \\ \emptyset & \emptyset & \emptyset & 33 & 34 \\ \emptyset & \emptyset & \emptyset & \emptyset & 44 \end{bmatrix}$$

Table 1: A 5×5 upper-right triangle.

For row 0 of the upper-right triangle, the $(0, i)$'th element is

$$S_0[i] = h_f(S[i+1], S[i].$$

Every row $[1, j)$ after this row follows the pattern of the (k, i)'th element being

$$S_k[i] = h_f(S_{k-1}[i+1], S_k[i]); k \leq i < j.$$

OTG uses two derivation methodologies to change this initial upper-right triangular matrix, $45 - 90$ split, OTG ues its parameter generator to decide the length x_j for the number of diagonal elements to use. The diagonal elements for the 45-section is

$$M[0,0], \ldots, M[x_j - 1, x_j - 1].$$

The 90-section is the subrow of $(x_j - 1)$:

$$M[x_j - 1, x_j], \ldots, M[x_j - 1, |M.cols| - 1].$$

A total of x_j derivative sequences can be calculated from this $45 - 90$ split (S_0, S_1). The first derivative sequence is simply the merged sequence $S_0[-1] \oplus S_1 = S^{(1)}$. For any derivative serquence after that operation, $S^{(q)}$ is comprised of numbers by this rule:

$$\overrightarrow{S}^{(q)}[i] = \begin{cases} S_0[-q] & \text{if } i = 0, \\ h_f(S^{(q)}[i-1], S^{(q-1)}[i-1]) & \text{otherwise.} \end{cases} \quad (2)$$

In the flip-derivation version, algorithm proceeds in one of two ways for M', the version of M flipped by either its 0-axis or 1-axis.

$$\begin{bmatrix} \emptyset & \emptyset & \emptyset & \emptyset & 44 \\ \emptyset & \emptyset & \emptyset & 33 & 34 \\ \emptyset & \emptyset & 22 & 23 & 24 \\ \emptyset & 11 & 12 & 13 & 14 \\ 00 & 01 & 02 & 03 & 04 \end{bmatrix}$$

Table 2: Matrix 1 flipped by 0-axis.

$$\begin{bmatrix} 04 & 03 & 02 & 01 & 00 \\ 14 & 13 & 12 & 11 & \emptyset \\ 24 & 23 & 22 & \emptyset & \emptyset \\ 34 & 33 & \emptyset & \emptyset & \emptyset \\ 44 & \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

Table 3: Matrix 1 flipped by 1-axis.

If M', an $m \times m$ matrix, is the 0-axis flip, M' is fed an integer seed s_i to produce one value from row 0, a row that has only one element, for a sequence $S^{(0)}$. For all q'th rows of M' after the first, follow this rule using sequence $S' = M'[-(q+1) :]$:

$$S^{(q)}[i] = \begin{cases} h_f(S^{(q-1)}[-1], S'[0]) & \text{if } i = 0, \\ h_f(S^{(q)}[-1], S'[i]) & \text{otherwise.} \end{cases} \tag{3}$$

A new upper-right triangular matrix M" is formed from these sequences, $S^{(0)}, \ldots, S^{(m-1)}$. The reason why an external integer seed is required for this derivation from M' to M" is to preserve the matrix dimension $m \times m$ (and the seed is an additional variable for the cryptanalyst's chagrin). A virtually identical procedure is ued for case of M' being a 1-axis flip, with a few accomodating changes in indexing. There is also a reproduction method in OTG's flip derivation. Using an integer seed s_n and the backward function h_b, OTG calculates a sequence S" of length $|S|$. This calculation is done by using the first row of M", R:

$$S"[i] = \begin{cases} h_b(s_n, R[0]) & \text{if i} = 0, \\ h_b(S"[i-1], R[i]) & \text{otherwise.} \end{cases} \tag{4}$$

This sequence S" is then used to produce a new upper-right triangle matrix.

OTG produces values using 45-90 split and flip-derivation. The matrices produced through these two methodologies, arbitrarily chosen through the course

of OTG generation with the use of its parameter generator or a default alternating $(01)^*$ have their upper-right triangular halves placed into the OTG's output queue. There is also the option of adding noise, using the parameter generator, to the numbers of these upper-right triangular halves.

The "randomness" of OTG's output very much rests on the quality of its parameter generator and its initial reference sequence S.

3.4 The Integer Decision Forest Generator

Integer Decision Forest Generator (**IDFG**) uses more complex calculations than OTG. As its name suggests, the generator uses a forest (an assortment of decision tree structures) to generate numbers. The form-fitting done by IDFG is on its reference sequence and is of two variants, both splitting the reference sequence into a decision tree. The first variant splits the sequence by factors, factors that exclusively belong to certain elements of S^*, a subsequence of S. The other groups elements of S^* into polynomial equations to be solved. IDFG requires that its reference sequence not contain the integers $\{0, 1, 2\}$. This requirement is obvious for factor-splitting, which is fairly straightforward once the decision tree is illustrated. In the other variant, algorithm first chooses two integers s_1, s_2 from S^*. Algorithm then finds coefficients for a polynomial P of arbitrary k order, such that $P(s_1) = P(s_2)$. Again, there are constraints on the polynomial degree and maximum x value per degree, in order to avoid the "exploding value" problem. For the remaining integers $s_i \in S^* - \{s_1, s_2\}$, find a polynomial P_i to satisfy $P_i(s_i) = P(s_1|s_2)$. This set of polynomials $\{P\} \bigcup \{P_i\}$ make up the splitting conditions for one node's conditional.

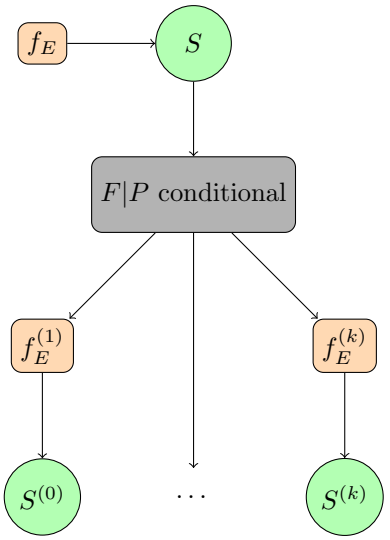

Figure 1: A decison tree with sequence S splits S into k nodes using its factor or polynomial conditional.

IDFG loads an arbitrary t number of trees, t specified by the user. Every tree is from a splitting process on a sequence S, initially an instantiation parameter of IDFG and also updated by IDFG's parameter generator. IDFG starts by generating one tree at timestamp 0. When the generation timestamp reaches a parameter counter checkpoint, IDFG instantiates another tree. This pattern repeats until IDFG holds a total of t trees. IDFG has an ordered queue Q that holds real numbers to be output. When Q is empty, IDForest chooses one of four schemes to generate the next batch of numbers, via the pair (S', T): sequential, iso-sequential, inflow, and splat.

In sequential generation, for every $s \in S$ (in the indexed-based ordering of S), s traverses T node by node until termination. At every node $V^{(i)}$ in this travel path, output a real number s_i from its entry function $f_E^{(i)}$, $f_E^{(i)}(s) = s_i$. Store this sequence of s_i numbers into Q. Iso-sequential is a "double" version of sequential generation. Another sequence S', of length equal to S, is used to travelling T. For every $s'_i \in S'$, find the path $P_T^{(i)} = traverse(T, s'_i)$. Then travel s_i of S along this path $P_T^{(i)}$ for the sequence S_I of numbers output from the path nodes' entry functions. Store S_I into Q. Iso-sequential is so named because the paths taken by elements of S are isomorphic to their index equivalents of S'. Inflow generation first partitions S into the subsets $S_1, \ldots, S_k$. For every S_i, load all elements of S_i into the root node of T. Numbers are produced from traversing all of these elements for one edge per one timestamp. Every element of S_i continues travelling T until they each reach termination condition. The numbers produced from application of the node entry functions are placed into Q. Inflow generation does not merely have to be a permutation of elements generated through sequential mode. There are certain other differences in cases where the node entry functions operate with mutable second-order seeds. In splat generation, the elements of S' are assigned nodes from T to produce the corresponding entry function output. There is no node-to-node traversal by this approach.

As with the previous generators described, whenever IDFG has to make a decision, such as choosing which scheme to generate the next batch of numbers, it uses its parameter generator to make the decision.

3.5 The R-Chainhead Generator

The R-Chainhead generator (**RCHG**) has an entropy accumulating mechanism. It uses a data structure called an R-Chainhead, the R standing for the prefix of "re" in words such as "response", "reconfigure", et cetera. The R-Chainhead is a structure extended from a graph that is a unidirectional path. The R-Chainhead is comprised of a variable number of nodes. R-Chainhead nodes have mutable qualities. Each of these nodes is associated with an entry function f_E, either a linear combination expression or a polynomial expression. An entry function accepts one real number as input. In the case of a linear combination, an additional function f_I (inner function) is required. Function f_I is associated with a vector V_R of degree equal to the linear combination of f_E, and is applied

first to real number input r to yield a vector fit for application with f_E. V_R, f_I, and f_E are mutable entities in RCHG. In short, linear combination nodes produce output from input q by

$$f_E(f_I(V_R, q)),$$

and polynomial nodes produce output by

$$f_E(q).$$

For an R-Chainhead of m nodes, an input value produces $(m + 1)$ values, one at each of the m nodes and the input value itself. Every time the R-Chainhead is called for values to be generated by RCHG, it outputs one value, that is the one from the end node of the R-Chainhead. The input for the j'th node of an R-Chainhead, $j > 0$, is the previous node's output. The other values produced, not from the end node, are loaded into a pre-cache for use with future calculations, such as new reference sequences (entropy accumulation). RCHG changes its R-Chainhead, with the use of its parameter generator that decides what timestamps during the ongoing generator activity to change the R-Chainhead.

There are a lot of little technicals in the code implementation of RCHG. This paper omits those details.

3.6 The Class of Decimal Generators

There are two main primitive structures used specifically to generate decimal values. They are the Q-Value Generator (**QG**) and Point Interpolation Differential (**PIDG**). QG is based around dividing integers of a reference integer sequence, while PIDG calculates differences between two ways to interpolate two-dimensional points (ordered by x-axis): classical Lagrangian polynomial interpolation and one that draws exponential or logarithmic curves connecting every two contiguous points.

3.7 Brief Mentions of Other Primitive Structures

Shadow Generator (**SG**) is one generator that is, in brief terms, a form-fitter of form-fitters over a reference pseudo-random number generator. It uses algorithms from Modular Decomposed Representative (MDR), Operation Triangle Generator OTG), ternary vectors, and polynomial/factor splitting of sequences to "shadow" over the original output from its reference PRNG. Search Space Iterating Network Operator Generator (**SSINOG**) is a generator that uses the output from a search space iterator (not described) and a network of input/output nodes, similar to the nodes of Integer Decision Forest Generator (IDFG), to calculate numbers to output.

These two primitive structures are relatively advanced machinery in comparison to the previous primitive structures mentioned. As of this point in time, code development of these two types of PRNGs has not been finished past adequate stages for quality verification.

3.8 The API Language: Comm Lang

Comm Lang, shorthand for Command Language, is a relatively simple language built on top of the program *Seqbuild*, written in Python. Users can initialize any number generator (primitive structure) out of those mentioned in this section. Users can also "merge" these primitive generators such as with an arithmetic operator or into a tree-like structure.

For instance, suppose there are these generators, $\{G_1, G_2, G_3, G_4\}$, a user initializes. User can merge G_1 and G_2 together with addition, $G_5 = G_1 + G_2$, so that every output of G_5 is $G_1() + G_2()$. Recall that all primitive structures in this program are inputless number generators. User can also merge $\{G_2, G_3, G_4\}$ into a tree-like generator G_5. Running G_5 produces the numbers

$$G_2(), G_3(), G_4(), \ldots, G_2(), G_3(), G_4(), \ldots.$$

Lastly, Comm Lang allows for the possibility of one-way encryption. A user can use a generator that they initialized in Comm Lang to "write over" a plaintext file of ASCII characters. This "writing over" is a simple addition between the generator's number and the ordinal value of the ASCII character.

4 Patterns in Algorithmic Cryptography

Some patterns are not meant to be easily predictable, as in cases of numerical-based cryptography. Attempts to understand the underlying pattern, some say this is the "nature" of their subject of study, come from all walks of life. Pattern discovery is one mark of intelligence. And on this mark, computerized machines excel over human beings, given the condition that they can be adequately trained through informatic feedback and that they are indeed trained for these pattern-based specialties.

The primitive structures in the PRNG system, *Seqbuild*, share some attributes (patterns of the source) worthy of explicit mention. One is that they are deterministic algorithms. The structures more advanced than the standard LCG require a parameter generator G. These parameter generators are used to make decisions. Decisions fall predominantly into these two categories: conditional decisions and sorting of sequences. The parameter generators can also be used to modify their host generator's second-order seed, thus effecting great differences in output quality, a simple example being one input x at timestamp t not yielding the same output as that at timestamp $t + j$. Hierarchical systems can be formed with one primitive structure serving as a parameter generator for another. As a result of this, the mutable seed hierarchy can become quite convoluted in the timing by which their host structures modify them according to their decision makers. If every primitive structure, with one or more seeds of second-order and higher, is considered an independent entity, then modification of their seeds by their programming is self-modification. However, the opaque aspect of this self-modification is that there is no objective answer on if these structures "break" any pattern they are already pre-disposed to, since

they are still deterministic algorithms, just not the kind of determinism easy for external agents to easily or completely comprehend. The self-modification of number generators' seeds of second-order and beyond occur because those generators' decision-makers decided to invoke the change, and those decision-makers themselves are deterministic. So it is important to clarify on the fact that a number generator's ability to self-modify does not necessarily imply the number generator is self-reflective. There is a difference in these two terms of prefix "self". Self-reflection by the entity necessitates the entity can recognize its own characteristics, internally (the primitive structures' variables such as its i'th order seeds) and externally (the generator's output values). Recognition, in the metric of full accuracy, would simply be complete and perfect replication so that nothing of the subject of recognition can be altered without there being distortion. Recognition of specific characteristics, on the other hand, produces different matters, and it requires the necessary programming and accurate application of that programming. Self-reflection is more effective than merely self-modification because self-modification only requires difference in expression (the immediately perceptible details), while self-reflection recognizes characteristics that would have otherwise been ignored in basic self-modifying processes. And this precise recognition is the grounds to calculate and execute change that is actually divergent. In the context of deterministic algorithms, more mathematical definition would benefit comprehension of these differences between modification,due to reflection, and pre-determined modification. Practically though, the quality and quantity of information known about the adversary's generator is the determinant in ruining the adversary's security goals of a truly random generator, and this determinant implies the ability to predict the timing of the generator's self-modification to what due to what (the third theory of randomness, mentioned at the beginning of this paper).

These primitive structures, in light of their "representative power" given their initializing parameters, can be considered **form-fitters**. MDRGen is exemplar of form-fitting. It uses a Modular Decomposed Representative to find a solution of (multiple,additive) pairs, along with a sequence of modulii for after every chunk of a reference sequence, so that the solution is meant to replicate the sequence. IDTG can also be considered form-fitting: it splits any reference sequence according to the sequence integers' qualities, namely, unique factors and equal polynomial outputs. Form-fitting methodologies practically find the "kernel" of an arbitrary numerical sequence. After the calculation of this "kernel", mutable seeds are applied onto or into this kernel to yield values that follow the kernel's specified quality of said kernel's form.

4.1 Abstract of the Guided Generator

There was an attempt in the development of *Seqbuild* to implement a self-reflective primitive structure. The structure is called **AGV2GuidedGen**, and is also deterministic. The "guidance" the generator takes is from a structure called an Abstract Pseudo-Random Number Generator Gauge. The gauge is able to perform metrological calculations on the output values from the struc-

ture. These metrics include output subsequences' "coverage" of arbitrarily-set number ranges, those subsequences' unidirectional weighted point distances (values in the range $[0, 1]$), and categorization of those subsequences' according to the previous two metrics. When this generator reaches some condition during its output of real numbers, the generator changes its second-order seed so that the next subsequence of arbitrary length does not match (or is not similar) in measurement with records of the previous subsequences' measures. As of this point in time, the generator is still developmentally incomplete.

5 Runtime Testing of Sample Generators

Description and runtime of selected generators is displayed in this section. Results from quality testing of these generators is not included. The computing hardware these tests were run on is Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 1992 Mhz, 4 Cores, 8 Logical Processors. Programming language is Python 3. Every one of these generators was run for 10^5 numbers.

- 1. Type: LCG

  parameters: $v_0 = 17, v_1 = 200, v_2 = 421, v_3 = -900,$

  $$f(x_{t+1}) = (v_1 x_t + v_2) \mod v_3.$$

  runtime: ≈ 0.053 seconds.

- 2. Type: LCG

  parameters: $v_0 = 26, v_1 = 202, v_2 = 425, v_3 = 900,$

  $$f(x_{t+1}) = (v_1 x_t + v_2) \mod v_3.$$

  runtime: ≈ 0.054 seconds.

- 3. Type: LCG

  parameters: $v_0 = 26, v_1 = 2020, v_2 = 4250, v_3 = 9000,$

  $$f(x_{t+1}) = (v_1 x_t + v_2) \mod v_3.$$

  runtime: ≈ 0.051 seconds.

- 4. Type: LCGv3

  parameters:

  start=500,multiple=23,additive=45,

  parameter generator=#1,

  ranged modulo of pre-image= $[0, 100000]$,

  ranged modulo of image= $[0, 100000]$,

exclude zero from ternary vector=False,
reflective modification=True,
runtime: ≈ 0.954 seconds.

- 5. Type: LCGv3
  parameters:
  start=500,multiple=23,additive=45,
  parameter generator=#3,
  ranged modulo of pre-image= $[0, 1000]$,
  ranged modulo of image= $[0, 1000]$,
  exclude zero from ternary vector=False,
  reflective modification=True,
  runtime: ≈ 0.951 seconds.

- 6. Type: LCGv3
  parameters:
  start=500,multiple=23,additive=45,
  parameter generator=#3,
  ranged modulo of pre-image= $[0, 1000000]$,
  ranged modulo of image= $[0, 1000000]$,
  exclude zero from ternary vector=False,
  reflective modification=True,
  runtime: ≈ 0.932 seconds.

- 7. Type: MDRGen
  parameters:
  starting reference sequence: 20 numbers from #4 (not shown),
  parameter generator=#2,
  exclude negative= False,
  generative type= 1,
  runtime: ≈ 0.077 seconds.

- 8. Type: MDRGen
  parameters:
  starting reference sequence: same as that for #7,
  parameter generator=#2,
  exclude negative= False,

generative type= 2,

R/C switch= False,

index selector= False,

seed selector= False,

sequential integer cycle selector= True,

runtime: ≈ 436.092 seconds.

- 9. Type: MDRGen

  parameters:

  starting reference sequence: same as that for #7,

  parameter generator=#2,

  exclude negative= False,

  generative type= 2,

  R/C switch= False,

  index selector= False,

  seed selector= False,

  sequential integer cycle selector= True,

  runtime: ≈ 436.092 seconds.

- 10. Type: OTG

  parameters:

  integer seed= 27.

  starting upper-right triangle matrix: 10×10, formed from 11 values of #2.

  parameter generator=#2.

  forward function= +,

  backward function= −,

  generative type= 1,

  add noise = False

  runtime: ≈ 8.778 seconds.

- 11. Type: OTG

  parameters:

  integer seed= 27.

  starting upper-right triangle matrix: 21×21, formed from 22 values of #2.

  parameter generator=#2.

forward function= +,

backward function= −,

generative type= 2,

add noise = False

runtime: ≈ 0.446 seconds.

- 12. Type: OTG

  parameters:

  integer seed= 27.

  starting upper-right triangle matrix: 10×10, formed from 11 values of #2.

  parameter generator=#2.

  forward function= +,

  backward function= −,

  generative type= 1,

  add noise = False

  runtime: ≈ 14.164 seconds.

- 13. Type: OTG

  parameters:

  integer seed= 27.

  starting upper-right triangle matrix: 21×21, formed from 22 values of #2.

  parameter generator=#2.

  forward function= +,

  backward function= −,

  generative type= 2,

  add noise = False

  runtime: ≈ 0.300 seconds.

- 14. Type: IDFG

  parameters: unspecified,

  runtime: ≈ 31.666 seconds.

- 15. Type: IDFG

  parameters: unspecified,

  runtime: ≈ 22.885 seconds.

- 16. Type: QV

  parameters:

  reference sequence

  $$\overrightarrow{S} = \{4567, 234, 15564, 1134, 51, 78, 95547, 46892, 1416, 747345223\}$$

  runtime: ≈ 1.105 seconds.

- 17. Type: QV

  parameters:

  reference sequence

  $$\overrightarrow{S} = \{1, 0, 1, 0, 10, 0, 1, 0, 10, 1, 0\}$$

  runtime: ≈ 5.405 seconds.

- 18. Type: PIDG

  parameters:

  reference two-dimensional sequence: 15 points $\in [0, 9007)$

  runtime: ≈ 8.694 seconds.

- 19. Type: PIDG

  parameters:

  reference two-dimensional sequence: 30 points $\in [0, 9007)$

  runtime: ≈ 8.55 seconds.

Most of the generators finish the task of outputting 10^5 values in under 10 seconds. The exception is MDRGen. MDRGen's programming, at this point in time, makes frequent changes in its reference sequence. Every reference sequence needs to be refitted by a Modular Decomposed Representative. The MDR is a kind of machine-learning algorithm on sequences, and MDRGen would surely need to be revised for demands of timely number generation.

6 Challenges in Decryption

In the beginning of this paper, RSA encryption was categorized as a solid-state encryption methodology. This categorization considers how an adversary attempts to solve RSA. An adversary without the private key would know the RSA method, that revolving around the problem of finding the two primes p_0 and p_1 that make up $n_0 = p_0 p_1$. The adversary would iterate through their hypothesized search spaces in $\mathbb{Z}$. Their task, however great the integer n_0 is, has been narrowed down towards this specific prime multiplication problem.

This paper argues that *Seqbuild* is a fluid-state encryption methodology. Suppose that an adversary wants to know the encryption sequence of real numbers $S_q \in \mathbb{R}^*$ for a plaintext sequence $\overrightarrow{M}$, $|\overrightarrow{M}| = |S_q|$. The adversary would have to be able to find the set of primitive structures $\{P\}$, those structures' first-order seeds $\{Q\}$, and the connections between those primitive structures $\{C\}$; $|\{P\}| = |\{Q\}|, |\{C\}| < |\{P\}|$. On the one hand, brute-forcing through the search space to find these three variables is exponential, $|\{P\}|$ serving as one variable exponent. But there is no guarantee that the number generator $G = init(\{P\}, \{Q\}, \{C\})$ is bijective. This is due to the coincidences of equal output values from different generators, and these coincidences are caused in part by the form-fitting mechanisms of algorithms such as Modular Decomposed Representative. The problem of *Seqbuild* generators' non-bijectivity has not been adequately explored, so there are no hard numbers provided on this area. Suppose that the practical generators of *Seqbuild* are not bijective through the infinite course of their number generation. However, a subsequence S from an arbitrary one of these practical generators, G_1, is equal to that of another generator G_2. Adversary can decrypt $\overrightarrow{M'} = \overrightarrow{M} + S$, by knowing S from G_1 or G_2 at their output timestamps t_1, t_2, respectively. But for the next plaintext message $\overrightarrow{M}_2$, the sequence S_1 is from G_1 and the sequence S_2 from G_2; $|S_1| = |S_2|$ and $S_1[i] \neq S_2[i] \forall 0 \leq i \leq |S_1| - 1$. If the adversary does not know the contents of M_2, then they cannot know the actual sequence S_1 if they do not know G_1 but know G_2. Their decryption attempt results in $\overrightarrow{M}'_2 - S_2 \neq \overrightarrow{M}_2$. The adjective of "practical" should be taken to be tied to the mathematics subfield of pathological functions, a famous representative being the Weirstrauss function.[Joh17]

The issue of *temporal equality* between two or more generators leads to a spottiness in encryption security such that the adversary will have partial knowledge of the plaintext communication. Temporal equality has its set of problems that effective adversary and effective defender will attempt to "flip" back onto each other, in cases where the adversary will always be able to calculate an alternative generator that is temporally equal to the defender's current generator in use. In these scenarios, game theoretics of information and physical systems are quite pivotal to continue meaningful discussion.

Categorization can prove to be folly. *Seqbuild* is categorized as fluid-state encryption, in comparison to RSA, due in great part to what has been discussed so far in this section. However, there are ways to reduce the output from *Seqbuild*'s primitive structures into groups, analogically subspaces of $\mathbb{Z}$ in RSA.

Sampling of first-order seeds is strongly advised for secure encryption. A noteworthy problem is the use of reference sequences S that follow patterns simple enough to be representable by one-liner expressions. Patterns such as $F(x_i) = mx_{i-1}$ (multiplicative) and multiplicative congruence, for example, are strongly discouraged for continuity of security. Randomness testing, such as through NIST Randomness Test Suite.[Ae10] This test suite and viable equivalents should be used in helping to determine the subspaces of first-order seeds most satisfying to the underlying theory of randomness. Schneier, an applied

cryptographer and inventor of the Fortuna algorithm, has his practices to share on ideologies of randomness.[NK10] In his description of the Fortuna algorithm, the algorithm accepts "random" streams of information into its computing storage. The algorithm uses this "random" contribution to alter its pre-defined numerical processes. This is the defining characteristic of his algorithm that allows for more secure PRNGs. I personally argue that, by the third theory of randomness, the Fortuna algorithm is still deterministic, albeit the distributed computing invoked for the number generating algorithm to accept external data, this data called "random" out of the lack of knowledge of it, is convincably random. Fortuna is practically an opening up of one or more computing devices (the Fortuna hosts) to other computing devices' information streams. And the Fortuna hosts do not possess or control the qualities found in these information streams. It is not foolhardy to argue that Fortuna is still not truly random in the context of perfect information over the network of hosts and others. Out of the lack of control to calculate the next number or decision without the input of these other computing devices, Fortuna is a non-deterministic process strictly with respect to its host devices. Determinism is, in my own words, "perfect control for one direction and for all", a statement that carries presumptions of the coupling between control and knowledge, one that is not very geometrically precise, and one that omits the specificities comprising the whole through simple expression.

I hypothesize that Seqbuild's one-way encryption scheme (SB-OWE), even when given the constraint of the key generator being comprised of no more than, say, twelve primitive structures of a diverse assortment, is at least as hard as RSA encryption. This paper does not provide a reduction proof of computability from RSA to SB-OWE.

There are some other details about SB-OWE worthy of mention. A glaringly obvious one is that the size of the ciphertext is exactly equal to the size of the plaintext. An adversary, with knowledge and resources of the natural language used in this communication, can already narrow down many of the possibilities of the plaintext. The cryptographic technique of padding would help alleviate this deficit. Suppose arbitrary generator G, used in SB-OWE, is used to encrypt some number of plaintexts. The adversary is able to acquire those plaintexts. Then generator G should be completely deleted from use or modified, past a point of recognition, to another generator G_2. Suppose for a (plaintext,ciphertext) $(\overrightarrow{M}, \overrightarrow{M'})$, adversary could determine the encrypting sequence S_q. But that is only if the ciphertext can be matched back to its specific plaintext. *Seqbuild*'s primitive generators could generate S_q, $S_q + \overrightarrow{M}$, or $S_q - \overrightarrow{M}$ as the encrypting sequence. The adversary, after acknowledging this possibility, would be at a point of uncertainty in determining the encrypting sequence, without any more adequate information on plaintext M. At this point in time, there has not been much work in the project for easily finding a generator G' that outputs encrypting sequence $S_q(+ \oplus -)\overrightarrow{M}$, given a generator that outputs S_q. The one-layer mask provided by SB-OWE is deceptively flimsy, by a certain choice of words, and also weak from poor choice.

Sources

[Mar68] George Marsaglia. "Random Numbers Fall Mainly in the Planes". In: *Mathematics Research Laboratory, Boeing Scientific Research Laboratories, Seattle, Washington* (1968).

[MT02] George Marsaglia and Wai Wan Tsang. "Some difficult-to-pass tests of randomness". In: *Journal of Statistical Software* (2002).

[Ae10] Juan Soto Andrew Rukhin and etc. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. National Institute of Standards, Technology Administration, US Department of Commerce, 2010.

[Gol10] Oded Goldreich. *Primer on Pseudorandom Generators*. Department of Computer Science and Applied Mathematics,Weizmann Institute of Science, Rehovot, Israel., 2010.

[NK10] Bruce Schneier Niels Ferguson and Tadayoshi Kohno. *Cryptography Engineering*. Wiley Publishing Inc., 2010.

[Hey17] Howard Heys. "A Tutorial on Linear and Differential Cryptanalysis". PhD thesis. Memorial University of Newfoundland, 2017.

[ZM23] Ruowen Liu Zhengping Jay Luo and Aarav Mehta. "Understanding the RSA algorithm". PhD thesis. Rider University, 2023.

[Joh17] Jon Johnsen. "Simple Proofs of Nowhere-Differentiability For Weierstrass's Function and Cases of Slow Growth". In: *Arxiv* (October 2017).