Secure and Computationally Efficient Cryptographic Primitive Based on Cellular Automaton

Rade Vuckovac
School of Information and Communication Technology
Griffith University
Parklands Drive
Southport, QLD 4222, Australia

The cellular automaton generator (CAG), a random number generator based on the one-dimensional cellular automaton (CA), is presented. Three procedures of secure implementation using the CAG are proposed and discussed. Implementations are very efficient in a wide range of hardware and software scenarios. That includes the advanced application of internet of things (IoT) and cyber-physical systems, which are both needed for computationally efficient cryptographic primitives. Furthermore, the proposed primitive is inherently resistant against the side-channel attack (SCA), where many currently available ciphers, such as the advanced encryption standard (AES), require additional hardware or software effort to prevent the SCA line of attack.

Keywords: cellular automata; cryptographic primitive; side-channel attack; stream cipher performance

1. Introduction

Cellular automaton (CA) cryptographic use is limited. One significant reason is performance. A survey of CA stream ciphers [1] shows that encrypting one megabyte of data requires five seconds at best. One exception is my array generator (MAG) [2], which is a cellular automaton generator (CAG) predecessor. It is a one-dimensional CA. It belongs to a class of three complexity classification schemes [3, p. 12]. That means nearly every initial state evolves in a pseudorandom or chaotic fashion. There are two major attributes that make MAG exceptional:

- MAG is invariant to the cell size. Both 32-bit and 64-bit cells are investigated, and they show the same behavior. That fact has a huge impact on performance.
- The MAG update rule is not entirely Boolean, and SAT solvers tools, which are generally better than brute force [1], could not be applied.

https://doi.org/10.25088/ComplexSystems.28.4.457

Apart from that, there are many more important reasons for revisiting MAG and for building CAG on MAG foundations:

- MAG was the main building block for stream cipher entry (eSTREAM and ECRYPT Stream Cipher Project [2]). That entry did not progress to the second round because of the available analysis at that time. On the second round of decision making, the analysis consisted of two attacks [4, 5]. Other attacks [6, 7] are variants of the previous two. More analyses were published [8] after round two. Those cryptanalyses contest attacks from [4, 6]. In the same analysis, it was implied that the second type of attack [5, 7] is avoidable using one minor alteration proposed in [9]. According to published analysis, at least one MAG variant remains secure.
- The second reason for the renewed interest in MAG is excellent software performance; see testing published on the eSTREAM webpage [10]. For a short set of results, see Table 1. It compares MAG with some well-known stream cipher algorithms.

Primitive	Stream (Cycles per Byte)
MAG-v3	2.20
TRIVIUM	4.14
Salsa20	7.64
RC4	14.52
AES-CTR	18.51

Table 1. An extract of eSTREAM software performance table [10].

- Another reason is MAG compactness. While the advanced encryption standard (AES) is widely used in symmetrical encryption, the emergence of IoT (internet of things), with constrained computation power, limits AES usability in that area. Therefore, lightweight symmetrical encryption schemes are sought. The MAG hardware footprint is 512 bytes of memory for the automaton state, plus a couple of variables. Operational cost is a CA updating rule consisting of one conditional branching, a couple of exclusive or logical operations, one one's complement and one addition (five basic operations). That should match an extensive range of IoT hardware with limited capabilities. See Listing A.1 for details.
- MAG, like other cellular automata (CAs), has inherent resistance to side-channel attacks (SCA), where AES and many other block ciphers implementations in that regard are relatively complex [11]. The lightweight cipher designs using S-boxes are affected by SCA as well [12].

Please note that the predecessor of our cryptographic primitive CAG, namely MAG, and its design choices for various parameters were never discussed or published before. Thus, in this paper, these issues will be addressed as well.

The rest of the paper is organized as follows: Section 2 introduces one-dimensional CAs and then defines the CAG proposition. It also shows where the CAG improves over MAG, making the CAG proposition even simpler. Appendix A includes C language general implementation. Section 2 shows three different ways to implement the proposed CAG securely. Section 3 deals with an analysis of MAG known attacks and how they might impact the CAG proposal. Section 4 discusses the CAG and its potential use in a variety of applications, SCA and input flexibility.

2. Cellular Automaton Generator

The CAG is a one-dimensional CA. The concept of the CA was first discovered in the 1940s by Ulam and von Neumann. The CA is used as a modeling tool in various scientific fields: computer and complexity science, mathematics, physics and biology. Wolfram was the first to propose the use of the CA (rule 30) [13] in cryptography [14].

Figure 1 showing Wolfram's rule 30 is used to explain the general working of a one-dimensional CA. This particular example uses cells with two possible states (black and white). The one-dimensional CA initial state is a row. In the example, it is the binary string (row 1):

0 000 000 000 000 001 000 000 000 000 000

Row 2 is derived from row 1 and so on, with final row 16:

1 101 111 001 101 001 011 111 001 111 111

Rules to determine an update of the cell are shown as eight cases. Each case shows one combination of three cells on the top and the derivative cell on the bottom. A new cell looks at the three cells from the row above (positioned above right, immediately above and above left). Depending on the configuration, one of the cases is applied. For example, a cell from row 2, column 14 is derived by case 8; a cell from row 2, column 15 is derived by case 7; a cell from row 2, column 16 is derived by case 6, and so on. Edge cells do not have above left or above right cells to choose a case. In that situation, the first or the last cell from the previous column is used for the ruling. For example, the left-edge cell from row 16, column 1 looks at cells from row 15, columns 31, 1, 2 and uses case 7 to create the cell. The right-edge cell from row 16, column 31 uses the cells from row 15, columns 30, 31, 1 for lookup and case 4 for cell determination. Derived rows (rows 2, 3, …) are outputs of the CA.

If a random stream of bits is required, column 16 could be used:

1 101 110 011 000 101…

A stream generated in this fashion passes many randomness statistical tests, and it is used as a random number generator in the Wolfram Language.

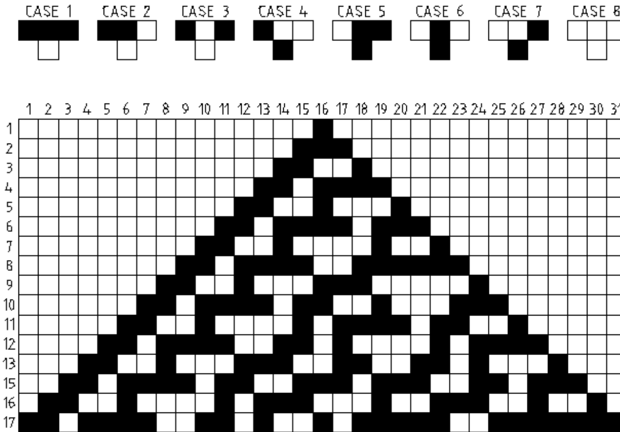

Figure 1. Rule 30 one-dimensional CA example.

The CAG is also a one-dimensional CA, but it differs from the example in the following ways:

- Cells are multi-bit words (32 or 64 bits in size).
- The update of CAG cells is serial, left to right, because the rule needs the outcome of the previous cell update (carry).
- The CAG rule also appears to be invariant to cell size because it shows the same random behavior for 32- and 64-bit size cells.

The CAG CA is governed by the CA rule and the state of neighboring cells. A row of cells, in a CAG case array of elements, is updated from left to right. One evolution cycle is when all elements in the array are updated. The next generation is another evolution cycle, and so on. The original and modified parameters are shown in Table 2.

MAG	CAG
$a = 127$	$a = 128$
$b = 32$	$b = 32$
c calculated	$c = 987\,654\,321$
$d = 0x11\,111\,111$ (HEX)	$d = 010\,101\ldots$ (binary)
$e = 1.5a$	$e = 4a$

Table 2. MAG/CAG parameters.

CAG parameters definitions:

- The number of cells is $a = 128$. That choice forces 2^{128} possible execution paths during one evolution cycle and assumes at least 128-bit–level security if the stream is used as a basis for the cipher.
- The cell size is $b = 32$ bits. The CAG CA rule appears to be invariant concerning the cell size. $b = 64$ bits is used and tested for randomness [15] and there is no change, although the performance doubles because the same generating cost produces double the stream.
- Instead of taking the carry value from the array element in the original version, it is initialized as $c = 987\,654\,321$ (decimal). The value of 987 654 321 was chosen, and there is no special meaning behind this choice.
- The constant d is now initialized as $d = 01\,010\,101\ldots$ (32 bits binary). Again there is nothing special in the constant value. In MAG, d was described as an arbitrary value (which it is). Related to this, an initialization attack [4, 6] where $d = 0$ was proposed. It eliminates any adding procedure, which simplifies the whole process significantly. The analysis [8] showed that any nonzero value of d is sufficient to prevent initialization attacks.
- In a modified version, the mixing period e is four evolutions $e = 4 * a = 512$. It assures proper mixing because the original one and a half evolutions occasionally produced biases in the first couple of generated rows. The same could be observed in the rule 30 case (Figure 1), where the first several rows still retain some patterns.
- The seed f is any binary string equal to or smaller than a row of cells $f_{\text{size}} \leq a * b$ and $f = k + s + \ldots$, meaning that the key k, the salt s and … (IV, pepper and so on) are concatenated to form seed f.

CAG operation is divided into initialization and update.

CAG initialization: Originally, the array of 128 elements, 32 bits wide is initialized to 0. The seed f is repeatedly concatenated until the resulting concatenation is equal to or greater than the array in size. The MAG array's first 127 elements are the initial row, and remaining elements become the carry c. For example, if the seed is $f =$ seed and the array is 10 cells (one byte each), the resulting initial array will be:

seedseedse

In the CAG modified version, the seed f is copied to the zero initialized array and the carry is given as an initial value $c = 987\,654\,321$. The pattern, with $f =$ seed and array of 10 bytes, looks like:

seed000 000

CAG update: When initialized, the rows are created by updating cells from left to right. The edge cases (A_i happens to be on the array end)

are handled as rule 30 edge-cell cases. The rule elements are shown in Figure 2. The cell update consists of three steps:

- The first step is to create a new state of carry c':

$$c' = \begin{cases} c \oplus A_{i+1}, & \text{if } A_{i+2} > A_{i+3} \\ c \oplus \overline{A}_{i+1}, & \text{otherwise.} \end{cases} \tag{1}$$

  Carry c' is updated by $\oplus$ (exclusive or) with previous value of c and one state of the first element to the right A_{i+1}, depending on the relation between the other two cells on the right (A_{i+2}, A_{i+3}). The states of A_{i+1} are current value (A_{i+1}) or its one complement ($\overline{A}_{i+1}$). In one evolution cycle (whole array is updated), each cell is changed once and the carry is calculated for every cell transformation. Note that the first cell update uses the initial value of carry $c = 987\,654\,321$.

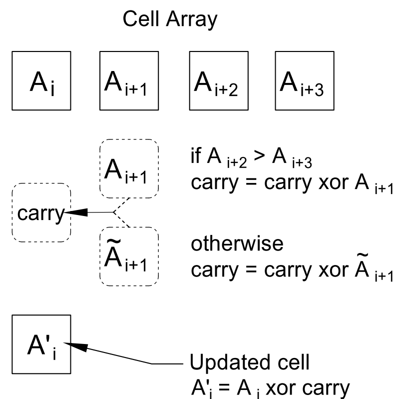

Figure 2. CAG cell update rule.

- The second step is the actual change of element A_i to A_i':

$$A_i' = A_i \oplus c'. \tag{2}$$

- The third step is updating the current c' value for the next cell transformation:

$$c' = c' + d. \tag{3}$$

Knowledge of the array state renders the CAG algorithm cryptographically unsound. For securing the stream from CAG, three strategies for concealing the CA state are proposed and discussed.

2.1 Reducing Output (cag-v1)

One of the concealing methods was already used by Wolfram on his CA rule 30 [16, Section 10.10]. For example, column 16 from

Figure 1

1 101 110 011 000 101…

is transformed by taking bits 1, 3, 5, 7, 9, … and producing a secure stream

10101000….

A similar approach was used in the MAG eSTREAM proposal. A small change in operation is added to achieve a secure stream property. Instead of copying every updated cell to the stream, only the first byte of the cell is added to the secure stream *s* (now array of bytes).

For example, Figure 3 shows updated cells as a stream of bytes where each pixel represents a byte and four bytes are an updated cell. To make a secure stream, every fifth byte (every fifth pixel from left to right) is taken and fed to the secure stream. That is bytes:

1, 5, 9, 13, 17, ….

Consequently $b_1 b_5 b_9 \ldots$ bytes create a secure stream. This approach was found insecure, and details are shown in Section 3.

Again the small change in pattern avoids weaknesses from the previous design. If the series of bytes taken is changed to: from first cell, first byte; second cell, second byte; third cell, third byte; fourth cell, fourth byte; fifth cell, first byte…

1, 6, 11, 16, 17, 22, 27, 32….

The stream of bytes $b_1 b_6 b_{11} b_{16} b_{17} b_{22} b_{27} b_{32} \ldots$ from Figure 3 will form a secure stream.

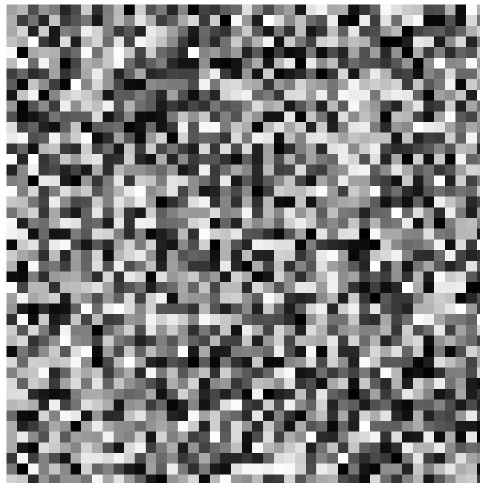

Figure 3. CAG evolution history; one pixel is one byte (256 grayscales).

2.2 Combining Streams (cag-v2)

One way of making a stream secure is to combine two or more streams. For example, linear-feedback shift register (LFSR) outputs were combined to make a shrinking generator (planned to be used as a stream cipher [17]). The shrinking generator uses two streams: one is the source, and the other is used to decide which bits of the source stream are output.

In the CAG case, the idea is to apply exclusive or between two generated streams (stream α and β) to produce secure stream s:

$$s_i = \alpha_i \oplus \beta_i. \tag{4}$$

One implementation attempt at the combined MAG approach was published in [15]. It includes source code. The easiest way to implement combined CAG is to initialize streams separately. Table 3 contains initial parameters. The seed f for each stream now includes corresponding IV. For example, the seed for α is $f_\alpha = k + s + \text{IV}$, where k is the key, s is the salt, IV is the initialization vector from Table 3, and + is the concatenation of strings.

CAG Stream	Stream IV
α stream	1234567890/3
β stream	9876543210/3
γ stream	…

Table 3. Combined stream parameters.

2.3 Stream Masking (cag-v3)

The idea with masking is to combine (xor) CAG output with a secret string. The original idea was to use a key as the secret string, but Bernstein and Lange noted that the same attacks [5, 7] apply for that proposal as well. Alternatively, the secret could be sourced from the execution path history. The branching from the previous evolution can generate string m. In the case of a 128-element array, there is 128 branching in one evolution cycle, making a 128-bit string m. Bits of m are determined by branching; the *if* branch will concatenate 0 and the *else* branch 1 to the mask m. From m the four 32-bit element mask array M is created:

$$M = [B_1; B_2; B_3; B_4]. \tag{5}$$

The secure stream s is now obtained by exclusive or result from CAG output cells (A) and mask array of four elements (M):

$$s_i = A_{i\%a} \oplus B_{i\%4}. \tag{6}$$

The new mask M is calculated for every evolution cycle and is used as a mask for the next cycle.

3. Cellular Automaton Generator Analysis

cag-v1: The eSTREAM proposal to strengthen the CA MAG was to output just the first byte from every updated cell. It did not work. The CA can continue updating on just that part of the cell without information from the hidden part:

- Since the first bytes of $A_{i\%a}$ and $A'_{i\%a}$ are known, the first byte of c' is known as well because:

$$A'_{i\%a} = A_{i\%a} \oplus c'. \tag{7}$$

- Knowing the first byte of c, predicting the next unknown value of the carry c' takes guessing the branching outcome. That guess is even easier because of knowledge of the first byte, which is compared.

This kind of attack is detailed in [5, 7]. For avoiding this line of attack, Fare proposed two amendments in [9]. The idea was to alternate extraction points because the original idea did not hide the evolution of the first-byte cell. Table 4 shows various extraction patterns for the proposed MAG secure stream. The first amendment (second row, Table 4) is broken as well. The one gap between exposed bytes of the cell did not prevent the same attack, although the guessing cost was increased [8]. The second amendment (third row, Table 4) does have three gaps between visible bytes and is still resisting analysis.

Series of Bytes Used in the Stream	Status
1, 5, 9, 13, 17, ...	×
1, 2, 7, 8, 9, 10, 15, 16...	×
1, 6, 11, 16, 17, 22, 27, 32...	✓

Table 4. CAG secure stream extraction patterns.

cag-v2: The result from exclusive or of two CAG streams is secure stream s. The relevant relations of knowns and unknowns are shown below. Stream s is known and streams α and β and carry c are unknown:

$$\text{row-wise: } s_i = \alpha_i \oplus \beta_i; \;\; s_{i+1} = \alpha_{i+1} \oplus \beta_{i+1}; \; \ldots$$

$$\text{evolution-wise: } c'_{\alpha\, i} \oplus c'_{\beta\, i} = s_i \oplus s'_i.$$

Both relations are used to attack CAG Section 2.1 [5, 7]. For strengthening the two-stream variant, additional streams could be included. The combination of three CAG streams producing a secure stream might look like:

$$s_i = \alpha_i \oplus \beta_i \oplus \gamma_i. \tag{8}$$

https://doi.org/10.25088/ComplexSystems.28.4.457

cag-v3: The relations with known s only are shown below concerning the CAG attacks [5, 7]:

$$\text{row-wise: } s_i \oplus s_{i+4} = A_i \oplus B_i \oplus A_{i+4} \oplus B_i = A_i \oplus A_{i+4}$$

$$\text{evolution-wise: } c'_i = s_i \oplus s'_i = A_i \oplus B_i \oplus A'_i \oplus B'_i.$$

4. Cellular Automaton Generator Advantages

In this section, a few vital CAG features are discussed. One feature is applicability concerning various hardware platforms. There is also CAG resistance to the SCA and CAG flexibility to inputs other than a key.

4.1 Cellular Automaton Generator Implementation and Performance

Three CAG cipher variants are presented:

- cag-v1 from Section 2.1 is the simplest. For producing one byte, a set of operations from Table 5 is needed plus an overhead of extracting the byte from the cell A' and array navigation. The array containing CAG cells is only 512 bytes in size and with the mentioned set of operations, lowers the entry hardware requirements bar significantly. That includes a wide array of IoT implementations. The cag-v1 performance, the same efficiency as mag-v1, is very comparable with AES; see Table 6.

CAG Single Cell Update
$c' = \begin{cases} c \oplus A_{(i+1)}, & \text{if } A_{(i+2)} > A_{(i+3)} \\ c \oplus \overline{A}_{(i+1)}, & \text{otherwise} \end{cases}$
$A'_i = A_i \oplus c'$
$c' = c' + d$

Table 5. The operations needed for a one-step cell update.

Crypto Primitive	Stream (Cycles per Byte)
RC4	14.52
AES-CTR	18.51
MAG-v1	20.43

Table 6. An extract of eSTREAM software performance table [10].

- cag-v2 from Section 2.2 is a little bit more complex. On the other hand, efficiency is improved. By roughly doubling effort, output increases four times. That is four bytes per two CA steps compared to one byte

per one CA step (Section 2.1). There are other ways to improve performance. One way relies on the fact that combining streams could be created in parallel. By that technique, performance is four bytes per one CA step (two steps in parallel). This technique also enables adding streams if needed without affecting performance. Another efficiency approach could be increasing the size of the CA cell from 32 to 64 bits, producing eight bytes per step. For details, see Table 7.

- cag-v3 from Section 2.3 is a notch more complex than previous variants. Developing a mask for each evolution step is the reason. This approach also improves performance concerning cag-v1. That improvement does not need parallelism. Although cag-v3 is relatively more complex, it is still significantly simpler than AES from a hardware and software point of view, delivering better performance. Table 1 shows mag-v3 (the same as cag-v3) 64-bit implementation versus AES and other primitives. Table 7 shows performances between various CAG variants, where one step from Table 5 produces 1 to 8 bytes toward a secure stream, depending on the variant used.

MAG Variants	Performance
cag-v1	1 byte/1 step
cag-v2	2 bytes/1 step
cag-v2 parallel	4 bytes/1 step
cag-v2 parallel 64 bit	8 bytes/1 step
cag-v3	4 bytes/1 step
cag-v3 64 bit	8 bytes/1 step

Table 7. Various performance details.

4.2 Side Channel Resistance

When discussing the SCA on AES, the quite often-cited work [18] cannot be avoided. There is an assertion mentioned in the abstract that attacks come from an AES design flaw rather than AES implementation. Furthermore, this report is also a call for research into functions with constant-time execution. The CAG as CA appears to have the rule that runs in constant time, but some attention is still needed. When implementing the CAG in SCA-resistant mode, the algorithm branching structure should be addressed. There are two issues:

- The first one is to compare secret cells in constant time. By comparing operations, some properties of the CAG cells could be determined. For example, comparing the equal cells byte by byte will take the longest time to execute. The solution is to use constant-time comparison functions. Some examples can be found in [19]. Note that the solutions are not entirely portable; therefore, finished program assemblies for particular hardware should be checked for correctness in any case.

- The second CAG issue is branched execution times. For example, CAG timing for each branch differs; consequently, the mask parity of cag-v3 in Section 2 could be easily determined. The solution is equal execution time for each branch. That is accomplished by introducing two pre-calculated intermediate variables v and w before branching. The update of CAG carry with v and w has equal execution of branches:

$$v = A_{i+1} \tag{9}$$

$$w = \overline{A}_{i+1} \tag{10}$$

$$c' = \begin{cases} c \oplus v, & \text{if } A_{i+2} > A_{i+3} \\ c \oplus w, & \text{otherwise.} \end{cases} \tag{11}$$

4.3 Input Flexibility

Generally, there is a requirement of an initialization vector (IV) in symmetrical encryption. For example, AES CBC (AES in cipher block chaining) needs a unique 128-bit IV for every message processing. If more than a 128-bit IV is needed for some reason, the key derivation function (KDF) is required to deliver a properly sized key and IV.

In that respect, the CAG allows an additional 480 bytes for the nonce, salt, pepper and … if needed. It can be used for resisting various repeat attacks, for example. That can be accomplished without using KDF, as is the case with AES.

5. Conclusion

The cellular automaton generator (CAG) offers an entirely new cryptographic primitive. It has a straightforward and compact implementation. It also provides side-channel attack (SCA) resistance, which is very important for hardware that is not physically secured, such as internet of things (IoT). Variant cag-v1 also benefits from the fact that its predecessor mag-v1 had relatively in-depth analysis and remains secure [8].

Appendix

A. Cellular Automaton Generator Implementation Details

Implementation is written in C language and source code is in Listing 1.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

void stir(uint32_t * array, uint32_t updates, int init)
{
	const uint32_t mixer = 1431655765;		//010101...
	uint32_t carry = 987654321;
	int a = 128;					//array size
	uint32_t i;
	for(i = 0; i < updates; i++)
	{
		if(array[(i+2)%a]>array[(i+3)%a])
			carry ^= array[(i+1)%a];
		else
			carry ^= ~array[(i+1)%a];

		array[i%a] ^= carry;
		carry += mixer;

		if(init == 1)
			fwrite(&array[i%a] , sizeof (array[i]) ,1 , stdout );
	}
}

int main(int argc, char **argv)
{
	uint32_t bytesfour;
	uint32_t skip = 512;
	uint32_t array[128] = {0};

	size_t length = strlen (argv[2]) ;

	bytesfour = atoi(argv[1]);
	memcpy(array, argv[2], length);

	stir(array, skip, 0);
	stir(array, bytesfour, 1);

	return 0;
}
```

Listing 1. cag.c.

Let us assume that the above file is compiled to *cag*. The execution command might look like:

cag 1024 entropy0 > entropy0.raw

where the first argument, 1024, represents how many cells are sent to the standard output (4096 bytes). The second argument is the seed entropy0 and > entropy0.raw redirects standard output to the file.

Figures A.1 and A.2 show graphical representations of the outputs when input differs by one bit only. The seeds are strings “entropy0” and “entropy1,” respectively. Every pixel is one byte shown in 8-bit grayscale.

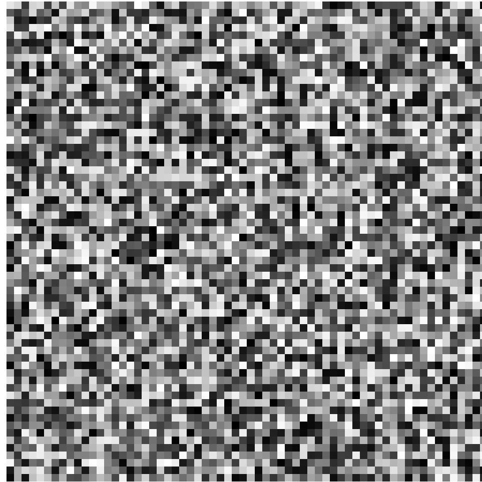

Figure A.1. Seed entropy0, (argv[2]).

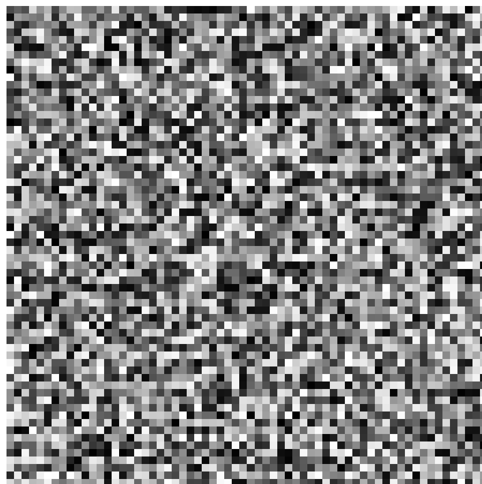

Figure A.2. Seed entropy1, (argv[2]).

B. Cellular Automaton Generator Statistical Properties

B.1 Standard Randomness Tests

MAG output streams were tested for patterns in every stage of development (several gigabytes of data), and no patterns were found. Systematic (recorded) pattern testing was performed on 500 megabytes of output data and no weaknesses were found. All testing details can be found in [20]. The 50Tests folder includes test data and corresponding test results divided into 50 subfolders (01, 02, 03…). Each sample from each folder was independently seeded. The 50Tests folder includes source code, executable and seeds used to produce test data (RNGSource folder). It also includes randomness test tools (Ent, Diehard and CRYPT-X'98 in the TestTools folder).

Test suites used for randomness testing of MAG:

- Diehard [21]: The Diehard battery of tests includes: birthday spacings, overlapping permutations, ranks of matrices, count the 1s, parking lot test, minimum distance test, random spheres test, the squeeze test, overlapping sums test, runs test and the craps test. The result of each test is a p value, which should be between 0 and 1. The testing of MAG returned a couple of p values close to zero or one with two first decimal places being 00 or 99, which is expected due to the volume of testing. Even a failure, p 0 or 1 to six or more places, is within expectation for 50 tests. Please visit [20] for details.
- ENT [22]: The ENT results for the same Diehard-tested samples are a pass (please see https://www.fourmilab.ch/random/README and [20] for details). Typical ENT tests results for MAG look like:

  Entropy = 7.999984 bits per byte.

  Optimum compression would reduce the size of this 11 466 000-byte file by 0 percent.

  Chi square distribution for 11 466 000 samples is 256.70, and randomly would exceed this value 50.00 percent of the time.

  Arithmetic mean value of data bytes is 127.5070 (127.5 = random).

  Monte Carlo value for pi is 3.141592883 (error 0.00 percent).

  Serial correlation coefficient is –0.000415 (totally uncorrelated = 0.0).
- CRYPT-X'98 [23]: CRYPT-X'98 tests the same data (as in the Diehard and ENT cases). The software is developed by QUT Information Security Research Centre and Centre in Statistical Science and Industrial Mathematics. The test used for MAG is the Stream Cipher Test option. This option has the following tests: frequency, binary derivative, change point, subblock, runs, sequence complexity and linear complexity. The results are interpreted. For example:

Input file: E:\50Tests \21 \binout.32

Total bits: 91 728 000
Number of ones (x): 45 860 357
Expected ones (mean): 45 864 000.0
Proportion of ones: 0.5000
Significance probability (p): 0.4468

Interpretation. Frequency test:

– 44.68% of bit streams of length 91 728 000 will have a number of ones further from the mean of 45 864 000.0 for the hypothesized distribution than this sample.

- This sample satisfies the frequency test.

Summary: all 50 samples pass all Stream Cipher CRYPT-X'98 tests.

B.2 Nonstandard Randomness Tests

The CAG was examined for the avalanche effect [24] property. It is a requirement for the block ciphers and hash functions. This property is akin to the butterfly effect. Flipping one bit of a key or a hash input should flip at least half of the bits in the resulting ciphertext or hash. This property is apparent in the CAG; Figure B.1 illustrates that. Predecessor MAG [2] shows similar results when the input seed is varied by just one bit. The distributions of ones and zeros in the produced stream are significantly different. The Diehard test [21] shows p values differences in Figure B.2.

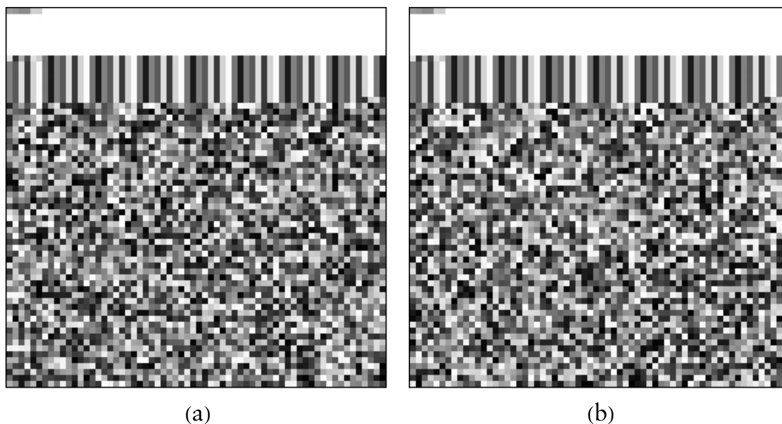

Figure B.1. Two CAG evolutions where initial state differs by just one bit. The seeds (strings (a) bits01 and (b) bits02) are the first six top-left pixels (bytes); the sixth pixel on the right image is just one gray shade darker (not visible) than the corresponding pixel on the left. The whole initial state is the seed plus the white region on the top (the top 512 pixels of the image). Even very low entropy produces a quite complex outcome. It also shows elements of chaos behavior, where a small initial change produces different outcomes.

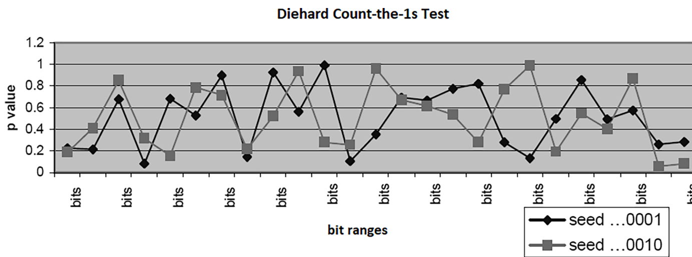

Figure B.2. The distribution of the 1s (for bit ranges 1–8, 2–9, 3–10, 4–11...) for streams initialized by 32-bit unsigned integers 1, 2 (0...0001, 0...0010).

References

[1] J. S. Testa, "Investigations of Cellular Automata-Based Stream Ciphers," Ph.D. thesis, Computer Science (GCCIS), Rochester Institute of Technology, Rochester, NY, 2008. scholarworks.rit.edu/theses/129.

[2] R. Vuckovac, "MAG: My Array Generator (A New Strategy for Random Number Generation)," Report 2005/014, ECRYPT eStream Project, 2005. www.ecrypt.eu.org/stream/ciphers/mag/mag.pdf.

[3] A. Ilachinski, *Cellular Automata: A Discrete Universe*, River Edge, NJ: World Scientific, 2001.

[4] L. Simpson and M. Henricksen, "Improved Cryptanalysis of MAG," in *Australasian Conference on Information Security and Privacy (ACISP 2006)*, Melbourne, Australia (L. M. Batten and R. Safavi-Naini, eds.), Berlin, Heidelberg: Springer-Verlag, 2006 pp. 64–75. doi:10.1007/11780656_6.

[5] S. Kunzli and W. Meier, "Distinguishing Attack on MAG," Report 2005/053, ECRYPT eStream Cipher Project, 2005. www.ecrypt.eu.org/stream/papersdir/053.pdf.

[6] E. Dawson and M. Henricksen, "Ensuring Fast Implementations of Symmetric Ciphers on the Intel Pentium 4 and Beyond," in *Australasian Conference on Information Security and Privacy (ACISP 2006)*, Melbourne, Australia (L. M. Batten and R. Safavi-Naini, eds.), Berlin, Heidelberg: Springer-Verlag, 2006. doi:10.1007/11780656_5.

[7] S. Fischer, S. Vaudenay and W. Meier, "Analysis of Lightweight Stream Ciphers," Ph.D. thesis, School of Computer and Communication Sciences, EPFL, Lausanne, Sweden, Security and Cryptography Laboratory, 2008. doi:10.5075/epfl-thesis-4040.

[8] A. Mirzaei, M. D. Alian and M. M. Hashemi, "Distinguishing Attack on a Modified Version of MAG Stream Cipher," in *6th International ISC Conference on Information Security and Cryptology (ISCISC '09)*, Isfahan University of Technology, Isfahan, Iran, 2009. hdakhilalian.iut.ac.ir/sites/dakhilalian.iut.ac.ir/files/u32/c26.pdf.

[9] R. Vuckovac, “MAG Alternating Methods Notes,” Report 2005/068, 2005, ECRYPT eStream Cipher Project, 2005. www.ecrypt.eu.org/stream/papersdir/068.pdf.

[10] eSTREAM: the ECRYPT Stream Cipher Project, “Latest Performance Figures (AMD Athlon 64 X2 4200+ 2.20GHz).” (Oct 30, 2019) www.ecrypt.eu.org/stream/perf/#results.

[11] T. Arcieri. “Cream: The Scary SSL Attack You’ve Probably Never Heard Of,” *Tony Arcieri* (blog). (Oct 30, 2019). tonyarcieri.com/cream-the-scary-ssl-attack-youve-probably-never-heard-of.

[12] A. Heuser, S. Picek, S. Guilley and N. Mentens, “Lightweight Ciphers and Their Side-Channel Resilience,” *IEEE Transactions on Computers*, **99**, 2017 pp. 1–1. doi:10.1109/TC.2017.2757921.

[13] S. Wolfram, “Computation Theory of Cellular Automata,” *Communications in Mathematical Physics*, **96**(1), 1984 pp. 15–57. doi:10.1007/BF01217347.

[14] S. Wolfram, “Cryptography with Cellular Automata,” in *Advances in Cryptology CRYPTO ’85 Proceedings* (H. C. Williams, ed.) Berlin, Heidelberg: Springer, 1985 pp. 429–432. doi:10.1007/3-540-39799-X_32.

[15] R. Vuckovac, “A New Kind of Complexity,” arxiv.org/abs/1309.0296.

[16] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc., 2002.

[17] D. Coppersmith, H. Krawczyk and Y. Mansour, “The Shrinking Generator,” in *Advances in Cryptology CRYPTO ’93* (D. R. Stinson, ed.), Berlin, Heidelberg: Springer, 1994 pp. 22–39. doi:10.1007/3-540-48329-2_3.

[18] D. J. Bernstein. “Cache-Timing Attacks on AES.” (Oct 30, 2019) cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[19] J.-P. Aumasson, A. Perrin, vixentael and MicheleCicciottiWork. “Cryptocoding.” (Oct 30, 2019) github.com/veorq/cryptocoding.

[20] R. Vuckovac. “RadeVuckovac/MAG-Statistics.” (Oct 30, 2019) github.com/RadeVuckovac/MAG-Statistics.

[21] G. Marsaglia, “The Marsaglia Random Number CDROM, Including the Diehard Battery of Tests of Randomness,” Tallahassee, FL: Florida State University, 1995.

[22] J. Walker, “ENT: A Pseudorandom Number Sequence Test Program.” (Oct 30, 2019) www.fourmilab.ch/random.

[23] E. Dawson, A. Clark, H. Gustafson and L. May “CRYPT-X ’98, (Java Version) User Manual,” Queensland University of Technology, 1999.

[24] H. Feistel, “Cryptography and Computer Privacy,” *Scientific American*, **228**(5), 1973 pp. 15–23. www.apprendre-en-ligne.net/crypto/bibliotheque/feistel.