Cover Sheet

Title: NKS 2D Cellular Automata Hash

Submitter: Geoffrey Michael Park

email: Geoffrey.Park@gmail.com
telephone: 416 690-4158
fax: 416 690-4158
organization: none
address: 183 Chisholm Ave., Toronto ON, Canada, M4C 4V9

Algorithm developer: Geoffrey Park

Algorithm Inventors: Geoffrey Park

Algorithm Owners: Geoffrey Park

Signature:

Table of Contents

Algorithm Specifications and Supporting Documentation

1 Algorithm specification

1.a The algorithm

The cryptographic hash algorithm described below is an adaptation of the 2D cellular automata generator for totalistic generation rules as described in "A New Kind of Science" by Stephen Wolfram, ISBN I-57955-008-8 [referred to below as 'NKS'].

A cellular automaton is an algorithm operating on an array of cells, each in one of a finite number of states.
A rule is applied to update all cells, creating a new generation. Rules are based on the state of a cell and those of its' neighbors. Cell arrays may be of any number of dimensions.

In NKS, Stephen Wolfram studies the nature of processes that can generate chaotic, pseudo-random sequences. It seems that the complexity of the process is unrelated to the statistical randomness of the sequence generated. In fact, some of the simplest such processes, cellular automata, can generate sequences as chaotic as any other.

In choosing a chaos generating process for use in a cryptographic hash, it seems a simple process is preferable to a complex one, other considerations being equal. A hash with very small internal state can be completely analysed by exploring all possible states. If the same algorithm can be extended to larger state spaces, and if it can reasonably be inferred that it will behave in a similar way, the analysis of the small hash may be extrapolated to ones too large for complete analysis.

NKS chapter 5 describes a family of simple cellular automata, in which cells have two states and generation rules consider only the state of a cell and the total value of adjacent neighbor cells. In NKS, generation rules for these “totalistic” cellular automata are specified using the following convention:

> *The last bit specifies what color the center cell should be if all its neighbors were white on the previous step, and it too was white.*
> *The second to last bit specifies what happens if all the neighbor cells are white, but the center cell itself is black.*
> *Each pair of earlier bits then specifies what should happen if progressively more neighbor cells are black.*

In one dimension, a cell has only two adjacent neighbors. In two dimensions cells can be counted as 'neighbors' four ways:

1. *4 rectangular positions e.g. above, below, left, right*
2. *4 diagonal positions e.g. above-left, above-right, below-left, below-right*
3. *6 hexagonal neighbor positions on hexagonal grid*
4. *8 adjacent positions, e.g. 4 rectangular plus 4 diagonal positions*

To keep memory use finite, a one dimensional automaton can be confined to a line segment by taking

the neighbor of the leftmost cell to be the rightmost cell, and vice versa - topologically a ring. Two dimensional automata can similarly be confined to a rectangle that wraps around at the edges - topologically a torus. An automaton can also be confined using reflection. Here the neighbor to the left of the leftmost cell is the neighbor to it's right, and so on.

Totalistic rules exist for one dimensional cellular automata that can generate good pseudo random sequences. One such rule is used for random number generation in Mathematica (NKS page 317, http://www.freepatentsonline.com/4691291.html). A two dimensional automaton has a better mixing rate, however, when used as a cryptographic hash.

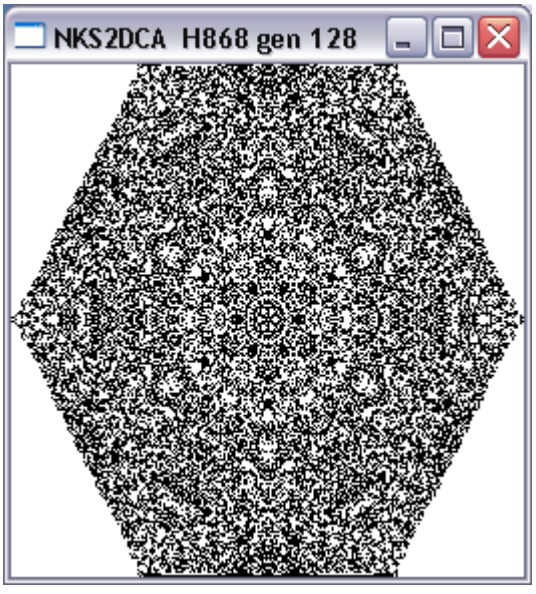
For rules considering only adjacent cells as neighbors, it is clear that the influence of new data can spread at a rate of no more than one cell in each direction, per generation. So for a one dimensional automaton operating over N cells, it will take at least N/2 generations for new data to influence all cells. For a two dimensional automaton on a square torus this "data diffusion rate" limit would be sqrt(N)/2. Cellular automata of high dimensions could have still faster data diffusion rates, but were not examined due to the size of the resulting rule spaces.

Using the above convention, one can see that it requires (1 + 4) * 2 = 10 bits to define a rule for a 2D cellular automaton using 4 neighbors per cell. The hexagonal rules require 14 bits, and 8 neighbors requires 18 bits, resulting in rule space sizes of 1024, 16384, and 262144 respectively. Most of the possible rules will reduce to static or repeating patterns in a few generations. Only a few rules will generate apparently random sequences. In NKS, Stephen Wolfram describes methods he used to automate the search of rule spaces for cellular automata that exhibit interesting behavior. A cellular automaton rule suitable for use in a cryptographic hash must, however, meet some specific criteria. Section 1.c below, describes some criteria that may be useful in assessing the suitability of a cellular automaton rule for use in a cryptographic hash.

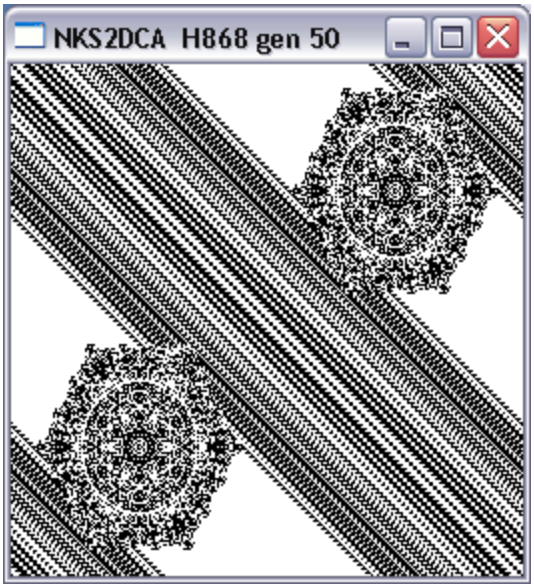
1.b Example patterns:

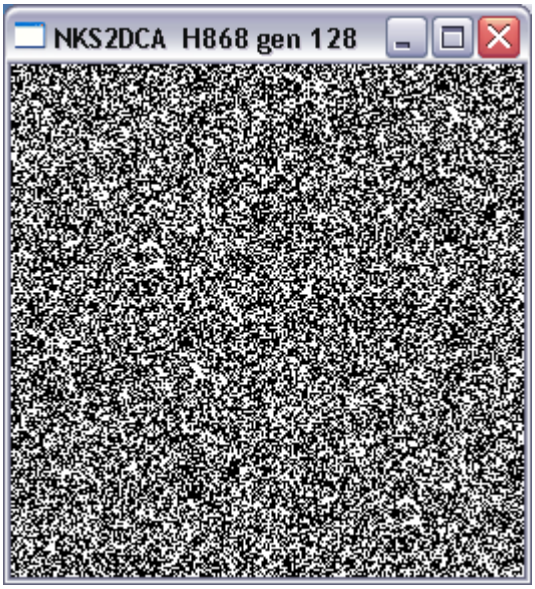
The following figures show some patterns produced by various cellular automata with either a single dot or a two dot and diagonal line pattern:

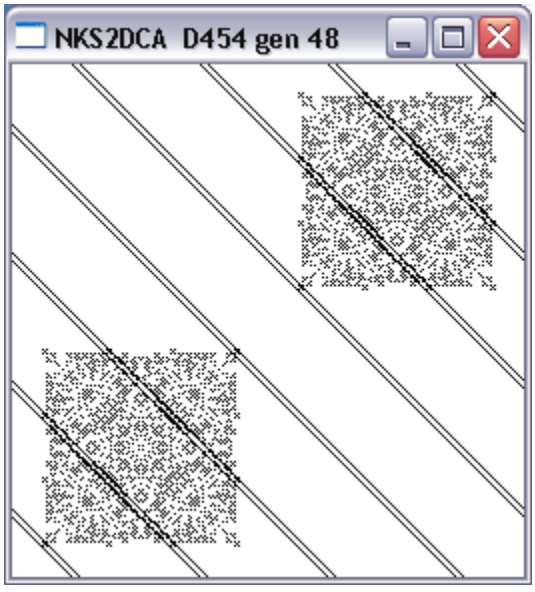
Single dot on a 256x256 plane, Hexagonal grid Rule 868:



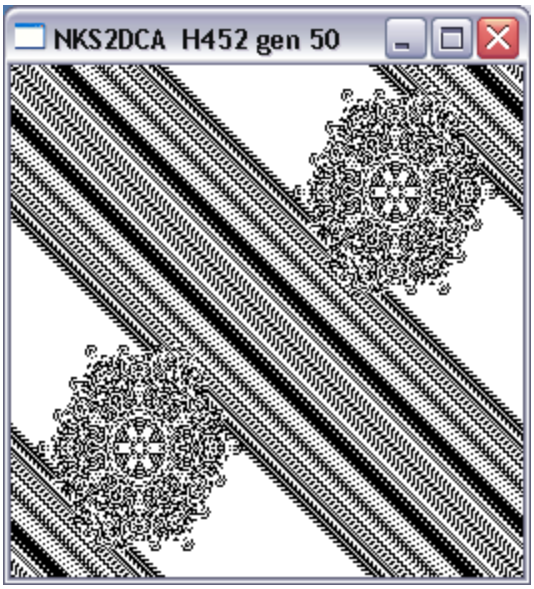
Two Dots + single pixel Diagonal line:

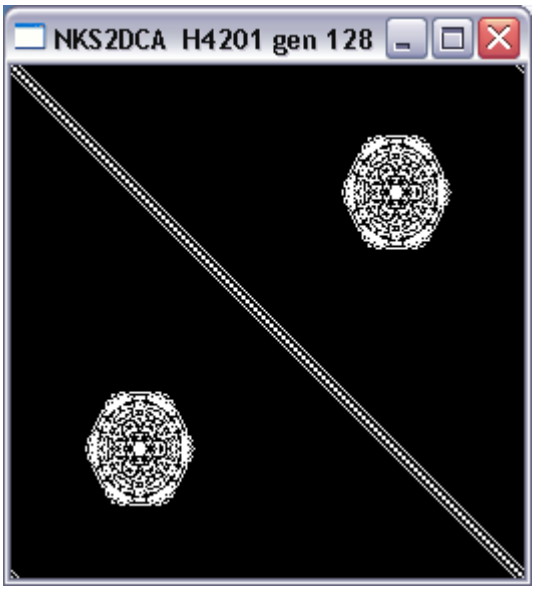


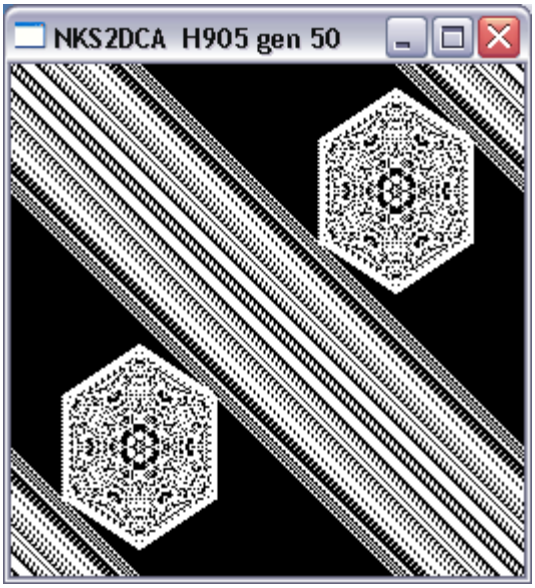


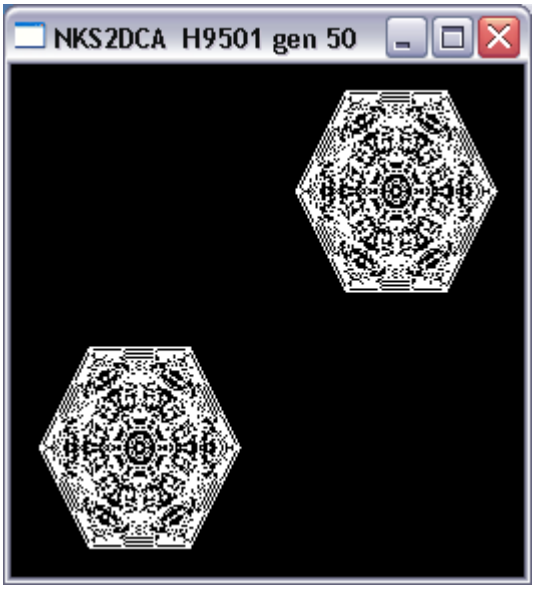


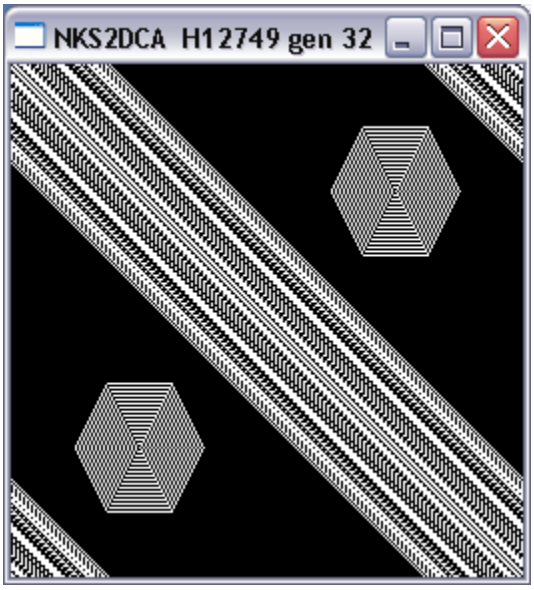
Some other rules:

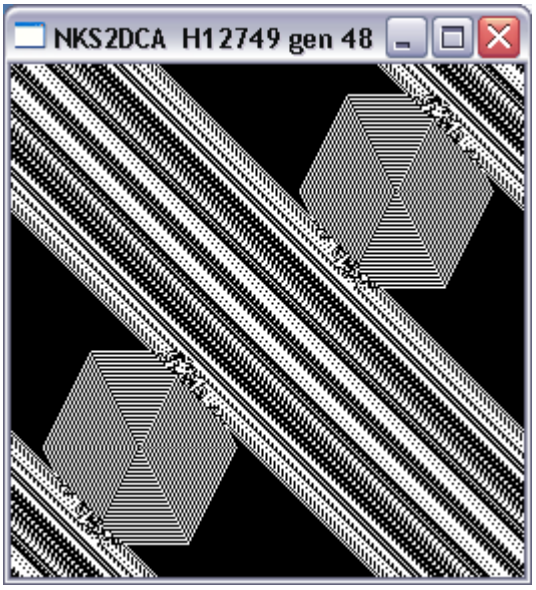


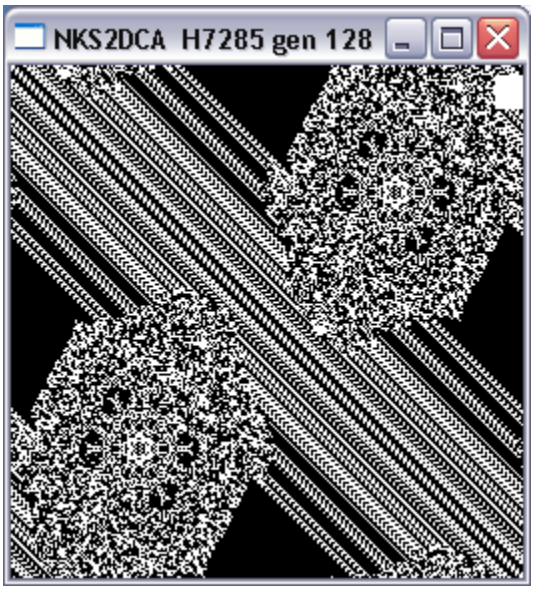


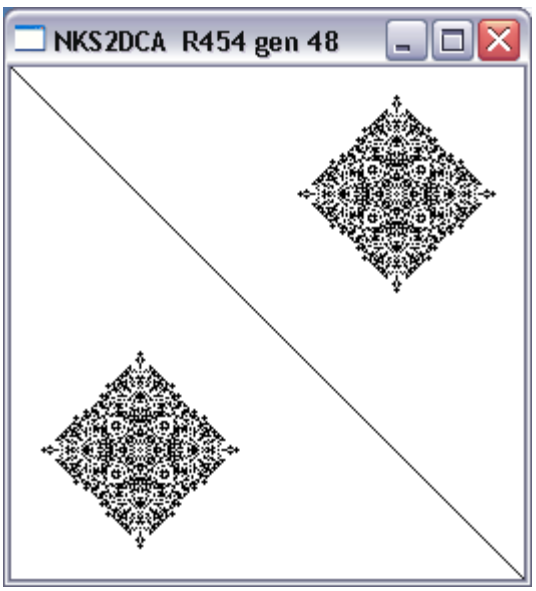





NKS2DCA H12749 gen 32


NKS2DCA H12749 gen 48


NKS2DCA H7285 gen 128


NKS2DCA R454 gen 48

1.c Criteria for choosing a generation rule

Some rules will result in a solid (all black or all white) color after just one or two generations, no matter the starting pattern. Others will slowly erode some or all starting patterns, going blank after at most w or 2w generations, where w is the width of a square toroidal bit plane. Still others will seem to generate suitably random sequences when started with random patterns, but will produce stable, fixed or cycling patterns when started with ordered patterns. In searching a rule space, simple tests will eliminate these.

The maximum rate that a pattern can grow using a totalizing nearest neighbor rule is 1 pixel per generation. Some rules produce patterns that grow slower that this, and may therefore require more generations than others to achieve equal mixing.

Beyond finding merely interesting behavior, a search for potential cryptographic hashes needs also to find rules that maximize randomness and collision resistance.

By reducing the dimensions of the plane to the minimum possible size, it is possible to try all possible input states for each rule in a rule space and count all collisions.

Table 1 shows the top 30 of 16384 totalizing cellular automata rules possible on a hexagonal grid, sorted in ascending order of collision count.

Table 1

rule	collisions	percent
9932	25611	0.39
6963	25948	0.4
6451	25979	0.4
3481	26050	0.4
13932	26136	0.4
2451	26167	0.4
6547	26198	0.4
10035	26212	0.4
3225	26319	0.4
3270	26489	0.4
3273	26494	0.4
4918	26495	0.4
9836	26542	0.41
12900	26555	0.41
7366	26565	0.41
7321	26604	0.41
4966	26618	0.41
9625	26744	0.41
4913	26765	0.41
3483	26790	0.41
2460	26822	0.41
7782	26849	0.41
1740	26867	0.41
6348	26875	0.41
12902	26895	0.41
7369	26925	0.41
14028	26930	0.41
13924	26952	0.41
2459	26952	0.41

These statistics were calculated for an 8x4 cell plane for which half the cells (two bytes) where initialized with data values from 1 to 65535. Each rule operated over each plane for 8 iteration. The simplified "hash" was a two byte value from the exclusive or of all the available state. The complete table is in the file balancecheckall.csv on the CD.

A two byte hash may have little use in cryptography, but because cellular automata rules compute their next iteration based entirely on immediate neighbors, they have very local behavior. There is some reason to believe that the basic behavior of a given rule will be independent of the size of the plane over which it operates. If so, we may be justified in extrapolating the results of table 1 to larger hash sizes.

A low collision rate is a necessary but not sufficient condition for selecting a rule. It is also necessary to filter out rules that generate unstable patterns. Of course, those rules that converge to a solid (all black or all white) color after a few generations will have high collision rates and will already have been eliminated. There may be some rules with low collision rates that nevertheless will converge to stable fixed patterns or short repeating cycles after some number of iterations of from some starting state. If the input data ends with, or is extended to by, a large zero pad, such a rule may converge to a predictable state.

In table 2 all the rules of table 1 with less that 50% collision rate were sorted by descending order of period.

Table 2

rule	collisions	fraction	balance	period	generations	stable
H8604	29175	0.45	0.92	977	"2001"	yes
H4558	31068	0.47	0.91	909	"1933"	yes
H5478	31967	0.49	0.9	853	"1877"	
H7473	30244	0.46	0.91	821	"1845"	yes
H1817	31319	0.48	0.91	721	"1745"	
H6601	28482	0.43	0.92	687	"1711"	yes
H8649	31336	0.48	0.91	656	"1680"	yes
H9651	31818	0.49	0.91	457	"1481"	yes
H13459	30427	0.46	0.91	429	"1453"	yes
H3478	28596	0.44	0.92	317	"1341"	yes
H12601	32015	0.49	0.91	299	"1323"	yes
H2865	31043	0.47	0.91	265	"1289"	yes
H2867	28621	0.44	0.92	233	"1257"	yes
H1633	29587	0.45	0.91	185	"1209"	yes
H4892	31502	0.48	0.91	155	"1179"	
H2841	29896	0.46	0.92	153	"1177"	yes
H868	28704	0.44	0.92	153	"1177"	yes
H12595	29958	0.46	0.91	149	"1173"	
H7601	31749	0.48	0.9	123	"1147"	yes
H12748	28988	0.44	0.92	93	"1117"	yes
H13718	31515	0.48	0.91	79	"1103"	yes
H3529	31737	0.48	0.91	65	"1089"	yes
H6931	31298	0.48	0.91	65	"1089"	yes
H4963	31937	0.49	0.91	53	"1077"	yes

A test was devised to measure a rule's repeat period. For this purpose a toroidal plane was used and seeded with a 4 x 4 grid pattern. Because of the symmetry of the starting pattern, the actual size of the plane is irrelevant – the pattern within any 4 x 4 pixel group would be identical, even for an infinite plane. [This is a useful property, because the included implementations won't operate on a 4 x 4 plane].

The test proceeds as follows:

The starting grid pattern is iterated for N generations. After N/2 generations the cell pattern is recorded. If during the next N/2 generations that pattern appears again, the period is recorded.

As another check, the Balance Measure as described in "Hash Function Balance and its Impact on Birthday Attacks" by Mihir Bellare and Tadayoshi Kohno, 401-418, Advances in Cryptology -EUROCRYPT 2004, ISBN 978-3-540-21935-4, was also computed.

A final visual check was done using a simple pattern of two dots and a 45 degree diagonal line on a 256 x 256 cell plane. If either the line or dots eroded in the first few generations, the rule was considered not stable enough for use in a hash.

For any given rule, there may be "self erasing" patterns. If such patterns are known, they may allow the hash to be attacked or weakened. A "yes" in the stable column indicates that both the line and dot patterns grew rapidly into complex visually random patterns.

1.d Implementing a Cryptographic Hash using Cellular Automata

The following describes a simple implementation of a hash using cellular automata.

Initialization of the algorithm begins by allocating a square or approximately square plane having hashlen or greater bits. This plane is seeded with a single black pixel (bit = 1) and the cellular automaton rule is iterated until the rule's characteristic noise pattern fills the entire plane. For a rule that grows one pixel per generation, this will take maxdim generations where maxdim is the largest dimension of the plane.

During the update phase of the hash algorithm, data is mixed into the process by XORing a block of data with the current plane, and iterating one or more times.

The finalize phase incorporates any left over data, as well as the the 64 bit data length, then iterates a further maxdim generations.

This implementation is not as secure as one might want for cryptographic use. If data can be contrived that will cause the algorithm to "wash out" resulting in a solid white or solid black bitplane this sequence could be appended to two different files giving them both the same hash. If input data were XORed with all the bits in the plane, simply using the XOR of the current plane would do this.

If only a fraction of the plane bits are XORed with input data, and the remaining bits in the plane allowed to continue evolving the original pattern, it would be difficult or impossible to completely "wash out" the bit plane. Nevertheless, XORing the accessible fraction with it's complement repeatedly would surely lessen the influence of previous data on the final hash. Depending on the fraction of the plane accessible a length extension attack of some kind would likely be possible.

A better solution is to run two parallel streams of automata using different starting conditions and/or different rules. If the same data block is XORed with both planes it should be unlikely that data could ever be contrived to zero both streams. This requires twice the processing but is more secure.

The included reference and optimized implementations extend the SHA3 API with two additional functions, HashEx(...) and InitEx(...), which can be used to configure the hash in several ways. The normal Hash(...) and Init(..) functions are implemented by calling HashEx(...) and InitEx(...) with default parameters.

Additional configuration parameters include the number of streams, the rule and initial state used for each stream, the number of generations computed for each block, and the data overlap for each pair of streams. In the Update(..) function, new data is mixed into each stream in turn. A data overlap of 100% for a stream means that it gets a duplicate of the data block used for the previous stream. 0% overlap means that it gets new data, with no duplication.

For example, one could configure a 512 bit hash with two streams, using rules H868 and H6502, 0% overlap on the first stream, 100% on the second stream. The second stream gets a duplicate of the first stream's data, but uses a different generation rule. The first stream gets a new data block on each generation.

For the purpose of this submission, the included code is configured as follows:

- 224 Bit hash: One stream, rule H8604, 25% overlap.
- 256 Bit hash: Two streams, rules H8604, H4558, 25% overlap each.
- 384 Bit hash: Three streams, rules H8604, H4558, H7473, 33.4% overlap each.
- 512 Bit hash: Three streams, rules H8604, H4558, H7473, 0%, 100%,100% overlap.

For all hash lengths, one generation per block is used by default.

2 Estimated computational efficiency

The reference implementation of NKS2D has modest memory requirements. For each stream there are two cell planes each equal in size to the hash length, and five int state variables. In addition there is one temp data array of 2*hash length and 10 more state variables of int size. For the 512 bit hash this is approximately 612 bytes.

The included 32 bit optimized implementation requires considerably more memory,as it generates lookup tables that are used to speed up the rule interpreter.

The included 8 bit implementation is a working 224 bit hash implementation which runs on a PIC16F690 which has only 256 bytes of RAM in total.

The table below shows results of speed tests of the 32 bit optimized implementation compiled with Microsoft Visual Studio 2008, running under Vista 32 on an AMD Phenom 9500 Quad Core Processor 2.21 GHz, hashing an 87 MB file:

Optimized 32 bit implementation

Hash Length	MB/ second	Cycles/ Byte
224	11.8	187
256	12.3	178
384	12.3	178
512	6.3	350

Note that the 512 bit hash is configured to duplicate the same input data over 3 streams to maximize security and is therefore slower. For a given configuration, longer hashes actually run slightly faster than shorter ones.The same code was compiled for 64 bit Windows operating system and tested on Vista 64. The results are in the table below:

Optimized 64 bit implementation

Hash Length	MB/ second	Cycles/ Byte
224	18.7	117
256	18.8	117
384	18.02	122
512	9.05	243

8 bit implementation

The 8 bit implementation is optimized for memory use, but not for speed. The speed of the 224 bit hash running on PIC16F690 at 20 MHz clock was approximately 812 bytes per second using 6155 cycles per byte. An improved 8 bit implementation may be submitted later, though it cannot be expected to reach the cycles/ byte of the optimized 32 and 64 bit implementations which make extensive use of the longer data words.

3 Known Answer Tests and Monte Carlo Tests

Known Answer and Monte Carlo tests were run using the code provided by NIST for the SHA3 contest. The results are on the CD in folder \KAT_MCT. The same tests were run on both the reference and optimized implementations and found to be identical.

3.a Intermediate values

Text files containing intermediate results of one block and two block hashes for hash lengths of

224, 256, 384, and 512 bits are included in the folder:

\Reference Implementation\IntermediateCalculations

The files Intermediate224_fmt.txt and Intermediate256_fmt.txt contain the same data as Intermediate224.txt and Intermediate256.txt, but more compactly formatted for readability.

4 Expected strength

NKS2D can be used for HMAC [Hash Message Authentication Codes] by prefixing a secret key to the the message. If a length extension resistant configuration such as the included 512 bit configuration, it should not be practical to add data to the message without knowing the key and obtain a valid MAC.

For a single rule configuration with no overlap, it is trivial to create data that will wash out all state due to the previous message, this could allow an attacker to prefix any desired data to the message, and get the same MAC. The included 224 bit configuration uses a single rule with overlapping data blocks. Unfortunately, the strength against extension may be reduced to the number of bits in the overlap. This configuration might be strengthened with a construct such as H(key1 || H(key2 || message)) where key1 and key2 are constructed from key by XOR with outer and inner pads having large hamming distance.

However it is probably preferable to simple use a multi stream configuration with different rules and 100% overlap. For HMAC use it is recommended that at least two rules with 100% overlap be used. There is no reason to suspect that attacking such a HMAC would require significantly less computation than a preimage attack.

NKS2D can be used for randomized hashing by prefixing a random salt in addition to the secret key to the message. There is no reason to suspect that attacking such a HMAC would require significantly less computation than a preimage attack.

The tests described in section 1.c on very small hashes, indicate that the included implementations should have collision resistance of n/2 bits or slightly better.

Preimage resistance should be approximately n bits.

Second preimage resistance has not yet been thoroughly analyzed. However resistance to length extension should make it more difficult to find messages of different lengths that all yield the same internal state after processing them. This is a necessary condition for the attack described in *Second Preimages on n-bit Hash Functions for Much Less than 2n Work* by John Kelsey and Bruce Schneier.

5 Known attacks

NKS2D can be attacked by brute force and the birthday attack. Because of the ease of implementation in hardware it may be possible to devise extremely fast brute force machines to attack it. No analysis has yet been done for resistance to differential cryptanalysis. It should however be possible to make differential cryptanalysis arbitrarily more difficult by configuring the hash with sufficient additional parallel streams using different rules and or initial conditions.

6 Advantages and limitations

In NKS, Stephen Wolfram makes the case that some of the simplest pseudo random processes, cellular automata, can generate sequences as chaotic as any other. The main advantage of using a simple process for a cryptographic hash is the ease of analysis. Totalizing cellular automata, in

particular, seem to operate in an identical manner over planes of any size equal or greater than 4 x 4 pixels. On such a small plane, it is possible to use brute force empirical analysis. By initializing the process with every possible starting pattern, and studying the sequences resulting from iterating the rule under study, one should be able to learn whether the rule is a good candidate for a cryptographic hash. There is of course an implicit assumption that the behavior of a totalizing cellular automaton will not change in any significantly qualitative way when scaled to useful sizes.

One might see the simplicity of analysis as a disadvantage, also: It may make it easier for a black hat to discover exploitable flaws in a hash algorithm. On the other hand, just because it is more difficult to find flaws in a more complex algorithm, does not imply that someone hasn't done it. Obscurity does not produce security.

Although analyzing a cellular automaton on a 4 x 4 plane with just 2^16 possible states, may be a great deal easier than analyzing SHA256, the analyses presented here are by no means exhaustive. Due to the larger rule space [262144 rules] of the eight neighbor rules, these were not analyzed at all. It may be that in that rule space there are automata that are more suitable for use in cryptographic hashes than those explored here.

It is also possible that an as yet unknown exploitable flaw is hidden in the presented algorithm. This could also be true of SHA2. If so, it is likely that flaws in a simple cellular automaton rule can be more easily discovered.

6.a Implementations

The included implementations are all in C and take no special advantage of modern CPU hardware. The 32 bit and 64 bit optimized versions combine a number of pixel operations in one 64 bit integer operation, for a primitive sort parallel processing. Proper use of Intel SIMD [Single Instruction Multiple Data] instructions should improve performance significantly. The multi stream configurations can be easily implemented with multiple threads on modern multi-core processors, for another significant improvement in speed.

The included 8 bit implementation is compact but not particularly efficient. Coding in assembler would allow much better use of the available registers and RISC instruction set of the PIC family of processors.

The NKS2D hash could also be easily implemented in HSL (high level shader language) or other shader or GPGPU [general purpose graphics processor unit] language.

FPGA [Field Programmable Gate Array] or ASIC [Application Specific Integrated Circuit] implementations should also be particularly easy. The operations required are just barrel shifts, bit tests, adds, and table look up. Few floating point or integer multiplications or divisions are used at all.

Furthermore, the core rule operation is inherently parallel: Each pixel in the next plane depends only on 4 to 8 pixels in the previous plane and could be computed independently of any results for the current plane. It should be possible to design custom hardware capable of executing large hashes either very fast using one small engine per pixel, or slower but with less silicon by reusing one or more small engines to compute each generation.

6.b Digest sizes

To compute the next value of a pixel for a totalistic cellular automaton requires input values from neighbors above, below, to the left and to the right. This implies that at least a 3 x 3 plane is necessary. In practice, at least 4 x 4 pixels are necessary before chaotic behavior is observed. The minimum digest

size is therefore 16 bits. The upper limit of digest size is bounded only by available memory. With the included optimized implementations, larger hashes are actually slightly more efficient.