MAG Hash Function Preliminaries

0.0 Abstract

This is a proposal for a new class of hash function using MAG alike algorithm [REF_M]. The proposal is accompanied with the prototype implementation. MAG Hash Function (MHF) uses a key for hashing, but that key can be public as well. Hash key and hash output lengths are not fixed and can be changed, although the prototype operates with 128 bits key and 128 bits hash output. Changing the size of hash key and the output size does not affect the algorithm complexity $O(1)$ as it is the case with other MAG algorithms.

1.0 Background

MHF is based on MAG computational irreducibility paradigm. The computational irreducibility implies that string produced by MAG can not be described by any other means other than MAG algorithm. That attribute partially satisfies Kolgomorov-Chaitin (KC) notion of randomness. KC approach to the randomness states that string is random if no shorter explanation of that string exists. In other words the random string can not be compressed in any possible way (represented by shorter string).

From KC point of view, MAG algorithm produces random string if MAG algorithm definition itself is excluded. More precisely the string produced by MAG is defined by the algorithm and the initial state only. That attribute makes departure from the classical PRNG. For example it is widely accepted that every PRNG will fail eventually some statistical test [REF_Goldreich], which is not in accordance with MAG hypothesis (computational irreducibility [NOTE1]). So far statistical testing can not find any pattern in MAG outputs, which is interesting phenomena because MAG is a very very simple algorithm.

Another MAG attribute is important for design of one way function. For steeping through MAG evolution, the knowledge of whole internal state is required. That attribute is also different from classical cellular automata (CA) (Wolfram's rule 30 and 45 [REF_W] and other various feedback shift registers). CA evolution can be partially observed because the cell evolution depends on neighboring cells and rule is well mathematically defined. MAG evolution can not be expressed by math formulas because selection criteria is underlying structure of MAG algorithm. Also MAG rule (selection criteria) depends on the words, where classical CA usually operates on bit scale, making MAG selection criteria outcome dependence far more complex. For example if two bytes is compared there is $2^8 * 2^8$ pairs which will produce true or false. If comparison is performed on bits there is only four pairs for comparison which will outcome true or false. That low complexity of evolution inputs is characteristic for other bit based CA.

2.0 MHF description

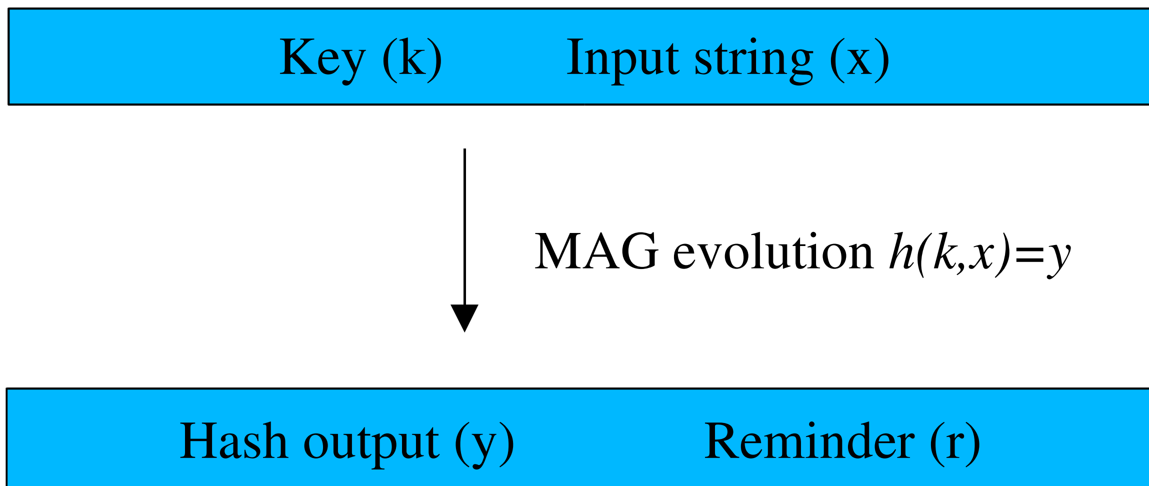
The idea of the proposed evaluational approach can be summarized as follows:

If input is a string (x) and if that string is evolved via MAG algorithm (h) than the evolved string should be pseudo random in KC sense (excluding MAG description of that state). A part (y) of evolved string can serve as a hash value.

Having the part (y) it is easy to find another string (x') which will produce the same string (x), because there is inverse MAG algorithm, which can reverse evolution.

But if string (x) is combined with the key (k) and that combo (k and x) is evolved by MAG, trying to reverse MAG evolution with the output string (hash y and arbitrary reminder r) becomes difficult task. The reversed string now has to contain key string (k). The key (k) can be public or secret (public key is implemented in accompanied code as a more simpler approach).

Practically, MHF implementation takes public key and intended message for hash as input. The key (k) and the message (x) are concatenated to form one string. That string is then evolved by MAG type of algorithm (selection criteria rule) for predefined number of cycles. From last evolution cycle a part of string is extracted and that part serves as a hash (output (y)).



3.0 MHF Details

Below are some general definition for a hash function taken from Handbook of Applied Cryptography book [REF_M_O_V].

compression – hash function h takes an input x , and produces an output $h(x)$ (usually $x > h(x)$ bitlengths wise).

ease of computation - given algorithm / function *h* and an input *string x*, it is easy to produce output $y=h(x)$.

preimage resistance - it is relatively hard to find some input string x' (preimage) to produce specific output y , if only output y and algorithm is known.

2nd-preimage resistance the same as above plus input string x ,which is mapped to specific output y,is known as well.

collision resistance - it is relatively hard problem to match two different inputs (x and x') to produce the identical output y even the variables (x x' and y) are not predetermined.

Below are some attributes of the proposed MHF:

compression – MHF maps input string (x) and key (k) to output string (y) and reminder (r).

ease of computation - it is easy to evolve string (x) using MAG rule.

collision resistance – It can be argued that if MHF is collision resistant then it posses *preimage resistance* and *2nd-preimage resistance* as well.

3.1 MHF collision resistance arguments

As mentioned, MHF is based on two premises.

- One is the selection criteria structure of the algorithm. That attribute provides apparent randomness and consequently the very good statistical distribution properties to the output stream. Therefore the chosen size of hash value should point to the collision resistance factor, meaning larger size of hash output increases collision resistance. For example, if (y) is 128 bits wide than probability of repeating the same string is one in 2^{64} Chances are that that eventual collision contains the same initial key (k) increases exponentially (2^{128}) .The hash size (y) is arbitrary value and does not make algorithm more complex ($O(1)$).

- Second is that knowledge of all cells (not just some cells) is required to step through evolution or to have any certain knowledge of the next evolved state. As discussed in [REF4 MAG] the claimed complexity of the algorithm depends on amount of cells in evolution if internal state is hidden as it is the case with PRNG MAG algorithm family. With hashing propositions the internal state is not hidden but the requirement of complete knowledge still plays important role. With hashing, aim is to find two different inputs with same hash output, meaning that inputs are varied to match predefined hash output (y). The variation of inputs can not be done partially meaning that divide and conquer strategies can not be applied in MHF case. For example if only one bit is changed from the original message (see MAG_REF) there is no

way to predict what will happen to the targeted evolution cycle. Only way is to try and see what will happen with prospect that acquired knowledge is valid for that configuration only. That phenomena forces exhaustive search through inputs and implies NP time cost solution to the problem.

It appears that there is two different ways to approach the MHF problem. Generally the problem is that with complete knowledge of hash process (known key (k), string (x), hash value (y) and reminder (r)), the string (x') has to to satisfy $h(x'+k)=y+z$ where (z)is arbitrary value.

- One approach is to start from the top with key (k) and changing string (x) to the (x') trying to get identical hash (y).
- Second is to start from bottom with hash (y) and changing reminder (r) to the (r') trying to get identical key (k).

This two approaches are actually equally difficult. Both have to change strings (x or r) to satisfy some pattern (k or y) going up or down through the evolution.

3.2 The MHF implementation details

First one is a key (k). The key size can be varied without impact to the overall algorithm complexity. Furthermore the key may be public as well. The prototype implements 128 bits public key which is set to zero and those zeros are hard coded. The public key variant is chosen because it appears to be simpler one.

Second one is hash output (y). The hash size is also arbitrary and should be the same size as the key (y). In that way the symmetry between the two approaches (starting from the top with key (k) and string (x) and from the bottom with hash (y) and reminder (r). The implementation uses 128 bit hash.

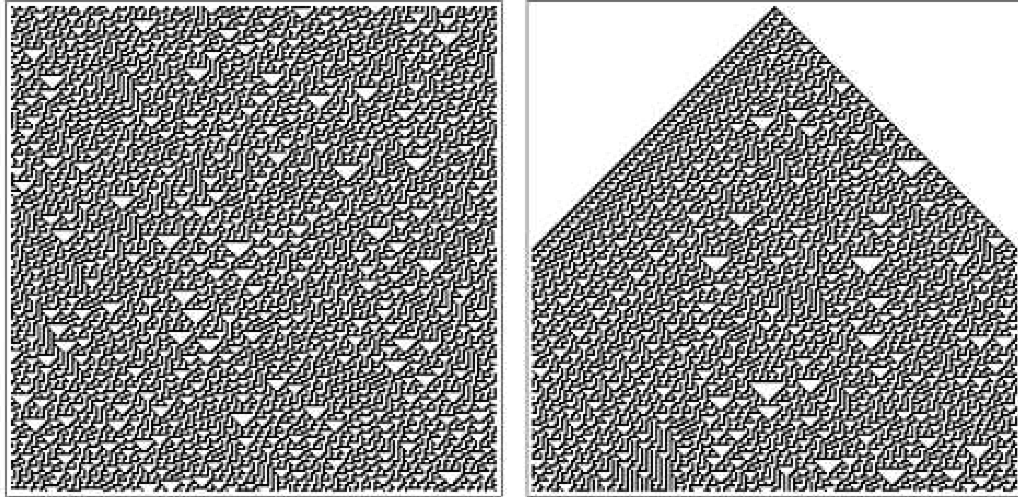
Third is the number of cells in evolution. The implementation accepts any size of string (x),but strings should be => 1024 bits. For example if string is 1024 bit wide, it will be divided to 32 words 32bit wide. If there is no better way to satisfy $h(k+x)=y \wedge h(k+x')=y$ excluding exhaustive search, then that search is around half of $(2^{32})^{32}$. However it may be the case that the amount of cells and the size of cells have different roles in MHF complexity. For example, it is questionable if MHF has same complexity if string 1024 bits wide (x) is processed via 8 words 128 bits wide or 128 words 8 bits wide.

Fourth is the size of the evolution cell. The prototype uses 32 bit cells because it is convenient. The size of the cell may become issue, because of the cell's size / cell's amount interaction in MHF complexity.

Fifth is the size of evolution cycle. For the present prototype cycle is connected to the amount of

cells in evolution. For example if string is 1024 bits wide (32 cells 32bit wide) then evolution will cycle 32 times. The assumption is that complexity propagation of MHF is in worst case scenario same as Wolfram's rule 30 CA. See following picture.

[REF_W nks page 261]



Comparison of the patterns produced by the rule 30 cellular automaton starting from random initial conditions and from simple initial conditions involving just a single black cell. Away from the edge of the second picture, the patterns look remarkably similar.

261

It can be observed that if amount of cells and amount of cycles are equal that ratio is enough for top right cell to impact bottom left one.

The size of the string (x) is not fixed in attached prototype (input file size is not fixed). Because complexity of MHF is directly related to the size of the string (x), some minimum size should be established. Starting point may be 1024 bits (probably going south from there). Maximum depends on the computing resources because complexity is . If required input is larger than available resources then input should be divided and processed separately.

4.0 Summary

In short, MHF takes string as an input. That string is evolved by MAG selection criteria rule [REF_M]. The sample taken from evolved string serves as representation (hash) of original string. Because MHF can go back and forth through evolution, it is trivial to find collisions. But if original string has fixed part (key) it is hard to temper original string and produce the same output sample as original evolved string sample. The same is valid if tampered evolved string is

used to reverse evolution and have the same fixed part (key) as original string. Tempering process is difficult because the MHF output is defined by input and rule only and cannot be expressed or compressed by any other means. Also the divide and conquer strategies does not apply to the MHF because complete knowledge of evolution state is required to step through evolution. That implies NP cost for finding any collision.

5.0 References and Notes

[NOTE_1] It is common argument that every PRNG will loop eventually meaning that in principle at least one pattern exists in PRNG output. But this argument, for example, does not apply for number pi [RNG]. The number pi passes statistical tests but will never enter repetition loop. In similar fashion MAG can add one cell to the evolution array every time when period is reached extending the period of the output to the infinity. Therefore MAG algorithm with specific initial state can be considered as a receipt for making a transcendental number.

[REF_M]
http://www.ecrypt.eu.org/stream/mag.html

[REF_W]

[REF_M_O_V]
Handbook of Applied Cryptography, by A. Menezes, P. van Oorschot, and S. Vanstone, CRC Press, 1996. For further information, see www.cacr.math.uwaterloo.ca/hac

[REF_Goldreich]
From a working draft of Goldreich's Foundation of Cryptography / Chapter 3 Pseudorandom Generators / 3.1.2 A Rigorous Approach to Pseudorandom Generators.

[REF_RNG] Pi scores well in RNG test
http://groups.google.com.au/group/sci.crypt.random-numbers/msg/eb68f0a5c502f741?hl=en

6.0 Source Code

For Unix / Linux OS copy source code to the files in comments. Example `/*Makefie*/` part should go to the Makefile file, `/*mag_hash.h*/` to the mag_hash.h file ...

Then run

$ make

and

$ ./maghash01 filename

where filename is file for hashing. The outcome should be four 32 bit integers (128 bit hash).

```
/*Makefie*/

# maghash Makefile

CC=   gcc
STD=  _GNU_SOURCE
OBJS= maghash01.o get_file_list.o create_hash.o

.c.o:
      $(CC) -c -Wall $(CFLAGS) -D$(STD) $<



#endifmag01: $(OBJS)
      $(CC) $(OBJS) -o maghash01

clean:
      rm -f *.o core

clobber: clean
      rm -f maghash01

#End Makefile



/*mag_hash.h*/

#ifndef mag_hash_h
#define mag_hash_h

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <sys/stat.h>
```

```
struct CellNode
{
	uint32_t Cell;
	struct CellNode *Next;
};

typedef struct CellNode CELLNODE;
typedef CELLNODE * CELLNODEPTR;

extern void get_file_list
(CELLNODEPTR *StartPtr, uint32_t *Carry, uint32_t FileSize, char *, int
HashSizeOutput);

extern void create_hash
(CELLNODEPTR *StartPtr, uint32_t Carry, uint32_t FileSize, int
HashSizeOutput);




/*get_file_list.c*/

#include "mag_hash.h"

void get_file_list
(CELLNODEPTR *StartPtr, uint32_t *Carry, uint32_t FileSize, char *FileName,
int HashSizeOutput)
{
	CELLNODEPTR NewItem, PreviousNode;
	uint8_t Ch;
	int i, j, k, Reminder;
	uint32_t Word, Store[4];
	FILE *FilePtr;

	FilePtr = fopen(FileName, "r");
	Reminder = FileSize % 4;

	NewItem = malloc(sizeof(CELLNODE));
	NewItem->Cell = 0;//creating plain text zeros
	*StartPtr = NewItem;
	PreviousNode = NewItem;
	PreviousNode->Next = *StartPtr;

	for(i = 1; i <  HashSizeOutput; i++)//i=1 one node already 0
	{
		NewItem = malloc(sizeof(CELLNODE));
		NewItem->Cell = 0;

		NewItem->Next = PreviousNode->Next;
		PreviousNode->Next = NewItem;
		PreviousNode = NewItem;
	}

	for(k = 0; k < (FileSize / 4); k++)
```

```
	{
		for(i = 0; i < 4; i++)
			Store[i] = (Ch = fgetc(FilePtr));

		Word =  (Store[3] << 24)|(Store[2] << 16)|(Store[1] << 8)|(Store
[0]);

		NewItem = malloc(sizeof(CELLNODE));
		NewItem->Cell = Word;

		NewItem->Next = PreviousNode->Next;
		PreviousNode->Next = NewItem;
		PreviousNode = NewItem;
	}

	//carry is formed from reminded bytes (if any) and zeros
	if(Reminder > 0)
	{
		for(j = 0; j < Reminder; j++)
			Store[j] = (Ch = fgetc(FilePtr));
		for(j = Reminder; j < 4; j++)
			Store[j] = 0;
		*Carry = (Store[3] << 24)|(Store[2] << 16)|(Store[1] << 8)|(Store
[0]);
	}
	else
		*Carry = 0;
}


/*create_hash.c*/

#include "mag_hash.h"

void create_hash(CELLNODEPTR *StartPtr, uint32_t Carry, uint32_t FileSize,
int HashSizeOutput)
{
	CELLNODEPTR CompA, CompB, Input, Output;
	uint32_t i, j, k;

	Output = *StartPtr;
	Input = Output -> Next;
	CompA = Input -> Next;
	CompB = CompA -> Next;

	for(i = 0; i < FileSize; i++)
	{
		for(j = 0; j < FileSize; j++)
		{
			if( CompA -> Cell > CompB -> Cell)
				Carry = Carry ^ Input -> Cell;
			else
				Carry = Carry ^ (~ Input -> Cell);

			Output -> Cell = Output -> Cell ^ Carry;
```

```
                  CompA = CompA -> Next;
                  CompB = CompB -> Next;
                  Input = Input -> Next;
                  Output = Output -> Next;
            }
      }
      for(k = 0; k < HashSizeOutput; k++)
      {
            printf("%lu\n", Output -> Cell);
            Output = Output -> Next;
      }
}


/*maghash01.c*/

#include "mag_hash.h"

int main(int argc, char **argv)
{
      CELLNODEPTR StartPtr = NULL;
      uint32_t Carry, FileSize;
      int HashSizeOutput = 4;//hash size = 4 * 32bit Word
      struct stat Sbuf;

      stat(argv[1], &Sbuf);
      FileSize = Sbuf.st_size;

      get_file_list(&StartPtr, &Carry, FileSize, argv[1], HashSizeOutput);
      create_hash(&StartPtr, Carry, FileSize, HashSizeOutput);
      return 0;
}
```