A new cryptographic hash function based on the Cellular Automaton Rule 30

Enrico Zimuel *(ezimuel@sci.unich.it)*
Departimento di Scienze - University "G.D'Annunzio" Chieti-Pescara (Italy)

Abstract:

New cryptographic hash functions based on cellular automaton have been proposed in the last years [1,2]. Most of this algorithms are based on the well-known *Merkle-Damgård* construction [3,4]. I investigated the possibility to use a complete different approach in order to generate new cryptographic hash functions with the use of simple cellular automaton in the spirit of NKS ("Simple rules can produce complex behavior" S.Wolfram). In order to prove the quality of the proposed cryptographic hash functions i used the standard *Avalanche* and *Collision* Tests [1,2].

Results:

Results of the conducted experiments have shown that it's possible to use simple cellular automaton like the 2 colors cellular automaton with rules 30 in order to produce good cryptographic hash functions.

Introduction

A hash function H is a transformation that takes a variable-size input m and returns a fixed-size string,which is called the hash value h (that is, h=H(m)).Hash functions with just this property have a variety of general computational uses,but when employed in cryptography the hash functions are usually chosen to have some additional properties.

The basic requirements for a *cryptographic hash function* are:

- the input can be of any length,
- the output has a fixed length,
- H(x) is relatively easy to compute for any given x,
- H(x) is *one-way*,
- H(x) is *collision-free*.

A hash function H is said to be *one-way* if it is hard to invert,where "hard to invert" means that given a hash value h,it is computationally infeasible to find some input x such that H(x)=h. If, given a message x, it is computationally infeasible to find a message y not equal to x such that H(x)=H(y) then H is said to be a weakly *collision-free* hash function. A strongly *collision-free* hash function H is one for which it is computationally infeasible to find any two messages x and y such that H(x)=H(y).

Applications of cryptographic hash functions

A typical use of a cryptographic hash would be as follows: *Alice* poses to *Bob* a tough math problem and claims she has solved it. *Bob* would like to try it himself,but would yet like to be sure that *Alice* is not bluffing.Therefore, *Alice* writes down her solution,appends a random nonce,computes its hash and tells *Bob* the hash value (whilst keeping the solution secret).This way,when *Bob* comes up with the solution himself a few days later, *Alice* can verify his solution but still be able to prove that she had the solution earlier.In actual practice, *Alice* and *Bob* will often be computer programs,and the secret would be something less easily spoofed than a claimed puzzle solution.-The above application is called a *commitment scheme*.

Another important application of secure hashes is verification of *message integrity*. Determination of whether or not any changes have been made to a message (or a file),for example,can be accomplished by comparing message digests calculated before,and after,transmission (or any other event). A message digest can also serve as a means of reliably *identifying a file*.

A related application is *password verification*. Passwords are usually not stored in clear text,for obvious reasons,but instead in digest form. To authenticate a user,the password presented by the user is hashed and compared with the stored hash. For both security and performance reasons,most digital signature algorithms specify that only the digest of the message be "signed",not the entire message. Hash functions can also be used in the generation of *pseudo-random bits*.

A family of hash function

$$H(m) = \{h_1, \ ..., \ h_n\}, \ h_j = \bigoplus_{i=1}^{|m|} CA(r, m, j)$$

$$CA(r, m, j) = \mathrm{Last}[\mathrm{CellularAutomaton}[r, m, j]]$$

where m is the input, H(m) is the output of the hash function, n is the length of the hash function, and r is the rule of the CA.

The *Mathematica* code:

```
hashCA[rule_, text_, len_] :=
  (BitXor @@@ CellularAutomaton[rule, text, len])[[2 ;; len + 1]]
```

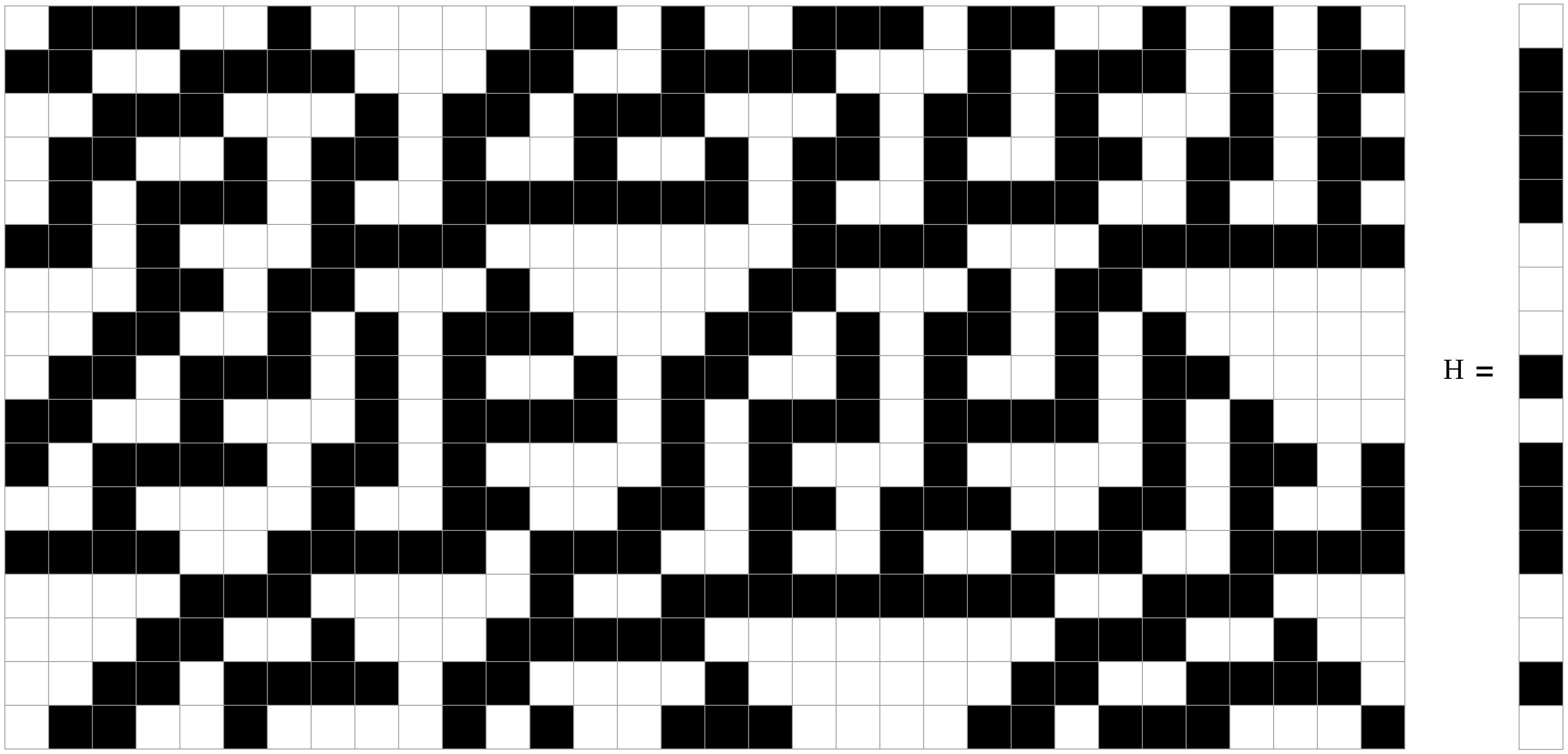
Why I used the CA rule number 30 in my tests?

- the rule 30 produces *pseudo-random* numbers;
- the rule 30 is *not reversible*;

Example

m = {0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0}

n = 16, | m | = 32, **r** = 30 (rule of the CA), **H** (***m***) = {1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0}



Testing the "security" of the hash function

In cryptography, the *Avalanche effect* refers to a desirable property of cryptographic algorithms, typically block ciphers and cryptographic hash functions. The *avalanche effect* is evident if, when an input is changed slightly (for example, flipping a single bit) the output changes significantly (eg, half the output bits flip). The *Strict Avalanche Criterion* (SAC) is a property of boolean functions of relevance in cryptography. The SAC builds on the concepts of completeness and avalanche and was introduced by *Webster* and *Tavares* in 1985. Definition: a function is said to satisfy the *strict avalanche criterion* if, whenever a single input bit is complemented, each of the output bits should change with a probability of one half.

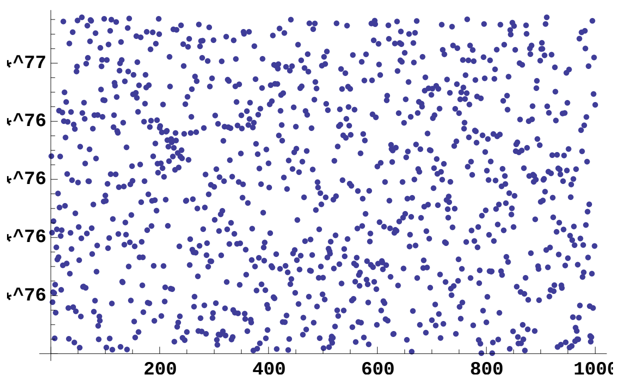
Test the avalanche effect of the *hashCA*:

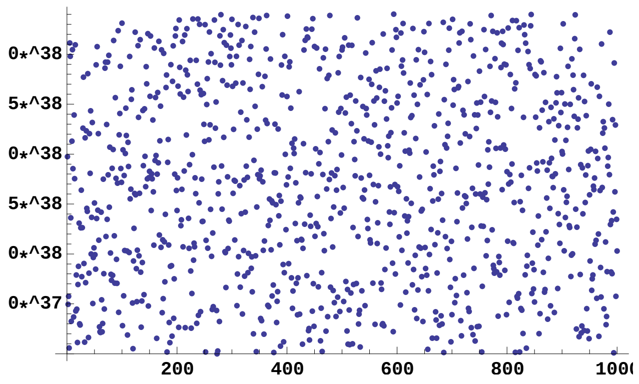
```
sigma[x_, y_] := Mean[BitXor[x, y]]

changeOneBit[a_] := Table[BitXor[a, IntegerDigits[2^(i - 1), 2, Length[a]]], {i, Length[a]}]

avalanche[t_, len_, num_] := Table[sigma[hashCA[30, t[[i]], 128],
   hashCA[30, changeOneBit[t[[i]]][[j]], 128]], {i, 1, num}, {j, 1, len}]
```

I generated 1000 random inputs of 128 and 256 bits and for each of these inputs i generated all the possible pairs of inputs with only one bit of difference.

length	mean	standard deviation	variance	min	max
128	0.500	0.044	0.002	0.312	0.679
256	0.499	0.044	0.002	0.312	0.703

Random inputs of 256 bit (1000 values)   HashCA output of 128 bit (1000 values)





Collision Test

```
findCollision[h_] := {#[[1, 1]], Last /@ #} & /@
  Select[Split[Sort[Table[{h[[i]], i}, {i, Length[h]}]], #[[1]] == #2[[1]] &], Length[#] > 1 &]

m = Table[Table[Random[Integer], {32}], {2^16}]

h = Table[hashCA[30, m[[i]], 128], {i, 1, 2^16}];
```

I generated *65536* hash codes of 128 bit from *65536* different inputs of 32 bit. I discovered only 8 collisions with the CA rule number 30. Moreover i investigate the possibility to use different CA rules for the generation of the *hashCA*. These are the results over 65536 different random inputs of 32 bit.

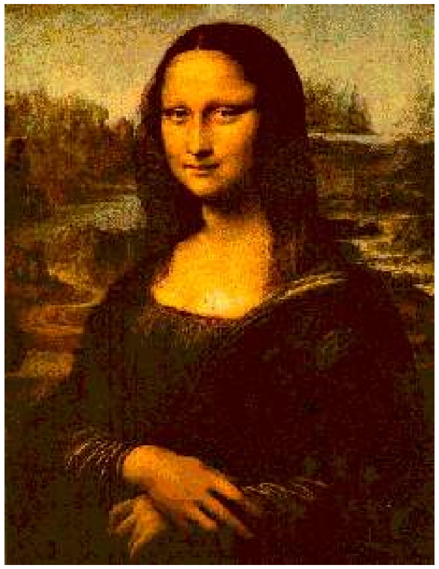
CA Rules	22	30	45	54	73	75	86	89	101	110	122	124	126	135	137	149
Collisions	9466	8	9	4972	31401	9	8	10	20	630	11579	646	35951	18	634	18

A nice application

What's the hash code of the *Monna Lisa* (or *La Gioconda*)?

```
m = hashCA[30, Flatten[IntegerDigits[Import["monnalisa.jpg", "RawData"], 2], 128];
```

m =



H(m) = 4 619 125 044 072 654 358 130 582 367 647 205 982

References

[1] M.J. Mihaljevic, H. Imai, *A family of fast dedicated one-way hash functions based on linear cellular automata over GF(q)*, IEICE Trans. Fundamentals, vol. 82-A, no.1, pp.40-47, Jan. 1999

[2] M. Bedau, R.E. Crandall, and M. Raven, *Cryptographic hash functions based on Artificial Life*, manuscript, 2004

[3] R.C. Merkle. *A Certified Digital Signature*. In Advances in Cryptology - CRYPTO '89 Proceedings, Lecture Notes in Computer Science Vol. 435, G. Brassard, ed, Springer-Verlag, 1989, pp. 218-238.

[4] I. Damgård. *A Design Principle for Hash Functions*. In Advances in Cryptology - CRYPTO '89 Proceedings, Lecture Notes in Computer Science Vol. 435, G. Brassard, ed, Springer-Verlag, 1989, pp. 416-427.