Finding Efficient Nonlinear Functions by Means of Genetic Programming

Julio César Hernández Castro[1], Pedro Isasi Viñuela[2], and
Cristóbal Luque del Arco-Calderón[2]

[1] Computer Security Group, Computer Science Department,
28911 Leganés, Madrid, Spain
jcesar@inf.uc3m.es

[2] Artificial Intelligence Group, Computer Science Department,
28911 Leganés, Madrid, Spain
{isasi,cluque}@ia.uc3m.es

Abstract. The design of highly nonlinear functions is relevant for a number of different applications, ranging from database hashing to message authentication. But, apart from useful, it is quite a challenging task. In this work, we propose the use of genetic programming for finding functions that optimize a particular nonlinear criteria, the avalanche effect, using only very efficient operations, so that the resulting functions are extremely efficient both in hardware and in software.

1 Introduction

The design of highly nonlinear functions is useful for a number of different applications, from the construction of hash functions for large databases to the development of different cryptographic functions (pseudorandom functions, message authentication codes, block ciphers, etc.)

For many of these applications, a desirable and conflicting property is efficiency. It is not difficult to design highly nonlinear functions, nor very efficient functions, but finding algorithms with both properties is quite a challenging task.

1.1 Compression Functions

A function $F : Z_2^m \rightarrow Z_2^n$ is said to be a compression function if *m*>*n*. Different compression functions are extensively used in diverse cryptographic primitives, such as pseudorandom number generator or hash functions (used for digital signatures and message authentication). The main advantage of this approach is to allow the simple processing of variable-length input, following the recurrence relation:

$$
\begin{aligned}
&M = M_0 M_1 \ldots\ldots\ldots M_n \\
&h_0 = IV \\
&h_{i+1} = F(M_i, h_i) \\
&h(M) = h_{n+1}
\end{aligned}
$$

This scheme is very versatile, as it can be found with some minor differences in block cipher chaining and cryptographic hash functions, and it can also be used to produce pseudorandom numbers. We will focus our search to a quite general kind of compression functions, where *m=2n.*

2 The Avalanche Effect

Nonlinearity can be measured in a number of ways or, what is the same, has not a complete unique and satisfactory definition. In this work we focus our attention in a special property, named avalanche effect because it tries to reflect to some extend the intuitive idea of high-nonlinearity: a very small difference in the input producing a high change in the output, thus an avalanche of changes. Mathematically:

$$\forall x, y \mid H(x,y) = 1 \;\; H(F(x), F(y)) \approx \frac{n}{2}$$

So if *F* is to have the avalanche effect [1], the Hamming distance between the outputs of a random input vector and one generated by randomly flipping one of the bits should be, on average, *n/2*. That is, a minimum input change (one single bit) produces a maximum output change (half of the bits).

This definition also tries to abstract the more general concept of independence of the output from the input. Although it is clear that this is impossible to achieve (a given input vector always produces the same output) the ideal *F* will resemble a perfect random function where inputs and outputs are statistically unrelated. Any such *F* would have perfect avalanche effect, so it is natural to try to obtain such functions by optimizing the amount of avalanche.

3 Genetic Programming

Genetic Programming [2] is a method for automatically creating working computer programs from a set of high-level statements of a given problem. This is achieved by breeding a population of computer programs using the principles of Darwinian natural selection and other biologically inspired operations that include reproduction, sexual recombination (crossover), mutation, and possibly others. Starting from a initial population of randomly created programs derived from a given set of functions and terminals, populations gradually evolve, giving birth to new, more fitted individuals.

This is performed by repeating the cycle of fitness evaluation, Darwinian selection and genetic operations until a certain ending condition is met. Each individual (or program in the population) is evaluated to determine how fit is at solving a given problem, and then programs are selected probabilistically from the population according to their fitness values for being applied the rest of genetic operators. It is important to note that, while fitter programs have higher probabilities of being selected, all programs have a chance. After some generations, a program may emerge that solves, completely or approximately, the problem at hand.

Genetic Programming combines the expressive high-level symbolic representations of computer programs with the learning efficiency of genetic algorithms. Genetic Programming techniques have been successfully applied to a number of different problems: apart from classical problems such as function fitting or pattern recognition, where other evolutionary computation techniques also work fine, they have even produced results that are competitive with humans in some non-trivial tasks as designing electrical circuits[3] (some of which have been patented) or at classifying protein segments[4].

4 Implementation Issues

We have used the lilgp genetic programming system, available at http://garage.cps msu.edu/software/lil-gp/lilgp-index html, but a number of modifications were needed for our problem.

Firstly, we need to define the set of functions: This is critical for our problem, as they are the building blocks of the algorithms we would obtain. Being efficiency one of the paramount objectives of our approach, it is natural to restrict the set of functions to include only very efficient operations; so the inclusion of the basic binary operations as **rotd** (right rotation), **roti** (left rotation), **xor** (addition mod 2), **or** (bit wise or), **not** (bit wise not), and **and** (bit wise and) are an obvious first step. Other operators as the **sum** (sum mod 2^{32}) are necessary in order to avoid linearity, being itself quite efficient. Another interesting operator introduced was **kte**, an operation that, whatever its input, returns the 32-bit constant value 0x9e377969, which are the most significant digits of the expression of the golden ratio in hexadecimal notation).The idea behind this operator was to provide a constant value that, independently from the input, could be used by the aforementioned operators to increase nonlinearity , and idea suggested by [5].

The inclusion of the **mult** (multiplication mod 2^{32}) operator was not so easy to decide, because, depending on the particular implementations, the multiplication of two 32 bit values could cost up to fifty times more than an xor or and operation, so it is relatively inefficient, at least when compared with the other operators used. In fact, we did not include it at first, but after extensively experimentation, we conclude that its inclusion was beneficial because, apart from improving non-linearity; it at least doubled and sometimes tripled the amount of avalanche we were trying to maximize, so we finally include it in the function set.

The set of terminals in our case is easy to establish; as said above, we are focusing our attention in compression functions where m=2n, so the input will be formed by two 32 bits integers ***a0*** and ***a1***, and these will be the branches of the function trees that the genetic programming algorithm will construct, with functions from the function set in the nodes.

The fitness of every individual (algorithm or function) was evaluated by generating 1024 64-bit random vectors, then randomly flipping one of the bits and calculating the Hamming distance over their outputs. For each of these 1024 experiments a Hamming distance between 0 and 32 was obtained and the fitness of the given function was the observed average Hamming distance.

Although this is a quite natural way of measuring the avalanche effect as defined in the introduction, some additional explanations are needed. Obviously; the ideal observed value should be *32/2=16*, so a more natural approach for the fitness function will be not simply this average but is deviation from the optimum value of 16, that is *|16-average|* Anyway, after some experiments we observed that, at least for the depths studied in this work, the resulting functions fitness were far and below the 16 optimum value, so simply using the average number of changes as the fitness function to maximize worked perfectly well.

When using genetic programming approaches, it is necessary to put some limits to the depth and to the number f nodes the resulting trees could have. We preferred to vary only the depth and not to put any limitation (apart from the depth itself) to the number of nodes possibly used.

We selected a population size of 100 individuals, a crossover probability of 0.8, which produced better results than the default 0.9 probability proposed, and an ending condition of reaching 500 generations. In many cases, ten different runs were performed, each one seeded with the 6 most significant digits of the expression (314159)^i. When only one run was possible, the seed was 314159.

5 Results

Selected functions are shown in the Appendix.

Table 1 below summarizes some of the results.

Basically, Table 1 shows that, by increasing the allowed depth of the function trees, better avalanche effects could be obtained. This is intuitively clear, but not trivial to get. The idea that simply adding more and more complexity (more branches with functions and terminals) to any tree will increase the avalanche effect of the function can be easily dismounted by a simple inspection to the first two individuals shown at the Appendix.

After all the experiments, we can conclude we have found one super-individual, shown in the appendix, that with only a depth of 5 and a little number of nodes is capable of contesting in terms of avalanche with much more complex rivals.

Furthermore, this highly fitted function has many points in common with some of the fittest individuals for every depth (as shown at the end of the appendix), so, in a way, this is also revealing us a new promising method of construction or a new kind of function design. There are other very fitted functions which have great similitude with the aforementioned, but we cannot include them in the appendix due to lack of space.

Table 1. Avalanche effect as a function of depth. Averages exclude the maximal and minimal values. For some values there are only one result

\Avalanche Depth\	Maximun	Average	Minimun
2		8.8696	
3	9.4424	9.0725	8.8174
4	10.1396	9.5619	9.1152
5	11.2148	9.9919	9.6533
6	11.1479	10.2296	9.6621
7	11.3232	10.7037	10.3145
8		11.5200	
9		10.9482	
10		11.2310	
11		11.3513	
12		11.8284	
13		11.5164	
14		10.9707	
15		12.4451	
16		11.6604	

6 Conclusions

The average avalanche effect on Table 1 steadily increases when the depth is increased; so a natural question is which is the minimum depth at which any algorithm exists that reaches the perfect avalanche of 16.

For trying to guess that, we performed a series of additional searches in deeper depths (but only one run, as they are much more time consuming) until reaching depth 16. By the slow increase in the avalanche levels observed, our current best guess is that this could perhaps be achieved at depth 20. This makes that some questions arise naturally as, for example, would it better to dig in some such deep trees, with the associated combinatorial explosion (for example in the number of nodes used) or is it more sensible to search for short, faster functions and apply then twice? What would it better, the best function we can find with this approach at level 12 or the best function found at depth 6, with the same amount of resources, applied twice?

Many other questions remain. One of the more interesting is if it would be possible to add to every function in the function set a weight or some other measure of its complexity or efficiency (for example in number of instructions needed at the processor level, microseconds taken to perform, etc) and allow the genetic programming technique itself to decide whether it is interesting to use that particular function or not in terms of avalanche effect introduced for a given cost. This and other questions and extensions to this work are currently under consideration.

Acknowledgements

Supported by the Spanish Ministerio de Ciencia y Tecnologia research project TIC2002-04498-C05-4

References

[1] Feistel, H.: Cryptography and Computer Privacy. Scientific American, 228 (5) 15-23, 1973

[2] Koza, J.: Genetic Programming. In: Encyclopedia of Computer Science and Technology, v.39, 29-43, 1998

[3] Koza, J., et. al. Automated Synthesis of analog electrical circuits by means of genetic programming. In IEEE Transactions on Evolutionary Computation 1(2), 1997

[4] Koza, J., Andre, D.: Automatic discovery of protein motifs using genetic programming. In Evolutionary Computation: Theory and Applications. World Scientific Publications, 1996

[5] Wheeler, D., Needham, R.: TEA, a Tiny Encryption Algorithm. In: Proceedings of the 1994 Fast Software Encryption Workshop, and at http://www ftp.cl.cam.ac.uk/ftp/papers/djw-rmn/djw-rmn-tea html

Appendix: Experimental Results

```
Depth 5 Run #4 fitness: 11.2148
TREE:
 (mult (kte (rotd a0))
       (rotd (sum (roti (xor a0 a1))
                  (xor a0 a1))))
```

```
Depth 14: fitness: 10.9707
TREE:
 (xor (sum (xor (rotd (mult (and a0
                                 (mult (mult (mult a1
                                                   (mult a1 a1))
                                             (rotd a0))
                                       (mult a0
                                             (mult a1 a0))))
                            (rotd (rotd (xor a1
                                             (rotd (xor a1
                                                        (rotd (sum a0 a1))))))))
                (xor (xor a0 a1)
                     (sum a0
                          (rotd (sum a0 a1)))))
```

```
         (sum (xor (rotd (roti a1)) a0)
             (sum a0 a1)))
     (xor (sum (xor (sum (mult (mult a1 a1)
                           (sum (sum a0 a0)
                               (xor a1
                                   (xor a0
                                       (sum a0
                                           (roti a1))))))
                      (xor a0
                          (sum a0 a1)))
                  (xor a0 a1))
             (not a0))
         (mult (sum a0 a1)
              (mult a0
                   (mult a1 a0)))))
==========================================================

    Run #4 fitness: 11.0957
TREE:
 (mult (kte (xor (roti (roti a1)) a1))
      (rotd (sum (sum (rotd a0)
                   (rotd a1))
               (roti (xor a0 a1)))))


    DepthMax=7
==========
Run #4 fitness: 11.3232
TREE:
 (mult (kte (rotd a0))
      (rotd (sum (rotd (sum (sum a0 a0)
                        (roti (rotd a1))))
               (rotd (sum (sum (rotd a0)
                            (rotd a1))
                        (rotd a0))))))
```