Breaking the RSA with Reversible Multiplier Circuits

Gustavo Estrela de Matos 925003559

April 28, 2016

1 The Problem

RSA is an algorithm for data encryption used in many different security system and considered one of safest nowadays. This algorithm is based on the difficulty of the problem of factorizing the product of two large prime numbers, known as Integer Factorization Problem, which has no polynomial algorithm to solve (not considering that there are some polynomial algorithms for this problem in quantum computers).

1.1 How to Break RSA Security

The RSA algorithm has two different keys, a public and a private one. The private key is composed by two large prime numbers and the public key is composed by the product of these two primes, therefore, if we are able to factorize the public key we can get the private key, which allows us to decrypt the message.

As mentioned before, the problem of factorizing integers is a hard problem and until today no one was able to give a polynomial algorithm to solve it. In this project we focus on creating a heuristic to find the prime factors that compose the public key of the RSA and crack the code.

The heuristic we are going to implement is based on the creation of a circuit that performs a reversible multiplier. This circuit is going to be composed of toffoli gates, which is a gate composed of control points and a controlled point. In a toffoli gate the open (closed) control point is satisfied if the input bit is 1 (0) and when all the control points are satisfied we toggle the controlled point. The circuit of a reversible multiplier is simply a set of toffoli gates.

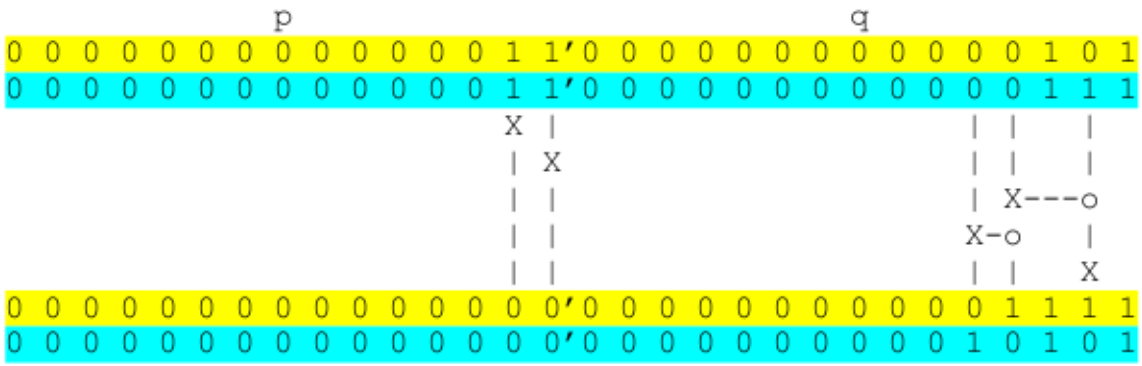

Figure 1: Example of a reversible Multiplier composed of toffoli gate. This multilpier is composed of five toffoli gates and it is capable of multiplying 3 and 5 (in yellow) and also 3 and 7 (in cyan). *Figure available in http://courses.cse.tamu.edu/daugher/misc/PPP/homeworks/420-16spring-project.pdf*

1.2 Goal of this Project

In this project we intend to use search algorithms that we were exposed in class to solve the problem of finding the best multiplier possible, i.e the multiplier that correctly multiplies

the greatest number of pairs of primes correctly. To do that we are going to use two search methods: genetic algorithms and local beam search. We are going to implement these two heuristics showing how we set up their parameters and also compare their efficiency.

1.3 Restrictions and Limitations

It should be a know fact for the reader that the number of primes are infinite, therefore, as we explained before we do not intend to create a universal circuit capable of multiplying any pair of primes. The first limitation imposed in our algorithm is that toffoli gates and multilplier circuits only handle up to 30-bits number. Another limitation is that the input of a multiplier should be a 30-bits number $\overline{p}'\overline{q}$ where $\overline{p}$ and $\overline{q}$ are 15-bits representation of p and q respectively, and $p \leq q$, and this is what guarantee this circuit to be reversible (otherwise order changing would create a different solution).

In the development of this project we also had to restrict the number of primes that we were considering for multiplication, so then we would be able to see more effective results of both genetic algorithms and local beam search. Our limitation is that we are only considering the first fourty prime numbers, i.e primes between 2 and 173 (we can round up to 8 bit primes).

2 Solving the Problem

As mentioned before we implemented used two different approaches to solve this problem: genetic algorithms and local beam search. In this section we describe how we implemented the solution using these two different approaches and how we set up parameters for them based on experimentation.

2.1 Solution Evaluation

The first problem we faced in this project was to find a fast way to compute the fitness or the score of a solution. The first approach we used was to run the multiplier being tested to all possible combinations of 30-bits prime numbers. This approach showed to be too slow, which is not good because for both genetic algorithm and local beam search, the fitness function is called constantly.

To solve this problem we decided to add to our search method a parameter that stores the number of prime pairs (randomly chosen) that should be multiplied to calculate the fitness of a multiplier. This does not solve entirely the evaluation problem because now, since there are a lot of different pair of primes, it would be hard to distinguish a multiplier that works for one input from one that does not work for any input; that happens because the probability of choosing that one input is too small. Finally, we added a limit on the number of primes we consider for this problem, as we mentioned in the section 1.3.

2.1.1 Bit Entropy

Before explaining how we implemented both solutions present on this project we should understand the concept of entropy and how we can use it to guide our algorithms on the search for

the best solution. In the context of this project we can define that the entropy of a bit is the inverse of the amount of information that this bit has, and by information we mean how certain are we that this bit needs to be toggled - or, complimentary, not toggled - from input to output.

For instance, let's consider that the i-th bit has been correct for 90% of the evaluations, then we can say that probably this bit will be correct for other pair of primes, and similarly, we can say that this bit will not be correct if it wasn't correct in the past evaluations. Knowing that, we can use this high informative bits to place our toffoli gates on bits that are normally incorrect and place the control points on bits that are normally correct.

2.2 Genetic Algorithm

The genetic algorithm approach consists of creating an initial set of multipliers, in this context called individuals, each one evaluated for some fitness function, and a sequence of iterations with crossovers and mutations. Each iteration generates a new generation of individuals that, in theory, should converge to a better fitting individual, which in this case is the individual that correctly multiplies the greates number of pairs of primes.

2.2.1 Population Start

Our first attempt to start the population

2.2.2 Crossover

We tried two different approaches for crossing over two individuals I_1 and I_2, both of them considering the relative fitness:

- Random crossing: for every gate in I_i, add this gate with probability $RelativeFitness_1(I_i)$
- Column random crossing: for every gate controlling the $j-$th bit of I_i, add all these gates to the child with probability $RelativeFitness_1(I_i)$. If the sum of gates in the j-th column of both individuals is less than 5, add all these gates to the child with probability 0.5. After some tests we concluded that the second approach was more effective than the first one.

2.2.3 Mutation

The mutation happens right after the crossover with probability of 0.5, and it exploits the Bit Entropy as we defined in section 2.1.1. The mutation only adds new gates to the child multiplier, placing the gate on a bit that is frequently incorrect with up to 6 (random) control points placed in frequently correct bits. We also tried totally random gates, but that haven't shown to be effective.

2.2.4 Fitness Function

The fitness function used in the genetic algorithm is simlpy the number of correct multiplications seen when evaluated the individual.

2.2.5 Parameter Settings

The genetic algorithm has several parameters including: population size, number of primes used to evaluate and probability of mutation. The most important parameter to set up between those is the number of primes to evaluate, because this number has big impact in the time needed to create a new child in a crossover, and, since we want the algorithm to have as many iterations as possible (so we can get evolved individuals) it is important to choose this parameter in a way that the evaluation of a new child happens fast.

2.3 Local Beam Search

The Local Beam Search algorithm is similar to a random walk with k different nodes and a function that we are going to call fitness function as we did for the genetic algorithm. The algorithm starts with a population of k empty multipliers, i.e there are no gates in the multipliers and for each step we do the following:

- Generate a number of successors for every k node;
- Choose the best k nodes between successors and current nodes all and repeat.

2.3.1 Fitness Function

With a few tests we realized that when we look into a node successors, many of them have the same score when considering the number of correct multiplications. Also, if we consider the number of correct bits we end up guiding the algorithm to solutions that, on average, has a good number of correct bits but not necessarily multiplies two primes correctly. Therefore, we decided to use a fitness function that has both information:

$$FitnessFunction_2(N_i) = 60 * FitnessFunction_1(N_i) + BitScore(N_i)$$

2.3.2 Choosing Next Step

Every iteration we open a certain number (set as a parameter) of successors and to do that we copy all gates from the current node and we generate the new multiplier by either:

- Adding a new gate with probability $3/5$;
- Adding a new control point with probability $1/5$;
- Removing a gate with probability $1/5$

To accomplish these actions we use again the Bit Entropy concept we presented in section 2.1.1 in a similar way as we did for the genetic algorithm. When we add a gate we should choose as the control to be on a bit that is frequently wrong and when adding a control point we should choose a bit that is frequently correct.

2.4 Parameter Setting

The Local Beam Search algorithm we implemented have some parameters we can set up, including the same parameter of number of primes to evaluate as we had for the genetic algorithm. We should note that the local beam search generates way more multipliers in an iteration, therefore, there are more calls to the evaluation function and we should set these evaluation to test less prime numbers than we did for the genetic algorithm.

3 Results

3.1 Sample Run

3.2 Analysis

4 Conclusion

4.1 Lessons Learnt

4.2 Future Research