Pollard rho algorithm

Technology, Information Technology

Pollard’s Rho algorithm Pollard’s Rho algorithm is an algorithm which requires a computation driven solution which is well addressed beneath a multi-core architecture. As the number of digits in number increases, more cores are needed to factorize the number. The most significant application of this algorithm is with Discrete Logarithmic Problem (DLP). It is a probabilistic algorithm which works by sequential iterations of a random quadratic function. Random coefficients are selected for a standard quadratic function which generates numbers, which are reduced modulo the composite number. Sequential iterations of the random quadratic function based on a randomly selected initial number generate a series which starts looping after a specific point. If two points are on the same position of the loop, they are congruent to each other modulo a factor of the composite. This is because the loop is actually a subgroup generated by the initial element as the identity and with the random function as the group operation (Hankerson, 20). Thus, if two points are in the same equivalence class in the subgroup, they are equivalent to each other modulo the order of the subgroup, which divides the order of the entire group, which is the composite number. Subtraction yields a multiple of the order of the subgroup and taking the greatest common divisor of this and the modulus yields a factor of the modulus. It is important to note that not every function and initial point pair will yield a subgroup. This algorithm is probabilistic and thus does not find a factor every time. However, bounds can be placed on it that forces a restart after a timeout period. If the algorithm works the first time, it is very fast; at worst case it runs at the same speed as trial division does. Pollard's Rho algorithm [6, 8] is a heuristic, i. e., the running time of this algorithm cannot

be rigorously analyzed. It is said to work very quickly when the number to be factorized has small factors, i. e., typically of the size of 10-12 digits. It is also very parallelizable. This is, hence, our choice of algorithm for implementation. Algorithms such as trial division and sieve of Eratosthenes take a lot of time because they use the brute-force approach and are only suitable for finding factors of size 3-5 digits. The other integer factorization algorithms such as the General number field sieve and its open-source implementations (MSIEVE), compute factors of any size. But these algorithms are preferred when it is known that the number to be factorized has large integers. This is because these algorithms take the same amount of time to find either large or small factors Thus the Pollard's Rho algorithm functions like this. Suppose we want to generate some numbers; say k numbers, (x1… xk), the challenge is we cannot. The following best step is to generate the random numbers one at a time while checking two successive numbers. We will keep repeating this step, and we will be successful luckily. for a while and hopefully we are going to get lucky. The function f which will be used will generate pseudo random numbers. That is, we will apply f and it will create numbers which " feel and look" random. Not all functions can generate the pseudo random numbers, but one such function which has the enigmatic pseudo random feature is F (x)= x2+amodN, Where a is generated by a random number generator. We begin with x1= 2 or any other number. Then we do x2= f(x2), x3= f(x2), … In this case the general rule is x n+1= f (xn). We can begin with a naive algorithm and start to fix the issues as we proceed. Algorithm scheme x:= 2; while(.. exit criterion ..) y:= f(x); p:= GCD(| y - x | , N); if(p > 1) return " Found factor: p"; x:= y; end return "

Failed. :-(" Lets begin by taking N= 55 and f (x)= x2+2mod55. Xn Xn+1 GCD(| xn? xn+1|, N) 2 6 1 6 38 1 38 16 1 16 36 5 From the table it can be conclude that this can work in many occasions. However, in some cases, it runs into an infinite loop since the function f cycles. As soon as this happens, we keep on doing the same with the same pair of values xn and xn+1 without a stop. For instance, we can create a pseudo random function f that generates a sequence such as 2, 10, 16, 23, 29, 13, 16, 23, 29, 13 ... For this case, we carry on with the cycle without finding a factor. The question is how can we detect that cycling has occurred? There are two solutions to our help: The first solution is to keep all the number we have seen so far; x1,..., xn and observe if xn= xk for a previous value k