Speeding up the Pollard's Rho algorithm

Silje Christensen
christensen@umail.ucsb.edu
&
Simen Johnsrud
simejo@umail.ucsb.edu

***Abstract*—In order to understand the threat model of elliptical curve cryptographic schemes it is important to have the knowledge of how you can attack it. Pollard's Rho algorithm is one possible way to solve the elliptic curve discrete logarithm problem (ECDLP)[1]. The algorithm was introduced in 1978 [2], and during the past four decades there have been several modifications to the original implementation to reduce the time and storage complexity.**

In our paper we aim to give the reader an overview of the Pollard's Rho algorithm in order to understand how we can speed it up. We will start by looking at the basic concepts of it, and then study the existing methods which can be applied to reduce the complexity.

Keywords*—*Pollard's Rho algorithm, speeding up, elliptic curve discrete logarithm problem, cycle-finding algorithm, Brent.

I. Overview

The Pollard's rho algorithm was introduced by the British mathematician John Pollard[3]. The origin of the name "Pollard's Rho" is based on the similarity in appearance between the Greek letter ρ (rho) and the sequence that the algorithm creates, as shown in Figure 1. The tail length (t) is the amount of steps before we enter a cycle, and the cycle length (s) is the length from the start of the cycle until we get a collision.

The input to the algorithm is the points P of the elliptic curve over the field $\mathbb{F}_q$. These points are divided into L sets, called branches, of approximately the same size, by using a partition function f. If f is a random function, we can make some assumptions about how long time it will take before we start the cycle, i.e. the tail length t and the expected cycle length s. We have that $t \approx \sqrt{\frac{\pi n}{2}}$ terms and $s \approx \sqrt{\frac{\pi n}{8}}$ [1], where n is the modulo. The output is the discrete algorithm $l = log_p Q$, which we will describe in the next sections.

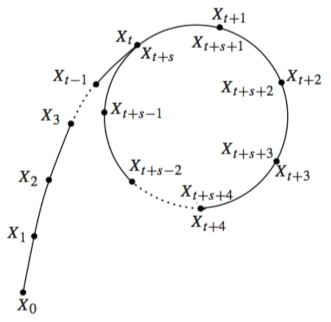

Fig. 1. Rho shape of the sequence X_i in Pollard's Rho algorithm.[1]

A. Context

Recall that when we want to solve the elliptic curve discrete logarithm problem (ECDLP), we have an elliptic curve (E) defined over a finite field $(\mathbb{F}_q)$, a point P of order n, a point Q which is one of the points in P, and an integer l in the range $[0, n-1]$ such that $Q = lP$. The integer l is called the discrete logarithm of Q to the base P, denoted $l = log_P Q$. [1]

B. Core idea

Pollards rho algorithm finds distinct pairs (c', d') and (c'', d'') of integers modulo n such that

$$c'P + d'Q = c''P + d''Q$$

which gives us

$$(c' - c'')P = (d'' - d')Q = (d'' - d')lP$$

and then

$$(c' - c'') \equiv (d'' - d')l(mod(n))$$

and from there we can find $l = log_p Q$ by computing

$$l = (c' - c'')(d'' - d')^{-1}(mod(n))$$

The algorithm finds the pairs (c', d') and (c'', d'') by defining an iterative function. When a point $cP + dQ$ occurs for the second time, we have a so called collision, and the pair has been found. A collision can be found by using Floyd's cycle-finding algorithm, also called the "tortoise and hare" cycle detection. This pointer algorithm uses two pointers which move through the sequence at different speed to find the first two points in the sequence with the same value[4].

II. Complexity

The time complexity of Pollard's rho is originally approximately $\sqrt{n}$, where n is the modulo, in terms of the input size in bits $O(2^{k/2})$.[9]

III. Algorithm

Algorithm 1 is a pseuducode of the Pollard's Rho algorithm for the ECDLP on a single processor[1], and should give the reader a better insight of how the algorithm works.

Algorithm 1 Pollard's rho algorithm for the ECDLP on a single processor

```
1:  procedure Pollard's Rho
2:      Select the number L of branches (e.g., L=16 or L=32)
3:      Select a partition function $H : \langle P \rangle \rightarrow \{1, 2, ..., L\}$
4:      for $j$ from 1 to L do
5:          Select $a_j$, $b_j \in_R [0, n-1]$.
6:          Compute R$_j$ = $a_j P$+$b_j Q$.
7:      end for
8:      Select $c'$, $d' \in_R [0, n-1]$ and compute $X' = c'P + d'Q$
9:      Set $X'' \leftarrow X'$, $c'' \leftarrow c'$, $d'' \leftarrow d'$.
10:     while $X' \neq X''$ do
11:         Compute $j = H(X')$. Set $X' \leftarrow X'+R_j$, $c' \leftarrow c' + a_j$ mod n, $d' \leftarrow d' + b_j$ mod n.
12:         for $i$ from 1 to 2 do
13:             Compute $j = H(X')$. Set $X'' \leftarrow X'' + R_j$, $c'' \leftarrow c'' + a_j$ mod n, $d'' \leftarrow d'' + b_j$ mod n.
14:         end for
15:     end while
16:     If $d' = d''$ then return ("failure")
17:     Else compute $l = (c' - c'')(d'' - d')^{-1}$ mod n and return $(l)$
18: end procedure
```

IV. Speeding up

The original algorithm created by Pollard has been modified several times in order to speed up the time complexity. We will take a look at some of the major improvements on the algorithm.

A. Brent's cycle finding method

In 1980 Richard P. Brent made a variant of the cycle-finding algorithm. In this version we are searching for the smallest power of two (2^i) that is larger than both t and s, were $i = 0, 1, 2, ...$ The idea is to compare 2^{i-1} with each following value up to 2^i until a collision occurs, i.e. the hare and tortoise have the same value.

By doing this he gained two advantages compared to the initial Floyd's cyclic finding algorithm. First, it finds the correct length of the cycle length (s) directly, and each step includes only one evaluation of the function that maps the set, rather than three evaluations.[5] If the cost of doing comparisons are low, the algorithm may actually improve the time complexity of the cycle finding algorithm by 30%.[6]. According to Brent, this would speed up the Pollard's rho algorithm by 24%.

B. Parallelized Pollard's Rho

Another way to speed up the algorithm is to run it on multiple processors. The naïve way to speed up Pollard's Rho when you got M processors is to run the algorithm independently on each processor with different randomly chosen starting points X_0 until one of the processors finds a solution. The expected number of elliptic curve operations performed by each processor before one terminates is about $3\sqrt{n/M}$, where n is the modulo. As a result of this observation the expected speedup is a factor of $\sqrt{M}$. [1]

This can be improved to a factor of M by letting the sequences generated by the different processors collide with each other. To be more specific, the M processors choose it's own independently starting point X_0, but they all share the same iterating function f to compute the next X_i points. As a result of this, if two sequences from different processors collide then the two sequences will be identical from that point on.

C. Automorphism

By creating a group automorphism, which loosely speaking is the symmetry group of the object, we can make further improvement to the Pollard's Rho algorithm. In other words, this is a way of mapping the object to itself while preserving all of its structure. It is defined by $\Psi : \langle P\rangle \rightarrow \langle P\rangle$, where $P \in E(\mathbb{F}_q)$.

The equivalence class $[R]$, where $R \in P$, will consist of all of the points satisfying $R_1 = \Psi^j(R_2)$ where j is an integer in the range $[0, t-1]$. Here, t is the order of the group and hence, the size of the equivalence class.

How will this affect the algorithm? The idea is to find a function that runs over the equivalence classes rather then the points one by one, and replace it in the Pollard's Rho algorithm. Since Ψ is a automorphism group, $\Psi(R) = \lambda(R)$ for all points R in $\langle P\rangle$, where λ is an integer between zero and $n-1$. For each equivalence class we create a unique representation $\overline{[R]}$, and then we can define the iteration function g such that $g(R) = \overline{f(R)}$. This function could then be used in the parallelized Pollard's Rho as the iterating function. As a result, we achieve a speedup with the factor of $\sqrt{t}$ (where t is the size of equivalence class) over the parallelized version. [1]

An example of an automorphism group is the use of the negation mapping $\Psi(P) = -P$. This has the order of two, hence we will gain a speedup improvement of $\sqrt{2}$.

V. Speeding up with trade-offs

All improvements wont necessarily affect the algorithm in a pure positive way. Some of these may win performance by sacrificing other resources or risks, such as memory. We call it a trade-off.

A. GCD improvement

The earlier mentioned cycle finding algorithm could be even faster, by implementing the improvements made by Pollard and Brent based on calculating the greatest common divider. They showed that if $gcd(a, n) > 1$ then $gcd(ab, n) > 1$, where b is a positive integer. Instead of computing $gcd(|x-y|, n)$ for each step, one can simply calculate $gcd(z, n)$ where z is the product of every $|x-y|$ - ending up with a huge performance gain. However, this may cause the algorithm to fail due to a repeated factor which may be introduced if n has a certain value, e.g. if n is squared. If the algorithm turns into a repetitive cycle, the algorithm should go back to the last computed gcd and do the initial formula. [7]

B. Faster cycle finding algorithm

There are a number of possible algorithms to use in order to find the cycle. Some of these will decrease the time complexity, but as a drawback increase the memory usage. In general, these methods store computed sequence values, and validates with a hash function whether or not the new value equals one of the previous ones. Sedgewick, Szymanski and Yao provided one that can guarantee a number of function evaluations, while Nivasch on the other hand came up with one that does not use a fixed amount of memory. However, these time improved methods usually cannot be implemented into the Pollard's rho algorithm.[4]

VI. Summary

From the very first release of the Pollard's rho algorithm, we have seen many enhancements. First, Brand made an improvement to the cycle finding algorithm resulting in a possible 30% gain compared to Floyd's version. Later in '$\mathbf{sectionV.B}$' we saw that by trading off memory, we could create an even faster cycle finding algorithm. However, most of these is not compatible to be used in the Pollard's rho algorithm.

Another improvement was made regarding the use of multiple processors. Actually we may gain a factor of a root square to the number of processors, M, due to the speed complexity. By letting the sequences generated by the different processors collide with each other, i.e. sharing state, the factor can be improved to M.

Automorphism is another topic for speeding up Pollard's rho. With some mathematical operations, we showed that instead of running the algorithm for every single point, you could simply run it over a group of points. This improvement would speed up the algorithm with the square root to the number of points in each group. As an example the negation map gains a speedup of $\sqrt{2}$.

Finally, we mentioned some improvement that came with a trade-offs. The theorem made by Pollard and Brent could possible save a lot of gcd-computations, but along with this improvement there is a possibility of terminating the algorithm, which is a drawback. We also discussed the opportunity of improving the cycle finding algorithm, by trading off memory.

VII. Conclusion

In this paper we have gained insight in how an algorithm can be modified to improve the speed complexity. This is an important factor when it comes to solve elliptic curves discrete logarithm problems. Further, we analyzed a collection of improvements made to the Pollard's rho, and witnessed many interesting variations. All in all we have learned that Pollard's rho algorithm has been through a lot of changes resulting in a massive speed improvement.

References

[1] Hankerson, Menezes, Vanstone: Guide to Elliptic Curve Cryptography, 2004, Springer-Verlag New York, Inc.

[2] Pollard, J. M. (1978). "Monte Carlo methods for index computation (mod p)". Mathematics of Computation

[3] https://en.wikipedia.org/wiki/John_Pollard_(mathematician), November 29, 2015

[4] https://en.wikipedia.org/wiki/Cycle_detection, November 27, 2015

[5] http://maths-people.anu.edu.au/ brent/pub/pub051.html, November 29, 2015

[6] http://maths-people.anu.edu.au/ brent/pd/rpb231.pdf, November 29, 2015

[7] https://en.wikipedia.org/wiki/Pollard's_rho_algorithm, November 29, 2015

[8] http://maths-people.anu.edu.au/ brent/pd/rpb051i.pdf, November 29, 2015

[9] http://cs.ucsb.edu/ koc/ecc/docx/08dlog.pdf

[10] http://mathworld.wolfram.com/GraphAutomorphism.html