Solving Elliptic Curve Discrete Logarithm Problem Using Parallelized Pollard's Rho and Lambda Methods

Puneet Gill

May 18, 2019

Contents

List of Figures

1 Introduction

In public key cryptography, a principal goal is to allow two parties to exchange confidential information, even if they have not met before and the communication channel is monitored by an adversary. RSA and elliptic curve public encryption schemes are widely used for any two parties to securely communicate providing confidentiality, data integrity, authentication and non-repudiation. The security of these encryption schemes lies in the difficulty of solving the factorization problem for RSA and discrete logartithm problem for Elliptic curves. Elliptic curve encryption scheme is preferred over RSA when building cryptosystems because the fastest algorithm to solve the factorization problem is sub-exponential, where as the one to solve discrete logarithms is fully-exponential. This means that to achieve the same level of security as an elliptic curve based encryption scheme, RSA encryption scheme requires its public and private keys to be significantly longer than in the case with elliptic curves. In the Parallel Collision Search with Cryptanalytic Applications paper, Paul C. Van Oorschot and Michael J. Weiner present a new technique of parallelizing methods that aim to find collisions in pseudo-random walks, which can be implemented to solve the discrete logarithm problem.

There are many applications of collision search algorithms in cryptanalysis. These may involve searching the space of keys, plaintext or ciphertext. For public key cryptosystems, they may be aimed at solving difficult mathematical problems such as computing factorization and discrete logarithms. In the paper, Oorschot and Weiner present an efficient method to parallelize Pollard's rho and lambda methods for computing discrete logarithms in cyclic groups. This analysis can also be extended to efficiently computing the elliptic curve discrete logarithm problem over a finite field $\mathbb{Z}_p$.

2 Elliptic Curve Discrete Logarithm Problem (ECDLP)

In the discrete logarithm problem in the finite field $\mathbb{F}_p^*$ based cryptosystem, Alice publishes two numbers g and h, and her secret is the exponent x that solves the congruence:

$$h \equiv g^x \ (mod\, p)$$

Solving the discrete logartithm problem thus requires an adversary, Eve, to find an x such that

$$h \equiv \underbrace{g \cdot g \cdot g \cdots g}_{x \text{ multiplications}} \ (mod\, p)$$

Something similar can be done with the group of points $E(\mathbb{Z}_p)$ of an elliptic curve $E : Y^2 = X^3 + aX + b$ defined over a finite field of $\mathbb{Z}_p$ as well. Alice can chose to publish two points P and

$Q \in E(\mathbb{Z}_p)$ and her secret is and integer k such that

$$Q = \underbrace{P + P + \cdots + P}_{\text{k additions}} = kP$$

There are properties that the generator, P, for elliptic curve should satisfy, these are:

- P is the generator of $E(\mathbb{Z}_p)$, $P \in E(\mathbb{Z}_p)$ and $P \neq \infty$
- $nP = \infty$
- $E(\mathbb{Z}_p) = \{\infty, P, 2P, 3P, ..., (n-1)P\}$. n is the $\#E(\mathbb{Z}_p)$
- For any integer k, $kP = (k \mod n)P$

In ECDLP, the goal of an adversary is to find how many times P needs to be added to itself to get to Q. By analogy to the discrete logarithm problem, k can be denoted as $k = \log_P Q$ and is called the elliptic discrete logarithm of Q with respect to P.

3 Algorithms for solving ECDLP

The goal in collision search is to find two distinct inputs a and b to a function f for which $f(a) = f(b)$. ECDLP can be reduced to such a problem.

There are several algorithms that can be used when solving for the discrete logarithm of an elliptic curve. The naive brute force method involves computing points $P, 2P, 3P, ...$ until a point kP is found that equals Q. This method does $O(n)$ elliptic point additions and each point addition takes $O((log_2 n)^2)$. Hence, the method is fully exponential because input size is $log_2 n$, as shown in A.1.

Other algorithms that solve the ECDLP in a considerably faster run-time are discussed in the proceeding sections. Shank's algorithm, Pollard's-rho method and Pollard's lambda method are discussed in this report. After the serialized version of the algorithms, parallelized versions are discussed, followed by their analysis.

3.1 Shank's Algorithm

Shank's algorithm is the generic algorithm for solving ECDLP, which is significantly faster than the brute force method. Despite the speed up, it is still fully exponential.

Given two points $P, Q \in E(\mathbb{Z}_p)$ where $Q = kP$ for some positive integer k, Shank's algorithm computes the discrete logarithm $k = \log_P Q$. It is similar to a meet-in-the-middle attack as it tries to find k by storing (rP, r) in a sorted table and searching the table for all values of q. It stops when a match satisfying the condition $Q - qM \overset{?}{=} rP$ is found, where $M = \lceil\sqrt{n}\rceil P, Q = kP$. The pseudocode of the algorithm is shown in Algorithm 1. It can be observed that the run-time of the algorithm is $O(\sqrt{n})$. There are two loops in *ComputeDiscreteLogarithm* and each runs at most $O(\sqrt{n})$ times and call to *RepeatedDoubleAndAdd* takes $O((\log n)^3)$ operations. This makes the total run-time to be $2\sqrt{n} + (\log n)^3$. In terms of the storage, Shank's algorithm requires $O(\sqrt{n})$ storage. This may not be feasible if n is large. For 128-bit security ($n \approx 2^{256}$), Shank's algorithm would require $2^{128} = 4.25 \times 10^{28}$ GBytes of storage, which is infeasible.

Algorithm 1 Shank's algorithm

```
1: procedure COMPUTE DISCRETE LOGARITHM($P, Q$)
2:     $m \leftarrow \lceil\sqrt{n}\rceil$
3:     $q \leftarrow k/m$ and $r \leftarrow k \mod m$        ▷ $k = qm + r$ multiplying by $P$ gives $rP = Q - q(mP)$
4:     for Each $r \in [0, m-1]$ do
5:         Compute $rP$ and store $(rP, r)$ in a table sorted by first entry
6:
7:     $M \leftarrow RepeatedDoubleAndAdd(m, P)$
8:
9:     for Each $q \in [0, m-1]$ do
10:        $R \leftarrow Q - qM$
11:        if $R$ matches any $(rP, r) \in$ sorted table then
12:            $k \leftarrow qm + r$
13:            return $k$
14:
15: procedure REPEATED DOUBLE AND ADD($x, P$)
16:     $x_{binary} \leftarrow toBinary(x)$        ▷ converts $x$ to binary array
17:     $A \leftarrow P$
18:     if $x_{binary}[0] = 0$ then
19:         $B \leftarrow \infty$        ▷ $0P = \infty$
20:     else
21:         $B \leftarrow P$
22:     for Each $x_b \in x_{binary}$ from $1, 2, ...$ do
23:         $A \leftarrow A + A$
24:         if $x_b = 1$ then
25:             $B \leftarrow B + A$
26:     return $B$
```

Parallelization of Shank's algorithm results in the speed up in writing and reading from the table but does not help reduce the memory problem. Due to this, parallelization of Shank's algorithm remains impractical. [2]

4 Pollard's rho-method

In 1978, Pollard came up with a *Monte-Carlo* method to solve discrete logarithm problem. Since then it has been modified to solve the Elliptic Curve discrete logarithm problem. Pollard's rho-method is an improvement over Shank's algorithm as it has the same expected run-time of $O(\sqrt{n})$ but uses negligible storage. As Pollard's rho-method is the fastest known method for solving ECDLP, the security of the elliptic curve cryptosystems depends on the efficiency of this algorithm.

To use the rho-method, a function f is selected such that it has the same domain and range (i.e. $f : S \to S$) and is a random function. Pollard's rho-method involves selecting a starting value x_0 and computing the next values as $x_i = f(x_{i-1})$ for $i = 1, 2, ...$ Since f has the same domain and range and S is finite, we would expect the values to repeat at some point. To detect a collision, the method involves storing only the points that satisfy a *distinguished property*, such as a fixed number of leading zero bits. A collision is detected when a distinguished point is encountered twice. This requires storage of very few points if the distinguished property is selected carefully.

4.1 Direct parallelization

The efficiency of collision search is dependent on increasing the probability of finding one. The direct parallelization of Pollard's rho-method involves multiple processors independently producing a sequence of points to find a collision. Since each processor works independently from the others, it does not alter the probability of other processors finding a collision. In addition, the probability that the next point on the sequence will result in a collision is dependent on the number of the points computed so far. As the number of points computed previously increase, the probability of the next point on the sequence resulting in a collision increases as well. As a result, none of the parallel processors reaches the success probability as high as it would in the case with a single processor. This is because multiple processors do not increase the number of points computed per unit time. In parallelization, each processor runs on a set with n/m elements, resulting in the expected number of steps taken by each processor before a collision is detected is $\sqrt{\pi n/2m}$. This is an inefficient use of paralleleization as direct parallelization for Pollard's rho-method only provides a speed up of a factor of $\sqrt{m}$ compared to the single processor version.

4.2 New parallelization method

To perform a parallel collision search, each processor does the following. It selects an arbitrary starting point $x_0 \in S$ and produces a trail of points defined by $x_{i+1} = f(x_i), i = 0, 1, 2, ...$ until a distinguished point x_d is observed. The distinguished point x_d is added to a common list for all processors. Once a processor adds a distinguished point, it starts to produce a new trail from a new starting point. This is done to avoid a processor from falling into a loop. A collision is detected when a processor finds a distinguished point that already exists in the common list.

In the parallelized rho-method, each processor does a pseudo-random walk and stops when a collision is detected, as shown in figure 1. After a collision occurs, the colliding trails coincide. This can be observed in figure 1 where trail 3 collides with trail 4 at x_3 and x'_2 and the collision is detected at x_5 and x'_4. In this method, if the first point on a trail collides with another trail, it is rejected, since the trail does not result in a collision in f. It is also possible that a processor may fall into a loop that does not contain any distinguished points and stops contributing to the collision search. It is avoided by setting a limit on the length a trail can grow to, which is described below.

For finding the discrete logarithm of elliptic curves using rho-method, an iterating function is defined and $E(\mathbb{Z}_p)$ is divided into 3 sets , S_1, S_2, S_3 roughly of same sizes based on some easily testable property. In the implementation, hashing is used to divide $E(\mathbb{Z}_p)$ into three equally sized disjoint sets.

This method is designed for the case when the discrete logarithm can be any value less than order, n. Given $Q = kP$ where $P, Q, R_i, R_{i+1} \in E(\mathbb{Z}_p), k$ is a positive integer, the iterating function is defined as:

$$R_{i+1} = f(R_i) = \begin{cases} Q + R_i, & R_i \in S_1 \\ 2R_i, & R_i \in S_2 \\ P + R_i, & R_i \in S_3 \end{cases}$$

Algorithm 2 shows the steps taken by each processor independently to solve ECDLP using pollard's rho-method. After the algorithm detects a collision, the discrete logarithm k can be computed as $\frac{a-c}{d-b}\,(mod\,n)$ provided $b \not\equiv d \ \ (mod\,n)$. This can be avoided because the probability of $b \equiv d \ \ (mod\,n)$ is low when n is sufficiently large.

4.3 Analysis

Let θ be the proportion of the points that satisfy the distinguishing property. The length of the trails are geometrically distributed with mean $1/\theta$. A trail is discarded if its length is 20 times longer than the average, $20/\theta$.

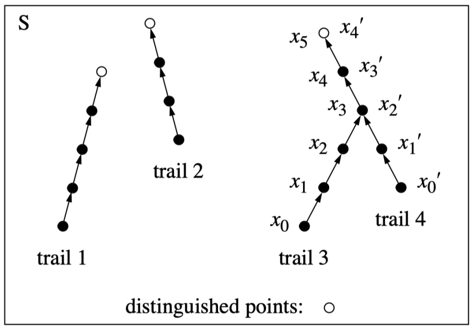

Figure 1: Parallelized Collision Search

Algorithm 2 Pollard's rho-method

```
1:  procedure COMPUTE DISCRETE LOGARITHM($P, Q$)
2:      Select $a, b \in_R [0, n)$                                    ▷ $n$ is the order of $E(\mathbb{Z}_p)$
3:      Compute $R = aP + bQ$
4:      while $R$ not found in common list of distinguished points do
5:          if $R$ is a distinguished point then
6:              Store $(R, a, b)$ to a common list of distinguished points
7:          $R, a, b =$ $ComputeNext(P, Q, R, a, b)$
8:
9:      Let the match of $R$ be $R^*$ and get the corresponding $c, d$ from list    ▷ $R^* = cP + dQ$
10:     return $\frac{a-c}{d-b}\ mod\, n$                              ▷ Provided $b \not\equiv d\ (mod\, n)$
11:
12: procedure COMPUTE NEXT($P, Q, R_i, a_i, b_i$)
13:     if $R_i \in S_1$ then
14:         return $Q + R_i, a_i, b_i + 1$
15:     else if $R_i \in S_2$ then
16:         return $2R_i, 2a_i, 2b_i$                                  ▷ $2a_i, 2b_i$ are $mod\, n$
17:     else if $R_i \in S_3$ then
18:         return $P + R_i, a_i + 1, b_i$
```

By the birthday paradox, the expected number of steps taken on a pseudo-random walk before a collision occurs is $\sqrt{\pi n/2}$, where $n = |S|$. Assuming there are m processors in the parallelized version of the rho-method, the expected number of steps taken by each processor before a collision occurs is $\sqrt{\pi n/2}/m$. After a collision occurs, the expected number of points computed before the collision is detected is $1/\theta$. [1] Therefore, the expected runtime for collision detection is:

$$T_\rho = \frac{\sqrt{\frac{\pi n}{2}}}{m} + \frac{1}{\theta}$$

.

Other than the collision detection, the second goal is to locate the collision. To solve for the location of the collision efficiently, one needs to know the starting point of each trail and its length. Once the starting points of the trails are known, the next step is to move along the longer trail until its length matches the second trail. Finally move along both trails until they reach the same point. If the value of the common point is such that $f(a) = f(b), a \neq b$ then the points a, b is the location of the collision. The expected runtime for finding the collision is $1.5/\theta$, as shown in the appendix of [1]. This makes the total expected runtime for detecting and finding a collision to be:

$$T_{\rho'} = \frac{\sqrt{\frac{\pi n}{2}}}{m} + \frac{2.5}{\theta}$$

.

[1]As described in the paper, there is an apparent paradox here because trails average $1/\theta$ in length, but the expected distance from a point of collision to the end of the trail is also $1/\theta$. Longer trails are more likely to be involved in a collision, which resolves the paradox.

5 Pollard's Lamda Method (Catching Kangaroos)

It is not always the case that the discrete logarithm k in $Q = kP$ can be any value less than the order, n. In practical implementations, k is limited to a restricted range of values for faster computation. The Pollard's rho-method solves discrete logarithms without restrictions but the lambda method is faster when k is restricted.

For a value of $k \in [0, b)$ for some $b < n$. The problem of finding k given $Q = kP$ can be solved using Pollard's lambda method. The lamda, or the catching kangaroos method, running on a single processor without distinguished points works as follows. In the lambda method, a number line can be thought of, as labelled by $P, 2P, 3P, ...$ and two kangaroos, a tame and a wild jump along this number line until the wild kangaroo catches the tame. To achieve this, an iterating function detemining the next location of the kangaroos is defined as $R_{i+1} = R_i + a(R_i)P$, where $a(R_i)$ is a function that outputs the next jump size for the kangaroo and the kangaroo moves forward by this value. The function $a(R_i)$ randomly selects values from a set A and only depends on the current location R_i. In the paper, A is a set containing powers of 2 starting with 2^0 up to a limit with the largest value in the set being such that the mean of the values in the set is a certain (optimized) value α.

To start the collision detection algorithm, a tame kangaroo starts at $R_0 = bP$ and makes $\alpha\beta$ jumps for an optimised value of β, keeping track of the distance travelled and the final resting spot of the tame kangaroo. Now a wild kangaroo begins at a starting point $R_0^{'} = Q$ and is allowed to jump with the same iterating function, keeping track of the distance travelled by it and checking whether it ever lands on the final resting spot of the tame kangaroo. Since both kangaroos use the same iterating function, if the wild kangaroo ever lands on any of the same spots as the tame, it will follow the same path thereafter and will eventually reach the final resting spot of the tame kangaroo. The discrete logarithm can then be calculated from the difference of the distance travelled by the two kangaroos.

The tame kangaroo starts from $R_0 = bP$ and make $\alpha\beta$ jumps, therefore the expected total distance travelled by it will be $\alpha^2\beta + b$ from $0P = \infty$ (expected value of all entries in A is defined as α). If the wild kangaroo travels this far, then the wild kangaroo has passed the final resting spot of the tame kangaroo and has failed. If the wild kangaroo fails, another wild kangaroo is set off from a starting poisiton of $R_0^{'} = Q + zP$, for a small known value of z. Algorithm 3 shows the pseudocode for the implementation of the lambda method.

5.1 Parallelization

Direct parallelization of the lambda method suffers from the same problem as the rho-method. Running m processors independently in parallel only provide the speed up of $\sqrt{m}$. A new method for parallelization is discussed in the paper that provides a linear speed up and works as follows. It starts with $m/2$ tame kangaroos launched from starting points $(b/2 + iv)P$, where $0 \leq i < m$

Algorithm 3 Pollard's Lambda-method

```
1: procedure COMPUTE DISCRETE LOGARITHM($P, Q$)
2:     Generate powers of 2 till $2^d$ such that the mean of the powers is $\alpha$
3:     $kangaroo\_tame = bP$
4:     $total\_distance\_tame = b$
5:
6:     for $jump \in [0, \alpha\beta)$ do
7:         $kangaroo\_tame$, $jump\_size = getNextJump(kangaroo\_tame)$
8:         $total\_distance\_tame += jump\_size$
9:         Store ($kangaroo\_tame$, $total\_distance\_tame$) in a sorted table
10:    $kangaroo\_wild = Q$
11:    $total\_distance\_wild = 0$
12:
13:    $jump\_counter = 0$
14:    $tame\_distance_{jump} = 0$                    ▷ tame distance till the $i^{th}$ jump
15:    while True do
16:        if $kangaroo\_wild = kangaroo\_tame$ then
17:            break
18:        if $total\_distance\_wild > total\_distance\_tame$ then
19:            Set value of $kangaroo\_wild = Q + zP$ for a small value of $z$
20:            $total\_distance\_wild = 0$
21:            $jump\_counter = 0$
22:        $kangaroo\_wild$, $jump\_size = getNextJump(kangaroo\_wild)$
23:        $total\_distance\_wild += jump\_size$
24:        $jump\_counter += 1$
25:        Get corresponding $tame\_distance_{jump}$ from the table
26:    return $b + tame\_distance_{jump} - total\_distance\_wild - z \pmod n$
27:
28: procedure GET NEXT JUMP($position$)
29:    $hash$ = Get hash value of $position$
30:    $jump\_size = 2^{hash (mod\ n)+1}$
31:    return position + jump_size*P, jump_size
```

and a small constant v (not a power of 2). Simultaneously, $m/2$ wild kangaroos are launched with starting points $Q + (b + iv)P$. Whenever a kangaroo lands on a distinguished point, the point with a flag indicating the kangaroo type (tame or wild) and the distance travelled are stored in a list common to all the processors. If the kangaroo lands on a distinguished point that exists in the list and the kangaroos are of the same type, then one of kangaroos is moved forward by a small random value. This is done to avoid the two from following the same path. Collision is detected when the two colliding kangaroos are of different types. The discrete logarithm can then be computed by subtracting the distance travelled by the colliding kangaroos.

5.2 Analysis

The tame kangaroo in the lambda method takes a total of $\alpha\beta$ jumps before reaching its final resting spot. After the wild kangaroo passes the point bP, it takes about $\alpha\beta$ (tame kangaroo makes $\alpha\beta$ jumps from bP) jumps before reaching the tame kangaroo. With each jump, the wild kangaroo has a probability of $1/\alpha$ of landing on one of the same spots that the tame kangaroo once landed on (collision). The probability of success is about $1 - e^{-\beta}$, as shown in A.2. The expected number of jumps that the wild kangaroo takes is $b/2\alpha + \alpha\beta$, since the expected starting point for it is $bP/2$. Before succeeding, the wild kangaroo is expected to fail $1/(1 - e^{-\beta}) - 1$ times, giving the total runtime to be $\alpha\beta - b/(2\alpha) + (b/\alpha + \alpha\beta)/(1 - e^{-\beta})$ operations. It can be observed that the runtime is a function of α and β, and can be minimized when $\alpha = \sqrt{(b(1 - e^{-\beta})/(2\beta(2 - e^{-\beta}))}$ as shown in A.4. Authors in the paper advice values of $\beta \approx 1.39$ and $\alpha \approx 0.51\sqrt{b}$ for minimizing the runtime, which are derived using numerical techniques.

In the parallel version of the lambda method, the tame and wild kangaroos are somewhere between 0 and $b/2$ distance apart from each other, because of their starting points. This makes the expected distance of separation to be $b/4$. This means that the trailing kangaroo takes $b/(4\alpha)$ jumps to cover this distance, from the relationship shown in A.3. After getting past the starting point of the leading kangaroo, the $m/2$ trailing kangaroos may land on the same spot as one of the leading kangaroos did. In the paper, the expected number of jumps taken by each trailing kangaroo before one one of them collides with one of the spots of the leading kangaroo is calculated to be $4\alpha/m^2$. This results in the total number of jumps taken by the trailing kangaroos to be $b/(4\alpha) + 4\alpha/m^2$. These jumps are a function of α and can be minimized to, as shown in A.5. If the proportion of the distinguished points is θ, then the number of jumps before a distinguished point is encountered after a collision is $1/\theta$. This results in the total runtime of the parallelized version of lambda method to be:

$$T_\lambda = \frac{2\sqrt{b}}{m} + \frac{1}{\theta}$$

Compared to the parallelized rho-method, the parallelized lambda method is 1.6 times slower when $b \approx n$ and only gets faster when $b < 0.39n$, as shown in A.6.

6 Implementation of the algorithms

As a part of this project, the non-parallelized versions of the three algorithms were implemented and the discrete logarithms for a test elliptic curve were computed. Although this does not tackle a real life ECDLP, it helped in understanding the algorithms better and the challenges involved in implementating them. These algorithms were run on a laptop and the following results were obtained:

$p = 1035418103$, 30-bit prime
$n = 1035437671$ (prime)
$E/\mathbb{Z}_p : Y^2 = X^3 + 45181635X + 124806060$
where $a = 45181635, b = 124806060$ are found such that $4a^3+27b^2 \neq 0$ and the generator of $E(\mathbb{Z}_p)$, $P = (299835419, 368012477)$

Table 1: Description of implementation results

Algorithm	P	Q	k	Time elapsed (ms)	Memory used (bits)
shank's	(299835419, 368012477)	(830731058, 402455075)	884158742	13398	2409132
rho	(299835419, 368012477)	(830731058, 402455075)	884158742	145706	negligible
lambda	(299835419, 368012477)	(1029827107, 152393852)	108302623	9823	negligible

The memory used in Shank's algorithm is calculated using the following formula, which can be directly derived from the objects stored in the list in the Algorithm 1, which is :

$$32178 \times (\log_2 32178 + 2 \times \log_2 1035418103) = 2409132\ bits$$

From the Table 1, it can be seen that the lambda method is the fastest. This is because the k value is chosen from a restricted range of $[0, 403813060]$, where $b = 0.39p$ while using a negligible amount of storage. If the k value is chosen without bounds then Shank's algorithm is $\approx$ 10 times faster than the rho-method but also uses 2409132 $bits \approx 301\ kbytes$ of storage. This is not significant compared to the memory available in modern computers. Shank's algorithm requires storage, which is a function of the number of bits of the prime. As the bit length of prime number increases, the algorithm requires a significant amount of storage. Although rho-method was slowest, it is still promising because it does not require significant storage and works for any k value without restrictions.

7 Conclusion

The security of public key cryptosystems rely on the difficulty of the underlying mathematical problem. In case of elliptic curves, it is the discrete logarithm problem. Since no known subexponential attack exists for ECDLP, the parallelized rho and the lambda methods introduced in the paper are the most efficient known attacks for finding the discrete logarithms. Direct parallelization only provides the $\sqrt{m}$ speed up. However, the new methods use the processors efficiently, giving a linear speed up. The new rho-method is a general method that greatly extends the reach of practical attacks on not only solving discrete logarithm problem but also finding collisions in hash functions as well as double and triple DES block ciphers.

A Appendix

A.1 Runtime for Brute force Algorithm

Let the point P be k-bits long and the brute force method is $O(n)$.

$$k = \log_2 n$$

$$n = 2^k$$

Therefore, in terms of the input size k, the runtime of the brute force method is $O(2^k)$, which is exponential.

A.2 Probability of success for wild kangaroo

The probability at each that the wild kangaroo will land on one of the same spots as the tame kangaroo once landed on is $1/\alpha$ and wild kangaroo makes $\alpha\beta$ jumps after reaching bP. We are interested in the probability of success of collision which is the probability of the wild kangaroo landing on one of the spots after $\alpha\beta$ jumps. This is given by

$$P(success) = 1 - P(failure)$$

$$P(failure) = (1 - (\frac{1}{\alpha})^{\alpha\beta}) \approx 1 - e^{-\beta}$$

In this, $failure$ is if the wild kangaroo fails to land on any of the spots as the tame kangaroo after making $\alpha\beta$ jumps.

A.3 Number of jumps to cover a distance

If the expected jump size from a set of jump values is α and the distance that needs to be covered is b then, the number of jumps is given by the following equation:

$$mean\ jump\ size \times number\ of\ jumps = distance\ covered$$

Hence, the number of jumps needed to cover distance b is b/α

A.4 Minimized parameters for Lambda method's runtime

The runtime for lambda method is a function of α and β and can be written as

$$f(\alpha) = \alpha\beta - b/(2\alpha) + (b/\alpha + \alpha\beta)/(1 - e^{-\beta})$$

$$f'(\alpha) = \beta + b/(2\alpha^2) + (\beta - b/\alpha^2)/(1 - e^{-\beta})$$

Runtime is minimized when $f'(\alpha) = 0$ and isolating for α gives

$$\beta + b/(2\alpha^2) + (\beta - b/\alpha^2)/(1 - e^{-\beta}) = 0$$

$$\alpha = \sqrt{(b(1 - e^{-\beta})/(2\beta(2 - e^{-\beta}))}$$

A.5 Minimize α for total number of jumps

The expected number of jumps by trailing kangaroo is a function of α and can be written as

$$f(\alpha) = \frac{b}{4\alpha} + \frac{4\alpha}{m^2}$$

$$f'(\alpha) = \frac{-4b}{(4\alpha)^2} + \frac{4}{m^2} = \frac{-4bm^2 + 64\alpha^2}{(4\alpha m)^2}$$

This is minimized when $f'(\alpha) = 0$ and isolating for α gives

$$\frac{-4bm^2 + 64\alpha^2}{(4\alpha m)^2} = 0$$

$$\alpha = \sqrt{\frac{4bm^2}{64}} = \frac{m\sqrt{b}}{4}$$

When $\alpha = (m/4)\sqrt{b}$, the expected number of jumps is

$$f(\frac{m\sqrt{b}}{4}) = \frac{b}{4(\frac{m\sqrt{b}}{4})} + \frac{4(\frac{m\sqrt{b}}{4})}{m^2} = \frac{2\sqrt{b}}{m}$$

A.6 Comparision of parallelized rho-method and lambda method

When $b \approx n$, the runtimes for the parallelized rho and lambda methods for collision detection are:

$$T_\rho = \frac{\sqrt{\frac{\pi n}{2}}}{m} + \frac{1}{\theta}$$

$$T_\lambda = \frac{2\sqrt{n}}{m} + \frac{1}{\theta}$$

$$\frac{T_\lambda}{T_\rho} = \frac{\frac{2\sqrt{n}}{m} + \frac{1}{\theta}}{\frac{\sqrt{\frac{\pi n}{2}}}{m} + \frac{1}{\theta}} \approx \frac{\frac{2\sqrt{n}}{m}}{\frac{\sqrt{\frac{\pi n}{2}}}{m}}$$

$$\frac{T_\lambda}{T_\rho} = 2\sqrt{\frac{2}{\pi}} = 1.5958$$

Assuming the time taken after a collision occurs and a collision detected $1/\theta$ is same for both methods, thus can be removed from the comparison.

Finding $b < n$, such that lambda method is faster than the rho method is:

$$T_\lambda < T_\rho$$

$$\frac{2\sqrt{b}}{m} + \frac{1}{\theta} < \frac{\sqrt{\frac{\pi n}{2}}}{m} + \frac{1}{\theta}$$

$$2\sqrt{b} < \sqrt{\frac{\pi n}{2}}$$

$$b < \frac{\pi n}{8}$$

$$b < 0.3927n$$

References

[1] Paul C. van Oorschot and Michael J. Wiener. *Parallel Collision Search with Cryptanalytic Applications*. 1996.

[2] Paul C. van Oorschot and Michael J. Wiener. *Parallel Collision Search with Application to Hash Functions and Discrete Logarithms*. pp. 6, 1994.