Testing of enhanced Pollard's kangaroo algorithm

September 2024

1 Introduction

It goes without saying that finding a discrete logarithm is a complex task, even for modern computers. However, Daniel J. Bernstein and Tanja Lange suggested in their article [1] a way to compute a small discrete logarithm faster based on Pollard's kangaroo algorithm. They introduced an algorithm [2] that uses different constant variables for computations and showed how increasing or decreasing some constant may impact processing time. They claim, that computing a discrete logarithm in an interval of order l takes only $1.93 \cdot l^{1/3}$ multiplications on average using a table of size $l^{1/3}$ precomputed with $1.21 \cdot l^{2/3}$ multiplications, and computing a discrete logarithm in a group of order l takes only $1.77 \cdot l^{1/3}$ multiplications on average using a table of size $l^{1/3}$ precomputed with $1.24 \cdot l^{2/3}$ multiplications. In that document, an improvement for the Bernstein and Lange algorithm is proposed, and results of different constant combinations are described, and suggestions are made on what constants are better to use in computations based on the results obtained.

2 Algorithm description

2.1 Original algorithm

The beauty of Pollard's kangaroo algorithm is its ability to compute a discrete logarithm universally in any group of prime order l in any interval of length l inside any group of prime order $p \geq l$. The algorithm is based on Shanks's baby-step-giant-step method and is as follows

- choose a base-g r-adding iteration function whose steps have average exponents $\Theta(l^{1/2})$;
- run a walk starting from g^y (the "tame kangaroo"), where y is at the right end of the interval;
- record the W^{th} step in this walk (the "trap"), where W is $\Theta(l^{1/2})$;
- run a walk (the "wild kangaroo") starting from h, checking at each step whether this walk has fallen into the trap.

Oorschot and Wiener also proposed a method of computing the discrete logarithm in parallel [3]. A parallel kangaroo algorithm is implemented where tame kangaroos initiate their paths from g^y for various values of y, all situated near the midpoint of the interval. Simultaneously, a comparable number of wild kangaroos begin from h^{gy} for numerous small values of y. Collisions are identified using distinguished points; however, the chosen distinguished-point property has a significantly higher probability than $1/W$. The walks proceed beyond the distinguished points, with adjustments made to prevent collisions among tame kangaroos as well as between wild kangaroos.

2.2 Article algorithm

The algorithm proposed in the article is divided into **preprocessing computations** and **main computations** parts.

In the preprocessing part, it is suggested that a table with so-called distinguished points be generated and used. According to it, a point is assumed to be distinguished if its n last bits are zeros. To build the table, simply start some walks at g^y for random choices of y. The table entries are the distinct distinguished points produced by these walks, together with their discrete logarithms. An implementation of this algorithm stores not just distinguished points but also the so-called most useful distinguished points in the table. A distinguished point is assumed to be most useful if it has the largest number of ancestors. In other words, if a point has the largest number of group elements that walk to this point, we say that it is the most useful. Authors also recommend generating, let's say, $N = T \cdot d$ elements where T - is the number of the most useful distinguished points and d - is some multiplier.

To find the discrete logarithm of h using this table, start walks at h^x for random choices of x, producing various distinguished points $h^x g^y$. Check for two of these new distinguished points colliding, but also check for one of these new distinguished points colliding with one of the distinguished points in the pre-computed table. Any such collision immediately reveals $\log_g h$.

Even since this implementation is quite good for small secrets (let's say, below 48-bit), solving DLP for 64-bit secrets and higher becomes really tricky. Basically, on pre-computations, it looks for a distinguished point by iterating over the whole table, which makes time complexity $\mathrm{O}(n^2)$. On the main computation step, it utilizes binary search, which makes $\mathrm{O}(n^{log(n)})$ time complexity.

2.3 Improved algorithm

We managed to improve this algorithm, so it has $\mathrm{O}(n)$ complexity on both pre- and main computations steps and could be run in a defined amount of CPU threads increasing its success in finding results faster.

Instead of using an array for the table, we use a hashmap, which produces constant time for retrieving and setting elements.

When running it in multiple threads, we run each thread with a different starting value and generate the next points based on it. Of course, sometimes a starting point needs to be regenerated. The usage of mutual exclusion locks (mutexes) increases computation time a bit but eliminates the chance of race conditions.

The above changes adjust the algorithm described in section 2.2. Thus, we don't need to operate with the concept of 'most distinguished points' since every point is retrieved from the table in constant time. Therefore, we use only N constants and no longer T and d.

Conigurable values that are used in the algorithm are:

- R - number of slog-s and corresponding s points to generate;
- m - multiplier of an upper bound of some generated secret on a s values initialization part;
- i - number of iterations for one loop on main computations;
- N - table size.
- W - number of steps to find a distinguished point from the chain of generated points;
- *table* - is a pregenerated table with distinguished points and their secrets. It's property *size* defines the current amount of elements in the table.

The algorithm is described in the section below.

Algorithm 1 Pollard's kangaroo algorithm

Phase 1: S values initialization

$\{slog_j \xleftarrow{r} [1; m \cdot 2^{(\log_2(2^{secret\_size-2}/W))}] \mid j = 0, 1, ..., R\}$
$\{s_j \leftarrow slog_j \cdot G \mid j = 0, 1, ..., R\}$

Phase 2: Preprocessing (table generation)

while $table.size < N$ **do**
  $wlog \xleftarrow{r} [1; 2^{secret\_size}]$
  $w \leftarrow wlog \cdot G$
  for $j = 0$ **to** $i \cdot W$ **do**
    if $isDistinguished(w)$ **then**
      $table[w] = wlog$
      break
    end if
    $h := hash(w) \mod R$
    $wlog := wlog + slog_h$
    $w := w \oplus s_h$
  end for
end while

Phase 3: Main computations (find discrete logarithm of target value t)

while true **do**
  $wdist \xleftarrow{r} [1; 2^{secret\_size-8}]$
  $w \leftarrow wdist \cdot G$
  for $j = 0$ **to** $i \cdot W$ **do**
    if $isDistinguished(w)$ **then**
      $wdist_{final} = table[w] - wdist$
      break
    end if
    $h := hash(w) \mod R$
    $wdist := wdist + slog_h$
    $w := w \oplus s_h$
  end for
  if $wlog_{final} \cdot G == t$ **then**
    return $wlog_{final}$
  end if
end while

In the above algorithm such functions are used:

- *hash* - some hash function that returns a hash value of the provided value;
- *isDistinguished* - a boolean function which defines whether the provided public key is distinguished or not.

Above algorithm can be described in a series of steps that are:

1. generate a set $\{slog_i\}$ of R random secrets each of size $m \cdot \log_2(2^{secret\_size-2}/W)$. For each $slog_i$ compute a set of points $\{s_i\}$ by multiplying $\{slog_i\}$ by the base point G;
2. generate discrete logarithm *table* (skip this step if a table for the specified secret size already exists):
   - 2.1. generate a random secret $wlog$ of size $secret\_size$ and multiply it by the base point G resulting w point;
   - 2.2. check whether the point w is distinguished. If it is, append it to the *table* and proceed with the step 2.1. If it is not, proceed with the next step;
   - 2.3. compute a value $h = hash(w) \pmod R$. Select a secret $slog_h$ and the corresponding point s_h from the set generated in the step 1;
   - 2.4. compute the next secret as $wlog = wlog + slog_h$ and the corresponding point as $w = (w \oplus s_h) \pmod p$, where p is the group prime field;
   - 2.5. repeat step 2.2 to check if s is distinguished. If the loop count exceeds $i \cdot W$, restart from the step 2.1;
   - 2.6. repeat steps 2.1 to 2.5 until the discrete logarithm table contains N elements;
3. start main computations for the provided target point t:
   - 3.1. generate a random secret $wdist$ of size $secret\_size - 8$ and multiply it by the base point G resulting w point;
   - 3.2. check whether w is a distinguished point. If it is, search for w in the *table*. If found, retrieve its corresponding secret $wdist_{table}$, and compute $wdist_{final} = wdist_{table} - wdist$. Compute a point by multiplying $wdist_{final}$ by the base point G. If it matches t, the secret has been found. If not, proceed to the step 3.3;
   - 3.3. compute a value $h = hash(w) \pmod R$. Select a secret $slog_h$ and the corresponding point s_h from the set generated in the step 1.
   - 3.4. compute the next secret as $wdist = wdist + slog_h$ and the corresponding point as $w = (w \oplus s_h) \pmod p$ where p is the group prime field;
   - 3.5. repeat the step 3.2. If the loop count exceeds $i \cdot W$, restart from the step 3.1;
   - 3.6. repeat steps 3.1 to 3.5 until the secret corresponding to the target public key is found.

We used an improved version of the algorithm described in this section for the next series of experiments.

3 Experiment

In the experiment, we wanted to test how much certain constants affect the next parameters:

- preprocessing time;
- generated table size;
- main computation time (best, worst, average);

For the purpose of testing, a few programs have been written and are available on GitHub [4]. The experiment is briefly described below. To launch it, follow the commands described in *README.md* file on GitHub.

JSON configuration file of above constants is generated using **config-generator**, which consists of a number of constants per some test. Constant ranges, as well as a number of constants in the range, are configurable. Since the preprocessing part could take a while, some of the tables are reused, which is chosen automatically. From now, for the sake of simplicity we call a number of tests as experiment.

For an experiment, a compiled binary file is launched, provided with constants defined in a JSON file. The experiment is launched in parallel using **experiment-launcher** by balancing tests between the defined number of threads, e.g., if an experiment is launched in 16 threads, 16 tests with different given parameters are launched at one time. Each test produces log files with time results, iterations number (number of x variable changes), and some additional information. For reasons of clarity, all tests are launched on the same inputs, which are generated using **secrets-generator** and stored in a .bin file. Number of threads in which a number of tests should be run is configurable. We ran the experiment in 1 thread, but each test of the experiment - in 30.

Log files could be analyzed using **result-analyzer** by outputting top K files with the best total time, which could help to understand the best parameters.

4 Expectations

Based on Bernstein and Lange doc, it is expected

- preprocessing time to increase with increasing a number distinguished points (N) (table size) and main computation time decrease respectively;
- preprocessing time to increase with increasing W parameter and main time decrease respectively.

Also, it is not stated in the original doc, but we also expect

- preprocessing time to increase slightly with increasing (R) and main computation time decrease respectively;
- preprocessing time to increase with increasing m parameter and main computation time decrease;
- we are not sure about how a number of iterations for one loop (i) could affect the results;
- we also expect the table size to expand as the secret size grows.

5 Benchmarks

A series of experiments was launched on two 12-core Intel Xeon CPU E5-2650 v4 @ 2.20GHz in 30 threads with 16gb of RAM. In general, around 3000 64-bit secrets were processed with different constants, and the results are obtained below.

Increasing R on range $(128 : 512]$ made no impact: the difference between best and worst time results haven't changed, as well as preprocessing time. We don't think there is a point in further changing R.

Increasing N on range $[800 : 8000]$ greatly impacted both pre- and main computations. Increasing N twice increases preprocessing twice. As for main computations, the results are shown in Figure 1, which looks like a logistic curve. On the Y axis, it shows $N/800$ value and X axis - mean time taken in seconds. From the chart, it is pretty clear that choosing a really high N almost makes no sense since the time difference is minimal.

Constant W should be chosen in such a way that W is the result of powering 2 to some value. In the first tests, it was decided to act according to the formula that is stated in the article and is as follows: $W = \alpha\sqrt{l/T}$. W was calculated by the formula with randomly chosen α. Since the chance of hitting W with the above condition is minimal, results showed a really long main computation time, which is unacceptable. In a series of experiments, we decided to choose W as $524288 * k$, where k is a natural number (524288 was taken from the code example attached to the Bernstein and Lange document). As well as for the N constant, increasing k produced a doubling of preprocessing time. The main computation time can be observed in Figure 2. Here, the Y axis represents the time taken in seconds, and the X axis - k. The produced curve may look like a parabola. According to the obtained results, it makes no sense to increase W really high and choose it low, since it could produce even worse results. The best results are at the bottom of the

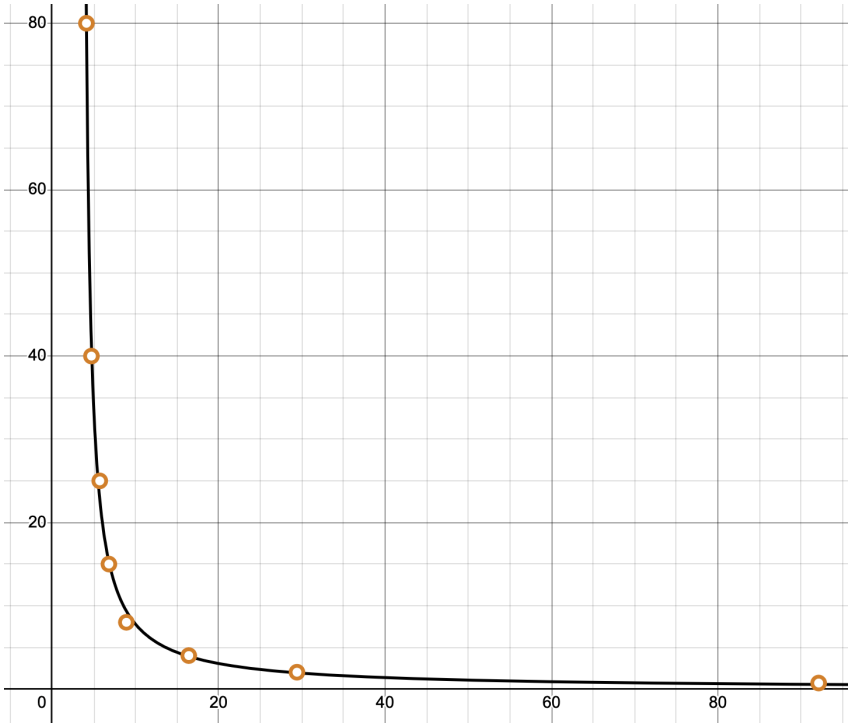

Figure 1: Dependency chart of N and main computation spent time. Y axis represents $N/800$ and X axis - taken time in seconds.

left branch. Also, minor changes of k at small values do not produce significant changes. Moreover, the larger k is taken, the bigger difference between the best and worst result is: the best becomes smaller, but the worst becomes larger respectively.

Increasing m in the range from $[0.5; 2]$ produced no result either on the preprocessing part or main computations. It makes sense to leave it as 0.5. Increasing i on multiple ranges also produced no results. Also, it was noticed that secrets that are closer to 0 (checked on size of 8 bits) and those that are closer to 64 bits have different main computation time.

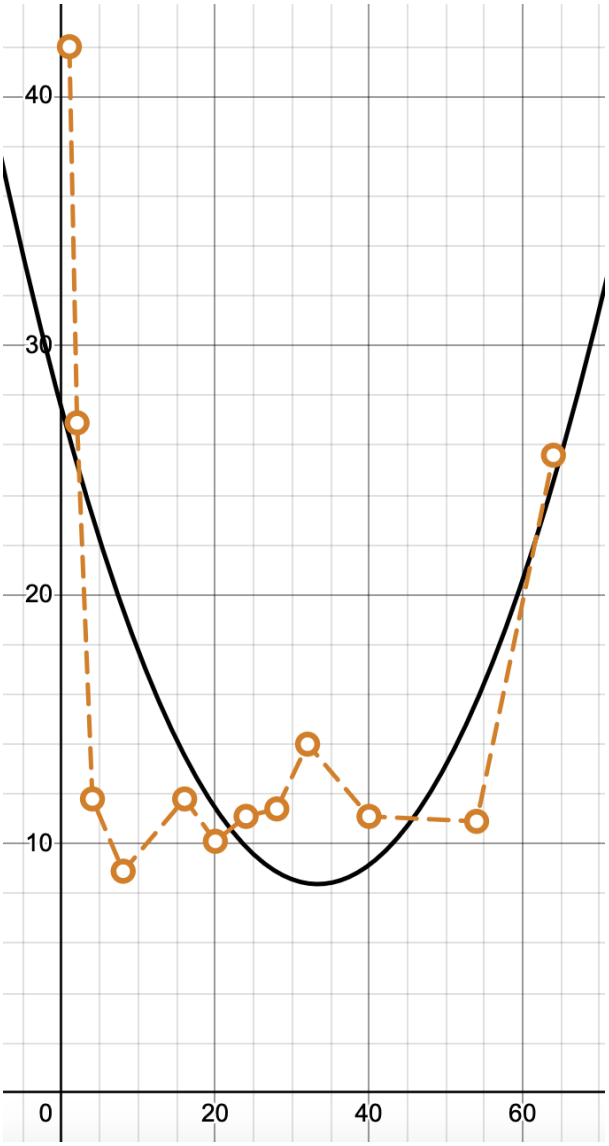

Figure 2: Dependency chart of k and main computation spent time. Y axis represents the taken time in seconds and X axis - k.

Based on the benchmarks, we picked the next best values for 64-bit secrets:

- R - 128;
- W - 4194304 (with k equal to 4);
- m - 0.5;
- i - 8;
- N - 32000;

Based on those parameters, around 100 tests were run, generating a 2.2mb table file on the preprocessing part, which took 15.8 minutes. The best result is 2.2 sec, the worst—13 sec. Meantime is 4.2 sec.

Also, a number of experiments were conducted on 72-bit secrets on the next parameters:

- N - 32000, 64000, 112000;
- W - 2097152, 4194304, 16777216 (with k equal to 4, 8, 16 correspondingly);
- m - 0.5;
- i - 8;

Produced time result relations were the same as for 64-bits: increasing N twice increases the preprocessing time by around two times resulting in 74, 147, and 258 minutes. The main computation times are as follows: 89, 48, and 34 seconds, correspondingly. As for W, preprocessing time increases almost twice when increasing W twice, and the same can be observed for main computations, too. The best results were obtained for N equal to 112000 and W - 16777216, resulting in 4.3 hours for preprocessing and around 34 seconds for main computations, with the best result of 8 seconds and worst - 85 seconds resulting in a 7.1 mb table.

6 Pitfalls

Conducting a series of tests, we faced some issues that must be taken into account when launching my own experiments.

For bigger secrets, bigger tables are expected. Bigger tables require more time to process and may take up to days on a single thread, as I have found out from the results I obtained. Also, the table is written into storage when it is fully generated, so if a test is stopped before a table is being written, starting the test one more time requires generating a table one more time. The same is for an experiment: if it is stopped before being ended, launching one more time requires regenerating all tables one more time (despite the case when tables were generated beforehand and *allowWriteFlag* in JSON file is marked as false). Of course, the main computations have to be started again.

The generated table size stated in this article is not really accurate. Choosing a better way to write data can reduce it even more. For instance, if the algorithm runs on the

same secrets, some data does not need to be written into the file. So, the final table file for the 64-bit secrets with the chosen best constants could be even less than 2.2 mb.

The received time results may not be accurate because of the other background processes and CPU time sharing: a slight deviation from the generated function graph can be seen in the Figure 1 and Figure 2 charts.

Also note, The more CPU cores used, the more overhead there will be for creating and stopping threads. That means that obtained results are not accurate and may be even less.

References

[1] Daniel J. Bernstein, Tanja Lange. Computing small discrete logarithms faster, https://cr.yp.to/dlog/cuberoot-20120919.pdf, (accessed Sep 30, 2024)

[2] Enhanced Pollard's kangaroo algorithm, https://cr.yp.to/dlog/walk1.cpp, (accessed Sep 30, 2024)

[3] Paul C. van Oorschot, Michael Wiener, Parallel collision search with cryptanalytic applications, Journal of Cryptology 12 (1999), http://members.rogers.com/paulv/papers/pubs.html, (accessed Sep 30, 2024)

[4] GitHub repository with the algorithm and test programs, https://github.com/distributed-lab/pollard-kangaroo-plus-testing, (accessed Sep 30, 2024)