Parallel Collision Search with Radix Trees

Gilles Dequen[1] and Sorina Ionica[1] and Monika Trimoska[1,2]

[1] Laboratoire MIS, Université de Picardie Jules Verne
33 Rue Saint Leu Amiens 80039–France
[2] SATT Nord
25 Avenue Charles Saint-Venant Lille 59800–France

Abstract. Parallel versions of collision search algorithms require a significant amount of memory to store a proportion of the points computed by the pseudo-random walks. Implementations available in the literature use a hash table to store these points to allow fast memory access. We propose to replace the traditional hash table by a radix tree structure, allowing both look-up and insertion in a single operation. We provide theoretical and experimental evidence that memory access is an important factor in determining the runtime of our algorithm. Our benchmarks indicate that our algorithm achieves linear parallel performance.

Keywords: radix tree, discrete logarithm, parallelism, collision, elliptic curves, meet-in-the-middle

1 Introduction

Given a function $f : S \to S$ on a finite set S, we call collision any pair a, b of elements in S such that $f(a) = f(b)$. Collision search has a broad range of applications in the cryptanalysis of both symmetric and asymmetric ciphers: computing discrete logarithms, finding collisions on hash functions and meet-in-the-middle attacks. The Pollard's rho method, initially proposed for solving factoring and discrete logs, can be adapted to find collisions for any random mapping f. The parallel collision search algorithm, proposed by van Oorschot and Wiener [16], builds on Pollard's rho method, and is expected to have a linear speedup compared to the sequential version. This parallel algorithm computes several walks in parallel and stores some of these points, called distinguished points, within the shared memory of an SMP system.

In this paper, we revisit the memory complexity of the parallel collision search algorithm, both for applications which need a small number of collisions (i.e. discrete logs) and those needing a large number of collisions, such as meet-in-middle attacks. In the case of discrete logarithms, collision search methods are the fastest known attacks in a generic group. In elliptic curve cryptography, subexponential attacks are known for solving the discrete log on curves defined over extension fields, but only generic attacks are known to work in the prime field case. Evaluating the performance of collision search algorithms is thus essential for understanding the security of curve-based cryptosystems. Several record-breaking implementations of this algorithm are available in the literature: over a prime field the current record reaches a discrete log in a 112-bit group on a curve of the form $y^2 = x^3 - 3x + b$ [5,9]. This computation was performed on a Playstation 3. More recently, Bernstein, Lange and Schwabe [6] reported on an implementation on the same platform and for the same curve, in which the use of the negation map gives a speed-up by a factor $\sqrt{2}$. Over binary fields, the current record is a FPGA implementation breaking a discrete logarithm in a 117-bit group [1]. As for the meet-in-the-middle attack, this generic technique is widely used in cryptanalysis to break block ciphers (double and triple DES, GOST [8]), hash functions [11,12] and lattice-based cryptosystems (NTRU [4,17]).

To the best of our knowledge, all existing implementations of parallel collision search algorithms use hash tables to organize memory and allow fast look-up operations. In this paper, we propose a radix tree data structure for storing the distinguished points computed by the algorithm. The basic idea is to associate to each point a string (given for instance by the x-coordinate of the point), partition this string into blocks of fixed length and store each block in a node of the radix tree. This technique has the advantage that both look-up and insertion of a new point in the data structure are done in logarithmic time, in a single operation. We further extend the analysis of the parallel collision search algorithm and show that the runtime complexity depends critically on the value of θ, the probability that a point on the curve is distinguished. In order to minimize the runtime, we deduce an optimal value of θ. This value is supported by our experiments and depends on the data structure and the machine architecture which are chosen for the implementation. We have implemented our radix tree algorithm for discrete logarithms on elliptic curves defined over prime fields. Our benchmarks demonstrate the performance and scalability of our method.

Organisation. Section 2 reviews algorithms for solving the discrete logarithm problem and for meet-in-the-middle attacks. Section 3 describes our choice for the data structure, complexity estimates and comparison with hash tables. In Section 4, we revisit the proof for the time complexity of the collision finding algorithm for a small and a large number of collisions. We furthermore show how to minimize the runtime, in function of the proportion of distinguished points. Finally, Section 5 presents our experimental results.

2 Parallel collision search

In this section we briefly review Pollard's rho method and the parallel algorithm for searching collisions. In order to look for collisions for a function $f : S \to S$ with Pollard's rho method, the idea is to compute a sequence of elements $x_i = f(x_{i-1})$ starting at some random element x_0. Since S is finite, eventually this sequence begins to cycle and we therefore obtain the desired collision $f(x_k) = f(x_{k+t})$, where x_k is the point in the sequence before the cycle begins and x_{k+t} is the last point on the cycle before getting to x_{k+1} (hence $f(x_k) = f(x_{k+t}) = x_{k+1}$). One may show that the expected number of steps taken until the collision is found is $\sqrt{\frac{\pi n}{2}}$, and therefore that the memory complexity is also $O(\sqrt{\frac{\pi n}{2}})$. This algorithm can be further optimized to constant memory time by using Floyd's cycle [10,7]. We do not further detail memory optimizations here since they are inherently of sequential nature and no way to exploit these ideas in a parallel algorithm is currently known.

The parallel version for the collision search was proposed by van Oorschot and Wiener [16] and assigns to each thread the computation of a trail given by points $x_i = f(x_{i-1})$ starting at some point x_0. Only points that belong to a certain subset, called the set of distinguished points, are stored. This set is defined by points having an easily testable property. Whenever a walk computes a distinguished point x_d, it stores in a common list of tuples (x_0, x_d). If two walks collide, this is identified only when they both reached a common distinguished point. We may then re-compute the paths and the points preceding the common point are distinct points which map to the same value.

Solving discrete logarithms. Our focus is on the elliptic curve discrete logarithm in a cyclic group $G = \langle P \rangle$, but the methods described in this paper apply to any cyclic finite group. We will assume that the curve E and the group G are defined over a finite field $\mathbb{F}_p$, where p is a prime number. Let $Q \in G$ and say we want to solve the discrete logarithm problem $Q = xP$, where $x \in \mathbb{Z}$. To apply the ideas explained above, we define a map $F : G \to G$ which behaves randomly and such that each time we compute $f(R)$ we can easily keep track of integers a and b such that $f(R) = aP + bQ$. Pollard's initial proposal for such a function was

$$f(R) = \begin{cases} R + P & \text{if } R \in S_1 \\ 2R & \text{if } R \in S_2 \\ R + Q & \text{if } R \in S_3 \end{cases} \tag{1}$$

where the sets S_i, $i \in \{1, 2, 3\}$ are pairwise disjoint and give a partition of the group G. As a consequence, whenever a collision $f(R) = f(R')$ occurs, we obtain an equality

$$aP + bQ = a'P + b'Q. \tag{2}$$

This allows us to recover $x = (a-a')/(b-b')$, provided that $b-b'$ is not a multiple of r. Starting from R_0, a multiple of P, Pollard's rho [13] method computes a sequence R_i of points where $R_{i+1} = f(R_i)$. Since the group G is finite, this sequence will produce a collision after $\sqrt{\frac{\pi n}{2}}$ iterations on average, where n is the cardinality of the group G. To define distinguished points, we take an easily testable property, such as a certain number of trailing bits of their x-coordinate being zero. Whenever a walk computes such a point, this is stored in a common list, together with the corresponding a and b. If two walks collide, this can not be identified until the computation of the common distinguished point. Then the discrete logarithm is recovered from an equality of type 2.

Meet-in-the-middle attacks. Meet-in-the-middle attacks require finding a collision of the type $f_1(a) = f_2(b)$, where $f_1 : D_1 \to R$ and $f_2 : D_2 \to R$ are two functions with the same co-domain. As explained in [16], solving this equation may be formulated as a collision search problem on a single function $f : S \times \{1, 2\} \to S \times \{1, 2\}$, where the solution we need is of the type:

$$f(a, 1) = f(b, 2). \tag{3}$$

and has some extra specific property. This collision is called *the golden collision*. The number of unordered pairs in S are approximatively $\frac{n^2}{2}$ and the probability that the two points in a pair map to the same value of f is $\frac{1}{n}$. There are $\frac{n}{2}$ expected collisions for f and there may be several solutions to equation 3. Hence one typically assumes that all collisions are equally likely to occur and that in the worst case, all possible $\frac{n}{2}$ collisions for f are generated before finding the golden one. Because so many collisions are generated, memory complexity can be the bottleneck in meet-in-the-middle attacks and the memory constraint becomes an important factor in determining the running time of the algorithm. We further explain this idea in Section 4.

Data structure. To store distinguished points, one may use any data structure allowing efficient look-up and insertion. The most common structure used in the litterature is a hash table. In order to make parallel access to memory possible, van Oorschot and Wiener [16] propose the use of the most significant bits of distinguished points. Their idea is to divide the memory in segments, each corresponding to a pattern on the first few bits. Threads read off these first bits and are directed towards the right segment. Each segment is organized as a hash table, to allow efficient look-up and insertion.

Notation. In the remainder of this paper, we denote by θ the proportion of distinguished points in a set S. We denote by n the number of elements of S. We denote by E an elliptic curve defined over a prime finite field $\mathbb{F}_p$. Whenever the set S is the group $E(\mathbb{F}_p)$, n is the cardinality of this group. For simplicity, in this case, we assume that n is prime (which is the optimal case in implementations).

3 Our approach for the data structure

In this section, we describe the data structure we used for our implementation of the parallel collision search algorithm and explain its advantages over hash tables.

To store the list of distinguished points, we use a radix tree structure. Each point is represented as a number in a base of our choice, denoted by b. For simplicity, we take $b = 10$ here. For example, in the case of attacks on the discrete logs on the elliptic curve, we may represent a point by its x-coordinate. The first entry in the representation of this number gives the root node in the tree, the next entry is a child and so on. According to graph theory, this leads to define an acyclic graph which consists of b connected components (i.e. a forest). As an illustration, Figure 1 shows how to store 7 numbers with 5 digits.

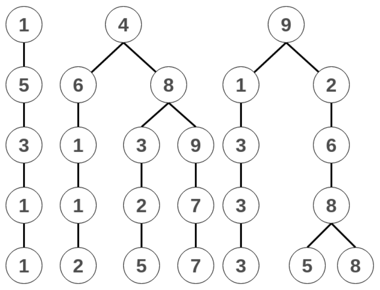


Fig. 1: Example of the structure for the set 15311, 46112, 48325, 48977, 91333, 92685, 92688

The search for a collision and the storage of a new point are done in a single operation. Indeed, the insertion algorithm for a radix tree always starts with a search. This means that when we fail to find the point we already have processed a part of the insertion algorithm. Let us look at an example. We consider the tree in Figure 1. The distinguishing property is having 3 trailing zero bits and the distinguished point that the thread found has the x-coordinate 48956000. Since we do not store the 3 trailing zeros, we are looking for the word 48956. We find the node 4, the node 8 and the node 9 and then, the node 5-child-of-9 does not exist. We realize that the point is not in the tree, so we wish to add it. To add the point, we do not have to start from the root. We are currently on the node 9 and so we can insert a new child with the value of 5, and continue with the insertion algorithm up to the end of the word. The look-up and insertion take $\log p$ time.

Making changes to a shared structure in parallel imposes the use of locks. Note that with a radix tree storing a point does not require locking the entire structure. Since we associate one lock per node, only those affected by the insertion will be locked. As the tree grows bigger, the probability of a thread being stuck on a lock is remarkably small.

Let c be the length of words we store in the tree and K the number of distinguished points computed by our algorithm. To estimate the memory complexity of our approach, we give upper and lower bounds for the number of nodes that will be allocated in the radix tree before a collision is found.

Worst-case scenario. In the worst case scenario, for each new word added in this structure we will create as much nodes as possible. This means that the x-coordinates of the added points have the shortest possible common prefix, as shown in Figure 2. For the first 10 points, we will use $10c$ nodes. After that, the first distinguished point that we find will take $c-1$ nodes, since all possibilities for the first letter in the string were created. We repeat this operation $9 \cdot 10$ times, provided that $K > 10 + 9 \cdot 10$.

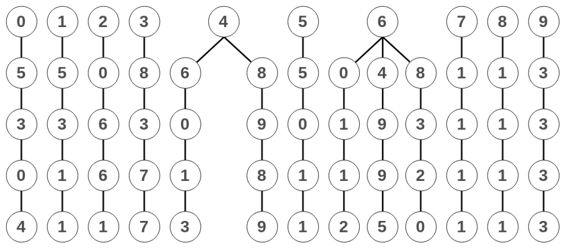


Fig. 2: Worst-case scenario example

More generally, let $k = \lfloor \log_{10} K \rfloor - 1$. We build the tree by allocating nodes as follows:

- $10c$ nodes for the first 10 points
- $9 \cdot 10 \cdot (c-1)$ for the next $9 \cdot 10$ points
- $9 \cdot 10^2 \cdot (c-2)$ for the next $9 \cdot 10^2$ points etc.
- $9 \cdot 10^k \cdot (c-k)$ for $9 \cdot 10^k$ points.

For each of the remaining $K - (10 + \sum_{i=1}^{k} 9 \cdot 10^i)$ points we will need $c-k-1$ nodes. To sum up, the total number of nodes that will bound our worst-case scenario is given by:

$$N(K) = 10c + \sum_{i=1}^{k} 9 \cdot 10^i (c-i) + (K - 10 - 90 \sum_{i=0}^{k-1} 10^i)(c-k-1).$$

The simplification of this formula is detailed in Appendix A and shows that:

$$N(K) \leq 10^{k+1} + K(c-k-1) - \frac{10^2}{9}. \tag{4}$$

Best-case scenario Let K be the number of distinguished points that we need to store and let $k = \lfloor \log_{10} K \rfloor + 1$. In the best-case scenario, we may assume without loss of generality that each time a new point is added in the structure, the minimal number of nodes is used, i.e. the x-coordinates of the added points have the longest possible common prefix. For example, for the first point c nodes are added, for the next 9 nodes, one extra node is allocated and so on, until all

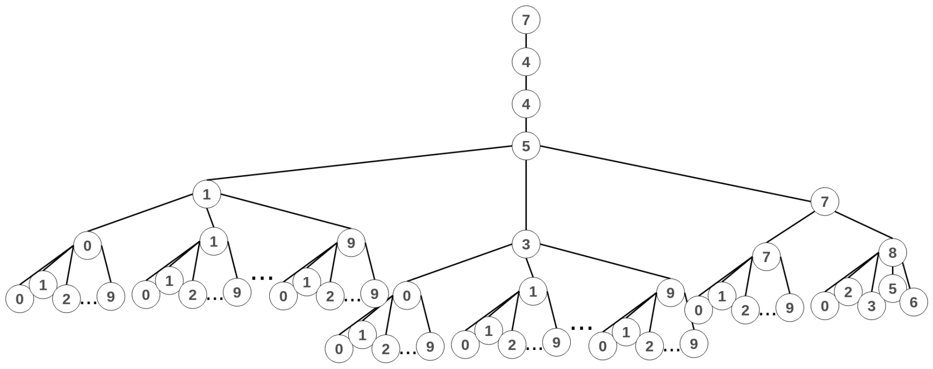


Fig. 3: Best-case scenario example

subtrees of depth 1, 2 etc. are filled one by one. Figure 3 gives an example of how 215 points are stored. If $K > 10^c$, we fill the first tree and start a new one. Let x_i, for $i \in \{0, 1 \ldots, k\}$, denote the i-th digit of K, from right to the left. In full generality, since $c > k$, we use:

- x_k complete subtrees of depth k and a (x_k+1)-th incomplete tree of depth k;
- the (x_k+1)-th tree of depth k has x_{k-1} complete subtrees of depth $k-2$ and a (x_{k-1}+1)-th incomplete tree of depth $k-1$;
- $c - k$ extra nodes.

Summing up all nodes, we get the following formula:

$$N(K) = \sum_{i=0}^{k} x_i \sum_{j=0}^{i} 10^j + k + c - k = \frac{1}{9} \sum_{i=0}^{k} x_i (10^{i+1} - 1) + c = \frac{10}{9} K - c - \frac{1}{9} \sum_{i=0}^{k} x_i.$$

We conclude that:

$$N(K) \geq \frac{10}{9} K - c - k - 1. \tag{5}$$

To conclude we have proved the following:

Proposition 1. *The expected number of nodes in the radix tree verifies the following inequalities:*

$$\frac{10}{9} K - c - \log K - 1 \leq N(K) \leq (c - \log K + 1) K - \frac{10^2}{9}. \tag{6}$$

Experimental evidence for the average case. To have an initial estimate of the average case, we experimented by counting the number of nodes created and

Key size	Points	Nodes	Word length	$\frac{\text{Nodes}}{\text{Points}}$	Worst-case offset	Best-case offset	Between best and worst
45 bits	2720	18925	11	7.10	+3904	-15894	0.80
50 bits	7394	57235	12	7.86	+9885	-49011	0.83
55 bits	21042	178629	13	8.61	+21318	-155239	0.87

Table 1: Average number of nodes in the radix tree data structure for solving ECDLP

comparing them to the number of points stored while solving discrete logarithms on elliptic curves. The results in Table 1 are an average of 200 runs.

The last column shows where the average-case stands on a scale from 0 to 1, where 0 represents the best-case scenario and 1 represents the worst-case scenario calculated by equations 5 and 4 respectively. Columns 6 and 7 give the offset of these values compared to the experimentally obtained average. In column 5 we compare the word length with the nodes to points ratio, we see how much we gain from the common prefixes.

Radix trees versus hash tables. Let's say that the expected number of distinguished points computed by the random walks before a collision is found is K. In order to estimate the amount of memory needed to store distinguished points in the hash table, we have to determine the size of indexes. Assume that the hash values are b-bit numbers, so that we create a large hash table of 2^b numbers, which will be indexed using the hash values. To bound the cost of look-up and insertion, we follow an analysis given in [10, Sect. 6.3.2] and choose the size of b so that we avoid obtaining multicollisions on the hash functions on elements from the list of distinguished points computed during the algorithm. Assuming that hash values behave as random numbers, then to avoid 3-multicollisions for instance, we take

$$K^3/6 \sim 2^{2b}.$$

By approximating $K \sim \theta\sqrt{\frac{\pi n}{2}}$, this yields $b \sim \frac{3}{2}\log\theta + \frac{3}{4}\log\frac{\pi n}{2}$ and a memory complexity of the hash table algorithm of :

$$O(\theta^{\frac{3}{2}}\left(\frac{\pi n}{2}\right)^{\frac{3}{4}}). \tag{7}$$

In the case of discrete logarithms, we will compare this against the worst-case complexity of our radix tree data structure. By a theorem of Hasse [14], we know that the number of points on the curve is given by $n = p + 1 - t$, with $|t| \leq 2\sqrt{p}$. Since we assume that n is prime, we approximate $\log n \sim \log p$. Hence an approximation of the number of nodes in the radix tree in the the worst-time is:

$$(\frac{1}{2}\log n - \log\frac{\sqrt{\pi}\theta}{\sqrt{2}} + 1)\theta\sqrt{\frac{\pi n}{2}}.$$

This shows that the memory complexity obtained for the algorithm by using our data structure is better than the one of the hash table version.

As radix trees, hash tables have $\log p$ look-up and insertion times, since one needs to take into account the cost of hashing the key. In the worst case, look-up and insertion may take longer, depending on how collisions are handled. To all these, we have to add the cost of comparing the two keys, the one to be inserted with the one already stored. We have to do this comparison to see if the discovered collision is a distinguished point collision or just a hash table one. Whereas with radix trees, when we find the corresponding node and it already has a value, we are sure to have found the desired collision.

4 Time complexity

This section provides a time analysis of the parallel collision search algorithm. Van Oorschot and Wiener [16] proved that the runtime of their algorithm is

$$O\left(\frac{1}{L}\sqrt{\frac{\pi n}{2}}\right),$$

with L the number of threads we use.

We revisit the steps of their proof and show a careful analysis of the running time. Our theoretical model provides a more rigorous argument for linear scalability and indicates that the actual running time of the algorithm depends on the proportion of distinguished points.

Theorem 1. *The total running time of the parallel collision search algorithm is*

$$f(\theta) = (\frac{1}{L}\sqrt{\frac{\pi n}{2}} + \frac{1}{\theta})t_c + (\frac{\theta}{L}\sqrt{\frac{\pi n}{2}})t_s, \tag{8}$$

where t_c and t_s are constants of the time it takes to compute and to store a point respectively.

Proof. Let $T = 1$ be the moment when every thread has found the first distinguished point. The number of distinguished points stored in the common structure at $T = 1$ is L. Note that the length of a short walk is $\frac{1}{\theta}$, thus every thread has calculated $\frac{1}{\theta}$ points at the moment $T = 1$. The probability of not having a collision at $T = 1$ is

$$1 - \frac{L}{n\theta}.$$

Even though the collision is discovered only when a distinguished point is found, two trails can collide at any given point on the short walk. Furthermore, any of the L threads can cause a collision. Thus, the probability for all threads of not finding a collision on any point on the short walk is:

$$(1 - \frac{L}{n\theta})^{\frac{1}{\theta}L},$$

at the moment $T = 1$.

Let X be the number of distinguished points calculated per thread before duplication. The probability of not having a collision after each thread has found and stored T distinguished points is

$$P(X > T) = (1 - \frac{L}{n\theta})^{\frac{L}{\theta}} \cdot (1 - \frac{2L}{n\theta})^{\frac{L}{\theta}} \cdot \ldots \cdot (1 - \frac{TL}{n\theta})^{\frac{L}{\theta}}.$$

To do this multiplication we are going to take a shortcut. When x is close to 0, a coarse first-order Taylor approximation for e^x as:

$$e^x \approx 1 + x.$$

Now we can rewrite our expression to:

$$\begin{aligned} P(X > T) &= (e^{-\frac{L}{n\theta}} \cdot e^{-\frac{2L}{n\theta}} \cdot \ldots \cdot e^{-\frac{TL}{n\theta}})^{\frac{L}{\theta}} = \\ &= (e^{\frac{-(L+2L+\ldots+TL)}{n\theta}}))^{\frac{L}{\theta}} = \\ &= (e^{\frac{-T(T+1)L}{2n\theta}})^{\frac{L}{\theta}} = \\ &= (e^{\frac{-T^2L}{2n\theta}})^{\frac{L}{\theta}} = e^{\frac{-T^2L^2}{2n\theta^2}}. \end{aligned} \tag{9}$$

This gives us the probability

$$P(X > T) = e^{\frac{-T^2L^2}{2n\theta^2}},$$

thus the expected number of distinguished points found before duplication, is

$$E(X) = \sum_{T=1}^{\infty} T \cdot P(X = T) = \sum_{T=1}^{\infty} T \cdot (P(X > T-1) - P(X > T)) = \sum_{T=0}^{\infty} P(X > T).$$

We approximate

$$E(X) = \sum_{T=0}^{\infty} e^{\frac{-T^2L^2}{2n\theta^2}} \approx \int_0^{\infty} e^{\frac{-x^2L^2}{2n\theta^2}} dx \approx \frac{\theta}{L}\sqrt{\frac{\pi n}{2}}.$$

Since the length of a short walk is $\frac{1}{\theta}$, the number of calculated points (distinguished or not) before a collision occurs is

$$\frac{1}{L}\sqrt{\frac{\pi n}{2}}.$$

However, a collision might occur on any point on the walk and it will not be detected until the walk reaches a distinguished one. We add $\frac{1}{\theta}$ to the number of calculations for the discovery of a collision. Finally, the expected number of calculated points per thread is:

$$\frac{1}{L}\sqrt{\frac{\pi n}{2}} + \frac{1}{\theta}.$$

The two main operations in our algorithm are computing the next point on the random walk and storing a distinguished point. Thus, the time complexity of our algorithm is:

$$f(\theta) = (\frac{1}{L}\sqrt{\frac{\pi n}{2}} + \frac{1}{\theta})t_c + (\frac{\theta}{L}\sqrt{\frac{\pi n}{2}})t_s. \tag{10}$$

□

Remark 1. Note that the analysis above shows that the number of points computed by the algorithm is $O\left(\theta\sqrt{\frac{\pi n}{2}}\right)$. This was proven by van Oorschot and Wiener in the first place.

As we can see in Equation (8), the proportion of distinguished points we choose can influence our time complexity. The optimal value for θ is the one that gives us the minimal time complexity. The time complexity is minimal at a point at which its derivative with respect to θ is 0. In our implementation for discrete logarithms, tests show that both computing and storing a point take 1 microsecond. We replace constants t_c and t_s with 1 to look for the minimal value of our runtime function, by computing the zeros of its derivative:

$$f'(\theta) = \frac{1}{L}\sqrt{\frac{\pi n}{2}} - \frac{1}{\theta^2}.$$

We deduce that the optimal number of trailing bits is $\frac{\log n}{4}$. In Section 5 we give timings for our implementation of the attack, which support these theoretical findings. Although we do not dispose of an implementation of the hash table variant of the algorithm and we are unable at the time of the submission to give a full analysis in this case, we are inclined to believe that the choices that were made in implementations available in the literature [9,6] were optimal or close to optimal. We detail their choices in Section 5. Most importantly, our analysis puts forward the idea that the optimal choice for θ depends essentially on the choices made for the implementation and the memory management.The optimal value for θ calculated here justifies our results in Figures 6 and 7.

4.1 Finding many collisions: Meet-in the middle attacks

Using a simplified complexity analysis, van Oorschot and Wiener [16] put forward the following heuristic.

Heuristic. Let n be the cardinality of the set S. For a memory which can hold w distinguished points, the (conjectured) optimum proportion of distinguished points is $\theta \sim 2.25\sqrt{\frac{w}{n}}$. Under this assumption, the expected number of iterations required to complete a meet-in-the-middle attack using these parameters is $O(\frac{n^{3/2}}{w})$.

This heuristic suggests that in the case of meet-in-the-middle attacks, our parallel collision search with radix tree algorithm will yield a memory improvement, and also a better time complexity. We revisit their proof and give a more refined analysis for the running time of a parallel collision search for finding $\frac{n}{2}$ collisions.

Theorem 2. *In the parallel collision search algorithm, an upper bound for the expected time to find $\frac{n}{2}$ collisions with a memory constraint of w words is given by:*

$$\frac{1}{L}\left(\frac{w}{\theta}+(\frac{n}{2}-\frac{\log\left(\frac{(\alpha-1)\omega\sqrt{2}}{\theta\sqrt{\pi n}}+1\right)}{\log\alpha})\frac{n\theta}{w}\right)+\frac{n}{2\theta}. \tag{11}$$

Proof. Let X be the number of distinguished points calculated per thread before duplication. Let T_1 be the number of distinguished points computed until the first collision was found, and T_i, for any $i > 1$, the number of points stored in the memory after the $(i-1)$th collision was found and before the ith collision is found.

As shown in Theorem 1, the expected number of points stored before finding the first collision is $T_1 = \theta\sqrt{\frac{\pi n}{2}}$. The probability of not having found the second collision after each thread has found and stored T distinguished points is

$$P(X > T) = (1-\frac{L+T_1}{n\theta})^{\frac{L}{\theta}}\cdot(1-\frac{2L+T_1}{n\theta})^{\frac{L}{\theta}}\cdot\ldots\cdot(1-\frac{TL+T_1}{n\theta})^{\frac{L}{\theta}}.$$

As in the proof of Theorem 1, we approximate this expression by

$$P(X > T) = e^{\frac{-T^2L^2-2LT_1T}{2n\theta^2}}.$$

Hence the expected number of distinguished points computed by a thread before the second collision is:

$$\begin{aligned} E(X) &= \sum_{T=0}^{\infty} e^{\frac{-T^2L^2-2LT_1T}{2n\theta^2}} \approx \int_0^{\infty} e^{\frac{-x^2L^2-2xLT_1}{2n\theta^2}}dx = \\ &= e^{\frac{T_1^2}{2n\theta^2}}\int_0^{\infty} e^{\frac{-(xL+T_1)^2}{2n\theta^2}}dx = \frac{\theta\sqrt{n}}{L}e^{\frac{T_1^2}{2n\theta^2}}\int_{\frac{T_1}{\theta\sqrt{2n}}}^{\infty} e^{-t^2}dt \\ &= \frac{\theta\sqrt{2n}}{L}e^{\frac{T_1^2}{2n\theta^2}}\left(-\frac{e^{-t^2}}{2t}\Bigg|_{\frac{T_1}{\theta\sqrt{2n}}}^{\infty} - \int_{\frac{T_1}{\theta\sqrt{n}}}^{\infty}\frac{e^{-t^2}}{2t^2}\right). \end{aligned}$$

Since the last integral is negligible, we get that the number of distinguished points stored during the search for the second collision T_2 is given by $\approx \theta\sqrt{\frac{2n}{\pi}} = \alpha T_1$, with $\alpha = \frac{2}{\pi}$. By redoing the computation above in order to estimate T_i one may show that $T_i \le \alpha T_{i-1}$. To simplify the computation, we assume that in the worst case $T_i = \alpha T_{i-1}$. Hence the total number of points stored until i collisions are found is bounded by:

$$T_1 + T_2 + \ldots T_{i-1} = \frac{1-\alpha^i}{1-\alpha}T_1.$$

Hence the memory will fill after computing the first $k = \frac{\log\left(\frac{(\alpha-1)\omega\sqrt{2}}{\theta\sqrt{\pi n}}+1\right)}{\log \alpha}$ collisions and the expected total time for one thread is:

$$\frac{1}{L}\sum_{i=1}^{k}\alpha^{i-1}\frac{1}{\theta}T_1 = \frac{w}{L\theta}.$$

On the other hand, as shown in [16], when the memory is full, the time to find a collision is $\frac{n\theta}{Lw}$. To sum up, the total time to find $\frac{n}{2}$ collisions is:

$$\frac{1}{L}\left(\frac{w}{\theta} + (\frac{n}{2} - \frac{\log\left(\frac{(\alpha-1)w\sqrt{2}}{\theta\sqrt{\pi n}}+1\right)}{\log\alpha})\frac{n\theta}{w}\right) + \frac{n}{2\theta}$$

□

By minimizing the complexity function obtained in Theorem 2 , we obtain an estimate for the optimal value of θ to take in order to minimize the running time of the algorithm.

Corollary 1. *The optimum proportion of distinguished points minimizing the time complexity bound in Theorem 2 is $\theta \sim \frac{\sqrt{w(2w+Ln)}}{Ln}$. Furthermore, by choosing this value for θ, the running time of the parallel collision search algorithm for finding $\frac{n}{2}$ collisions is bounded by:*

$$O\left(\frac{n}{L}\frac{\sqrt{2w+Ln}}{2L\sqrt{w}}\right). \tag{12}$$

Proof. In order to be able to formally compute a minimal value of θ depending on n and w, we approximate the logarithm in Equation 11 by $\frac{(\alpha-1)w\sqrt{2}}{(\alpha-1)w\sqrt{2}+\theta\sqrt{n}}$. Hence, the time complexity is bounded by:

$$f(\theta) = \frac{1}{L}\left(\frac{w}{\theta} + (\frac{n^2\theta}{2w} - \frac{\sqrt{n}}{w})\right) + \frac{n}{2\theta}.$$

By minimizing this expression, we obtain that by taking $\theta \sim \frac{\sqrt{w(2w+Ln)}}{Ln}$, the time complexity is $O\left(\frac{n}{L}\frac{\sqrt{2w+Ln}}{2L\sqrt{w}}\right)$. □

This confirms the heuristic findings in [16]. Most importantly, the running time complexity in Equation (12) suggests that in the case of meet-in-middle applications which fill the memory available, our approach will yield a faster algorithm, since we can store a large number of words.

5 Implementation and benchmarks

To support our findings, we implemented the parallel collision search with radix trees for discrete logarithms on elliptic curves defined over prime fields. Our implementation is in C and the external libraries we used are The GNU Multiple

Precision Arithmetic Library [2] for large numbers arithmetic, and the OpenMP (Open Multi-Processing) interface [3] which supports shared memory multiprocessing programming. Our tests were performed on a 16-core Intel Xeon E5-2680 processor and we experimented using between 2 and 16 threads.

Additive walks. Teske [15] showed experimentally that the walk proposed by Pollard 1 originally performs on average slightly worse than a random walk. Teske proposes alternative mappings that lead to the same performance as expected in the random case: additive walks and mixed walks. The additive walks are presented as follows. Let r be the number of sets S_i which give a partition of the group G, and let M_i be a linear combination of P and Q: $M_i = a_iP + b_iQ$, for $i = \overline{1, r}$. We choose the iterating function of the form:

$$R_{i+1} = f(R_i) = \begin{cases} R_i + M_1, & R_i \in S_1; \\ R_i + M_2, & R_i \in S_2; \\ R_i + M_3, & R_i \in S_3; \\ \dots \\ R_i + M_r, & R_i \in S_r. \end{cases} \tag{13}$$

In the case of mixed walks, we introduce squaring steps to r-additive walks. However, Teske's experimetal results show that apart from the case $r = 3$, the introduction of squaring steps does not lead to a significantly better performance. After experimenting with both of them, we confirmed his conclusion and decided to use additive walks.

Teske shows experimentally that if $r \geq 20$ then additive walks are close to random walks. We therefore chose $r = 20$ in our implementation.

Use of automorphisms If the function f is chosen such that $f(R) = f(-R)$ then we may regard f as being defined on equivalence classes under $\pm$. Since there are $\frac{n}{2}$ equivalence classes, this would lead to a theoretical speed-up of $\sqrt{2}$. However, it was observed that the use of the negation map leads to so-called fruitless cycles, cycles that trap the random-walks. In practice, since these cycles need to be handled, the actual speed-up is significantly less than $\sqrt{2}$ and actually depends on the platform one uses [6]. In this paper, we aim at evaluating the performance of our algorithm independently of the platform one may choose for its implementation. Therefore, we do not use automorphisms in our implementation.

Long walks vs. short walks As we explained in section 2 every thread selects a starting point, which is a multiple of P, and computes the random walk until a distinguished point is found. After the distinguished point is stored in the radix tree, the thread starts a new walk from a new starting point. We refer to this as a short walk because the walk stops at the first distinguished point and has an average length of $1/\theta$. A second possibility is that the thread would continue the walk from a distinguished point rather than start from a new one. We refer to this as a long walk. This approach is used in the Pollard's rho method because it allows the walk to enter a cycle and is an indispensable factor in finding a collision using the Floyd's cycle finding algorithm[10]. However, in the Parallel Collision Search algorithm every distinguished point is stored, and thus the cycle property is irrelevant.

Furthermore, using the short walk method we are not required to calculate the coefficients a and b every time. We calculate only the value of R for each iteration, and when we find a distinguished point we store the coefficients of the starting point (only the coefficient a is stored because the starting point being a multiple of P, the b coefficient is zero). It is only when a collision is found that we start iterating from the beginning of the short walk, this time computing a and b. This convenience makes short walks the better choice. Furthermore, we experimented with both short walks and long walks, finding that short walks give slightly better runtime results. All our results presented here use short walks.

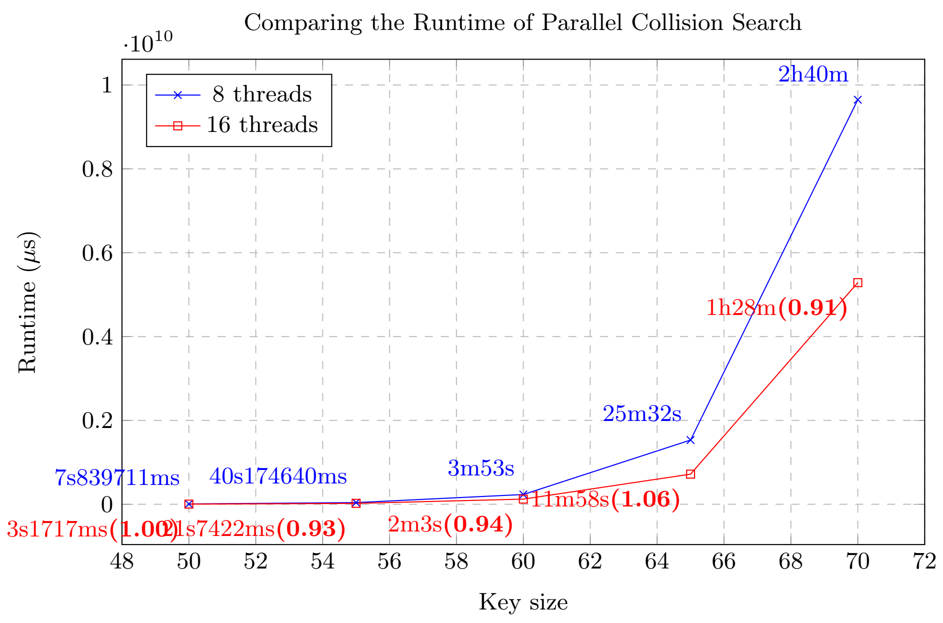
Parallel Performance We were interested in the parallel performance i.e. how efficient our program is when increasing the number of parallel processing elements. A program is considered to scale linearly if the speedup is equal to the number of threads used. In the theoretical model [16], the Parallel Collision Search is considered to have a linear scalability. This means that if we implement L threads, we will find the discrete logarithm L times faster than we would using one thread. The parallel performance was measured from two points of view:

- The runtime.
- The number of calculated (distinguished and non distinguished) points.

Figures 4 and 5 present our results. Every value is an average of 100 runs and the parallel performance is indicated in parentheses. When working with 50-bit or bigger field curves, the parallel performance of our method is linear.

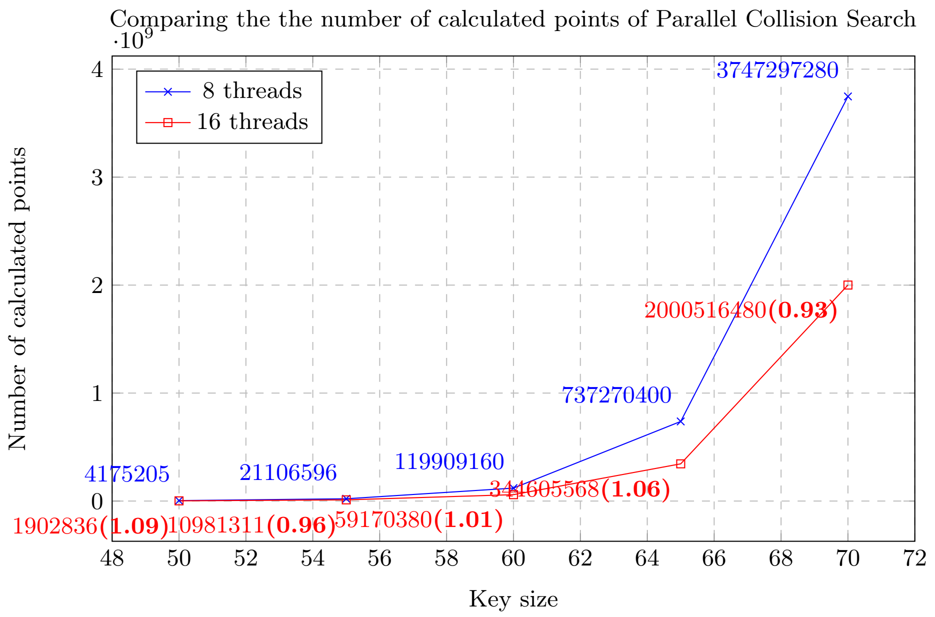
Distinguishing property The probability θ of a point being distinguished is a significant factor in the time complexity of the algorithm, as demonstrated in in section 4. The results in Figures 6 and 7 show how the runtime depends on the value of θ. Once more, every value is computed as an average of 100 runs. The x-axis represents the number of trailing zero bits that the x-coordinate of a point must have in order to satisfy the distinguishing property. Note that when the number of trailing zero bits is s, the value of θ is $\dfrac{1}{2^s}$.

We conclude that the optimal choice for the number of trailing zero bits is $\frac{B}{4}$, for a B-bit curve. Note however that all values in the interval $[\frac{B}{6}, \frac{B}{3}]$ for a B-bit field curve are close to this optimal value, while very small or very large values of θ increase the runtime significantly. Our results are justified by the optimal value for θ that we obtain theoretically in Section 4. For comparison, Bos et al. [5] chose 24 trailing bits for the 112-bits curve and Bernstein, Lange and Schwabe [6] took 20 trailing bits for the same size.



Comparing the Runtime of Parallel Collision Search
$\cdot 10^{10}$
8 threads
16 threads
Runtime (μs)
Key size
2h40m
1h28m(0.91)
25m32s
3m53s
11m58s(1.06)
7s839711ms
40s174640ms
3s1717ms(1.00)
21s7422ms(0.93)
2m3s(0.94)

Fig. 4



Comparing the the number of calculated points of Parallel Collision Search
$\cdot 10^{9}$
8 threads
16 threads
Number of calculated points
Key size
3747297280
2000516480(0.93)
737270400
119909160
344605568(1.06)
4175205
21106596
1902836(1.09)
10981311(0.96)
59170380(1.01)

Fig. 5

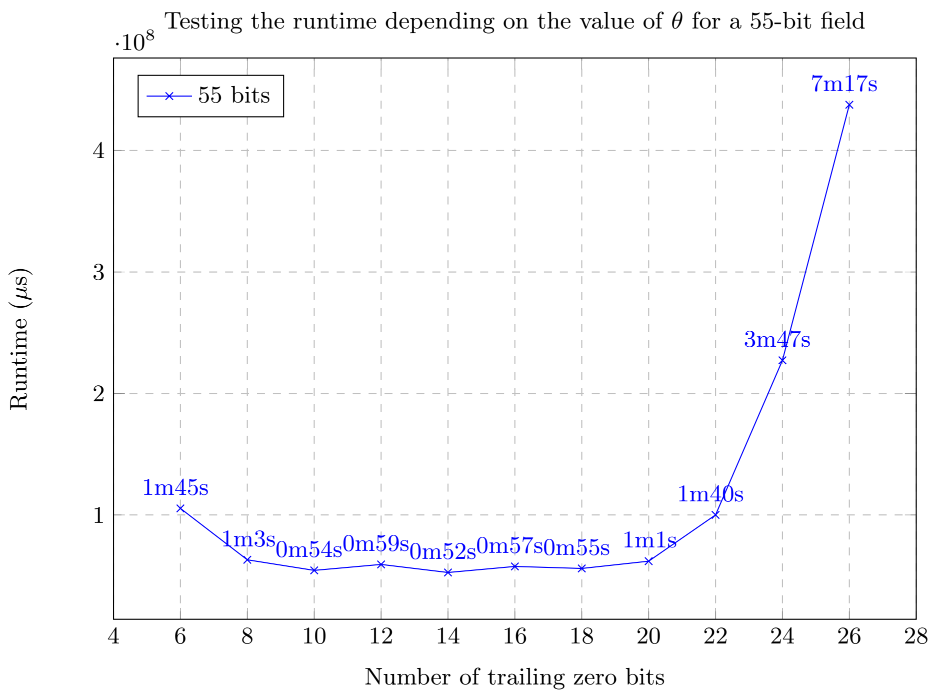

Testing the runtime depending on the value of θ for a 55-bit field
$\cdot 10^8$
55 bits
7m17s
3m47s
1m45s
1m40s
1m3s
0m54s
0m59s
0m52s
0m57s
0m55s
1m1s
Runtime (μs)
Number of trailing zero bits

Fig. 6

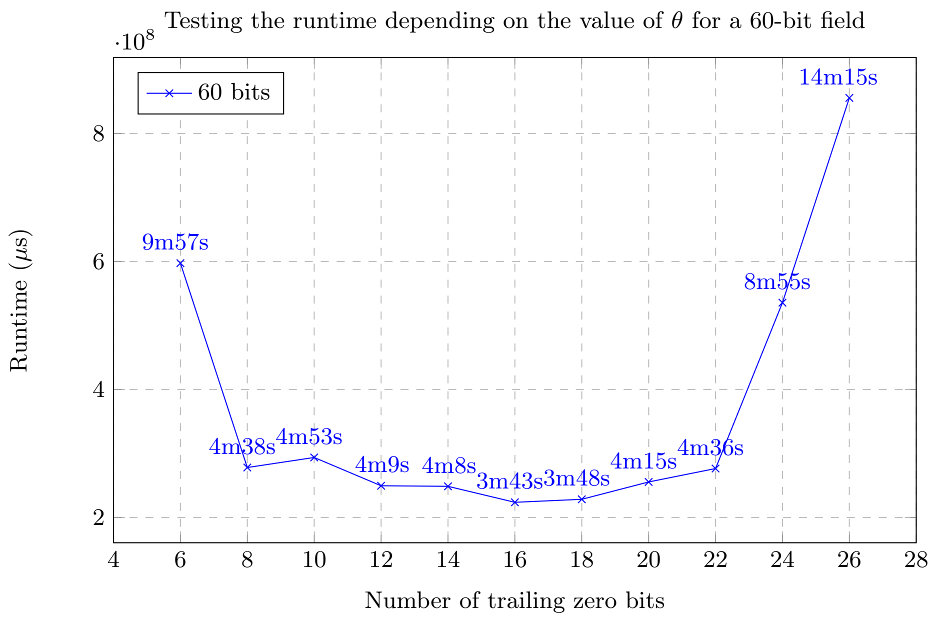

Testing the runtime depending on the value of θ for a 60-bit field
$\cdot 10^8$
60 bits
14m15s
9m57s
8m55s
4m38s
4m53s
4m9s
4m8s
3m43s
3m48s
4m15s
4m36s
Runtime (μs)
Number of trailing zero bits

Fig. 7

6 Conclusion

We proposed an alternative memory structure for the parallel collision search algorithm proposed by van Oorschot and Wiener [16]. We show that this structure yields a better memory complexity than the hash table variant of the algorithm. We revisited the time complexity of the parallel collision search and explained how to choose the optimal value for the proportion of distinguished points when implementing this algorithm. Moreover, using the new memory structure, we obtained a better bound for the time complexity of the parallel collision search, in the case where a large number of collisions is needed. Finally, we implemented the radix tree parallel collision search algorithm for solving discrete logarithms and showed its scalability.

References

1. Faster elliptic-curve discrete logarithms on FPGAs. `https://eprint.iacr.org/2016/382`.
2. GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`.
3. Open Multi-Processing Specification for Parallel Programming. `https://gmplib.org/`.
4. A meet-in-the-middle attack on an NTRU private key. Tehnical report, NTRU Cryptosystems, 2003.
5. Playstation 3 computing breaks 2^{60} barrier; 112-bit prime ECDLP solved. `http://lacal.epfl.ch/112bit_prime`, 2009.
6. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. On the Correct Use of the Negation Map in the Pollard rho Method. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC 2011: 14th International Conference on Theory and Practice of Public Key Cryptography*, volume 6571 of *Lecture Notes in Computer Science*, pages 128–146, Taormina, Italy, March 6–9, 2011. Springer, Heidelberg, Germany.
7. Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20:176–184, 1980.
8. Takanori Isobe. A single-key attack on the full GOST block cipher. In Antoine Joux, editor, *Fast Software Encryption – FSE 2011*, volume 6733 of *Lecture Notes in Computer Science*, pages 290–305, Lyngby, Denmark, February 13–16, 2011. Springer, Heidelberg, Germany.
9. Peter L. Montgomery Joppe W. Bos, Marcelo E. Kaihara. Pollard rho on the playstation 3. Workshop record of SHARCS'09 `http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf`, 2009.
10. Antoine Joux. *Algorithmic Cryptanalysis*, chapter 7, pages 225–226. Chapman & Hall/CRC, 2009.
11. Dmitry Khovratovich, Ivica Nikolic, and Ralf-Philipp Weinmann. Meet-in-the-middle attacks on SHA-3 candidates. In Orr Dunkelman, editor, *Fast Software Encryption – FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 228–245, Leuven, Belgium, February 22–25, 2009. Springer, Heidelberg, Germany.
12. Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The rebound attack: Cryptanalysis of reduced Whirlpool and Grøstl. In Orr

Dunkelman, editor, *Fast Software Encryption – FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 260–276, Leuven, Belgium, February 22–25, 2009. Springer, Heidelberg, Germany.
13. John Pollard. Monte Carlo methods for index computation $\pmod p$. *Math. Comp.*, (32):918–924, 1978.
14. J. H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, 1986.
15. Edlyn Teske. On random walks for Pollard's rho method. *Math. Comp.*, 70(234):809–825, 2001.
16. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
17. C. van Vredendaal. Reduced memory meet-in-the-middle attack against the NTRU private key. *LMS Journal of Computation and Mathematics*, 19(Issue A (Algorithmic Number Theory Symposium XII)):43–57, 2016.

A Appendix : Worst-case scenario

The total number of nodes in the worst-case scenario is given by:

$$N(K) = 10c + \sum_{i=1}^{k} 9 \cdot 10^i (c-i) + (K - 10 - 90 \sum_{i=0}^{k-1} 10^i)(c-k-1).$$

First, we simplify the geometric progression: $\sum_{i=1}^{k} 9 \cdot 10^i (c-i)$, denoted by S_k.

$$\begin{aligned} S_k &= 9 \cdot 10^1 \cdot (c-1) + 9 \cdot 10^2 \cdot (c-2) + \cdots + 9 \cdot 10^k \cdot (c-k) \\ 10S_k &= 9 \cdot 10^2 \cdot (c-1) + \cdots + 9 \cdot 10^k \cdot (c-(k-1)) + 9 \cdot 10^{k+1} \cdot (c-k) \end{aligned}$$

$$\begin{aligned} 10S_k - S_k &= -(9 \cdot 10^1 \cdot (c-1)) + \quad 9 \cdot 10^2 + 9 \cdot 10^3 + \cdots + 9 \cdot 10^k \quad + 9 \cdot 10^{k+1} \cdot (c-k) \\ &= -(9 \cdot 10^1 \cdot (c-1)) + \quad \sum_{i=2}^{k} 9 \cdot 10^n \quad + 9 \cdot 10^{k+1} \cdot (c-k) \\ &= -(9 \cdot 10^1 \cdot (c-1)) + 10^{k+1} - 10^2 + 9 \cdot 10^{k+1} \cdot (c-k) \end{aligned}$$

Thus we have that

$$\begin{aligned} N(K) &= 10c - 10(c-1) + \frac{1}{9}10^{k+1} - \frac{10^2}{9} + 10^{k+1}(c-k) + (K - 10^{k+1})(c-k-1) \\ &= \frac{10}{9}10^{k+1} + K(c-k-1) + 10c - 10(c-1) - \frac{10^2}{9}. \end{aligned}$$

We approximate:

$$N(K) \approx 10^{k+1} + K(c-k-1) - \frac{10^2}{9}.$$