Rigorous Time-Memory Trade-offs for Parallel Collision Search

Monika Trimoska and Sorina Ionica and Gilles Dequen

Laboratoire MIS, Université de Picardie Jules Verne
33 Rue Saint Leu Amiens 80039 - 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 and allow fast memory access. We provide rigorous theoretical evidence that memory is an important factor in determining the runtime of this method. We propose to replace the traditional hash table by a simple structure, inspired by radix trees, which saves space and provides fast look-up and insertion. In the case of many-collision search algorithms, our variant has a constant-factor improved runtime. We give benchmarks that evaluate the linear parallel performance of the attack on ECDLP.

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

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 [13], 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 [17], builds on Pollard's rho method, and is expected to have a linear speedup compared to the sequential version. This 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 [4,18]).

First, our contribution is to extend the analysis of the parallel collision search algorithm and present a formula for the expected runtime to find any given number of collisions, with and without a memory constraint. We show how to compute optimal values of θ, allowing to minimize the running time of collision search, both in the case of discrete logarithms and meet-in-the-middle attacks. In the case where the available memory is limited, we determine the optimal value of θ, proving rigorously that the value conjectured by van Oorschot and Wiener was asymptotically correct. Going further in the analysis, our formulae show that the actual running time of finding-many-collisions algorithm is critically reduced if the number of words w that can be stored in memory is larger.

Secondly, we focus on the data structure used for the algorithm. 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 investigate the use of 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 a point on the elliptic curve), partition this string into blocks of fixed length and store each block in a node of the radix tree. By considering only the dense upper part of this tree, we introduce a new structure, called Packed Radix-Tree-List, and show that the use of this structure leads to a better use of memory in implementations and thus yields improved running times for many-collision applications.

Using the PRTL structure, we have implemented the parallel collision search algorithm for discrete logarithms on elliptic curves defined over prime fields. Our benchmarks demonstrate the performance and scalability of this method.

Organisation. Section 2 reviews algorithms for solving the discrete logarithm problem and for meet-in-the-middle attacks. In Section 3, 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. Section 4 describes our choice for the data structure, complexity estimates and comparison with hash tables. 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 [17] 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 (ECDLP) 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 [14] 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 (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 [17], 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 3.

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 literature is a hash table. In order to make parallel access to memory possible, van Oorschot and Wiener [17] 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 memory structure on its own.

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 Time complexity

Van Oorschot and Wiener [17] gave formulae for the expected running time of parallel collision search algorithms. In this section, 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 yields runtimes which asymptotically coincide to those in [17]. However, our refined formulae indicate that the actual running time of the algorithm depends on the proportion of distinguished points and allow us to rigorously determine the optimal choice of θ for actual implementations.

3.1 Finding one collision: elliptic curve discrete logarithm

Van Oorschot and Wiener [17] proved that the runtime for finding one collision is

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

with L the number of threads we use.

Theorem 1. *In the parallel collision search algorithm, the expected running time to find one collision is*

$$f(\theta) = (\frac{1}{L}\sqrt{\frac{\pi n}{2}} + \frac{1}{\theta})t_c + (\frac{\theta}{L}\sqrt{\frac{\pi n}{2}})t_s, \tag{4}$$

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

Proof. We call short path the chain of points computed by a thread between two consecutive distinguished points. Let $T = 0$ 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 $T = 1$, for one thread 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:

$$\begin{aligned} P(X > T) &= (e^{-\frac{L}{n\theta}} \cdot e^{-\frac{2L}{n\theta}} \cdot \ldots \cdot e^{-\frac{TL}{n\theta}})^{\frac{L}{\theta}} = \\ &= (e^{\frac{-(L+2L+\ldots+TL)}{n\theta}}))^{\frac{L}{\theta}} = \\ &= (e^{\frac{-T(T+1)L}{2n\theta}})^{\frac{L}{\theta}} = \\ &= (e^{\frac{-T^2L}{2n\theta}})^{\frac{L}{\theta}} = e^{\frac{-T^2L^2}{2n\theta^2}}. \end{aligned} \tag{5}$$

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:

$$f(\theta) = (\frac{1}{L}\sqrt{\frac{\pi n}{2}} + \frac{1}{\theta})t_c + (\frac{\theta}{L}\sqrt{\frac{\pi n}{2}})t_s. \tag{6}$$

□

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 (4), the proportion of distinguished points we choose will influence our time complexity. The optimal value for θ is the one that gives the minimal run complexity. 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. From this formula, we easily deduce that if the proportion of distinguished points is too small or too large, the running time of the algorithm increases significantly.

By estimating the ratio t_s/t_c for a given implementation, one can extrapolate the optimal value of θ by computing the zeros of the derivative:

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

Figure 1 gives timings for our implementation of the attack, using a hash table to store distinguished points. Timings shown on the figure are a mean for 100 tests on a 65-bits curve and support our theoretical findings.

Note that most recent implementations available in the literature choose the number of trailing bits giving the distinguished point property in a range between $0.178\log n$ and $0.256\log n$ (see [1,6,9]). This value was determined by experimenting on curves defined over small size fields. Our theoretical findings confirm that these values were close to optimal, but we support the idea that for future record-breaking implementations, the value of θ should be determined as explained above.

3.2 Finding many collisions: Meet-in the middle attacks

Using a simplified complexity analysis, van Oorschot and Wiener [17] 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 $O(\frac{n}{L}\sqrt{\frac{n}{w}})$.

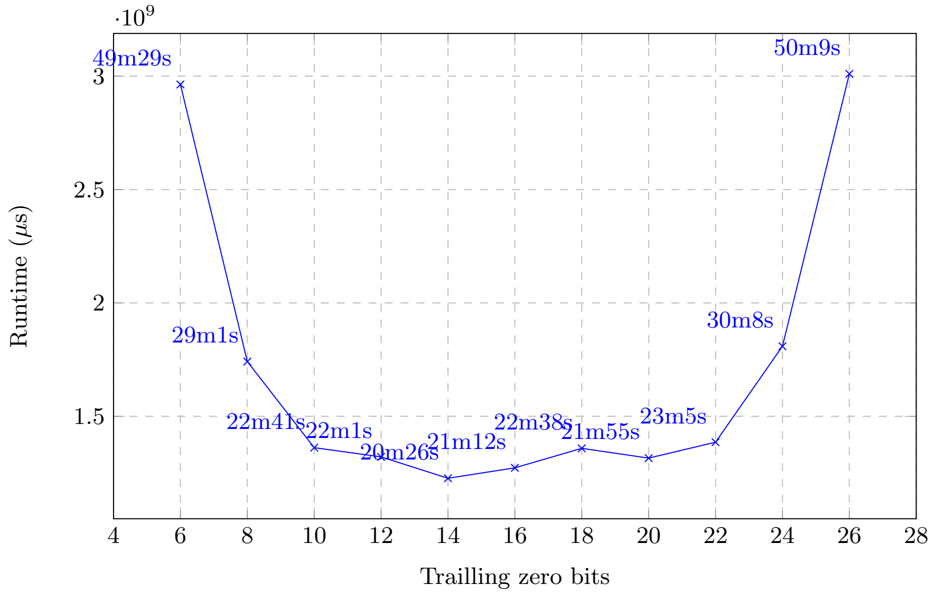

Fig. 1: Timings of solving ECDLP for different values of θ, 65-bits curve, 28 threads

This heuristic suggests that in the case of meet-in-the-middle attacks, a memory data structure allowing to store more distinguished points will yield a better time complexity. To prove the conjectured runtime, we first give a more refined analysis for the running time of a parallel collision search for finding m collisions.

Theorem 2. *In the parallel collision search algorithm, the expected running time to find m collisions with a memory constraint of w words is:*

$$\frac{1}{L}\left(\frac{w}{\theta}+(m-\frac{w^2}{2\theta^2 n})\frac{\theta n}{w}\right)+\frac{m}{\theta}. \tag{7}$$

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:

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

We denote by

$$U_k = T_1 + T_2 + \ldots T_k.$$

By applying repeatedly the formula above (and neglecting the last integral), we have that $T_k = \frac{\theta^2 n}{LU_{k-1}}$. Therefore we have $U_k = U_{k-1} + \frac{\theta^2 n}{LU_{k-1}}$. By letting $V_k = \frac{LU_k}{\theta\sqrt{n}}$, we obtain a sequence given by the recurrence formula

$$V_k = V_{k-1} + \frac{1}{V_{k-1}}.$$

We will use the Cesaro-Stolz criterion to prove the convergence of this limit. First, we note that this sequence is increasing and tends to ∞. Moreover we have that $V_k^2 = V_{k-1}^2 + 2 + \frac{1}{V_{k-1}^2}$. Hence $\frac{V_k^2 - V_{k-1}^2}{k+1-k} \to 2$ and as per Cesaro-Stolz we have $V_k \sim \sqrt{2k}$. We conclude that $U_k \sim \frac{\theta\sqrt{2kn}}{L}$.
Since U_k is the number of distinguished points computed per thread, the total number of stored points is $\theta\sqrt{2kn}$. Hence the memory will fill when $\theta\sqrt{2kn} = w$. This will occur after computing the first $k_w = \frac{w^2}{2\theta^2 n}$ collisions and the expected total time for one thread is:

$$\frac{w}{L\theta}.$$

When the memory is full, the time to find a collision is $\frac{\theta n}{Lw}$. To sum up, the total time to find m collisions is:

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

□

Remark 2. According to formula obtained for U_k in the proof of Theorem 2, we see that if the memory is not filled when running the algorithm for finding $\frac{n}{2}$ collisions, as in meet-in-the-middle applications, then we store θn distinguished points, i.e. all distinguished points in S.

Recall that in the meet-in-the-middle attack, one needs to compute $\frac{n}{2}$ collisions. 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* $\theta = \frac{\sqrt{w^2+nLw}}{n}$*. 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:*

$$O\left(\frac{n\sqrt{w^2+nLw}}{wL}\right). \tag{8}$$

Proof. From Theorem 2, the runtime complexity is given by:

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

By computing the zeros of the derivative:

$$f'(\theta)=\frac{n^2\theta^2-w^2-nLw}{2Lw\theta^2},$$

we obtain that by taking $\theta = \frac{\sqrt{w^2+nLw}}{n}$, the time complexity is $O\left(\frac{n\sqrt{w^2+nLw}}{wL}\right)$. □

This confirms and proves the heuristic findings in [17]. Most importantly, Theorem 1 suggests that in the case of applications which fill the memory available, the number of words we can store is an important factor in the running time complexity. More storage space yields a faster algorithm by a constant factor. We propose such optimization in Section 4.

4 Our approach for the data structure

In this section, we evaluate the memory complexity of parallel collision search algorithm. As explained in Section 2, van Oorschot and Wiener's [16] proposed to divide the memory into segments to allow simultaneous access by threads. We revisit this construction, with the goal in mind to minimize the memory consumption as well. To this purpose, our idea is to consider a radix tree and give best case and worst case estimates for the number of nodes needed to store distinguished points in this tree during the search. Since in Section 3 we showed that the time complexity of collision search depends strongly on the available amount of memory, we describe an alternative structure called a Packed Radix-Tree-List, which will be referred to as PRTL in this paper. We explain how to choose the densest implementation of this structure for collision search data storing in Section 5.

4.1 Radix tree structure

We first describe the classic radix tree and give memory complexity analysis. Each distinguished point from the collision search is represented as a number in a base of our choice, denoted by b. 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 in base b 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 2 shows how to store 7 numbers having 5 digits, where $b = 10$.

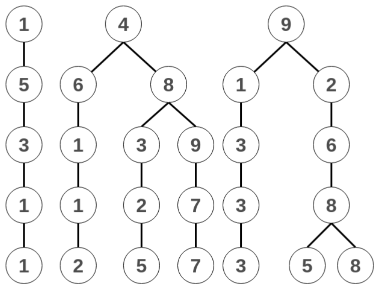

Fig. 2: Example of the structure in base 10 for the set 15311, 46112, 48325, 48977, 91333, 92685, 92688.

The search for a collision and the storage of a new point are done in a single operation and take $\log p$ time. Furthermore, within a parallel framework, 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.

In regard to the memory consumption, we take advantage of common prefixes to have a more compact structure. Here is some analysis on how much common prefixes help to reduce the overall memory consumption. 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 3. For the first b points, we will use bc 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 $(b-1)b$ times, provided that $K > b + (b-1)b$.

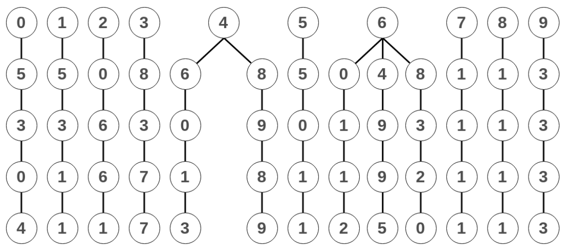

Fig. 3: Worst-case scenario example, $b = 10$

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

- bc nodes for the first b points
- $(b-1)b(c-1)$ for the next $(b-1)b$ points
- $(b-1)b^2(c-2)$ for the next $(b-1)b^2$ points etc.
- $(b-1)b^k(c-k)$ for $(b-1)b^k$ points.

For each of the remaining $K - (b + \sum_{i=1}^{k}(b-1)b^i)$ 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:

$$N(K) = bc + \sum_{i=1}^{k}(b-1)b^i(c-i) + (K - b - b(b-1)\sum_{i=0}^{k-1} b^i)(c-k-1).$$

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

$$N(K) \leq \frac{b}{b-1} b^{\lfloor \log_b K \rfloor} + K(c - \lfloor \log_b K \rfloor). \tag{9}$$

Best-case scenario Let K be the number of distinguished points that we need to store and let $k = \lfloor \log_b K \rfloor$. 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-coordinate of the added point has the longest possible common prefix with some other point that was previously stored. For example, for the first point c nodes are allocated, for the next b-1 nodes, one extra node is allocated and so on, until all subtrees of depth 1, 2 etc. are filled one by one. Figure 4 gives an example of how 215 points are stored. If

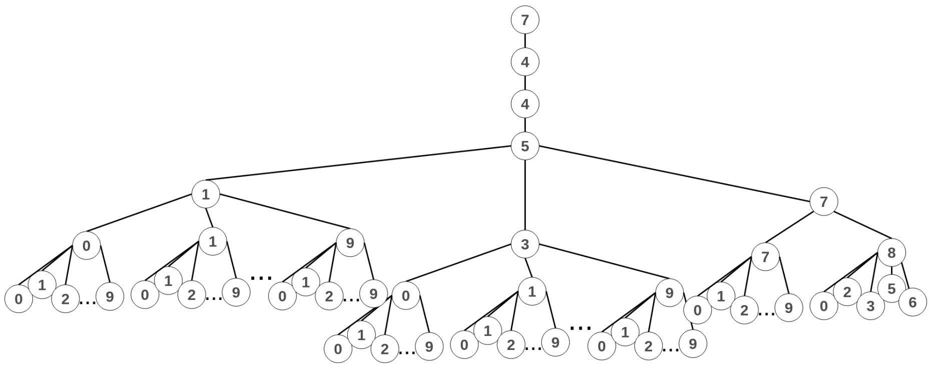

Fig. 4: Best-case scenario example

$K > b^{c-1}$, 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 $k-1$ and a $(x_{k-1}{+}1)$-th incomplete tree of depth $k-1$;
- $c - k - 1$ extra nodes.

Summing up all nodes, we get the following formula:

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

We conclude that:

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

We have proved the following:

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

$$\frac{b}{b-1} K - c - \log K - 1 \leq N(K) \leq (c - \log K + \frac{b}{b-1}) K. \tag{11}$$

Traditionally, nodes in a radix tree are implemented as arrays of pointers to child nodes. This representation will lead to excessive memory consumption when the data to be stored follows a uniform random distribution, leading to sparsely populated branches. The next paragraph provides experimental proof that, in the case of cryptographic application, the average distribution of nodes in the tree is closer to the worst case than to the best case.

The difference between the worst case value and the best case value is approximatively $(c - \log K)K$. Depending on the application, this value may be large. We take the case where a single collision is stored for solving the ECDLP. By a theorem of Hasse [15], 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 this number is:

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

In the case of many collisions algorithms, $c \sim K$ and this standard deviation becomes negligible, resulting into a space-reduced data structure. We show how to handle sparse trees efficiently in Section 4.2.

Memory consumption of a radix tree for solving ECDLP. To have an initial estimate of the average case, we experimented by counting the number of nodes created and comparing them to the number of points stored while solving discrete logarithms on elliptic curves. The results in Table 1 are an average of 100 runs.

Key size	Points	Nodes	Word length	$\frac{\text{Nodes}}{\text{Points}}$	Worst-case offset	Best-case offset	Between best and worst
60 bits	33065	303750	14	9.18	+38010	-267000	0.87
65 bits	90939	905400	15	9.95	+106039	-804346	0.88

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

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 10 and 9 respectively. Columns 6 and 7 give the offset of these values compared to the experimentally obtained average. And if we compare the word length with the nodes to points ratio, we see how much we gain from the common prefixes.

In the case of attacks on the discrete logs on the elliptic curve, we store the starting point of the Pollard walk aP and the first distinguished point we find, represented by the coefficient a and the x-coordinate correspondingly. In practice, implementing this structure is very costly. To have a $\log p$ look-up and insertion time each node has to store the pointers to all of its b children in an array. Furthermore, internal nodes, as well as leaves, have to hold a pointer to a coefficient a, since x-coordinates can be of different lengths.[1] For reasons pointed out earlier, there is also a lock associated to each node.

We label a pointer wasted when it is pending (i.e. unallocated child or a coefficient) and we denote by rate of use the ratio between the memory that is used and the allocated amount of memory. We show in Table 2 the experimental results that we got for the rate of use of the radix tree structure in the case of collision search for the discrete log attack.

Key size	base 2	base 4	base 8	base 10
60 bits	51.47%	35.46%	22.4%	18.58%
65 bits	51.39%	35.36%	21.95%	18.50%

Table 2: Rate of use of the radix tree structure for solving ECDLP according to base 2, 4, 8 and 10. Each value is an average of 100 tests.

4.2 Packed Radix-Tree-List

Intuitively, we see that the radix tree is dense at the upper levels and sparse at the lower ones. Hence it would be less memory consuming to construct a radix tree up to certain level and then add the points to linked lists, each list starting from a leaf on the tree. We call this a Packed Radix-Tree-List[2]. Figure 5 illustrates an example of an abstract Radix-Tree-List in base 4.

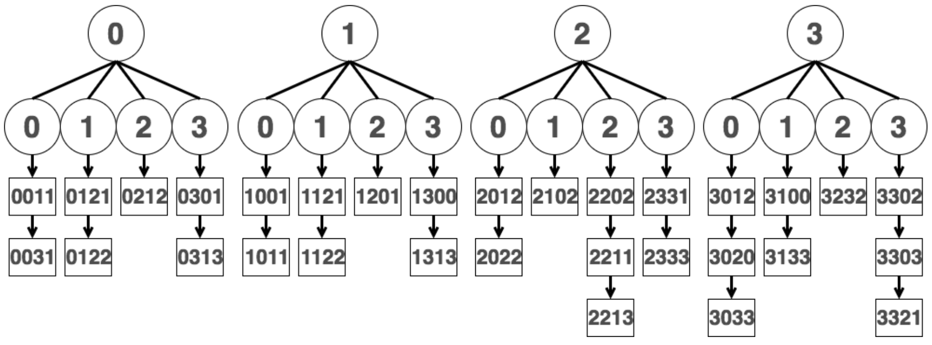

Fig. 5: Radix-Tree-List structure with $b = 4$ and $l = 2$

We look to estimate up to which level the tree is complete for our use case. Let K be the number of stored points before a collision is found and let l be the

[1] Words can be of uniform length if we add zeros at the beginning.

[2] The 'packed' property is addressed in Section 5, where we give implementation details.

level up to which we build the radix tree. The number of leaves in a complete radix tree of depth l is b^l. As per the coupon collector's problem, all the linked lists associated with a leaf will contain at least one point when the following inequality is verified:

$$K \geq b^l(\ln b^l + 0.577). \tag{12}$$

We consider the highest value of l which satisfies this inequality to be the optimal level, as it allows us to obtain the shortest linked lists while having 100% rate of use of the memory structure. We verify this experimentally by inserting a given number of randomly obtained points of length $c = 65$ bits to the PRTL structure. The results are in Table 3.

We do 100 runs per K using a base 2 tree implementation and we count the number of empty lists at the end of each run. None of the 300 runs finished with an empty list in the PRTL structure, which confirms that the obtained l is small enough to have at least one point per list.

Then, to confirm that l is the highest possible value that achieves this, we do the same number of tests, taking $l + 1$, which is the lowest value that does not satisfy the inequality 12. The results show that $l + 1$ is not small enough to produce a 100% rate of use of the memory, therefore l is in fact the optimal level to choose.

Nb. of points	Chosen level		Average nb. of empty lists per run	
	l	$l+1$	l	$l+1$
5 million	18	19	0	37
7 million	18	19	0	0.84
10 million	19	20	0	75

Table 3: Verifying experimentally the optimal level.

We notice that in the case of 7 million points, we have very few (or none) empty slots even by taking the level $l + 1$. This is explained by the fact that a PRTL of level 19 needs 7207281 points to have all of its lists filled, as per 12. Given that 7 million is very close to this number, some of the runs finish with 100% rate of use.

The attribution of a point to a leaf is determined by its prefix and we know in advance that all of the leaves will be allocated. Therefore, in practice we do not actually have to construct the whole tree, but only the leaves. Hence, we allocate an array indexed by prefixes beforehand and then we insert each point to the corresponding prefix's list. The operation used to map a point to an index is faster than a hash table function. More precisely, we perform a logical AND operation between the point's x-coordinate and a precomputed mask to extract the prefix. Furthermore, the lists are sorted. Since we are doing a search-and-add operation, sorting the lists does not take additional time and proves to be more efficient than simply adding at the end of the list. Figure 6 illustrates the implementation of this structure.

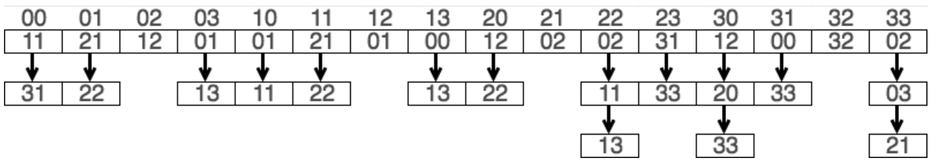

Fig. 6: PRTL implementation. Same points stored as in Figure 5.

Remark 3. When implementing the attack for curves defined over sparse primes, we advise taking an l-bit suffix instead of an l-bit prefix. Prefixes of numbers in sparse prime fields are not uniformly distributed and one might end up only with prefixes starting with the 0-bit, and therefore a half empty array.

Remark 4. Describing the structure, we took the example of the ECDLP. However, the analyses and choices we made for constructing the PRTL are valid for any collision search application which includes storing a pair $(key, data)$ and requires pairs to be efficiently looked up by key.

PRTL vs. hash table We experimented with the ElfHash function, which is used in the UNIX ELF format for object files. It is a very fast hash function, and thus comparable to the mask operation in our implementation. Small differences in efficiency are negligible since the insertion is the less significant part of the algorithm. Indeed, recall one insertion is done after $\frac{1}{\theta}$ computations.

As is the practice with the Parallel Collision search, we allocate K indexes for the hash table, since we expect to have K stored points. Recall that this guarantees an average search time of $O(1)$, but it does not avoid multi-collisions. Indeed, according to [10, Section 6.3.2], in order to avoid to avoid 3-multicollisions, one should choose a hash table with $K^{\frac{3}{2}}$ buckets. Consequently, we insert points in the linked lists corresponding to their hash keys, as we did with the PRTL. Every element in the list holds a pair $(key, data)$ and a link to the next element. The PRTL is more efficient in this case as we only need to store the suffix of the key.

With this approach, we can not be sure that a 100% of the hash table indexes will have at least one element. We test this by inserting a given number of random points on a 65-bit curve and counting the number of empty lists at the end of each run, like we did to test the rate of use for the PRTL. We try out two different table sizes: the recommended hash table size and for comparison, a size which matches the number of leaves in the PRTL. All results are an average of 100 runs.

Results in Table 4 show that when we choose a smaller table size, we have fewer empty lists, but the hash table is still not 100% full. Due to these results, when implementing a hash table we choose to allocate a table of pointers to slots, instead of allocating a table of actual slots which will not be filled. This is the optimal choice because we only waste 8 Bytes for each empty slot, instead of 24 (the size of one slot).

Nb. of points	Average nb. of empty lists for $size = K$	Average nb. of empty lists for $size = 2^l$
5 million	2592960 (51.85%)	98308 (37.50%)
7 million	3632679 (51.89%)	98304 (37.50%)
10 million	5138792 (51.38%)	196615 (37.50%)

Table 4: Test the rate of memory use of a hash table structure.

Since results in Table 3 show that the array in PRTL will be filled completely, when using this structure we allocate an array of slots directly. This makes PRTL save a constant of $8K$ Bytes compared to a hash table.

To sum up, the PRTL structure is less space consuming and has a memory rate of use of 1. Note however that by Equation (12), the average number of elements in a chained list corresponding to a prefix is $\frac{K}{b^l} \approx l \log b + 0.577$. This shows that the search time in our structure is negligible, and our benchmarks shown in Section 5 confirm that memory access has no impact on the total running time for the algorithm.

It is clear that when one implements the PRTL, this structure takes the form of a hash table where the hash function is in fact the modulo a specific value calculated using the correlation (12). It might seem counter-intuitive that the optimal solution for a hash function is the modulo function. However, collision search algorithms do not require a memory structure that has hash table properties, such as each key to be assigned to a unique index.

Indeed, a well distributed hash function is useful when we look to avoid multicollisions. With collision search algorithms, the number of stored elements is so vast that we can not possibly allocate a hash table of the appropriate size and thus we are sure to have longer than usual linked lists. Fortunately, this is not a problem since the insertion time is, in this case, not significant compared to the $\frac{1}{\theta}$ random walk computations needed before each insertion. For example, $\frac{1}{\theta}$ would be of order 2^{32} for a 129-bit curve. On the other hand, as shown in Section 3, the available storage space is a significant factor in the time complexity, which makes the use of this alternative structure more appropriate for collision searches.

5 Implementation and benchmarks

Packed RTL A link in the lists in the PRTL stores one $(key, data)$ pair. The intuitive structure implementation of such a link would be the following:

```
struct {
pointer to key-suffix;
pointer to data;
pointer to next;
} link;
```

In addition, we have the bytes storing the actual values for the *key*-suffix and the *data*.

In order to have the best packed structure, we look to avoid wasting space on addressing, structure memory alignment and unintended padding. Hence we propose to store all relevant data in one byte-vector. Our compact slot has the following structure:

```
struct {
byte vector[vector size];
pointer to next;
} link;
```

The *key*-suffix and *data* are bound in one single vector. We designed functions that allow us to extract and set a specific bit. In this way, we have at most 7 bits wasted due to alignment.

The described implementation of a PRTL yields a more optimal memory occupation, but most importantly, manipulating this structure does not slow down the overall runtime of the attack. We show tests that verify this in Table 5, where we insert a given number of random points on a 65-bit curve, using both a hash table and the PRTL. To have a measurement of the runtime that does not depend on point computation time, we take $\theta = 1$, meaning every point is a distinguished one. The *key* length is thus $c = 65$. All results are an average of 100 runs.

K	Memory		Runtime	
	PRTL	Hash table	PRTL	Hash table
5 million	106 MB	324 MB	5.05 s	5.20 s
7 million	148 MB	454 MB	6.74 s	7.01 s
10 million	213 MB	649 MB	9.84 s	10.2 s

Table 5: Comparing the insertion runtime and memory occupation of a PRTL vs. a hash table.

We show similar experiments in Table 6. This time, we performed actual attacks on the discrete log over elliptic curves, instead of inserting random points. Since the number of stored points is now random and can be different between two sets of runs, the runtime per stored point and memory per stored point are more relevant results.

The results are an average of 100 runs and they show that by using a PRTL for the storage of distinguished points we optimize the memory complexity by a factor of 3. The vector method that we describe in this section can be applied for a classic hash table as well. In that case, the PRTL would have a gain factor of 1.5 over the packed hash table, according to these experimental results.

However, the number of stored points for the presented tests is relatively small, corresponding to the number of points needed for one collision. For applications that demand a large number of collisions, such as meet-in-the-middle

Field	Memory		Memory per point		Runtime		Runtime per point	
	PRTL	Hash table	PRTL	Hash table	PRTL	Hash table	PRTL	Hash table
55-bit	402 KB	1172 KB	19 B	59 B	35.16 s	36.42 s	1.69 ms	1.81 ms
60-bit	618 KB	1801 KB	20 B	59 B	210.33 s	212.83 s	6.88 ms	6.91 ms
65-bit	1856 KB	5212 KB	21 B	60 B	1292 s	1291 s	14.90 ms	14.95 ms

Table 6: Runtime and the memory cost for the attack on ECDLP using PRTLs and hash tables.

attacks, the ratio between the prefix and the suffix grows in the favour of the PRTL. Moreover, the difference between the memory occupied by the hash table and the one taken by the PRTL grows linearly with the number of stored points, and can be calculated as follows:

$$M_{hybrid} = M_{hash} - K(\lceil \frac{l}{8} \rceil + 8).$$

We suppose that all of the points can be addressed using 8-Byte addresses, i.e. $K < 2^{64}$, and let us look at the example of a meet-in-the-middle attack on the 3-DES with three independent keys. Following van Oorschot and Wiener [17], this has a runtime of $7n_2\sqrt{\frac{n_1}{K}}$, where the memory available is K words and $n_1 = 2^{56}$ and $n_2 = 2^{112}$. We calculate the memory requirements to store K pairs $(key, data) \in \{0,1\}^{64} \times \{0,1\}^{56}$. Table 7 shows the memory occupation for the packed hash table and the PRTL and the ratio between them when we increase the value of K.

K	Packed hash table	PRTL	Gain	Ratio	Complexity
2^{32}	137438 MB	85899 MB	$12K$	1.60	$7 \cdot 2^{124}$
2^{40}	35184372 MB	20890720 MB	$13K$	1.68	$7 \cdot 2^{120}$
2^{48}	9007199254 MB	5066549580 MB	$14K$	1.77	$7 \cdot 2^{116}$
2^{56}	2305843009213 MB	1224979098644 MB	$15K$	1.88	$7 \cdot 2^{112}$
2^{64}	590295810358705 MB	295147905179352 MB	$16K$	2.00	$7 \cdot 2^{108}$

Table 7: Comparing structure-specific memory requirements for a parallel collision search MITM attack on 3-DES.

5.1 ECDLP implementation details and scalability

To support our findings, we implemented the parallel collision search using both PRTLs and hash tables 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 28-core Intel Xeon E5-2640 processor and we experimented using between 1 and 28 threads. For completeness, we enumerate the choices we made in our implementation and which are common in the literature.

Additive walks. Teske [16] showed experimentally that the walk proposed by Pollard 1 originally performs on average slightly worse than a random walk. She 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 experimental 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 her 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 [17], the Parallel Collision Search is considered to have a linear scalability and our time complexity in Theorem 1 also proves this.

We experimented with $L \in \{1, 2, 7, 14, 28\}$ threads, solving the discrete log over a 60-bit curve. Table 8 shows the Wall clock runtime and the parallel performance of the attack when we double the number of threads.

L_1		L_2		Parallel performance
Threads	Runtime	Threads	Runtime	
1	2459 s	2	1699 s	0.72
7	776 s	14	411 s	0.94
14	411 s	28	210 s	0.97

Table 8: Runtime and Parallel performance of the attack on ECDLP. Results are based on 100 runs per $L_i \in L$.

We conclude that the parallel performance is not as good as expected for 2 threads, yet it gets closer to linear as the number of threads grows.

6 Conclusion

We proposed an alternative memory structure for the parallel collision search algorithm proposed by van Oorschot and Wiener [17]. 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. J. M. Pollard. Monte carlo methods for index computation ($\pmod p$). *Mathematics of Computation*, 32(143):918–924, 1978.
14. John Pollard. Monte Carlo methods for index computation $\pmod p$. *Math. Comp.*, (32):918–924, 1978.

15. J. H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, 1986.
16. Edlyn Teske. On random walks for Pollard's rho method. *Math. Comp.*, 70(234):809–825, 2001.
17. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
18. 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:

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

First, we simplify the expression $S_k = \sum_{i=1}^{k}(b-1)\cdot b^i(c-i)$. We have:

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

By subtracting these expressions we get:

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

Thus we have that

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

We approximate by:

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

Since $k = \lfloor \log_{10} K \rfloor - 1$, we have that

$$N(K) \approx \frac{b}{b-1}b^{\lfloor \log_b K \rfloor} + K(c - \lfloor \log_b K \rfloor).$$