Time-Memory Trade-offs for Parallel Collision Search Algorithms

Monika Trimoska and Sorina Ionica and Gilles Dequen

Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France *
{monika.trimoska,sorina.ionica,gilles.dequen}@u-picardie.fr

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 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 show the linear parallel performance of the attack on elliptic curves discrete logarithms and improved running times for meet-in-the-middle applications.

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

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 [Pol78], 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 [vW99], builds on Pollard's rho method, and is expected to have a linear speedup compared to its sequential version. This algorithm computes several walks in parallel and stores some of these points, called distinguished points.

In this paper, we revisit the memory complexity of the parallel collision search algorithm, both for applications that need a small number of collisions (i.e. discrete logs) and those needing a large number of collisions, such as meet-in-middle

* This work is co-financed by the European Union under the 2014/2020 European Regional Development Fund (FEDER).

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 $y^2 = x^3 - 3x + b$ [BKK+12,BKM09]. This computation was performed on a Playstation 3. More recently, Bernstein, Lange and Schwabe [BLS11] 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 an FPGA implementation breaking a discrete logarithm in a 117-bit group [BEL+]. 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 [Iso11]), hash functions [KNW09,MRST09], lattice-based cryptosystems (NTRU [HGSW03,vV16]) and isogeny-based cryptosystems [ACC+18].

Two models of computation can be considered for this algorithm. The first one follows the shared memory paradigm, in which each thread will compute distinguished points and store it in the common memory. The second one is a message-passing model, where the threads computing points, called the clients, send the distinguished points to a separate process, running on a different machine called the server, who will handle the memory and check for collisions.

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 - the proportion of distinguished points, 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 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 lookup operations. In this paper, we introduce a new structure, called Packed Radix-Tree-List (PRTL), which is inspired by radix trees. We show that the use of this structure leads to 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 and experimented using a Shared-Memory Parallelism (SMP) system. Our benchmarks demonstrate the performance and scalability of this method. While in the case of a single discrete log, the PRTL variant implementation yields running times similar to those of a hash table approach, our experiments demonstrate that the new data structure gives faster limited-memory multi-collision attacks.

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, as a 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. Let S be a finite set of cardinality n. 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 complexity by using Floyd's cycle-finding algorithm [Jou09,Bre80]. We do not further detail memory optimizations here since they are inherently of sequential nature and there is currently no known way to exploit these ideas in a parallel algorithm.

The parallel algorithm for collision search proposed by van Oorschot and Wiener [vW99] 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 thread computes a distinguished point x_d, it stores it in a common list of tuples (x_0, x_d). If two walks collide, this is identified 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 that map to the same value.

Solving discrete logarithms. In this subsection, S denotes a cyclic group of order n. We focus on the elliptic curve discrete logarithm problem (ECDLP) in a cyclic group $S = \langle P \rangle$, but the methods described in this paper apply to any finite cyclic group. We will assume that the curve E is defined over a finite field $\mathbb{F}_p$, where p is a prime number. Let $Q \in S$ 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 $F : S \to S$ which behaves randomly and such that each time we compute $R_{i+1} = f(R_i)$ we can easily keep track of integers a_i and b_i such that $f(R_i) = a_iP + b_iQ$. 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 S. As a consequence, whenever a collision $f(R_j) = f(R_k)$ occurs, we obtain an equality

$$a_jP + b_jQ = a_kP + b_kQ. \tag{2}$$

This allows us to recover $x = (a_j - a_k)/(b_k - b_j)$, provided that $b_k - b_j$ is not a multiple of n. Starting from R_0, a multiple of P, Pollard's rho [Pol78] method computes a sequence of points R_i where $R_{i+1} = f(R_i)$. Since the group S is finite, this sequence will produce a collision after $\sqrt{\frac{\pi n}{2}}$ iterations on average. In the parallel version, each thread computes a walk, and only certain points on this walk are stored in memory. These points are called distinguished points and defined by an easily testable property, such as a certain number of trailing bits of their x-coordinate being zero. Whenever a thread computes such a point, this is stored in a common list, together with the corresponding a and b. When two walks collide, this cannot be identified until the common distinguished point is computed. Then the discrete logarithm can be recovered from an equation of type (2).

Many collision applications. A first type of application of the van Oorschot and Wiener algorithm computing many collisions is the multi-user setting of both public and secret key schemes. In such a setting, it has demonstrated that it is more efficient to recover individual keys one by one, by using a growing common database of distinguished points, instead of running the algorithm for each key separately (see [KS01,FJM14]). A second type of applications concerns meet-in-the-middle attacks, which require finding a collision of the type $f_1(a) = f_2(b)$, where $f_1 : D_1 \rightarrow R$ and $f_2 : D_2 \rightarrow R$ are two functions with the same co-domain. As explained in [vW99], solving this equation may be formulated as a collision search problem on a single function $f : S \times \{1, 2\} \rightarrow S \times \{1, 2\}$, where the solution we need is of the type:

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

and S is a set bijective to D_1. This collision is called *the golden collision.* The number of unordered pairs in S are approximately $\frac{n^2}{2}$ 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.

Computational model and data structure. We consider a CPU implementation of the shared memory variant of the algorithm, where each thread involved in the process performs the same task of finding and storing distinguished points. In this case, the choice of a data structure for allowing efficient lookup and insertion is significant. 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 [vW99] propose the use of the most significant bits of distinguished points. Their idea is to divide the memory into 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.

In recent years, with the development of GPUs and programmable circuits, the client-server model has been widely used for implementing parallel collision search. In this setting, a large number of client chips are communicating with a central memory server over the Internet. For computing discrete logarithms, [BBB$^+$09] gives a comparison between implementations on different architectures in this model. Current record-breaking implementations of ECDLP also rely on this model [BKK$^+$12,BKM09,BEL$^+$].

Except for the need for a structure that allows efficient simultaneous access to memory, all results in this paper apply to both the client-server and the SMP versions of the PCS algorithm, even though our experimental results are obtained using a CPU implementation following the SMP paradigm.

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 $\mathbb{F}_p$ and by $E(\mathbb{F}_p)$ the group of points on E defined over $\mathbb{F}_p$. 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 [vW99] 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 both for computing a single collision or multiple collision applications. Our refined formulae indicate that the actual running time of the algorithm depends on the proportion of distinguished points and allow us to determine the optimal choice of θ for actual implementations.

3.1 Finding one collision: elliptic curve discrete logarithm

Van Oorschot and Wiener [vW99] 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. This is obtained by finding the expected number of computed points before a collision occurs and then intuitively dividing the clock time by L when L processors are involved. The proof of the following theorem, given in Appendix A, provides a more rigorous argument for the linear scalability of the algorithm.

Theorem 1. *Let S be a set with n elements and $f : S \to S$ be a random map. In the parallel collision search algorithm, denote by θ is the proportion of distinguished points and t_c and t_s denote the time for computing and storing a point respectively.*

1. *The expected running time to find one collision for f is*

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

2. *The worst case running time is*

$$T(\theta) = (\frac{1}{L}\sqrt{(2-\frac{\pi}{2})n} + \frac{1}{L}\sqrt{\frac{\pi n}{2}} + \frac{1}{\theta})t_c + \frac{\theta}{L}(\sqrt{(2-\frac{\pi}{2})n} + \sqrt{\frac{\pi n}{2}})t_s. \tag{5}$$

Remark 1. In the client-server model, clients do not have access to memory, but they send distinguished points to the server and thus t_s stands for cost of communication on the client-side. We suppose that all the client processors are dedicated to computing points. On the server-side however, the analysis is different. Theorem 1 and the means of finding the optimal value for θ apply both to the shared memory implementation adopted in this paper, and to the more common distributed client-server model.

As we can see in Equation (4) and (5), 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 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 of the function in Equation (4):

$$T'(\theta) = \frac{1}{L}\sqrt{\frac{\pi n}{2}}t_s - \frac{1}{\theta^2}t_c.$$

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

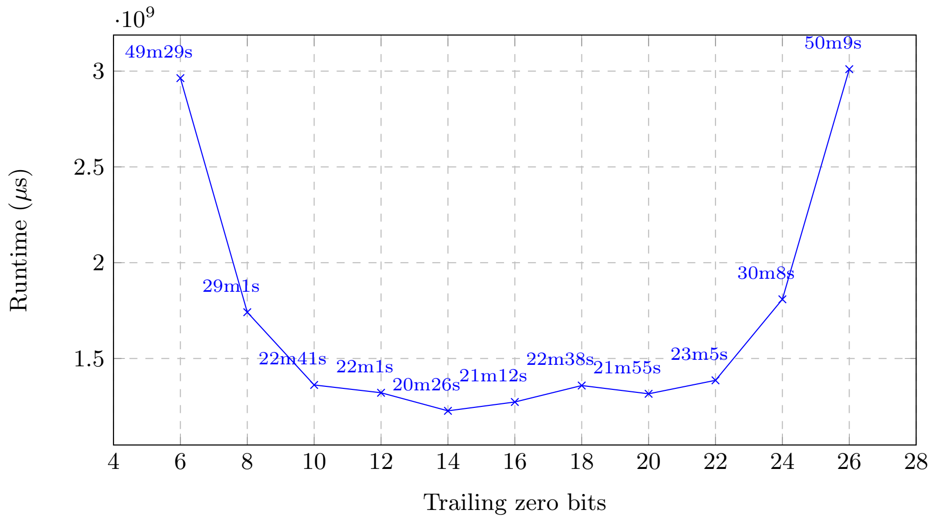

Fig. 1: Timings of solving ECDLP for different values of θ, 65-bits curve, 28 threads

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 [BEL$^+$,BLS11,BKM09]). 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 suggest that for future record-breaking implementations, the value of θ should be determined as explained above.

3.2 Finding many collisions

Using a simplified complexity analysis, van Oorschot and Wiener [vW99] put forward the following heuristic.

Heuristic ([vW99]). *Let $f : S \to S$ a random map and assume that the memory can hold w distinguished points. Then in the meet-in-the-middle attack 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 the attack using these parameters is $2.5\frac{n}{L}\sqrt{\frac{n}{w}}$.*

This heuristic suggests that in the case of many collisions 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. *Let S be a set with n elements and $f : S \to S$ a random map. We denote by θ the proportion of distinguished points in S. The expected running time to find m collisions for f with a memory constraint of w words is:*

$$\frac{1}{L}\left(\frac{w}{\theta} + (m - \frac{w^2}{2\theta^2 n})\frac{\theta n}{w} + \frac{2m}{\theta}\right). \tag{6}$$

Proof. Let X be the expected number of distinguished points calculated per thread before duplication. Let T_1 be the expected number of distinguished points computed until the first collision was found, and T_i, for any $i > 1$, the expected 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 one thread before the second collision is:

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

where the last equality is obtained by integration by parts. 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

$$U_k \sim \frac{\theta\sqrt{2kn}}{L}. \tag{7}$$

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 $\frac{\theta n}{w}$ (see [vW99] for detailed explanation). Finally, to actually locate the collision, we need to restart the two colliding trails from their start, which requires $2/\theta$ steps on average.

To sum up, the total time to find m collisions is:

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

Remark 2. According to the formula obtained in Equation 7, 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.

Note that the proof of Theorem 2 relies strongly on our formula for the expected total number of computed distinguished points for finding m collisions, when m is sufficiently large and the memory is not limited:

$$S_m \approx \theta\sqrt{2mn}. \tag{8}$$

We confirmed this asymptotic formula experimentally by running a multi-collision algorithm for a curve over a 55-bit prime field. The comparison of our formula with the experimental results is in Table 1. Each value in this Table is an average of 100 runs where we set $\theta = 1/2^{13}$. Furthermore, our formula coincides with the estimated workload for computing k discrete logarithms in [KS01], which is obtained using a different analysis valid when $k < n^{1/4}$.

Finally, 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.

Collisions	Experimental Avg.	S_k	Collisions	Experimental Avg.	S_k
100	238289	231704	500	530493	518107
1000	750572	732714	2000	1062581	1036215
5000	1681831	1638399	7000	1990671	1938581

Table 1: Comparing the asymptotic value of S_m to an experimental average.

Corollary 1. *The optimum proportion of distinguished points minimizing the time complexity bound in Theorem 2 is $\theta = \frac{\sqrt{w^2+2nw}}{n}$. 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:*

$$O\left(\frac{n}{L}\sqrt{1+\frac{2n}{w}}\right). \tag{9}$$

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

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

By computing the zeros of the derivative:

$$T'(\theta) = \frac{n^2\theta^2 - w^2 - 2nw}{2Lw\theta^2},$$

we obtain that by taking $\theta = \frac{\sqrt{w^2+2nw}}{n}$, the time complexity is $O\left(\frac{n}{L}\sqrt{1+\frac{2n}{w}}\right)$.

This confirms and proves the heuristic findings in [vW99]. Most importantly, Corollary 1 suggests that in the case of applications that fill the memory available, the number of distinguished points 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 an optimization in Section 4.

4 Our approach for the data structure

In this section, we evaluate the memory complexity of parallel collision search algorithms. As explained in Section 2, van Oorschot and Wiener's [vW99] 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. Since in Section 3 we showed that the time complexity of collision search depends strongly on the available amount of memory, we propose 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.

Since the PRTL is inspired by radix trees, we first describe the classic radix tree structure and then we give complexity analysis on why its straightforward implementation is not memory efficient. The PRTL structure has the memory gain of radix tree common prefixes but avoids the memory loss of manipulating pointers.

4.1 Radix tree structure

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 numerical digit of this number in base b gives the root node in the tree, the next digit is a child and so on. This leads to the construction of an acyclic graph which consists of b connected components (i.e. a forest).

In regard to memory consumption, we take advantage of common prefixes to have a more compact structure. Let c be the length of numbers written in base b that we store in the tree and K the number of distinguished points computed by our algorithm. To estimate the memory complexity of this 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.

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

$$\frac{b}{b-1}K - c - \log_b K - 1 \leq N(K) \leq (c - \log_b K + \frac{b}{b-1})K. \tag{10}$$

The proof of these inequalities is detailed in Appendix B.

Traditionally, nodes in a 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 and to the average distribution of nodes in the tree being closer to the worst case than to the best case.

The difference between the worst-case value and the best-case value can be approximated as $\Delta \sim K(c - \log_b K)$. Depending on the application, this value may be large. Let us consider the case where a single collision is required for solving the ECDLP. By a theorem of Hasse [Sil86], 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 Δ is:

$$\Delta \sim \theta\sqrt{\frac{\pi n}{2}}(\frac{1}{2}\log_b n - \log_b \sqrt{\frac{\pi}{2}}),$$

which implies that the tree is sparse. In the case of many collisions algorithms, $c \sim \log_b K$ 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.

4.2 Packed Radix-Tree-List

Starting from the analysis in Section 4.1, we look to construct a more efficient memory structure by avoiding the properties of the classic radix tree that make it memory costly for our purposes. Intuitively, we see that the radix tree is dense at the upper levels and sparse at the lower ones. Hence it would be more efficient to construct a radix tree up to a certain level and then add the points to linked lists, each list starting from a leaf on the tree. We denote by l be the level up to which we build the radix tree. We call this a Packed Radix-Tree-List[1]. Figure 2 illustrates an example of an abstract Radix-Tree-List in base 4.

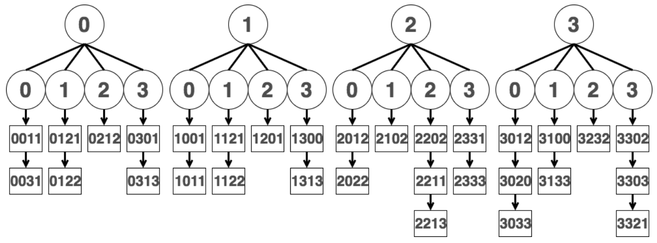

Fig. 2: Radix-Tree-List structure with $b = 4$ and $l = 2$

This idea was considered by Knuth [Knu98, Chapter 6.3] for improving on a table structure called trie, introduced by Fredkin [Fre60]. Knuth considers a forest of radix trees that stop branching at a certain level, whose choice is a trade-off between space and fast access. Indeed, the more we branch, the faster the lookup is, but the more memory we require. He suggests that the mixed strategy yields a faster lookup when we build a tree up to a level where only a few keys are possible. Starting from this level a sequential search through a list of the remaining keys is fast.

In our use case, we favor memory optimization to fast lookup, thus we use a different technique to decide on the tree level. First, we look to estimate up to which level the tree is complete for our use case. 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:

$$K \geq b^l(\ln b^l + 0.577). \tag{11}$$

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 verified this experimentally by inserting a given number of randomly obtained points of length 65, with $b = 2$, in the PRTL

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

structure. The results are in Table 2. We performed 100 runs for each value of K and counted 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 supports the claim 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 reproduced the experiments by taking $l+1$, which is the lowest value that does not satisfy Equation (11). 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.

K	l	Average nb. of empty lists per run	
		Level l	Level $l+1$
5 million	18	0	37
7 million	18	0	0.84
10 million	19	0	75

Table 2: Verifying experimentally the optimal level.

The attribution of a point to a leaf is determined by its prefix and we know in advance that all 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 in the list for the corresponding prefix. The operation used to map a point to an index is faster than a hash table function. More precisely, we perform a bitwise AND operation between the x-coordinate of the point 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 3 illustrates the implementation of this structure.

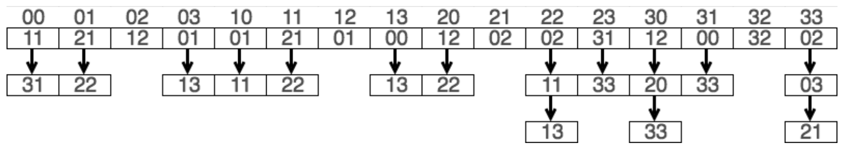

Fig. 3: PRTL implementation. Same points stored as in Figure 2.

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. To experiment with this structure, we chose the example of ECDLP. In this case, we store the starting point of the Pollard walk kP and the first distinguished point we find, represented by the coefficient k and the x-coordinate correspondingly. Consequently, we store a pair $(x\text{-}coordinate, k)$. However, the analysis and choices we made for constructing the PRTL are valid for every collision search application which needs to store pairs $(key, data)$ and requires pairs to be efficiently looked up by keys. For the ECDLP, Bailey et al. [BBB+09] propose, for example, to store a 64-bit seed on the server instead of the initial point, which makes the pair $(x\text{-}coordinate, seed)$.

Remark 5. In our implementation, we always use $b = 2$, and thus, the parameter b will no longer be specified.

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 that insertion is performed every $\frac{1}{\theta}$ iterations of the random map f.

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 [Jou09, Section 6.3.2], in order to avoid 3-multi-collisions, 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 regard 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 that matches the number of leaves in the PRTL. All results are an average of 100 runs.

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

Results in Table 3 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 that 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).

Since results in Table 2 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 (11), 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 Equation (11). 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 multi-collisions. 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

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 C implementation relies on the GNU Multiple Precision Arithmetic Library [MM11] for large numbers arithmetic, and on the OpenMP (Open Multi-Processing) interface [OPE] for shared memory multiprocessing programming. Our experiments were performed on a 28-core Intel Xeon E5-2640 processor using 128 GB of RAM and we experimented using between 1 and 28 threads. In this section, first we explain in detail the implementation of the PRTL structure and then we show experimental results.

Packed RTL. An entry in the lists in the PRTL stores one $(key, data)$ pair. 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. In this way, we have at most 7 bits wasted due to alignment. We designed functions that allow us to extract and set values in the vector. Our implementation of a PRTL yields a better memory occupation, but most importantly, manipulating this structure does not slow down the overall runtime of the attack. We show experimental results that verify this in Table 4, 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	106MB	324MB	5.05 s	5.20s
7 million	148MB	454MB	6.74 s	7.01s
10 million	213MB	649MB	9.84 s	10.2s

Table 4: Comparing the insertion runtime and memory occupation of a PRTL vs. a hash table.

We show similar experiments in Table 5. 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.

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

Table 5: Runtime and the memory cost for attacking ECDLP using PRTLs and hash tables.

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.

Calculating the exact memory occupation. Let f be the size of the field in bits and t be the number of trailing bits set to zero in a distinguished point. We keep the notation of K for the expected number of stored points and of l for the level of the PRTL structure. To calculate the expected memory occupation of the entire PRTL structure, we first calculate the size of a compact slot. Recall that a compact slot holds one byte-vector and a pointer to the next slot. Thus, one compact slot takes $\lceil (f-l-t+f)/8 \rceil + 8$ bytes ($f-l-t$ bits for the *key*-suffix and f bits for the *data*). The size of the slot is to be multiplied by K. To make sure that the access to the shared memory is asynchronous, we use locks on the shared data structure. However, for efficiency, the time that threads spend waiting on a lock needs to be minimized as much as possible. Thus, when we store a point we do not lock the entire structure. In the PRTL structure, there is a lock for every entry in the array, which makes a total of 2^l locks. This adds $8 \cdot 2^l$ bytes to the total memory occupation. Recall from Section 4 that the entries in our classic implementation of a hash table are linked lists of points. Every element (point) in the list holds a pair $(key, data)$ and a link to the next element. The size of one point is 24 bytes for the three pointers (pointer to the *key*, pointer to the *data* and pointer to the next element), plus $\lceil (f-t)/8 \rceil$ bytes for storing the *key* and $\lceil f/8 \rceil$ bytes for storing the *data*. Similarly to the PRTL structure, the hash table has a lock for every entry in the table, which makes a total of K locks, taking $8K$ bytes. Then, we have the size of the hash table, $8K$ bytes, plus the size of all points. Hence, the total memory occupation of the hash table is expected to be $(40 + \lceil (f-t)/8 \rceil + \lceil f/8 \rceil)K$. Table 6 shows examples of memory requirements of large ECDLP computations calculated in this way. The computation in [BEL$^+$] on the elliptic curve **target117** over $\mathbb{F}_{2^{127}}$ is performed using the 30 leading zero bits distinguishing property. For this curve, we have $n = 2^{117.35}$, and thus, the estimation of K is 379821956. However, the actual computation finished after 968531433 distinguished points were collected. We calculate the memory requirements both for the estimated and for the resulting value of K. In both cases, the l is calculated with respect to the estimation of K, as l always needs to be set beforehand. We also calculate the memory requirements for a discrete log computation on a 160-bit curve, with an estimated number of stored distinguished points.

ECDLP implementation details and scalability. Teske [Tes01] showed experimentally that the walk proposed by Pollard described in Equation (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. In our implementation, we adopted the approach of using additive walks and we chose $r = 20$ as the number of sets S_i that give a partition of the group S. Teske showed experimentally that if $r \geq 20$ then additive walks are close to random walks.

Field	θ	K	l	PRTL	Hash table
117.35-bit [BEL+] estimation	$1/2^{30}$	~ 379821956	24	9.6GB	23.1GB
117.35-bit [BEL+] computation	$1/2^{30}$	968531433	24	24GB	59GB
160-bit estimation	$1/2^{40}$	$\sim 2^{40}$	35	43155GB	82463GB

Table 6: Memory requirements of large ECDLP computations using PRTLs and hash tables.

In the theoretical model [vW99], the Parallel Collision Search is considered to have linear scalability and our time complexity in Theorem 1 confirms this. To assess the parallel performance of our implementation, we experimented with $L \in \{1, 2, 7, 14, 28\}$ threads, solving the discrete log over a 60-bit curve. Table 7 shows the Wall clock runtime and the parallel performance of the attack when we double the number of threads. The parallel performance is an indication of how the runtime of a program changes when the number of parallel processing elements increases. It is computed as

$$\frac{L_1 t_1}{L_2 t_2},$$

where t_i is the Wall clock runtime with L_i threads and $L_1 > L_2$. A program is considered to scale linearly if the speedup is equal to the number of threads used i.e. if the parallel performance is equal to 1 (or very close to 1, in practice). From our results, we conclude that the parallel performance is not as good as expected for a small number of threads, but gets closer to linear as the number of threads grows.

L_1	Runtime t_1	L_2	Runtime t_2	Parallel performance
1	2459s	2	1699s	0.72
7	776s	14	411s	0.94
14	411s	28	210s	0.97

Table 7: Runtime and Parallel performance of the attack on ECDLP. Results are based on 100 runs per $L_i \in L$.

Multi-collision search computation. To prove our claims from Section 3 that more storage space yields a faster algorithm, we ran a multi-collision search while limiting the available memory. When the memory is filled, each thread continues to search for collisions without adding new points. As a practical application of this computation, we chose the discrete logarithm in the multi-user setting [KS01,FJM14]. Hence, the data that we store for each distinguished point is the coefficient a, plus an integer representing the user. Results in Table 8 show that the PRTL yields a better runtime compared to a classic hash table due to the more efficient memory use.

Collisions	Memory limit	Runtime		Stored points	
		PRTL	Hash table	PRTL	Hash table
4000000	1GB	34.64 h	58.80h	46820082	12912177
16000000	2GB	88.18 h	137.46h	93640161	25824345
50000000	4GB	203.24 h	276.80h	168325978	51648716

Table 8: Runtime for multi-collision search for a 55-bit curve using PRTLs and hash tables. Values for 1GB memory limit are an average of 100 runs and values for 2GB and 4GB memory limits are an average of 10 runs.

6 Conclusion

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. We proposed an alternative memory structure for the parallel collision search algorithm proposed by van Oorschot and Wiener [vW99]. We show that this structure yields a better memory complexity than the hash table variant of the 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

ACC$^+$18. Gora Adj, Daniel Cervantes-Vázquez, Jesús-Javier Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic curves. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 322–343. Springer, 2018.

BBB$^+$09. Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130.

Cryptology ePrint Archive, Report 2009/541, 2009. `https://eprint.iacr.org/2009/541`.

BEL^{+}. D.J. Berstein, S. Engels, T. Lange, R. Niederhagen, C. Paar, P. Schwabe, and R. Zimmermann. Faster elliptic-curve discrete logarithms on FPGAs. `https://eprint.iacr.org/2016/382`.

BKK^{+}12. Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *Int. J. Appl. Cryptogr.*, 2(3):212–228, 2012.

BKM09. Joppe W. Bos, Marcelo E. Kaihara, and Peter L. Montgomery. Pollard rho on the Playstation 3. Workshop record of SHARCS'09 `http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf`, 2009.

BLS11. 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.

Bre80. Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20:176–184, 1980.

FJM14. Pierre-Alain Fouque, Antoine Joux, and Chrysanthi Mavromati. Multi-user collisions: Applications to discrete logarithm, Even-Mansour and PRINCE. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 420–438, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.

Fre60. Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960.

HGSW03. Nick Howgrave-Graham, Joseph H. Silverman, and William Whyte. A meet-in-the-middle attack on an NTRU private key. Tehnical report, NTRU Cryptosystems, 2003.

Iso11. 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.

Jou09. Antoine Joux. *Algorithmic Cryptanalysis*, chapter 7, pages 225–226. Chapman & Hall/CRC, 2009.

Knu98. Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

KNW09. 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.

KS01. Fabian Kuhn and René Struik. Random walks revisited: Extensions of Pollard's rho algorithm for computing multiple discrete logarithms. In Serge Vaudenay and Amr M. Youssef, editors, *Selected Areas in Cryptography*, pages 212–229, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

MM11. Martin Maechler and Maintainer Martin Maechler. GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`, 2011.

MRST09. 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.

OPE. Open Multi-Processing Specification for Parallel Programming. `https://gmplib.org/`.

Pol78. John Pollard. Monte Carlo methods for index computation (mod p). *Math. Comp.*, (32):918–924, 1978.

Sil86. Joseph H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer, 1986.

Tes01. Edlyn Teske. On random walks for Pollard's rho method. *Math. Comp.*, 70(234):809–825, 2001.

vV16. Christine 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.

vW99. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

A Appendix : Proof of Theorem 1

Proof. 1. We call short path the chain of points computed by a thread between two consecutive distinguished points. The expected number of distinguished points produced after a certain clock time T is $TL\theta$. The probability of not having a collision at $T = 1$, for one thread is

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

Note that 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:

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

at the moment $T = 1$.

Let X be the number of points calculated per thread before duplication. Hence:

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

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 as:

$$P(X > T) = (e^{-\frac{L}{n}} \cdot e^{-\frac{2L}{n}} \cdot \ldots \cdot e^{-\frac{TL}{n}})^L = (e^{\frac{-(L+2L+\ldots+TL)}{n}})^L =$$
$$= (e^{\frac{-T(T+1)L}{2n}})^L = (e^{\frac{-T^2L}{2n}})^L = e^{\frac{-T^2L^2}{2n}}. \tag{12}$$

This gives us the probability

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

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

$$E(X) = \sum_{T=0}^{\infty} e^{\frac{-T^2L^2}{2n}} \approx \int_0^{\infty} e^{\frac{-x^2L^2}{2n}} dx \approx \frac{1}{L}\sqrt{\frac{\pi n}{2}}.$$

Since the expected length of a short walk is $\frac{1}{\theta}$, the number of distinguished points before a collision occurs is

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

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:

$$T(\theta) = (\frac{1}{L}\sqrt{\frac{\pi n}{2}} + \frac{1}{\theta})t_c + (\frac{\theta}{L}\sqrt{\frac{\pi n}{2}})t_s. \tag{13}$$

2. To compute the worst time complexity, we compute the variance of the random variable X as $\sigma(X) = E(X^2) - E(X)^2$. Using a similar approximation as in Equation (12) we obtain that $E(X^2) \approx \int_0^{\infty} e^{\frac{-xL^2}{2n}} dx \approx \frac{2n}{L^2}$. Hence the worst case runtime is

$$T(\theta) = (\frac{1}{L}\sqrt{(2-\frac{\pi}{2})n} + \frac{1}{L}\sqrt{\frac{\pi n}{2}} + \frac{1}{\theta})t_c + \frac{\theta}{L}(\sqrt{(2-\frac{\pi}{2})n} + \sqrt{\frac{\pi n}{2}})t_s.$$

Remark 6. 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.

B Appendix : Proof of Proposition 1

The lower and upper bound in Equation (10) are given by the worst-case and best-case scenario for the number of nodes.

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 4. 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. This case is repeated $(b-1)b$ times, provided that $K > b + (b-1)b$.

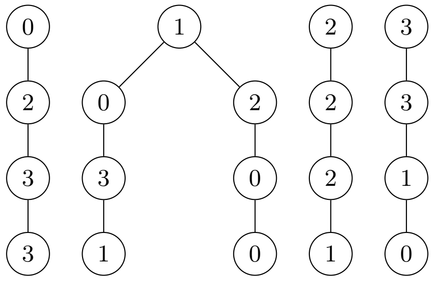

Fig. 4: Worst-case scenario example with parameters $K = 5$ and $b = 4$

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:

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

We simplify the sums and 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

$$N(K) \approx \frac{b}{b-1}b^{\lfloor \log_b K \rfloor} + K(c - \lfloor \log_b K \rfloor). \tag{14}$$

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 5 gives an example of how 215 points are stored. If $K > b^{c-1}$, we fill the first tree and start a new one. Let x_i, for $i \in \{0, 1 \ldots, k\}$, denote the

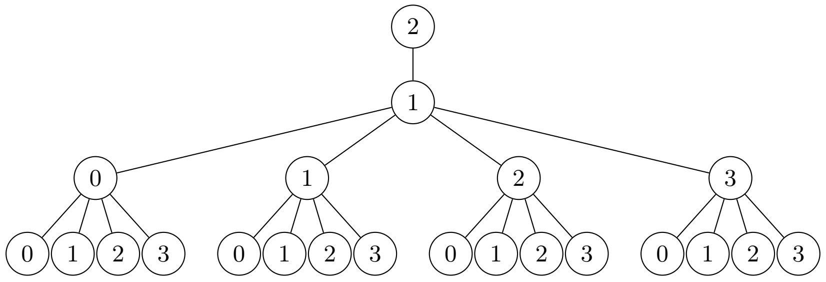

Fig. 5: Best-case scenario example with parameters $K = 16$ and $b = 4$

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:

$$N(K) \geq \frac{b}{b-1} K - c - k - 1. \tag{15}$$