UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Kangaroo configuration for criptoanalysis of public key protocols

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Andreu Villar, Mario

Tutor: López Rodríguez, Damián

ACADEMIC YEAR: 2021/2022

Resum

El problema del Logaritme Discret és un dels problemes que avui dia es consideren com a base per proposar protocols criptogràfics de clau pública (independentment de si aquests es proposen per al xifratge, signatura o identificació). Entre els sistemes que utilitzen aquest problema es troba el protocol d'intercanvi públic de claus de Diffie-Hellman, utilitzat habitualment en multitud d'aplicacions, des de la missatgeria instantània a l'accés privat a xarxes. Un dels algorismes proposats per resoldre aquest problema, i, per tant, utilitzable per atacar els protocols criptogràfics que ho consideren, és conegut com el "mètode del cangur". En aquest treball s'analitza el comportament d'aquest algorisme davant de claus de diferent tamany, alternatives en la seva implementació que modifiquen el seu comportament, així com possibles configuracions d'aquest algorisme analitzant el seu comportament en un atac criptogràfic, comparant el resultat respecte a la proposta original.

Paraules clau: Criptografia; Logaritme discret; Algorisme del cangur; Pollard lambda; Diffie-Hellman

Resumen

El problema del Logaritmo Discreto es uno de los problemas que hoy día se consideran como base para proponer protocolos criptográficos de clave pública (independientemente de si estos se proponen para el cifrado, firma o identificación). Entre los sistemas que utilizan este problema se encuentra el protocolo de intercambio público de claves de Diffie-Hellman, utilizado habitualmente en multitud de aplicaciones, desde la mensajería instantánea al acceso privado a redes. Uno de los algoritmos propuestos para resolver este problema, y por lo tanto utilizable para atacar los protocolos criptográficos que lo consideran, es conocido como el "método del canguro". En este trabajo se analiza el comportamiento de este algoritmo frente a claves de distinto tamaño, alternativas en su implementación que modifiquen su comportamiento, así como posibles configuraciones de este algoritmo analizando su comportamiento en un ataque criptográfico, comparando el resultado respecto la propuesta original.

Palabras clave: Criptografía; Logaritmo discreto; Algoritmo del canguro; Pollard lambda; Diffie-Hellman

Abstract

The Discrete Logarithm problem is one of the problems that today are considered as a basis for proposing public-key cryptographic protocols (regardless of whether these are intended for encryption, signing or identification). Systems using this problem include the Diffie-Hellman public key exchange protocol, commonly used in many applications, from instant messaging to private network access. One of the algorithms proposed to solve this problem, and therefore usable to attack the cryptographic protocols that consider it, is known as the "kangaroo method". In this paper, we analyze the behavior of this algorithm against keys of different size, alternatives in its implementation that modify its behavior, as well as possible configurations of this algorithm analyzing its behavior in a cryptographic attack, comparing the result with the original proposal.

Key words: Cryptography; Discrete Logarithm; Kangaroo algorithm; Pollard lambda; Diffie-Hellman

Contents

Appendices

List of Figures

List of Tables

List of Algorithms

CHAPTER 1
Introduction

This chapter exposes the reasons that have led us to choose this topic as a final degree project, analyzing the objectives we pursued throughout the experimentation. We also state how we have structured this memory that presents and compiles the results obtained.

1.1 Motivation

The appearance of information technology and the massive use of digital communications has caused the need to encrypt the information that we transmit. The transactions carried out through the network can be intercepted, and therefore, the security and integrity of the data must be guaranteed. This challenge has generalized the objectives of cryptography to protect information and provide security to communications and the entities that communicate.

The discrete logarithm problem is one of the main problems in cryptography today. This problem, together with the factorization of integers, covers practically all current public-key cryptography and is vital in the design of the most common cryptographic methods. As we will see later in chapter 4, companies, entities, and powers from all over the world trust their security and privacy in the difficulty of solving this problem.

For this reason, constant work is required by the scientific community to analyze new ways of attacking and solving this problem. The study of cryptanalysis algorithms is essential to maintain awareness of the protocols' security and to be able to be forewarned if a new algorithm is discovered that calls into question cryptographic security worldwide.

Consequently, this final degree project was born due to this need for constant study. The purpose is to analyze an algorithm designed in the late 1970s, *Pollard's Kangaroo Algorithm* [1], which, although it has been studied from a mathematical and statistical point of view, has not been analyzed much from a practical point of view.

The evolution of technology, the improvements in the hardware that we have, the expressiveness of the new programming languages, and many other external factors allow that, on certain occasions, the same algorithm that went unnoticed by the scientific community years ago can become highly relevant today. Crypto-attacks that were once economically unaffordable have become viable and efficient after a technological breakthrough. This work intends to see how this algorithm behaves today in the face of real-world problems, or at least with cryptographic keys of different sizes.

1.2 Objectives

The general objective of this work is to analyze *Pollard's Kangaroo Algorithm* from the practical point of view of a computer scientist engineer instead of doing it from a mathematical-theoretical point of view.

More specifically, we can state the following objectives:

- To implement the algorithm, taking advantage of the evolution in technology that has taken place since it was designed.
- To analyze the actual capacity of the implemented algorithm to solve the discrete logarithm problem, calculating its precision and the maximum size of the problem that it is capable of solving in a reasonable time (less than a day and a half).
- To develop different versions of the algorithm, including our contribution, with a new way of proceeding to save computation time.
- To measure times and analyze which combination of hyperparameters is the one that best solves the problem.

1.3 Memory structure

We present the different developed versions of the algorithm throughout this work, showing the most relevant results and indicating the best combinations of parameters.

In chapter 2, we start with a section of fundamental concepts to refresh some mathematical concepts that it is essential to keep in mind in order to be able to understand the content of this work. Afterward, in chapter 3, we briefly analyze the state of the art, indicating the advantages and disadvantages of the existing algorithms to solve the discrete logarithm problem.

Thereafter, in chapter 4, we review some of the practical applications of the discrete logarithm problem, focusing on the importance of this problem being computationally hard to solve [1] to ensure robust security. We also see how such security could be compromised using the proposed algorithm.

Later, in chapter 5, we formalize the problem we want to solve using *Pollard's Kangaroo Algorithm*, clearly delimiting it in time and space. Analyzing other algorithms is out of the scope of this work.

After this, in chapter 6, we present the experimentation carried out. We start with a zero version of a brute force algorithm to give us an upper bound to start from. Then, we implement *Pollard's Kangaroo Algorithm*, its version of 1978 [1], and later we expose the newly developed versions that try to optimize and reduce the computation time to be able to attack bigger problems.

Next, in chapter 7, we present and discuss the results obtained in the experimentation. Chapter 8 is the culmination of this memory, where we compile the definitive conclusions of the work carried out.

Finally, in chapter 9, we propose possible improvements to achieve even better results. These improvements could not be examined for being outside the length of the project.

[1] A problem that is difficult to solve at a computational level is one whose solution requires such an amount of resources of time (computing) or space (memory) that it makes it unfeasible to calculate a solution, regardless of the algorithm used.

CHAPTER 2
Basic concepts

In this chapter, we try to explain concisely and simply the mathematical concepts and principles that are basic to be able to understand the work carried out and the complexity of the problem to be solved. We also present a set of results from number theory that are especially practical in the field of cryptography.

2.1 Group

In algebra, a group is a pair formed by a finite or infinite set of elements (G) and a binary operation ($\oplus$), which satisfy the following properties:

- The set is closed under the operation.
- The operation is associative.
- There is an identity element in the group.
- Every element of the group has an inverse with respect to the operation.

A simple example of a group that we will use throughout this work is the cyclic group $\mathbb{Z}_p$, where p is an integer:

- **Set:** $G = \{0, 1, 2, \ldots, p-1\}$
- **Operation:** sum modulo p
- **Properties:**
  - Closure: $\forall a, b \in G,\ (a + b \mod p) \in G$
  - Associativity: $a + (b + c) \equiv (a + b) + c \pmod{p}$
  - Identity element: $0 \Leftrightarrow a + 0 \mod p = a$
  - There is an inverse for every value in $\mathbb{Z}_p$: $a + (-a \mod p) \mod p = 0$

In the following figure 2.1 we can see the representation of the group $\mathbb{Z}_{11}$:

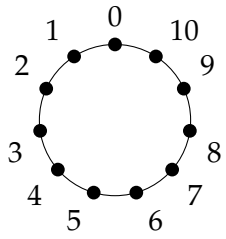


Figure 2.1: Cyclic group $\mathbb{Z}_{11} = \{0, 1, 2, \ldots, 10\}$

2.2 Euler's totient function

Euler's totient function, denoted $\varphi(n)$, counts the number of positive integers relatively prime [1] to n, from 1 to n.

For example:

- $\varphi(9) = 6$ because 1, 2, 4, 5, 7, 8 are relatively prime to 9, but 3, 6, and 9 are not.

```
gcd(9, 1) = 1      gcd(9, 4) = 1      gcd(9, 7) = 1
gcd(9, 2) = 1      gcd(9, 5) = 1      gcd(9, 8) = 1
gcd(9, 3) = 3      gcd(9, 6) = 3      gcd(9, 9) = 9
```

- $\varphi(11) = 10$ because, between 1 and 11, only 11 is not relatively prime to 11.

```
gcd(11, 1) = 1      gcd(11, 5) = 1      gcd(11, 9) = 1
gcd(11, 2) = 1      gcd(11, 6) = 1      gcd(11,10) = 1
gcd(11, 3) = 1      gcd(11, 7) = 1      gcd(11,11) = 11
gcd(11, 4) = 1      gcd(11, 8) = 1
```

In fact, if p is a prime number, $\varphi(p) = p - 1$.

2.3 Multiplicative group of integers modulo p

A multiplicative group of integers modulo p, denoted as $\mathbb{Z}_p^*$, is defined as the finite set of positive integers less than p, which are also relatively prime to p.

That is:

$$(x \in \mathbb{Z}_p^*) \Leftrightarrow ((gcd(p,x) = 1) \wedge (x < p) \wedge (x \in \mathbb{Z}^+))$$

For example:

$$\mathbb{Z}_9 = \{0,1,2,3,4,5,6,7,8\} \qquad |\mathbb{Z}_9| = 9$$
$$\mathbb{Z}_9^* = \{1,2,4,5,7,8\} \qquad |\mathbb{Z}_9^*| = \varphi(9) = 6$$

$$\mathbb{Z}_{11} = \{0,1,2,3,4,5,6,7,8,9,10\} \qquad |\mathbb{Z}_{11}| = 11$$
$$\mathbb{Z}_{11}^* = \{1,2,3,4,5,6,7,8,9,10\} \qquad |\mathbb{Z}_{11}^*| = \varphi(11) = 10$$

In general:

$$\mathbb{Z}_p = \{0,1,\ldots,p-1\} \qquad |\mathbb{Z}_p| = p$$
$$\mathbb{Z}_p^* = \{x \mid (gcd(p,x) = 1) \wedge (x < p) \wedge (x \in \mathbb{Z}^+)\} \qquad |\mathbb{Z}_p^*| = \varphi(p)$$

If p is prime:

$$\mathbb{Z}_p = \{0,1,\ldots,p-1\} \qquad |\mathbb{Z}_p| = p$$
$$\mathbb{Z}_p^* = \{1,\ldots,p-1\} \qquad |\mathbb{Z}_p^*| = \varphi(p) = p - 1$$

[1] Two numbers are relatively prime if and only if their greatest common divisor is 1, that is, if they have no common divisors.

2.4 Order of a finite group

Given a finite group formed by the pair $(G, \otimes)$ and $\alpha \in G$, we define:

$$\alpha^k = \underbrace{\alpha \otimes \alpha \otimes \cdots \otimes \alpha}_{k \text{ times}}$$

We also define $\langle \alpha \rangle$ as the subgroup generated by α, that is:

$$\langle \alpha \rangle = \{\alpha^i : i \geq 1\}$$

Thus, the order of an element $\alpha \in G$ can be defined as the smallest positive integer such that by raising α to that integer, we obtain the identity element of the group (e):

$$ord(\alpha) = j, \text{ being } j \text{ the smallest } j > 0 \text{ such that } \alpha^j = e$$

For example, taking the group as ($\mathbb{Z}_{11}^*$, product operation mod 11 [2]) and $\alpha = 2$:

$$\langle \alpha \rangle = \{2, 4, 8, 5, 10, 9, 7, 3, 6, 1\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$
$$ord(\alpha) = 10$$

Since:

```
2^1 mod 11 = 2     2^5 mod 11 = 10    2^9  mod 11 = 6     2^12 mod 11 = 4
2^2 mod 11 = 4     2^6 mod 11 = 9     2^10 mod 11 = 1     2^13 mod 11 = 8
2^3 mod 11 = 8     2^7 mod 11 = 7      (starts again)      2^14 mod 11 = 5
2^4 mod 11 = 5     2^8 mod 11 = 3     2^11 mod 11 = 2     ...
```

10 is the smallest positive integer such that $2^{10} \pmod{11}$ is equal to the identity element of the product modulo p, that is, to 1. From that point on, the rest of the powers repeat the cycle $2, 4, 8, 5, \ldots$

An interesting result from number theory is that for any finite group $(G, \otimes)$ and $\alpha \in G$, the following equality holds:

$$ord(\alpha) = |\langle \alpha \rangle|$$

2.5 Generator of a group

Given $\alpha \in \mathbb{Z}_p^*$, if the order of α is $\varphi(p)$ then α is said to be a generator of $\mathbb{Z}_p^*$. If $\mathbb{Z}_p^*$ has a generator, $\mathbb{Z}_p^*$ is said to be cyclic.

If α is a generator of $\mathbb{Z}_p^*$, then:

$$\mathbb{Z}_p^* = \{\alpha^i : 1 \leq i \leq \varphi(p)\}$$

For example:

Given $\mathbb{Z}_{11}^*$ and $\alpha = 2$, α is generator of the group, since $ord(\alpha) = |\langle \alpha \rangle| = |\mathbb{Z}_{11}^*| = \varphi(11) = 10$. As we have seen in the previous example, the generator generates the following sequence:

$$\langle \alpha \rangle = \{2, 4, 8, 5, 10, 9, 7, 3, 6, 1\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} = \mathbb{Z}_{11}^*$$

[2] Note that 1 is the identity element of the product mod p.

2.6 Discrete logarithm

Let us consider $(G, \otimes)$ a cyclic group of order n and α a generator of G. For $\beta \in G$, the *discrete logarithm of β to base α* is the unique integer k, $0 \leq k \leq n-1$, such that $\alpha^k = \beta$.

Unless otherwise specified, algorithms in this work are described in the general setting of a (multiplicatively written) finite cyclic group G of order n with generator α. For a more concrete approach, the reader may find it convenient to think of G as the multiplicative group $\mathbb{Z}_p^*$ of order $p-1$, where the group operation is simply multiplication modulo p.

In this way, the discrete logarithm problem can be summarized as:

> “ Given a prime number p, a generator $\alpha \in \mathbb{Z}_p^*$, and $\beta \in \mathbb{Z}_p^*$, find an integer $k \in \mathbb{Z}_p^*$ such that:
>
> $$\alpha^k \mod p = \beta$$
>
> ”

For example, given $p = 11$, $\alpha = 2$, and $\beta = 9$, the problem lies in finding an integer k such that $2^k \equiv 9 \pmod{11}$.

In this particular case, the solution to the discrete logarithm would be $k = 6$, since $2^6 \equiv 64 \equiv 9 \pmod{11}$.

While normal exponentiation can be solved with its inverse function, that is, by applying a logarithm, the discrete logarithm is a one-way function. Therefore, there are no analytical methods to resolve it. We can only use brute force methods or sophisticated algorithms to compute k.

Pollard's Kangaroo Algorithm that we study throughout this work tries to solve this exact problem: given α, β, and p, it tries to find k.

2.7 The Kruskal Count Card Trick

The *Kruskal Count* or the *Kruskal Principle* is a probabilistic concept discovered by Martin Kruskal based on the following idea [2]:

Take a deck of cards, shuffle them, and spread them out on the table, one card after another, face up. Start with the first card and count as many cards to the right as the card's value (ace is worth one and face cards are worth five). Repeat the “walk” from the new card, continuing until you are near the end of the deck. Drop a coin on the last card you arrive.

Now comes the exciting part: challenge your friend that if he chooses one of the first ten cards (different from yours) and follows the same mechanics, he will reach the coin card. If it happens, you take the coin. You will win 80% of the time [3, 2].

Why does this occur? Every time your friend jumps, he can land on one of the following ten cards. At least one of them will have been a part of your path. Since it has been shuffled, the next card is equally likely to be any card from the deck, so it has at least a $\frac{1}{13}$ chance of landing on a card of your path. If this happens, since you both use the same jumping mechanic, you both will follow the same route and end up on the coin card. That is the key idea, at every jump there is an opportunity for your paths to sync up, and if there are enough cards, this will happen eventually.

Let us see the following example[3]:

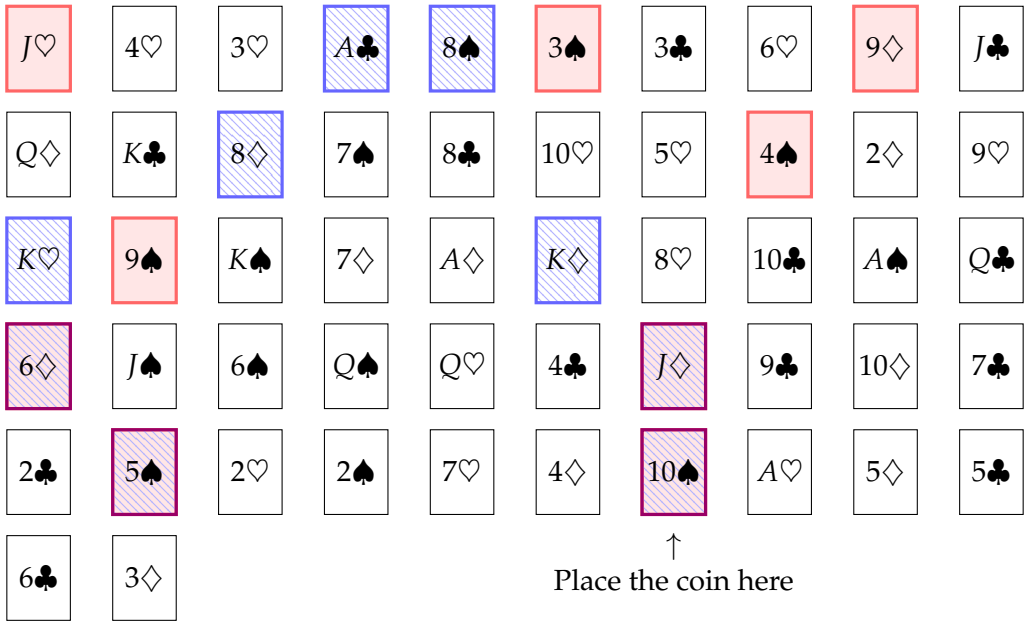


Figure 2.2: Kruskal Count example

We are red, and we start with the first card, which is $J\heartsuit$. As it is a face card, we count five cards to the right and arrive at $3\spadesuit$. Its value is three, so we count three cards and arrive at $9\diamondsuit$. We continue like this successively until we reach the $10\spadesuit$, where we put the coin.

Our friend is blue, and he chooses a card among the ten in the first row. In this example, he has chosen the $A\clubsuit$. Therefore, it moves one position to the right and reaches $8\spadesuit$. Its value is eight, so it jumps to $8\diamondsuit$ and so on until reaching $6\diamondsuit$, a card we had already passed through. From this moment, our paths have been synchronized, and the following cards will coincide with ours, thus arriving at the coin.

This same idea, applied in the field of cryptography to find the discrete logarithm, is what lies behind the algorithm we are studying. John M. Pollard uses a cyclic group as a deck, with the elements ordered as generated by a generator and a jump size defined by a function. When the paths collide, we can establish an equation and solve the discrete logarithm problem.

[3]The cards' distribution has been done randomly through the following website: https://faculty.uml.edu/rmontenegro/research/kruskal_count/kruskal.html

CHAPTER 3
State of the art

Once we have established the mathematical background and formally defined the discrete logarithm problem in chapter 2, we can analyse the problem's state of the art. In this chapter, we present the complexity of the problem under consideration and the advantages and disadvantages of the main algorithms that have been proposed to try to solve the discrete logarithm problem. A detailed description of the operation of each algorithm is beyond the limits of this work, and we leave it proposed for the interested reader. Some recommended sources are: [4, 5, 6, 7, 1].

The discrete logarithm is a problem whose time complexity is linear with respect to the group's order. Let n be that group's order. An exhaustive-search implementation of the solution would have an asymptotic cost of $\mathcal{O}(n)$.

Although this complexity is defined in the literature as linear, in practice this description is not very useful since the group order is usually an extraordinarily large number. Therefore, the problem size is generally defined as the size in bits that the group order occupies. Let t be this size, it would be defined as $t = \lceil \log_2(n) \rceil$.

Assuming as the group the multiplicative group of integers modulo a prime p, the order of the group would be $n = p - 1$ and $t = \lceil \log_2(p-1) \rceil$. Thus, unless otherwise specified, throughout this work we define t as the problem size and the time complexity of the discrete logarithm as $\mathcal{O}(2^t)$, exponential with respect to the size of the modular value, i.e., with respect to the bit size of p. There is no known algorithm for conventional computers that is able to solve the discrete logarithm in polynomial time with respect to the size of the modular value.

The known algorithms (whose complexity is mainly exponential) capable of solving the discrete logarithm problem can be categorized as follows:

- Generic algorithms that work in arbitrary groups, e.g., exhaustive brute-force search, *Baby-step giant-step* algorithm, *Pollard-rho* for logarithms, and the *Pollard's Kangaroo*.
- Algorithms which work in arbitrary groups but are especially efficient if the order of the group has only small prime factors, e.g., *Pohlig-Hellman* algorithm.
- Algorithms that only work in certain groups, e.g., *Index-calculus* algorithm.

As can be seen, it is interesting to be aware that *Pollard's Kangaroo* algorithm we are studying belongs to the first group of generic algorithms. This implies that although the domain of application changes, and on many occasions, elliptic curves are used instead of multiplicative groups of integers modulo a prime number, the problem we are considering is the same. Therefore, its difficulty when solving it (or seen in another way, the underlying computational complexity) is equivalent.

3.1 *Baby-step giant-step* algorithm

The *Baby-step giant-step* algorithm is a simple algorithm to implement, but it requires large amounts of RAM to work correctly. For this reason, its use always involves a strong trade-off between memory and time.

The idea of this method is to store a pair of sets, one of small steps and another of large steps, thanks to some specific mathematical properties of logarithms. If one of the calculated values appears in both sets, an equality can be established, and the discrete logarithm can be solved. For the curious reader, we recommend section 3.6.2 of the *Handbook of Applied Cryptography* [4], where a detailed description of the algorithm and its pseudocode can be found.

Let $p-1$ be the order of the generator α and t its size in bits. The algorithm has a time complexity of $\mathcal{O}(2^{t/2})$. This is a significant improvement over the brute-force approximation, whose temporal cost is $\mathcal{O}(2^t)$.

On the other hand, its spatial complexity is also $\mathcal{O}(2^{t/2})$, which makes it not feasible to solve significant problems with this method. An example of what this space complexity entails is given below:

Problem size	Memory needed (RAM)
32 bits	0.25 MB
64 bits	32 GB
256 bits	9.9e+27 TB
1024 bits	1.6e+144 TB

Table 3.1: *Baby-step giant-step* memory requirements.

One of the advantages of this algorithm is that it is a generic algorithm, i.e., it can be applied to any group, as long as the order of the group is known. In contrast, the large memory requirements make it unfeasible for attacking real-world problems.

3.2 *Pollard-rho* algorithm for logarithms

The *Pollard-rho* algorithm is a randomised algorithm that can be used to solve discrete logarithms. It works by defining a pseudo-random sequence and a pair of paths over it, which it tries to make collide. Once this collision occurs, an equality can be established, and the problem can be solved. The interested reader can find a detailed explanation of this method and its pseudocode in section 3.6.3 of the *Handbook of Applied Cryptography* [4].

This algorithm has the same asymptotic running time as the *Baby-step giant-step* algorithm, i.e., it has a time complexity of $\mathcal{O}(2^{t/2})$. However, it requires a negligible amount of memory, on the order of $\mathcal{O}(1)$, so it is preferred over the *Baby-step giant-step* to solve problems of practical interest.

In addition to using negligible memory, this algorithm is also generic, so it does not take advantage of any particular property of the groups and hence is of potential applicability to any group. Its main drawback is that it is a *Monte Carlo* algorithm. Therefore, there are certain conditions that, if not met, will cause the algorithm to fail with a very high probability.

3.3 *Pollard's Kangaroo* algorithm

The *Pollard's Kangaroo* algorithm, also known as *Pollard-lambda*, is a generic algorithm for solving discrete logarithms. It is based on simulating a pair of kangaroos (one wild and one tamed) running along the number line. The tame kangaroo leaves traps at each jump so that if the wild kangaroo passes through an area where the tame kangaroo has jumped, it will be trapped. Once trapped, we can establish an equation and solve the discrete logarithm. Since this is the algorithm we are analysing, a detailed description of it can be found in section 6.2.

The principal strength of this algorithm is that we do not need to seek the solution in the whole search-space if we know that it lies in an interval between a and b.

Let w be the width of the interval, i.e., $w = b - a$. The size of this interval is what really defines the size of the problem. In fact, if we do not know the interval in which the solution is found, we can always set $a = 1$ and $b = p - 1$ so that the width would be $w = p - 2$ and the asymptotic size of the problem would still be the order of the generator.

Thus, with t' being the bit size of the width, i.e., $t' = \lceil \log_2(w) \rceil$, the algorithm has a time complexity of $\mathcal{O}(2^{t'/2})$. Regarding memory usage, it depends on the implementation used, but in general it can be negligible, in the order of $\mathcal{O}(1)$.

As can be seen, the advantage of this algorithm over its analogue, the *Pollard-rho* algorithm for logarithms, is that we can narrow down the search space more strictly. This fact, together with its great precision, makes it one of the most promising generic algorithms for its practical applicability.

3.4 *Pohlig-Hellman* algorithm

The *Pohlig-Hellman* algorithm, also called the *Silver-Pohlig-Hellman* method, is a special-purpose algorithm for computing discrete logarithms in groups whose order has small prime divisors.

The basic idea of this algorithm is to iteratively calculate the p-adic digits [1] of the logarithm by repeatedly shifting all but one of the unknown digits in the exponent and computing that digit by elementary methods. A detailed description of the algorithm can be found in section 3.6.4 of the *Handbook of Applied Cryptography* [4] for the interested reader.

This algorithm has the limitation that the order of the group, let us denote it as n in this case, is factorizable in small primes. Therefore, it cannot be applied to every group, but where it can be used, it is more efficient than the generic algorithms.

Let $n = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_r^{e_r}$ where $e_i \geq 1$ be the factorization of n. The time complexity of the algorithm in the case where n has small factors is $\mathcal{O}(\sum_i e_i(\log n + \sqrt{p_i}))$.

As can be seen, this time bound is given by the order of the generator and not by its size in bits. In general, we can assume that the asymptotic bound of this problem in bits is, as in the rest of the algorithms, $\mathcal{O}(2^{t/2})$. However, under certain circumstances, the actual performance of the algorithm may be better. On the other hand, if n is a prime, the algorithm is equivalent to the *Baby-step giant-step* [4, *Note 3.67.i*].

[1] An p-adic number is an extension of the field of rationals such that the congruences modulo powers of a fixed prime p are close according to the "p-adic metric". This is a complex mathematical concept beyond the scope of this work. For the inquisitive reader we recommend the following source: https://mathworld.wolfram.com/p-adicNumber.html

3.5 *Index-calculus* algorithm

The *Index-calculus* algorithm is the most efficient known method for computing discrete logarithms. This algorithm is based on selecting a relatively small subset F of elements of the group set G, called the *factorial basis*, such that a significant fraction of elements of G can be efficiently expressed as products of elements of F. For the interested reader, we recommend section 3.6.5 of the *Handbook of Applied Cryptography* [4], where a detailed explanation of the algorithm, its pseudocode and the existing techniques for the most efficient implementation possible can be found.

This algorithm is not generic, so it can only be applied to specific groups. For example, while it is compatible with multiplicative groups of integers modulo a prime number, it does not hold for elliptic curves.

The loss of generality is compensated by the efficiency obtained by taking advantage of the particular properties of the groups with which it is compatible. Thanks to this, although the algorithm can be generally considered to have a time complexity similar to the others, $\mathcal{O}(2^{t/2})$, when certain conditions are met, its asymptotic behaviour is sub-exponential with respect to the size of the problem.

CHAPTER 4
Practical applications

The discrete logarithm problem is one of the fundamental problems for designing public-key cryptographic protocols. It provides the necessary mathematical foundation for the design of these protocols, thanks to the non-existence of an algorithm capable of solving it efficiently (in polynomial time).

Since the algorithm that we are studying in this work tries to solve this problem, the practical applications are evident: if we are able to solve the discrete logarithm, we can attack and undo the cryptographic protocols that make use of it. Moreover, this applies to all kinds of protocols, both encryption and digital signature and identification, widely used today.

Additionally, as we have commented in chapter 3, although the domain changes, the use of elliptic curves considers the same problem, so the method we are studying also applies to such protocols and services.

Among the systems that use the discrete logarithm, we must highlight the Diffie-Hellman public-key exchange protocol [8], whose use is widespread in many different applications and services worldwide. It is used from instant messaging and communication applications such as WhatsApp [9], Telegram [10], Signal [11] or Zoom [12], to different security protocols such as *Secure Shell* (SSH) [13, RFC 4253], *Transport Layer Security* (TLS) [14, RFC 5246], *Secure Sockets Layer* (SSL) [15, RFC 6101], *IP Security* (IPsec) [16, RFC 6071], and *Internet Key Exchange* (IKE) [16, RFC 6071].

This protocol is generally used to agree on symmetric keys that will be used to encrypt a session. It is very common in cryptography to use symmetric encryption protocols such as AES because they are generally faster than public-key protocols when encrypting and decrypting messages. The main drawback of symmetric protocols is not related to their security but to the key exchange:

> “ Once the sender and recipient have exchanged the keys, they can use them to communicate securely, but what secure communication channel have they used to transmit the keys to each other? It would be much easier for an attacker to try to intercept a key than to try all the possible key-space combinations. ”

This is the problem that the Diffie-Hellman public-key exchange protocol solves, ensuring that even if the attacker intercepts the messages that allowed the symmetric key to be established, he must solve an extremely difficult problem to obtain it: the *discrete logarithm problem*.

On the other hand, we can also highlight the ElGamal encryption system [17]. This system is also based on the discrete logarithm problem and has many applications. Its

outstanding versatility allows it to be used both as a protocol for data encryption, decryption, and digital signatures.

4.1 Diffie-Hellman public-key exchange protocol

In this section, we are going to represent in a simplified way what the Diffie-Hellman public-key exchange protocol consists of. Although this project's goal is not to analyze the various protocols that employ the discrete logarithm, if we have to pay attention to any of them, it is this one because it is included in the great majority of services and applications that we use daily. So, let us see how it works and how it can be attacked using the algorithm we are studying.

The primary motivation behind this protocol is the following: we have a couple of users, let us call them Alice and Bob. They want to communicate in a secure (encrypted) way over the insecure internet. There is no secure channel between them to communicate, and they are far enough away that they cannot speak in person and whisper the password. The question we must ask is:

> “ Is it possible they can agree on a secret key that they both know and that no one else can figure out, even if they are listening? ”

Yes, thanks to the Diffie-Hellman public-key exchange protocol.

The protocol works as follows:

Algorithm 4.1: Diffie-Hellman public-key exchange procedure

1. Alice and Bob **publicly** agree to use a multiplicative group of integers modulo p (where p is a prime number, i.e., $\mathbb{Z}_p^*$), and a generator α of this group;
2. Alice chooses a random number, a, but **does not** tell anyone;
3. Bob chooses a random number, b, but **does not** tell anyone either;
4. Alice sends Bob through an insecure channel, that is, **publicly**, the result of calculating $A = \alpha^a \mod p$;
5. Bob sends Alice through an insecure channel, that is, **publicly**, the result of calculating $B = \alpha^b \mod p$;
6. Alice has received B and calculates $s = B^a \mod p = \alpha^{b \cdot a} \mod p$;
7. Bob has received A and calculates $s = A^b \mod p = \alpha^{a \cdot b} \mod p$;

As can be seen, Alice and Bob have the same secret at the end of the algorithm: the value of s, which is the same for both, thanks to the commutative property and the properties of powers.

On the other hand, if a third party were listening on the insecure channel, they would only know the values of p, α, A and B but would not know the value of a or b. The only way to calculate s without these two values would be to solve the discrete logarithm defined by:

$$\alpha^a \mod p = A \qquad \text{or} \qquad \alpha^b \mod p = B$$

Both are equally complex to solve.

Let us see it more visually in the following example:

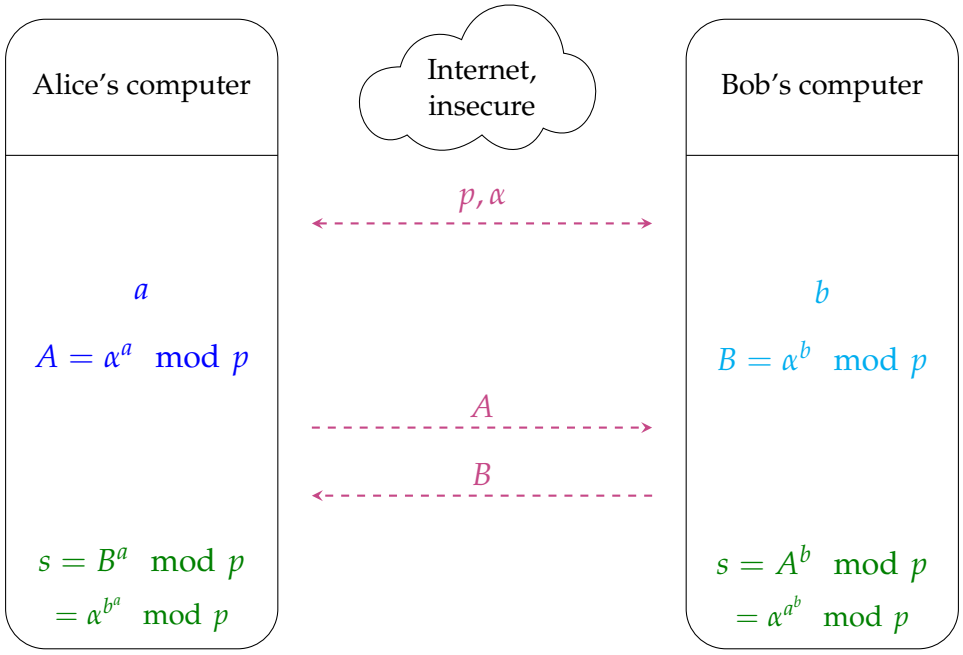


Figure 4.1: Diffie-Hellman public-key exchange representation.

Once Alice and Bob have the key s, they can initiate a symmetric encryption protocol such as AES, using its value as the encryption key, thus creating a secure (encrypted) communication channel.

The cryptographic attack that we can perform is simple: use our kangaroo algorithm to compute $a = Kangaroo(\alpha, p, A)$ and once we have the value of a, calculate, like Alice, $s = B^a \mod p$. Then the obvious question is:

> “ If our algorithm can solve the discrete logarithm, are these widely used Diffie-Hellman generated keys insecure? ”

The answer is, as the reader might expect, a resounding no. A straightforward solution is used to avoid this type of attack: to employ keys of enormous size. Although our algorithm can solve the discrete logarithm and, as we will see later in chapter 7, does so several orders of magnitude faster than a brute force algorithm, it is still an exponential algorithm. It should be remembered that this is why the discrete logarithm is so widely used: no known algorithm is capable of solving it in polynomial time. Therefore, it will still need billions of years to solve the discrete logarithm with keys of enormous size.

CHAPTER 5
Problem statement and formulation

This chapter attempts to clearly and concisely establish the problem we solve throughout this project. We define the hardware used for it, the results' acceptable bounds in terms of space and computation time, and the procedures that we follow before each test and solution.

5.1 Problem delimitation

The problem that we solve is the calculation of the discrete logarithm. As we have seen in section 2.6, this problem can be applied to different finite cyclic groups. In our case, we focus only, although without losing generality, on one of the groups where it is more manageable to understand the concept: in multiplicative groups modulo a prime p.

Thus, the problem can be summarized as calculating k given α, p, and β such that:

$$\alpha^k \mod p = \beta$$

To solve it, we employ *Pollard's Kangaroo Algorithm*, also known as *"A Lambda Method for Catching Kangaroos"* [1] or simply *Pollard-lambda*. However, it is recommended not to call it by the latter because the parallel version of the *Pollard-Rho Algorithm* is also known as *Pollard-Lambda* since the graphical representation of its trace resembles the shape of the Greek letter Lambda (λ).

Therefore, the study of algorithms other than *Pollard's Kangaroo*, such as *Pohlig-Hellman* or the well-known *Baby-step giant-step*, is outside this work's boundaries.

Also, to simplify the redesign and implementation of the algorithm, we are not going to apply the *distinguished points* technique [18, 19]. We discuss this strategy in more depth in chapter 9.

As indicated before, we work throughout this project with multiplicative groups modulo a prime p. It should be noted, however, that the order of the group must be complete, that is, the order of the group generator must be $p - 1$.

After implementing the first functional version of the algorithm, we noticed that it failed about 25% of the time but inquisitively: the algorithm stopped finding a solution k. However, this solution was not correct and did not meet the equality $\alpha^k \mod p = \beta$. This was remarkably strange since *Pollard's Kangaroo Algorithm* has two reasonable outputs: it either finds the solution that verifies the equation, or it does not. It is not expected that it stops with a wrong result. After analysing what could be failing, we noticed that when

generating the tests, we did not check if the generator α was of order $p-1$. Indeed, in the cases where it failed, it was because the order of the group's generator was smaller than $p-1$. Therefore, the solution was given modulo $ord(\alpha)$, being smaller values and thus not verifying the equation. Consequently, it is an essential requirement that the base used is a group generator, i.e., $ord(\alpha) = p-1$.

Finally, it is convenient to emphasise the rigour followed when carrying out the experimentation:

- The execution time that appears in the results of the following chapters is the average time of the execution of 30 problems.
- The problem size will be defined, as usual in cryptography, by the size in bits of the value of p, since it determines the size of the group G and the size of the cryptographic keys to be cracked.
- Each version and configuration has been subjected to the same problem tests. In this way, the complexity of the problems is the same for all the configurations examined.
- The average time measured is calculated only between the times of the executions that have found the correct solution. For example, if a problem has taken one second but has not found the solution and the rest have taken 20 seconds but have found it, the one is discarded when calculating the time results.

5.2 Time delimitation

One of the most critical aspects of the work is the time delimitation: to restrict the computing time. Given that the temporal complexity of the problem is linearly proportional to the order of the group and, thus, exponentially proportional to the bit size of the modular value, we must be cautious with the execution time since increasing the input size might substantially increase execution time.

Also, we must be aware that it is not enough to solve a problem of t bits. This problem has best and worst cases, so we have decided to perform a set of 30 problems for each key size and calculate the average execution time. In this way, we can reasonably estimate the actual time cost of solving each problem size.

For all this, we determined that it is appropriate to limit the execution time to a maximum of one day and a half to finish the 30 problems that constitute a test. Another viewpoint is that each problem can take around an hour to be solved (on average).

5.3 Space delimitation

The delimitation of the space to solve the problem is given mainly by the RAM memory available in the machine where the algorithm is executed.

One of the advantages of *Pollard's Kangaroo Algorithm* is that it does not require intensive use of memory to solve the problem. Therefore, we consider that the 16GB of RAM + 2GB of SWAP memory of the system where we carried out the experimentation is more than adequate. Any problem requiring a more significant amount of memory exceeds our space constraints, and we will abort it. If it really requires that much memory, it might be more reasonable to try to solve it with the *Baby-step giant-step* algorithm or similar.

5.4 Hardware used for experimentation

All the versions experimented with and all the tests executed have been carried out on the same system to avoid any interference in the results. The hardware and software of the system that has performed the experimentation is the following:

- CPU: `Intel i7-7700K @ 4.500GHz`
- Number of cores: `4 (8 threads)`
- RAM memory: `15,890 MiB`
- SWAP memory: `2 GiB`
- GPU: `Intel HD Graphics 630`
- OS: `Ubuntu 20.04.3 LTS x86_64`
- Kernel: `5.13.0-27-generic`
- Python version: `Python 3.8.10`

CHAPTER 6
Experimentation

Throughout this chapter, we present the different algorithms implemented, the experimentation carried out, and the tests performed. First, we implement an exhaustive solution search method to establish an upper time-bound. The following sections show the successive modifications performed to *Pollard's Kangaroo Algorithm* to optimize its performance.

6.1 Brute Force algorithm

As mentioned in chapter 3, the discrete logarithm problem is an exponential problem concerning the size of the modular value. There are no known algorithms for classical computers capable of solving the discrete logarithm deterministically in polynomial time.

Before starting to program *Pollard's Kangaroo Algorithm*, we believe it is interesting to have a first approximation that supposes an upper bound for the problem. We achieve this by implementing and measuring how long an exhaustive-search algorithm takes to solve various input sizes.

The implemented solution is an algorithm that performs a systematic search for the solution in the whole key-space:

Algorithm 6.1: Brute Force algorithm

Input: α, p, β
Output: k

$k \leftarrow 1$;
while $(\alpha^k \mod p) \neq \beta$ **do**
  $k \leftarrow k + 1$;
end
return k

This algorithm is straightforward, but its main problem lies in its temporal complexity. As we have mentioned, it looks for the solution in the entire key-space, that is, in the set $\{1, 2, ..., p-1\}$. Therefore, the algorithm is linear with the value of p. However, since in practice p is enormous, the size of the problem is usually determined by its value in bits. Let t be this value, i.e., $t = \lceil \log_2 p \rceil$. With this value, the problem is exponential, i.e., its temporal complexity is $\mathcal{O}(2^t)$.

Once we have the algorithm ready and we verify it works with small test cases, we proceed to large-scale experimentation. The results can be seen in table 6.1 and in figure 6.1.

The times shown in the upper part are the mean of 30 discrete logarithm problems. As mentioned above, the size of p gives the size of the problem. As usual in cryptography, we represent it as the number of bits that its binary representation occupies, i.e., $problem\ size = \lceil \log_2\ p \rceil$. The accuracy value is defined as the number of times the algorithm has found the solution divided by the number of problems to solve multiplied by 100.

Problem size	12	16	20	24	28	32	36
Time	0.0008s	0.039s	0.191s	11.1s	3.7min	1.4h	22.6h
% Accuracy	100.00	100.00	100.00	100.00	100.00	100.00	Estimated

Problem size	40	44	48	52	56	60	64
Time	15.1d	241.0d	10.6y	168.9y	2703.1y	43250.4y	692005y
% Accuracy	Estimated	Estimated	Estimated	Estimated	Estimated	Estimated	Estimated

Table 6.1: Execution time and accuracy of the brute force algorithm.

As can be seen, since it is a brute force algorithm, the solution is always found. However, the execution time excessively increases when exceeding 32 bits. It is logical since every time we increase the size of the problem by 4, we are multiplying by $2^4 = 16$ the the search-space size.

That is why, following the guidelines set out in section 5.2, we have chosen not to execute the tests with a size greater than 32 bits. Instead, we have calculated the estimated time it would take to execute such a brute-force search.

As can be appreciated, this problem overgrows. While a 60-bit problem would cost 43,250 years, which is already excessive (an ancestor from the Upper Paleolithic would have left running the problem for us to find it nearing completion today), an 80-bit problem would cost several lifetimes of the universe. The infeasibility of solving the discrete logarithm using a brute force algorithm is evident.

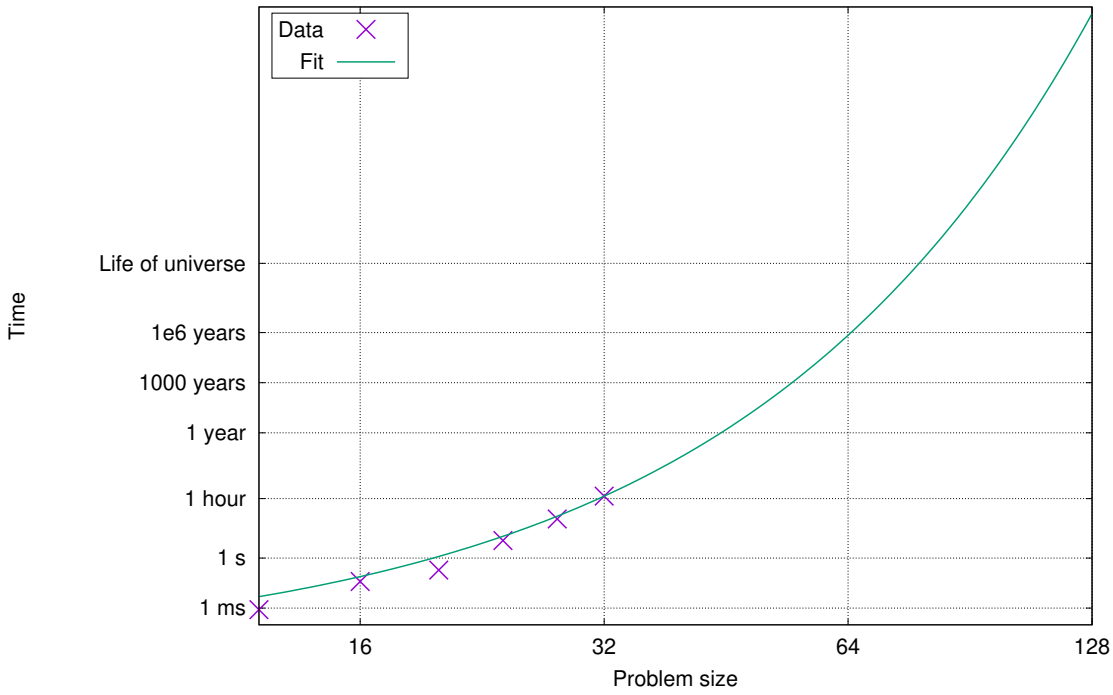


Figure 6.1: Time required to solve the discrete logarithm by brute force.

6.2 Kangaroo v.1 - Search in the whole interval

In this first version of the algorithm, we focus only on implementing a functional version of *Pollard's Kangaroo Algorithm* as simple as possible, leaving the optimizations for the following versions. To do this, we rely on the original article by John M. Pollard: "*A Lambda Method for Catching Kangaroos*" [1].

To understand this algorithm, John M. Pollard uses the following metaphor: the *kangaroo method* employs two kangaroos (the two players in Kruskal's Count) jumping identically across different terrains (different cards). The objective is to capture the wild kangaroo with the cooperation of another tamed kangaroo. These kangaroos are jumping, and the amount they jump depends on the quality of the terrain they are on (the card's value).

Every time our tame kangaroo jumps, it sets up a series of traps. The wild kangaroo will be entrapped if it lands on any of them along its route. If this happens, we can establish a simple equation and solve the discrete logarithm.

In [1], Pollard defines the jumping functions of the kangaroos (in the example of Kruskal's Count, it would be equivalent to defining how we go from one card to another). He also indicates how to get the result of the discrete logarithm when one kangaroo catches the other, and offers an example of a jump evaluation function (similar to being on a card and figuring out what value it has, to know how many cards we must move on).

The deck is equivalent to the set G of a group. In this case, we consider the multiplicative group $\mathbb{Z}_p^*$. A more straightforward way to see it is to imagine that the kangaroos jump over the number line, although ordered according to how the generator α generates the group G [1].

Furthermore, we define a set of integers S that represents the possible values for the jumps (in Kruscal's Count, S would be the set of card values, i.e., $S = \{1, 2, 3, \ldots, 9, 10\}$).

Finally, it should be remarked that one of the advantages of this algorithm is that it does not need to search the entire search space. We can limit the search by establishing boundaries a and b so that it only searches between those numbers.

The tame kangaroo starts at position x_0 and performs a series of jumps x_i. With each jump, it deposits a trap at position x_i and travels a total distance d_i. For each jump we store the pair (x_i, d_i). Since, in this case, the "deck" is cyclical and has no end, we define the value σ that represents the number of jumps the tame kangaroo will make before stopping and resting (equivalent to putting down the coin).

Similarly, the wild kangaroo starts from x'_0 and makes a series of jumps x'_i and covers a total distance d'_i that we store. If the wild kangaroo lands on a trap at any point in its journey, it will be irremediably trapped (like if our friend lands on a card from our path), and we can solve the discrete logarithm. On the other hand, if it avoids all the traps, the algorithm will stop, indicating that the solution has not been found.

Mathematically, kangaroos are represented as follows:

$$x_{i+1} = x_i \cdot \alpha^{f(x_i)} \mod p$$

$$d_i = f(x_0) + f(x_1) + \ldots + f(x_{i-1})$$

[1] For simplicity, in the illustrations of the number line we have omitted this ordering.

Where $f(x_i)$ is the jump evaluation function, which given a position (the terrain / the card), returns a value of S (quality of the terrain, i.e. how much it can jump / value of the card).

In [1], Pollard defines f as:

$$f(x_i) = \mu^j\,, \quad 0 \leq j \leq \mu - 1\,, \quad \text{where } j \equiv x_i \pmod{\mu}$$

Where $\mu = |S|$ and recommends that $\mu \ll \sqrt{b-a}$.

The starting points of the kangaroos are the following:

$$x_0 \equiv \alpha^b \pmod{p} \qquad x_0' \equiv \alpha^k \pmod{p} = \beta$$
$$d_0 = 0 \qquad d_0' = 0$$

The solution k to the discrete logarithm is found if at any time the wild kangaroo falls into a trap. This is equivalent to saying that $x_N = x'_M$ for some N and M. Let us develop this equality:

$$\begin{aligned}
x_N &\equiv x'_M && \pmod{p}\\
x_{N-1} \cdot \alpha^{f(x_{N-1})} &\equiv x'_{M-1} \cdot \alpha^{f(x'_{M-1})} && \pmod{p}\\
x_0 \cdot \alpha^{d_N} &\equiv x'_0 \cdot \alpha^{d'_M} && \pmod{p}\\
\alpha^b \cdot \alpha^{d_N} &\equiv \alpha^k \cdot \alpha^{d'_M} && \pmod{p}\\
\alpha^{b+d_N} &\equiv \alpha^{k+d'_M} && \pmod{p}\\
b + d_N &\equiv k + d'_M && \pmod{\phi(p)}\\
b + d_N &\equiv k + d'_M && \pmod{p-1}\\
k &\equiv b + d_N - d'_M && \pmod{p-1}
\end{aligned}$$

Therefore, once some x and some x' collide, solving the discrete logarithm is as simple as calculating $k \equiv b + d_N - d'_M \pmod{p-1}$.

On the other hand, the calculation of x' (wild kangaroo jumps) can stop [2] when $d'_M > b + d_N - a$, as this indicates that the wild kangaroo has avoided all the traps [1].

In algorithm 6.2, we can see the resulting pseudocode:

Algorithm 6.2: *Pollard's Kangaroo* procedure

```
Initialize the initial positions of both kangaroos;
The tame kangaroo performs σ jumps and sets σ traps;
while the wild kangaroo does not fall into a trap do
    Perform a wild kangaroo jump;
    if the stopping condition is met, the algorithm has failed, stop;
end
return b + d_N − d'_M (mod p − 1)
```

It is interesting to remark that we state that the tame kangaroo should complete all its jumps before the wild kangaroo starts to jump. Although this technique is the same as the one described in [1], it is not the only one that exists in the literature. For example, it would also be possible to alternate their jumps and check, after each pair, if a collision occurs.

[2] Note that the calculations of x (tamed kangaroo jumps) are by definition limited since $0 \leq N \leq \sigma$.

Now let us see a *minimal* example of how this algorithm works and how it actually finds the solution. It should be noted that we have mentioned a series of hyperparameters to which we have to assign a value:

a and b define the search interval. Although, in theory, this would allow us to reduce the search space, the reality is that the modulo function is widely used in cryptography precisely because it blurs any clues about the area in which we should search for the solution. Therefore, since we cannot define a specific search interval, we will search the entire interval, that is, $a = 1$ and $b = p - 1$.

On the other hand, we have the size of the group S, that is, $\mu = |S|$. We initially give the value $\mu = 4$, just as Pollard does in [1] and later in section 6.5 we will analyze if this value is the most suitable or if there is another better option.

Finally, we have σ, which determines the number of jumps the tame kangaroo makes and the number of set traps. For this small example, four traps are more than enough.

Therefore, this example is only 4 bits long and has the following parameters:

$$\alpha^k \mod p = \beta \qquad \text{where} \begin{cases} \alpha = 2 \\ p = 11 \\ \beta = 9 \end{cases} \quad \text{and} \begin{cases} a = 1 \\ b = p - 1 = 10 \\ \sigma = 4 \\ \mu = 4 \end{cases}$$

In this case, the solution can be calculated by hand, and we can see that it is six:

$$64 \equiv 9 \ (\text{mód } 11)$$

$$2^6 \equiv 9 \ (\text{mód } 11)$$

$$\Downarrow$$

$$k = 6$$

Now let us see how it would be solved with *Pollard's Kangaroo Algorithm*:

Jump function and power precalculation

$$f(x_i) = \mu^j\,, \quad 0 \leq j \leq \mu - 1\,, \quad \text{where } j \equiv x_i \pmod{\mu}$$

$$\Downarrow$$

$$f(x_i) = 4^j\,, \quad 0 \leq j \leq 3\,, \quad \text{where } j \equiv x_i \pmod{4}$$

Since $j \in \{0,1,2,3\}$, we know that $f(x_i) \in \{4^0, 4^1, 4^2, 4^3\}$ and therefore, $\alpha^{f(x_i)} \in \{\alpha^1, \alpha^4, \alpha^{16}, \alpha^{64}\} \pmod{p}$, so we can precalculate the powers:

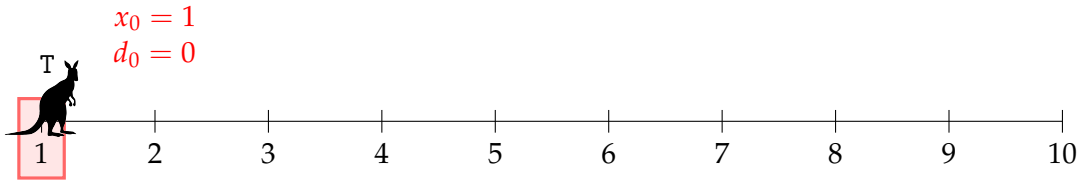
- $2^1 \pmod{11} = 2$
- $2^4 \pmod{11} = 5$
- $2^{16} \pmod{11} = 9$
- $2^{64} \pmod{11} = 5$

And therefore:

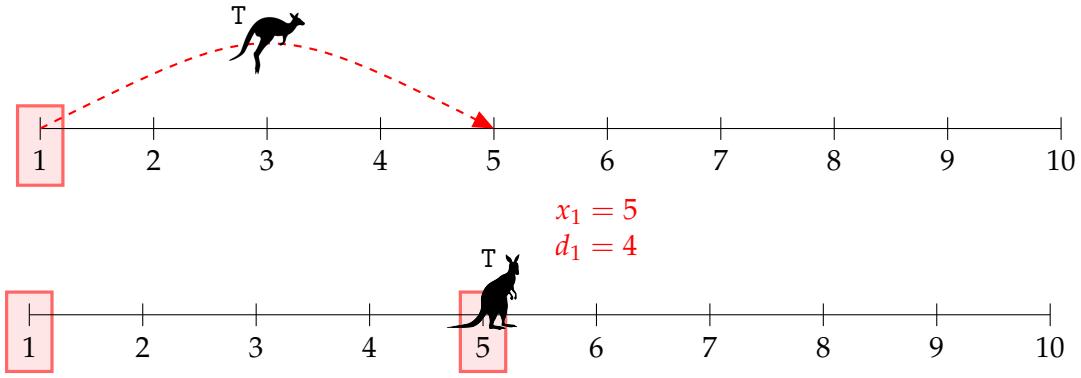
$$\texttt{pows = [2,5,9,5];} \qquad \alpha^{f(x_i)} = \texttt{pows[}x_i\texttt{ \% 4]}$$

Tame kangaroo

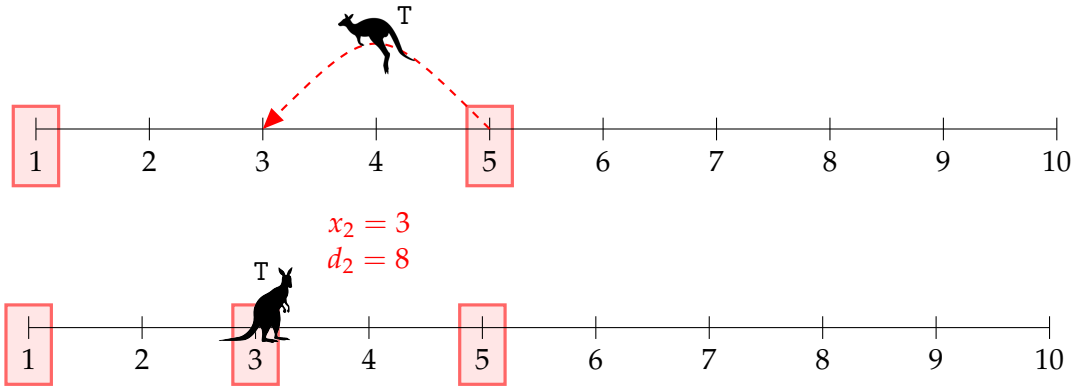
$$x_0 \equiv \alpha^b \pmod p \qquad \Rightarrow \qquad x_0 = 2^{10} \bmod 11 = 1$$



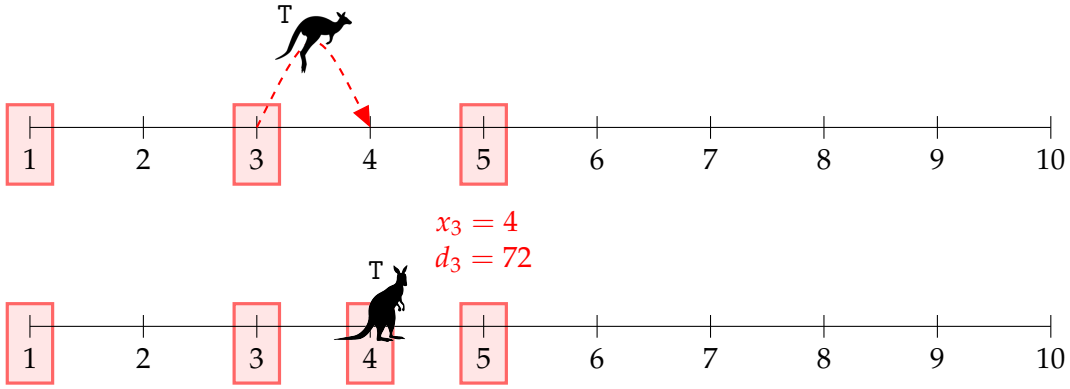
$$x_1 = x_0 \cdot \alpha^{f(x_0)} \pmod p = 1 \cdot 2^{f(1)} \pmod{11} = 1 \cdot 2^4 \pmod{11} =$$
$$= 1 \cdot \texttt{pows[}x_0\ \texttt{\%}\ 4\texttt{]} \pmod{11} = 1 \cdot \texttt{pows[1]} \pmod{11} = 5$$
$$d_1 = f(x_0) = f(1) = 4$$



$$x_2 = x_1 \cdot \alpha^{f(x_1)} \pmod p = 5 \cdot 2^{f(5)} \pmod{11} = 5 \cdot 2^{f(1)} \pmod{11} =$$
$$= 5 \cdot \texttt{pows[1]} \pmod{11} = 5 \cdot 5 \pmod{11} = 3$$
$$d_2 = f(x_0) + f(x_1) = d_1 + f(x_1) = 4 + f(5) = 4 + f(1) = 4 + 4 = 8$$

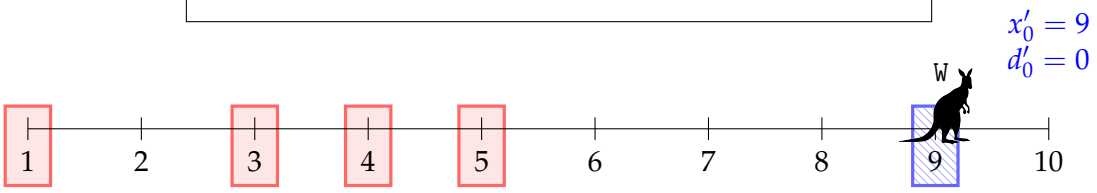


$$x_3 = x_2 \cdot \alpha^{f(x_2)} \pmod p = 3 \cdot 2^{f(3)} \pmod{11} =$$
$$= 3 \cdot \texttt{pows[3]} \pmod{11} = 3 \cdot 5 \pmod{11} = 4$$
$$d_3 = f(x_0) + f(x_1) + f(x_2) = d_2 + f(x_2) = 8 + f(3) = 8 + 64 = 72$$

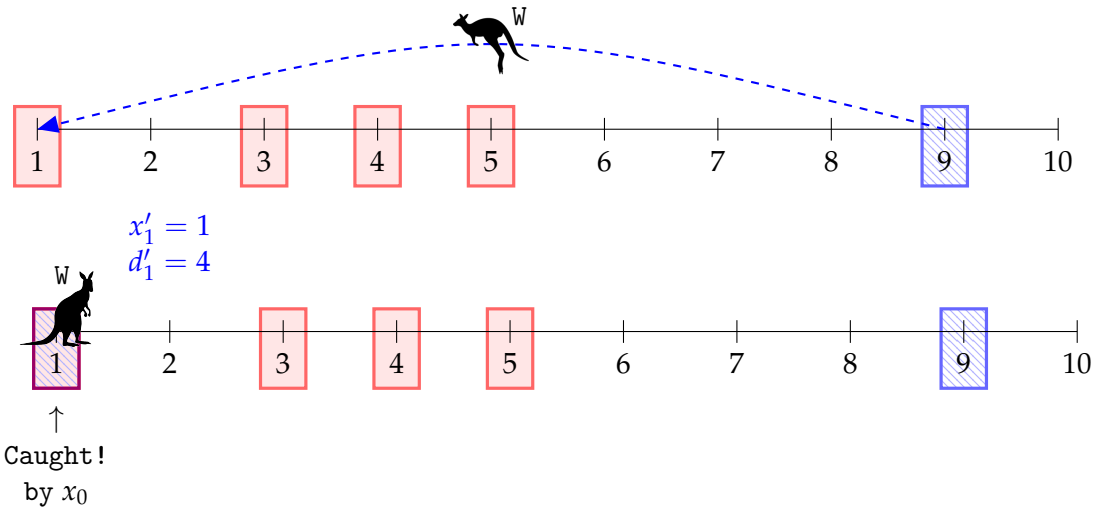


Wild kangaroo

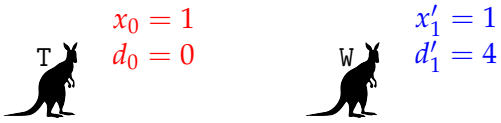
$$x'_0 \equiv \alpha^k \pmod p = \beta \qquad \Rightarrow \qquad x'_0 = \beta = 9$$



$$x'_1 = x'_0 \cdot \alpha^{f(x'_0)} \pmod p = 9 \cdot 2^{f(1)} \pmod{11} = 9 \cdot 2^4 \pmod{11} =$$
$$= 9 \cdot \texttt{pows[1]} \pmod{11} = 9 \cdot 5 \pmod{11} = 1$$
$$d'_1 = f(x'_0) = f(1) = 4$$



Collision



$$
\begin{aligned}
x_N &\equiv x'_M && \pmod{p} \\
x_{N-1} \cdot \alpha^{f(x_{N-1})} &\equiv x'_{M-1} \cdot \alpha^{f(x'_{M-1})} && \pmod{p} \\
x_0 \cdot \alpha^{d_N} &\equiv x'_0 \cdot \alpha^{d'_M} && \pmod{p} \\
\alpha^b \cdot \alpha^{d_N} &\equiv \alpha^k \cdot \alpha^{d'_M} && \pmod{p} \\
\alpha^{b+d_N} &\equiv \alpha^{k+d'_M} && \pmod{p} \\
b + d_N &\equiv k + d'_M && \pmod{\phi(p)} \\
b + d_N &\equiv k + d'_M && \pmod{p-1} \\
k &\equiv b + d_N - d'_M && \pmod{p-1} \\
&\Downarrow \\
k &\equiv b + d_0 - d'_1 && \pmod{p-1} \\
k &\equiv 10 + 0 - 4 && \pmod{10} \\
k &= 6
\end{aligned}
$$

As can be seen, in this case the algorithm has worked perfectly and has found the solution to the discrete logarithm.

Once this algorithm had been implemented in Python, we experimented with different values of σ to determine its optimal value for much more significant problems. We performed the experimentation with the largest values that were temporarily reasonable in the case of brute force. The results can be seen in table 6.2.

Configuration	$\sigma = 10$ $\mu = 4$	$\sigma = 50$ $\mu = 4$	$\sigma = 100$ $\mu = 4$	$\boldsymbol{\sigma = 500}$ $\boldsymbol{\mu = 4}$	$\sigma = 1000$ $\mu = 4$	$\sigma = 5000$ $\mu = 4$	$\sigma = 10000$ $\mu = 4$
24 bits problem-set	0.099s 23.33	0.098s 80.00	0.097s 100.00	0.098s 100.00	0.098s 100.00	0.099s 100.00	0.103s 100.00
28 bits problem-set	1.24s 26.67	1.85s 80.00	1.89s 96.67	1.89s 100.00	1.88s 100.00	1.89s 100.00	1.93s 100.00
32 bits problem-set	18.4s 16.67	29.4s 83.33	32.2s 100.00	32.5s 100.00	32.5s 100.00	32.7s 100.00	33.0s 100.00

Table 6.2: Execution time and accuracy of different configurations for the Kangaroo-v.1 algorithm.

As we can see, the results reveal two facts. First, a very small value of σ implies that very few traps are set, and then the problem is seldom solved. On the other hand, placing an enormous amount of traps does not excessively increase the computation time. However, storing many traps is not interesting due to the extra space cost, and it is reasonable to assume that setting σ too elevated will result in an unnecessary cost overrun that may be substantially higher for larger problems.

For all these reasons, we chose the value of $\sigma = 500$ as the optimal one in the absence of further experimentation with the combination of σ and μ. We leave this experimentation for a later, more optimized version, which should allow us to attack more significant problems (see section 6.5).

Once we have decided on the optimal value of σ, we have done a complete experimentation with $\sigma = 500$ and $\mu = 4$. The results can be seen in table 6.3 and figure 6.2.

Problem size	12	16	20	24	28	32	36
Time	0.0002s	0.0003s	0.005s	0.098s	1.88s	32.5s	9.6min
% Accuracy	96.67	100.00	100.00	100.00	100.00	100.00	100.00

Problem size	40	44	48	52	56	60	64
Time	2.6h	1.7d	27.6d	1.2y	19.4y	309.8y	4956.2y
% Accuracy	100.00	Estimated	Estimated	Estimated	Estimated	Estimated	Estimated

Table 6.3: Execution time and accuracy of the Kangaroo-v1 algorithm.

As can be seen, this algorithm is much more efficient than the brute force version. However, as expected, very large values will continue to be unfeasible due to the complexity of the problem itself.

Nevertheless, we were able to solve problems up to 40 bits with this approximation, with an average time of 2.6 hours, which is a significant reduction from the two weeks that the brute force version was expected to take. In general, this version has allowed us to solve problems in a search-space $2^8 = 256$ times larger, requiring a similar amount of time.

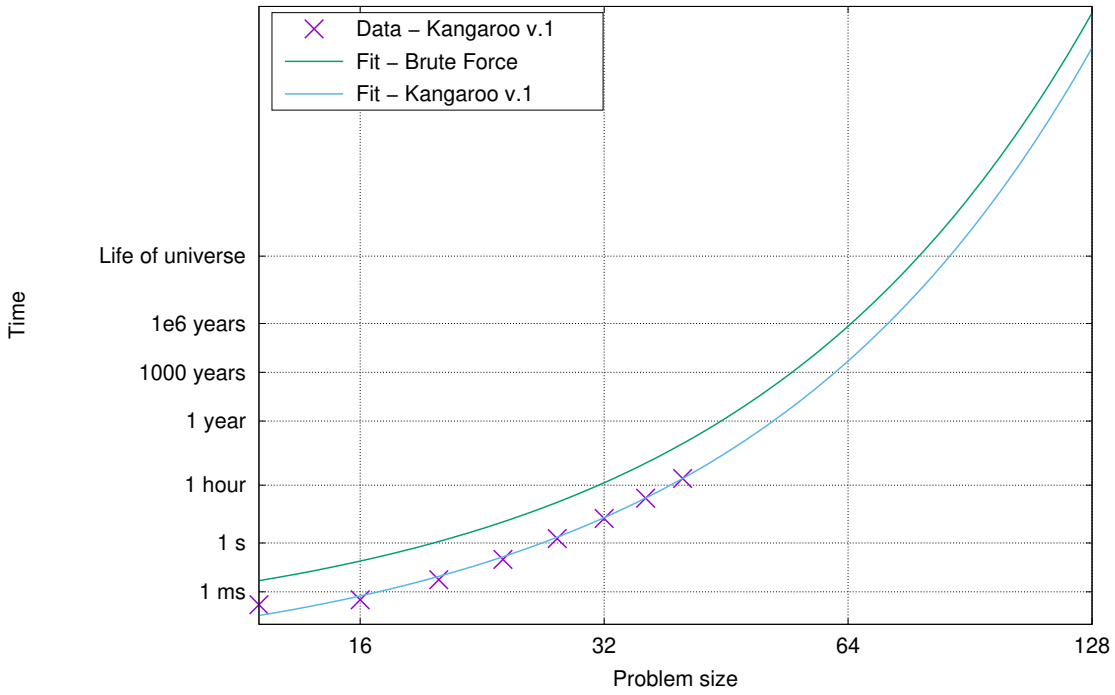


Figure 6.2: Time required to solve the discrete logarithm by the Kangaroo v.1 algorithm.

6.3 Kangaroo v.2 - Creating a *Wild Jumps Database*

In this second version of the algorithm, we make a variety of optimizations to the first version in order to reduce its execution time.

An initial step to achieving this optimization is modifying how we try to solve the problem. In this version, instead of searching the entire interval directly, i.e. setting $a = 1$ and $b = p - 1$, we divide the whole interval into chunks of fixed size, which we refer to as "sections".

In this way, we now apply *Pollard's Kangaroo Algorithm* to only one section at a time, setting the values of a and b as the start and end of the section. If the solution to the discrete logarithm is found in that section, it is returned, and the algorithm ends. Otherwise, it is searched in the next section, as shown in simplified form in algorithm 6.3:

Algorithm 6.3: Kangaroo v.2 - Basic procedure

```
Divide interval in sections;
foreach section ∈ sections do
    Determine the limits a and b;
    k ← Kangaroo(α, p, β, a, b);
    if k ≠ NULL then
        return k
    end
end
if k is not found and there are no sections left to explore then
    the algorithm has failed, stop;
end
```

Operating in this way, we can appreciate a repetitive calculation: once the tame kangaroo has calculated its σ jumps and has placed its σ traps (which depend on the value of b and, therefore, change every section), wild jumps are computed until a collision occurs or the stop condition is met. However, the jumps of the wild kangaroo do not depend on a or b, so they will always be the same for a given discrete logarithm problem.

Therefore, an optimization applicable to the algorithm is to create what we have named the "Wild Jumps Database" (WJD). In each section, the idea now is that once the tame kangaroo has calculated its σ jumps, these jumps intersect with the WJD.

If there is a collision, we directly have a solution to the problem, and the wild kangaroo does not need to make any jump. Otherwise, wild kangaroo jumps continue to be calculated and accumulated in the WJD until the solution is found, or the stop condition is met and the next section is processed.

In this way, we can achieve a significant reduction in the calculation. Generally, for the first section, many wild jumps will be carried out, and the Wild Jumps Database will be filled in. However, no wild jumps will be required for the remaining sections since either a collision will occur or d' will already be high enough to meet the halting condition, and we will not waste time looking at a section where the solution is not located.

Therefore, we have given a twist to the algorithm. Until now, the tame kangaroo has been setting traps, and, with luck, the wild kangaroo would fall into one of them. Now, being the tame kangaroo on a piece of land, we know if the wild has passed through here. If it does, we set the equation the same way as in version one. Thus, the tame kangaroo has become a detective instead of a hunter.

After implementing this algorithm in Python and verifying that it works correctly, we have performed a large-scale experimentation with $\sigma = 500$ and $\mu = 4$. The size of the sections is $2^{16} = 65\,536$, which *a priori* [3] seems large enough to avoid an excessive number of interval divisions, but not too large so that the kangaroo algorithm can solve the subproblem in a very short time because it must be done many times. The results can be seen in table 6.4 and figure 6.3.

Problem size	12	16	20	24	28	32	36
Time	0.0002s	0.0005s	0.004s	0.029s	0.473s	8.65s	2.7min
% Accuracy	96.67	100.00	100.00	100.00	100.00	100.00	100.00

Problem size	40	44	48	52	56	60	64
Time	40.9min	10.9h	7.3d	116.3d	5.1y	81.5y	1303.8y
% Accuracy	100.00	Estimated	Estimated	Estimated	Estimated	Estimated	Estimated

Table 6.4: Execution time and accuracy of the Kangaroo-v.2 algorithm.

As can be seen, this new way of finding the solution allows us to reduce the search time by more than a half. Specifically, the test-sets are solved about 3.75 times faster than in the first version of the algorithm. For example, a 44-bit problem would cost approximately 11 hours instead of the nearly two days of the `v.1` and the 241 days of the brute force algorithm, a considerable reduction.

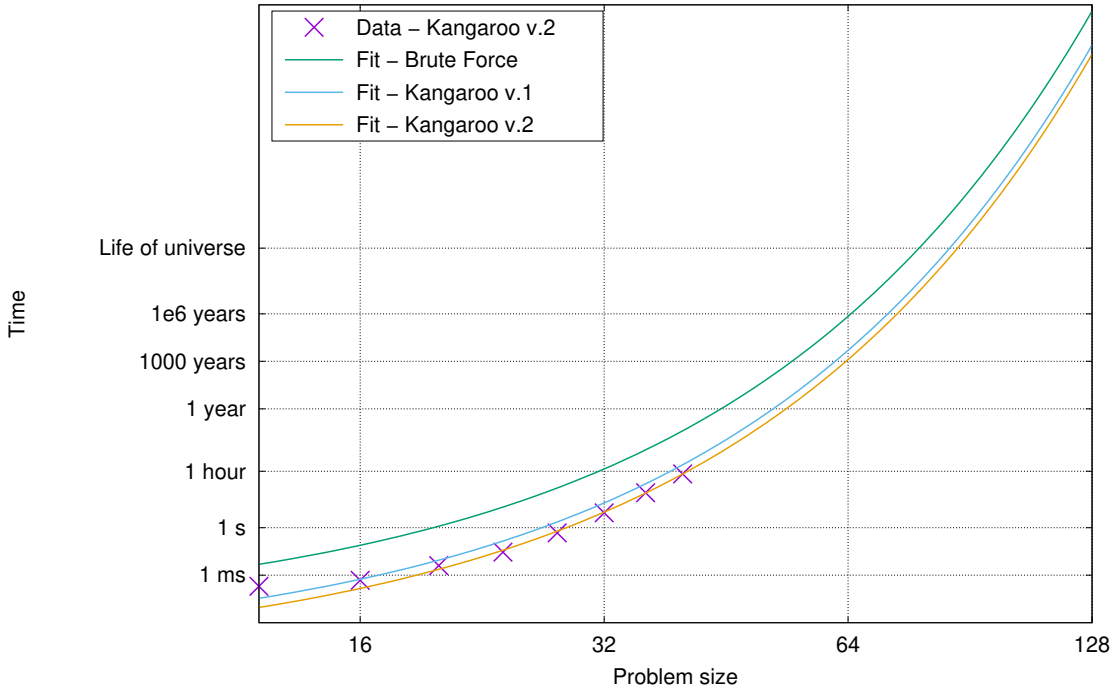


Figure 6.3: Time required to solve the discrete logarithm by the Kangaroo v.2 algorithm.

Despite these promising results, we need to check if the assumption we have made when setting the size of the sections to 2^{16} elements is adequate or if we can further improve this approach, as we see in section 6.4.

[3] In the third version of the algorithm we analyze if it is better to follow another more elaborate strategy.

6.4 Kangaroo v.3 - Dynamically sized sections

In this section, we perform an analysis of which is the ideal size for the sections in which we divide the number line. For this purpose, we define a new hyperparameter: λ. In the second version of the algorithm, we have assumed that this value can be constant regardless of the problem size and that sections of 2^{16} elements were a suitable dimension. Throughout this passage, we review if this assumption is accurate.

This new hyperparameter, λ, will determine the size of the sections as follows: the entire interval from 1 to $p-1$ will be divided into sections with 2^{λ} values. Thus, the second version of the algorithm had a value of $\lambda = 16$ since the sections contained $2^{16} = 65\,536$ values.

As discussed in section 6.3, it seems reasonable to think that a very small value of λ will produce too many sections with such trivial problems that more computation time will be spent initializing the algorithm than searching for the solution itself, causing a significant overhead. On the other hand, sections with a considerable size would lead us to a situation similar to the first version of the algorithm where we search the entire interval, thus not taking advantage of the speedup achieved thanks to the *Wild Jumps Database*.

For these reasons, while the value of $\lambda = 16$ was adequate for us to evaluate the second version of the algorithm and verify that the new search strategy and the *Wild Jumps Database* were able to reduce the times of the first version significantly, it seemed necessary to determine whether this value is the optimal one for achieving good results, or if it is too small and we should use larger values.

Thus, we have experimented with distinct values of λ for different problem sizes. The results can be seen in table 6.5 and plot 6.4.

Problem size	24	26	28	30	32	34	36	38	40
$\lambda = 12$	0.304s 100.00	0.864s 100.00	>5s killed	>10s killed	>20s killed	>80s killed	>5min killed	>21min killed	>1.4h killed
$\lambda = 16$	**0.026s** 100.00	0.073s 100.00	0.527s 100.00	0.900s 100.00	9.27s 100.00	43.4s 100.00	2.3min 100.00	6.3min 100.00	33.5min 100.00
$\lambda = 20$	0.039s 100.00	**0.052s** 100.00	**0.186s** 100.00	**0.301s** 100.00	**2.65s** 100.00	11.8s 100.00	36.7s 100.00	1.7min 100.00	8.9min 100.00
$\lambda = 24$	0.184s 100.00	0.308s 100.00	0.678s 100.00	0.609s 100.00	2.89s 100.00	**10.8s** 100.00	**32.4s** 100.00	**88.5s** 100.00	7.8min 100.00
$\lambda = 28$	0.183s 100.00	1.14s 100.00	2.69s 100.00	>10s killed	13.3s 100.00	21.3s 100.00	42.7s 100.00	97.6s 100.00	**7.7min** 100.00
$\lambda = 32$	0.187s 100.00	1.12s 100.00	2.72s 100.00	>10s killed	>20s killed	– out of RAM	– out of RAM	– out of RAM	– out of RAM

Table 6.5: Execution time and accuracy of different λ configurations for the Kangaroo-v.3 algorithm. Bold indicates the best result for that problem-set.

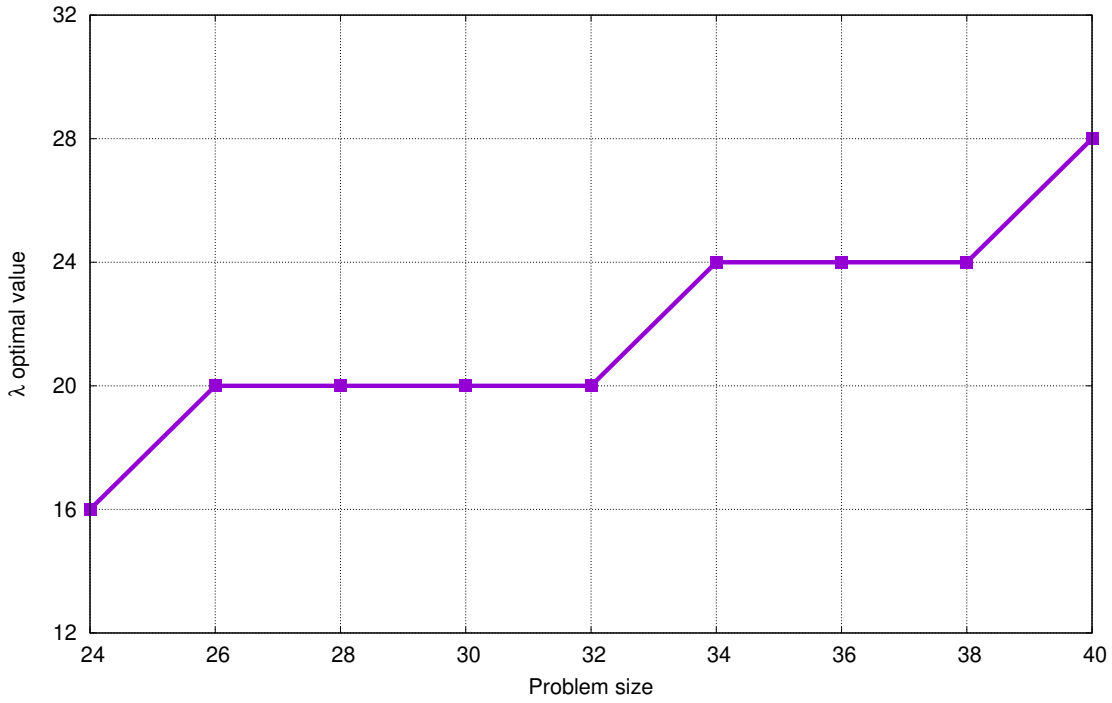


Figure 6.4: λ optimal value per problem size according to Table 6.5.

As can be seen, it seems that there is no optimal value for λ, but rather that the optimal value increases as the problem increases. In turn, this is a great inconvenience since we cannot simply increase its value because values as low as $\lambda = 32$ already cause the algorithm to be cancelled due to excessive RAM consumption [4].

On the other hand, another important conclusion that we draw is that $\lambda = 16$ will not generally be a too propitious value since other configurations can solve the discrete logarithm in less than half the time.

Given this situation and taking into account the results obtained, we have determined that the value of λ should be set dynamically according to the size of the problem to be solved.

Since the execution time for insignificant problems of the previous versions is small *per se*, it seems counterproductive to create such tiny sections due to the overhead it would cause, so we lower bound the size of the sections to $\lambda = 16$. On the other hand, due to RAM memory limitations, we have also decided to limit the value of λ further, leaving it at a maximum of $\lambda = 26$. We have chosen the value $\lambda = 26$ instead of 28 or 30 so that it does not use too much RAM memory and we have enough margin to parallelize the algorithm [5].

[4] It is convenient to remember that one of the strengths of *Pollard's Kangaroo Algorithm* is that it does not need (or should not need) large amounts of RAM. Therefore, we advise against performing experiments by configuring more RAM in the computer. If large amounts of RAM are available, it is probably better to try to solve the discrete logarithm problem using the *Baby-step giant-step* algorithm or similar.

[5] Since we have eight threads of execution, we thus limit the sequential algorithm so that it does not use more than $\frac{1}{8}$ of the total computer RAM.

Given all the above conclusions, we have defined the following piecewise linear function to define the optimal value of the hyperparameter λ:

$$t = \lceil \log_2 p \rceil \, ; \qquad \lambda^* = \begin{cases} 16 & \texttt{if } t < 28 \\ t - 12 & \texttt{if } 28 \leq t \leq 38 \\ 26 & \texttt{if } t > 38 \end{cases}$$

Once this modification has been implemented on the v.2 algorithm, we have done a complete experimentation with $\sigma = 500, \mu = 4$, and $\lambda = \lambda^*$. The results can be seen in table 6.6 and figure 6.5.

Problem size	12	16	20	24	28	32	36
Time	0.0002s	0.0005s	0.004s	0.029s	0.410s	2.43s	38.0s
% Accuracy	96.67	100.00	100.00	100.00	100.00	100.00	100.00

Problem size	40	44	48	52	56	60	64
Time	9.5min	2.5h	1.7d	26.9d	1.2y	18.8y	301.5y
% Accuracy	100.00	Estimated	Estimated	Estimated	Estimated	Estimated	Estimated

Table 6.6: Execution time and accuracy of the Kangaroo-v.3 algorithm.

Although it is true that this optimization mainly affects problems between 30 and 40 bits, we can see that the overall time is lowered for larger problems as well. Therefore we can assert that a value of $\lambda = 26$ is better than $\lambda = 16$ on large problems.

In conclusion, this proposal has achieved an acceleration in problems larger than 28 bits, solving them approximately four times faster. For example, on average, a 40-bit problem would take about 10 minutes instead of the 40 minutes of the second version.

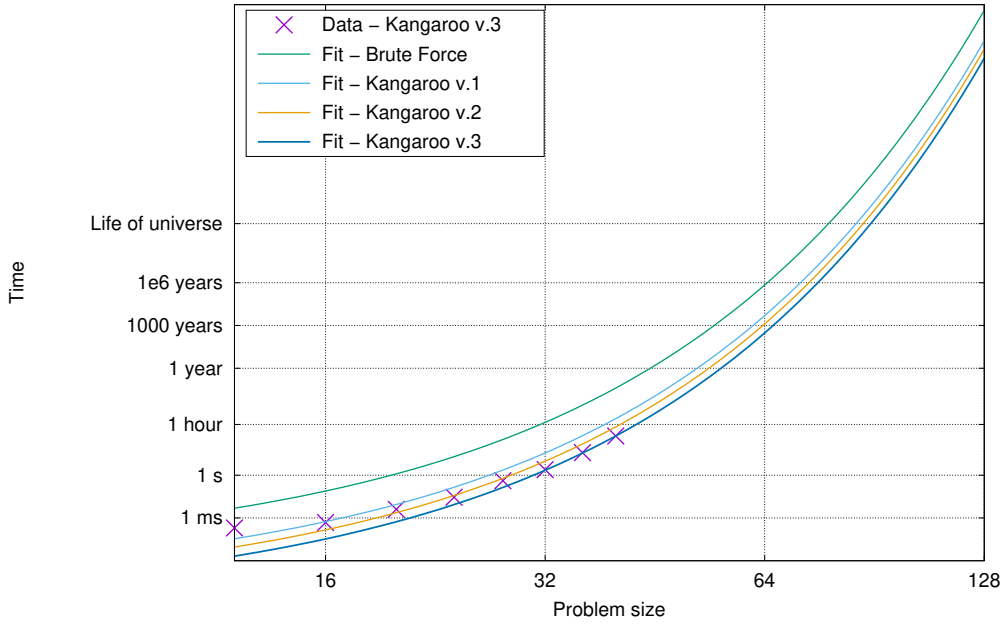


Figure 6.5: Time required to solve the discrete logarithm by the Kangaroo v.3 algorithm.

6.5 Kangaroo v.4 - Algorithm parallelization

The last significant modification we have made to our algorithm is to implement a parallel version of it in order to take advantage of all of the machine's cores and thus be able to solve larger problems in a similar time.

This new algorithm follows a coordinator-worker architecture by parallelizing the sequential algorithm with the *Wild Jumps Database*. It also implements the *Dynamically sized sections* as they have proven to be helpful in large problems.

Thus, being th the number of threads we have available, we keep one thread of execution as coordinator, and the rest of the threads ($th - 1$) will be the workers that apply our third version of *Pollard's Kangaroo Algorithm*.

The procedure applied is as follows: the coordinator divides the number line into $th - 1$ intervals and assigns each to a worker. Each worker, in turn, divides it into sections of size 2^λ and applies the strategy seen in section 6.3 (see Algorithm 6.3).

Once the distribution has been made, the coordinator is left to wait for a thread to (hopefully) find the solution, while the workers search for it in their sections. Algorithms 6.4 and 6.5 explain in pseudocode these procedures.

Algorithm 6.4: Kangaroo v.4 - Coordinator procedure

Divide the number line into $th - 1$ intervals;
foreach *worker* $\in$ *workers* **do**
  Send its sub-interval;
end
failure counter $\leftarrow 0$;
while *failure counter* $\neq th - 1$ **do**
  Wait to receive a result;
  if k is received, terminate all workers and **return** k;
  if `None` is received, it implies that a worker has exhausted its sections, increase the *failure counter* by one;
end
if the *failure counter* is equal to $th - 1$, no workers have found the solution and there are no sections left to explore, the algorithm has failed, **stop**;

Algorithm 6.5: Kangaroo v.4 - Worker procedure

Receive sub-interval and divide it in sections;
while *sections remain to be explored in my sub-interval* **do**
  Select one section to explore;
  Determine the boundaries a and b of that section;
  $res \leftarrow Kangaroo(\alpha, p, \beta, a, b)$;
  if $res = -1$, it means that the solution has not been found in that section, go to the next one;
  if $res > 0$, it implies that k has been found, send k to coordinator;
end
if k has not been found in any section, send `None` to coordinator;

After implementing this algorithm in Python and verifying that it works correctly, we decided to perform the experimentation that we had pending with different values of μ to determine its optimal value. We also decided to make another one with different values of σ since its optimum could have changed with the parallelization and the rest of the modifications made in versions two and three. The results can be seen in tables 6.7 and 6.8.

Config. \ Problem size	12	16	20	24	28	32	36	40
$\sigma = 5$ $\mu = 4 \quad \lambda = \lambda^*$	0.004s 100.00	0.005s 86.67	0.005s 33.33	0.010s 16.67	0.044s 30.00	0.560s 26.67	11.4s 30.00	2.6min 13.33
$\sigma = 50$ $\mu = 4 \quad \lambda = \lambda^*$	0.003s 100.00	0.008s 100.00	0.012s 100.00	0.015s 90.00	0.050s 80.00	0.713s 90.00	12.6s 86.67	2.4min 73.33
$\sigma = 500$ $\mu = 4 \quad \lambda = \lambda^*$	0.003s 100.00	0.003s 100.00	0.066s 100.00	0.081s 100.00	0.126s 100.00	0.763s 100.00	11.0s 100.00	2.3min 100.00
$\sigma = 5000$ $\mu = 4 \quad \lambda = \lambda^*$	0.005s 100.00	0.005s 100.00	0.008s 100.00	0.516s 100.00	0.934s 100.00	1.74s 100.00	12.4s 100.00	2.3min 100.00
$\sigma = 50000$ $\mu = 4 \quad \lambda = \lambda^*$	0.020s 100.00	0.021s 100.00	0.027s 100.00	0.055s 100.00	9.44s 100.00	13.0s 100.00	32.2s 100.00	4.8min 100.00

Table 6.7: Execution time and accuracy of different configurations of σ for the Kangaroo-v.4 algorithm.

Config. \ Problem size	12	16	20	24	28	32	36
$\sigma = 500$ $\mu = 1 \quad \lambda = \lambda^*$	0.004s 100.00	0.055s 100.00	0.069s 100.00	0.108s 100.00	0.815s 100.00	13.2s 100.00	3.8min 100.00
$\sigma = 500$ $\mu = 2 \quad \lambda = \lambda^*$	0.004s 100.00	0.053s 100.00	0.078s 100.00	0.104s 100.00	0.572s 100.00	9.03s 100.00	2.5min 100.00
$\sigma = 500$ $\mu = 3 \quad \lambda = \lambda^*$	0.003s 100.00	0.013s 100.00	0.075s 100.00	0.090s 100.00	0.255s 100.00	3.15s 100.00	49.3s 100.00
$\sigma = 500$ $\mu = 4 \quad \lambda = \lambda^*$	0.078s 100.00	0.079s 100.00	0.165s 100.00	0.189s 100.00	0.227s 100.00	0.825s 100.00	10.9s 100.00
$\sigma = 500$ $\mu = 5 \quad \lambda = \lambda^*$	0.003s 100.00	0.003s 100.00	0.005s 100.00	0.066s 100.00	0.097s 100.00	0.199s 93.33	1.61s 96.67
$\sigma = 500$ $\mu = 6 \quad \lambda = \lambda^*$	0.004s 100.00	0.003s 100.00	0.004s 100.00	0.026s 100.00	0.096s 96.67	0.129s 43.33	0.237s 20.00
$\sigma = 500$ $\mu = 7 \quad \lambda = \lambda^*$	0.004s 100.00	0.003s 100.00	0.004s 100.00	0.008s 100.00	0.059s 93.33	0.096s 33.33	0.140s 6.67
$\sigma = 500$ $\mu = 8 \quad \lambda = \lambda^*$	0.006s 100.00	0.003s 100.00	0.004s 100.00	0.010s 100.00	0.045s 83.33	0.093s 20.00	0.209s 3.33

Table 6.8: Execution time and accuracy of different configurations of μ for the Kangaroo-v.4 algorithm.

Despite the several optimizations and improvements we have performed to our algorithm, the optimal value of σ does not actually change. As can be seen, very small σ values make the process slightly faster but also cause it to fail frequently. On the other hand, huge σ values anticipate excellent accuracy at the risk of taking longer than necessary to solve the problem. For all these reasons and considering the results shown, we still choose $\sigma = 500$ as the optimal value.

For its part, the results of the μ analysis indicate that while this value does not matter too much in small problems, it becomes a critical parameter in more significant problems. On the one hand, we see that large values of μ achieve speedy times but sacrifice the percentage of successes too much. On the other hand, small values cause the algorithm to take longer than required. In conclusion, it seems that Pollard was not wrong when he chose the value of $\mu = 4$, although the results for $\mu = 5$ also seem reasonable and even more attractive.

For all of the above reasons, we have performed two large-scale experiments, one with $\sigma = 500, \mu = 4$, and $\lambda = \lambda^*$, and another with $\sigma = 500, \mu = 5$, and $\lambda = \lambda^*$, and we finally selected the one that provides the best overall performance. The results can be seen in tables 6.9, 6.10, and figure 6.6.

Problem size	12	16	20	24	28	32	36
Time	0.003s	0.003s	0.067s	0.080s	0.128s	0.798s	11.2s
% Accuracy	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Problem size	40	44	48	52	56	60	64
Time	2.5min	44.8min	11.9h	8.0d	127.4d	5.6y	89.6y
% Accuracy	100.00	100.00	Estimated	Estimated	Estimated	Estimated	Estimated

Table 6.9: Execution time and accuracy of the Kangaroo-v.4 algorithm. [$\mu = 4$]

Problem size	12	16	20	24	28	32	36
Time	0.003s	0.003s	0.004s	0.065s	0.098s	0.198s	1.65s
% Accuracy	100.00	100.00	100.00	100.00	100.00	93.33	96.67

Problem size	40	44	48	52	56	60	64
Time	19.0s	6.3min	1.7h	1.1d	17.8d	284.7d	12.5y
% Accuracy	93.33	86.67	Estimated	Estimated	Estimated	Estimated	Estimated

Table 6.10: Execution time and accuracy of the Kangaroo-v.4 algorithm. [$\mu = 5$]

As can be seen, if we compare the results of both experiments, we can reach two conclusions: on one side, the value of $\mu = 4$ provides us with more reliable results, obtaining 100% accuracy in all the tests performed. On the other side, the tests with $\mu = 5$ are around seven times faster, a pretty remarkable improvement but reducing the accuracy by about 10%.

As a proposal, our recommendation would be to select $\mu = 5$ since the time reduction is considerable and, if high accuracy is required, to try to increase the value of σ. According to previous results, by increasing it slightly we can achieve better values in accuracy

and still obtain times similar to those we have with $\mu = 5$ and $\sigma = 500$, without reaching the higher times of $\mu = 4$ and $\sigma = 500$.

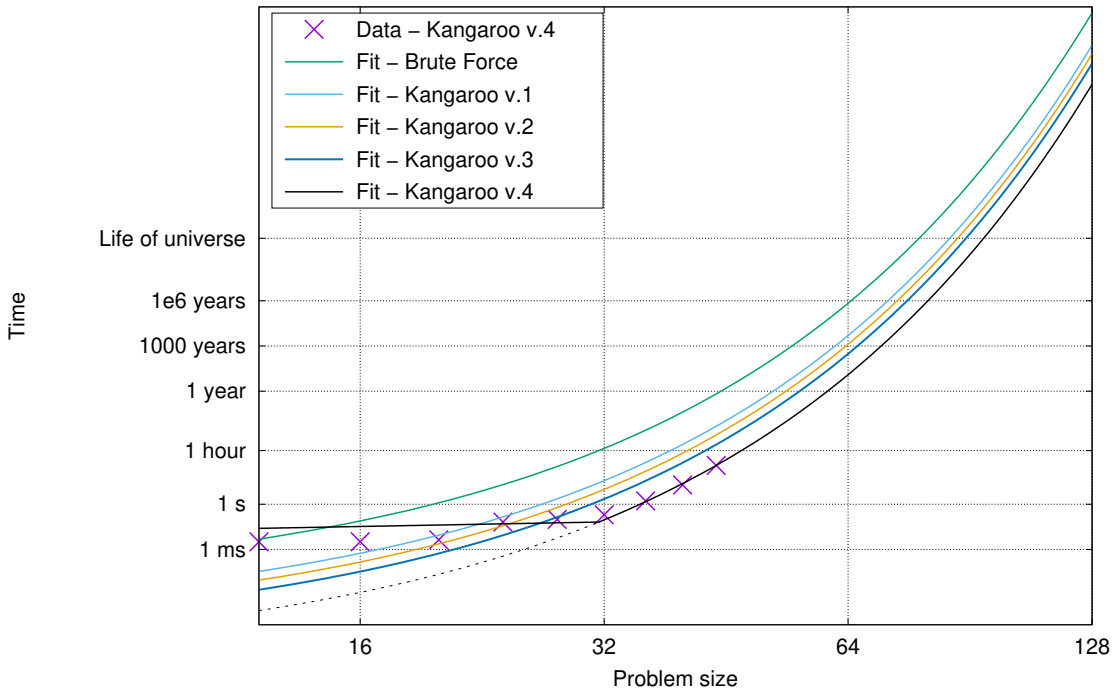


Figure 6.6: Time required to solve the discrete logarithm by the Kangaroo v.4 algorithm. [$\mu = 5$]

Finally, it is interesting to analyze the temporal performance of this version with respect to the previous ones. Comparing the values with $\mu = 4$ as they are the ones we also used before, we can conclude that the acceleration obtained by parallelizing the algorithm is somewhat low, obtaining results around 3.35 times faster despite using seven cores for the search of the solution. In chapter 7 we discuss the reason behind this low speedup.

However, comparing the times with both improvements (parallel search and switching to $\mu = 5$), we get a performance increase of around 24 times. This significant improvement allowed us to solve the 44-bit test-set for the first time, in 6.3 minutes, substantially faster than the 2.5 hours of the third version, the 1.7 days of the first version, or the 241 days of the brute force solution.

6.6 Kangaroo v.5 - Do extreme hyperparameters give extreme results?

This section focuses on analyzing and answering the title question: *Do extreme hyperparameters give extreme results?* Could a crazier combination of hyperparameters yield better results?

The algorithm's behaviour and the results obtained in previous sections motivate the study of these questions and lead us to extend it as if it were a new version. However, it is worth emphasizing that we do not introduce any changes to the algorithm, i.e., we continue with the parallel implementation of the fourth version (see section 6.5).

More specifically, the question in the title arises from the conclusions obtained in version four of the algorithm when we were looking for the ideal values for σ and μ. Figures 6.7 and 6.8 represent the results for 32-bit problems, but the trend is similar for all large problem sizes. Let us remember that σ defines the number of jumps the tame kangaroo makes and, hence, the number of traps it sets. In turn, μ models the function f that somehow represents the quality of the terrain over which the kangaroos jump. The larger the value of μ, the more different types of terrains there are and the larger the size of the set S.

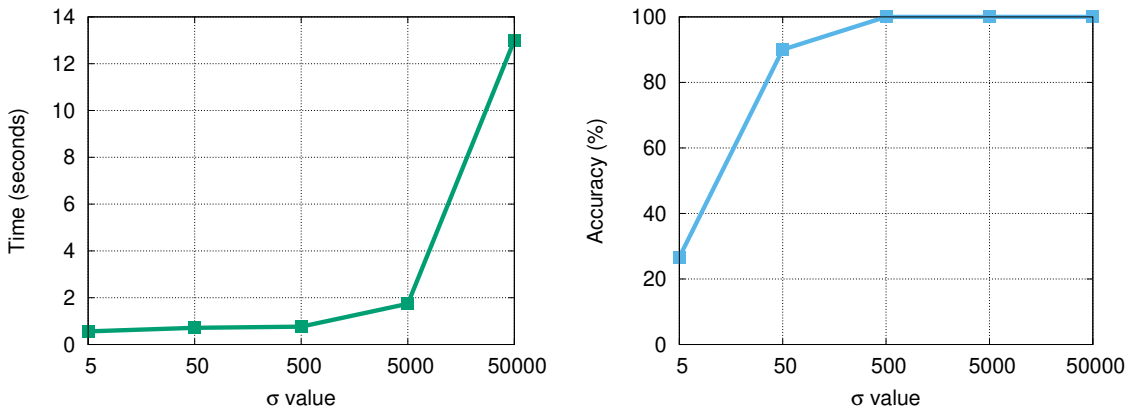


Figure 6.7: Average time required and accuracy to solve 32-bit problems with different σ values. Data extracted from table 6.7.

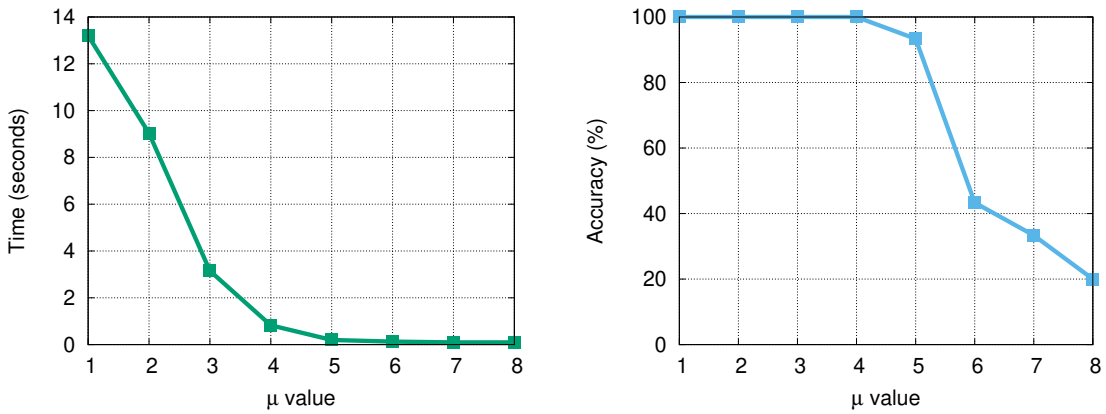


Figure 6.8: Average time required and accuracy to solve 32-bit problems with different μ values. Data extracted from table 6.8.

The results of tables 6.7 and 6.8, and figures 6.7 and 6.8 lead to the following conclusions:

- With **large** σ values, **high accuracy** is achieve, but very **slow times**.
- With **small** μ values, **high accuracy** is achieve, but very **slow times**.
- With **large** μ values, **fast times** are achieve, but too **low accuracy**.

Given these results, we propose a hypothesis: is it possible to bring these two parameters to the limit so that their effects are balanced? In other words: what would happen if we set extremely high values of σ and μ? Is it possible to obtain both advantages of fast times and high accuracy?

Perhaps this theory is not so far-fetched: large values of μ seem to solve the problem rapidly, but at the expense of precision. On the other hand, setting many traps requires wasting time calculating (maybe excessive) tamed kangaroo jumps, but it has been demonstrated that it can increase accuracy. So the question arises: are the adverse effects likely to be counteracted?

To answer this question and test our hypothesis, we have performed a large-scale experimentation as in the previous chapters. For the choice of the configuration values, we have followed these guidelines:

- For the selection of σ, it is crucial to be aware that we are storing in a dictionary each trap (position and distance travelled), and therefore, if there are no collisions in the keys, we will store up to 2σ integers. Given this space constraint, we have chosen the largest value of σ that our RAM limits allowed us: $\sigma = 10\,000\,000$.
- For the choice of μ, we need to consider the constraint $\mu \ll \sqrt{b-a}$ [1]. In [3], Pollard confirms that "powers of two are a good choice", although he "does not claim they are the best choice". Since our best value was $\mu = 5$, we decided to opt for a power of this value and conduct the experiment with $\mu = 25$.

  The value of $b - a$ is equivalent to the section size where we apply the kangaroo algorithm. This size is delimited by λ, so it has at least 2^{16} elements and at most 2^{26} elements. Thus, its square roots are $\sqrt{2^{16}} = 2^8 = 256$ and $\sqrt{2^{26}} = 2^{13} = 8192$. Therefore, the value of $\mu = 25$ satisfies the constraint since $25 \ll 256 \ll 8192$.
- For λ, we have decided to continue with the strategy of the *Dynamically sized sections* due to the good results obtained. Consequently, we set $\lambda = \lambda^*$.

In conclusion, we have performed a complete experimentation, with the parallel algorithm of version 4, and with the following configuration values: $\sigma = 10000000$, $\mu = 25$, and $\lambda = \lambda^*$. The results can be seen in table 6.11 and in figure 6.9.

Problem size	12	16	20	24	28	32	36
Time	-	-	10.7s	11.2s	10.8s	7.84s	11.9s
% Accuracy	killed	killed	100.00	100.00	100.00	100.00	100.00

Problem size	40	44	48	52	56	60	64
Time	9.23s	11.9s	26.0s	2.2min	28.1min	5.3h	3.6d
% Accuracy	100.00	100.00	100.00	100.00	100.00	100.00	Estimated

Table 6.11: Execution time and accuracy of the Kangaroo-v.5 algorithm.

As can be seen, the panorama has changed significantly compared to the results of the fourth version of the algorithm. On the one hand, we have successfully solved much larger test-sets. For example, on average, the 60-bit problems have been solved in slightly more than five hours. As a reminder, the brute force algorithm required 43 250 years and the fourth version required 285 days.

As mentioned above, it is worth noting that the implementations of this algorithm and version four are exactly the same. We have only changed the configuration parameters. A plausible assumption to explain this behaviour is that by increasing the value of σ so significantly, we have caused a *high density of traps* in each section. This modification, however, does not seem to introduce any major temporal drawbacks, although it does increase memory consumption by storing such a large number of traps.

On the other hand, we see that "small" problems now require much more time to be solved than in previous versions, around 10 seconds instead of fractions of a second. This extra time is mainly due to the overhead caused by such a large number of traps in problems where the sections are small (basically, there are more traps than land). It is more noticeable in the case of 12 and 16 bits, where the test-set execution did not finish.

After examining the traces of these two cases, we discovered that one problem of each size took so long (hours) that the experiment was stalled and had to be cancelled. Nevertheless, some 12-bit and 16-bit problems were solved in similar times as the others, ranging between 4 and 10 seconds on average and with 100% accuracy.

Altogether, the results are encouraging compared to previous configurations, with a speedup over the fourth version of more than a thousand in some cases.

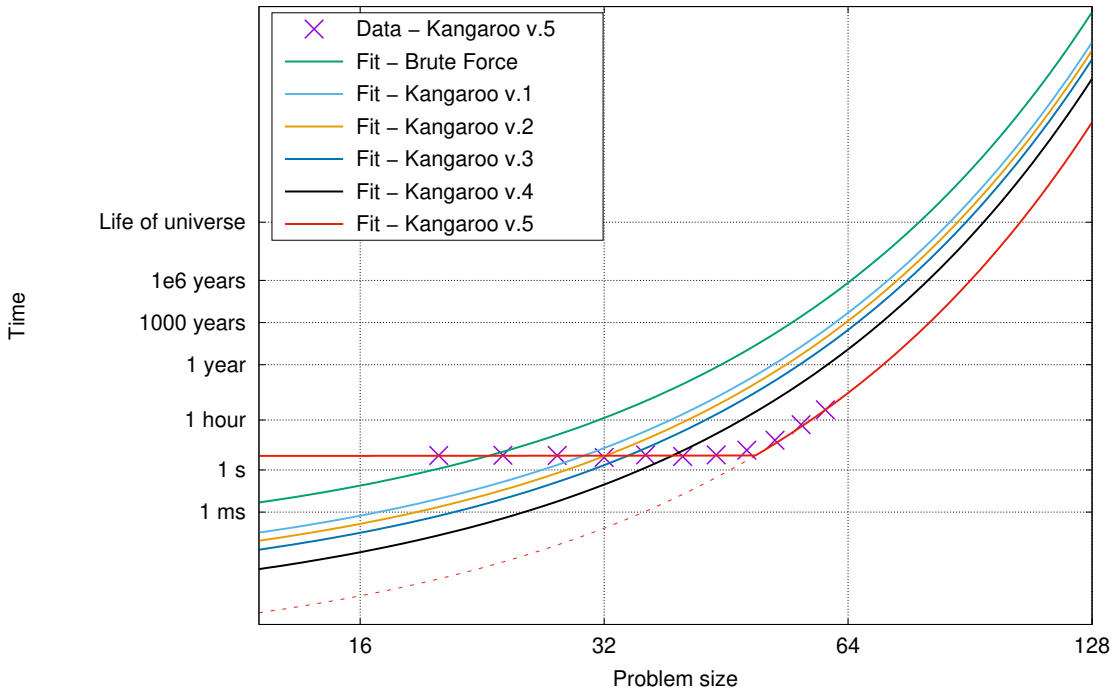


Figure 6.9: Time required to solve the discrete logarithm by the Kangaroo v.5 algorithm.

CHAPTER 7
Results and discussion

This chapter collects and discusses the different results obtained during the experimentation. We present the several modifications made and the motivation that led us to implement each one of them. We also analyse the different options that have emerged after each version was developed and the evolution of the algorithm's capacity to solve the discrete logarithm problem when facing progressively larger cryptographic keys.

First, as discussed in detail in section 6.1, we have implemented a brute-force algorithm. While this algorithm has nothing to do with the kangaroo method, it had a clear objective: to indicate which was the upper bound of the complexity of the problem. In other words, it allowed us to know how long it would take to find the solution "manually", without using any heuristics or sophisticated algorithms.

In this regard, it accomplished its objectives perfectly. As seen in table 6.1, this exhaustive search is feasible for solving problems up to 24 bits. However, the execution time skyrockets if the key size is much larger. A reliable indicator that has been used throughout the project to determine the actual algorithm capabilities is the maximum key size that can be solved in less than 36 hours. In the case of the brute-force implementation, this size was 36 bits.

Once we had established the upper time bound of the problem, we were ready to implement *Pollard's Kangaroo Algorithm*. As discussed in section 6.2, for the first version, we tried to implement an algorithm as clean as possible, based on Pollard's description of the method in [1].

However, this publication has no pseudocode to solve the problem, so the implemented code follows our interpretation of the kangaroo method. Thus, at the time of implementation, we found the need to define a pair of hyperparameters that define the essential configuration variables of the algorithm:

- σ: represents the number of jumps and traps set by the tame kangaroo.
- μ: models the way the kangaroos jump and the number of different terrains there are.

Once verified that the algorithm works correctly, we proceed to evaluate its capabilities to solve discrete logarithms. As can be seen in table 6.3, with this new version, the algorithm is able to decrypt keys of up to 42 bits. Therefore, we can determine that *Pollard's Kangaroo Algorithm* does indeed reduce the complexity of the problem as expected, solving it about a hundred times faster.

Once Kangaroo v.1 was finished, we wondered what improvements we could implement to extend the algorithm's capabilities. The natural path to follow was to try to exploit one of this method's main advantages: defining some a and b boundaries to avoid

searching through the whole space of solutions. Nevertheless, the chaotic behaviour introduced by the modulus function prevents us from having clues about where the solution might fall.

The hypothesis proposed was the following: since there is no way of knowing where the solution can be found in the interval, we will search everywhere. However, instead of doing it all at once, we can divide the search space into sections and search sequentially in each of them. If we find it, we can stop the execution immediately without needing to continue searching through the rest. It is reasonable to estimate that, on average, the algorithm will need to search through about half of the sections before finding the solution. The question is whether this search is faster than searching across the entire interval in a single pass.

This approach has another fundamental advantage: the jumps of the wild kangaroo are always the same regardless of the section. Thus, we could implement the so-called *Wild Jumps Database* to avoid unnecessarily recomputing the wild kangaroo jumps. With both techniques applied, the algorithm works almost three times faster, being able to decrypt 44-bit keys in less than a day and a half.

The next logical point to study was the size of the sections. To test the hypothesis of the second version, it was decided to use an arbitrary value. However, we did not perform any study to check whether this value was optimal or not. In the third version of the algorithm, we studied this feature, which we set with the hyperparameter λ.

The study's conclusions were unambiguous: the optimal value increases as the problem size increases. However, this size cannot grow enormously as it leads to high RAM consumption. Given this drawback, we decided to use a piecewise-defined function that considers this constraint. With these modifications, the execution times were reduced by a third, and the algorithm increased its capabilities to solve the discrete logarithm, being able to decrypt 46-bit keys in 10 hours.

Once all these modifications had been made, there were few heuristic options left to increase the algorithm's capabilities further. Therefore, we employed the technique of parallelization. At first glance, our method seemed to benefit directly from parallelization's advantages, so we attempted to exploit the CPU's full computational power and implement the section search in parallel.

The results were clear: parallelization helps solve the problem, but the achievable speedup is not linear but somewhat lower. Around three times faster than the sequential version using seven workers and one coordinator. To explain this phenomenon, we rely on the state of the art. In [18], Paul C. van Oorschot and Michael J. Wiener analyse parallelization in algorithms based on collision search, including *Pollard's Kangaroo Algorithm*. One of the conclusions they present is:

> “ Obvious methods of parallelizing collision search algorithms do not give linear speedup; when m processors are used, collision search is only $\sqrt{m}$ times faster than when one processor is used. The new method for parallelizing Pollard's rho-method presented herein, based on the use of distinguished points, gives linear speedup.
>
> *Paul C. van Oorschot and Michael J. Wiener [18]* ”

Although they mention the *Pollard rho* algorithm, the conclusion is also applicable to our algorithm. Therefore, parallelization results are correct and the expected ones given that no techniques such as the *distinguished points* method have been applied.

In addition, at this stage of the experiment, we also decided to re-evaluate the optimal values of the basic hyperparameters. Surprisingly, while σ did not change, μ did. Consequently, we decided from that moment to use the new optimum. Thus, with both changes (parallelization and new μ value), the algorithm increased its capabilities again, vulnerating keys of up to 52 bits in less than 36 hours.

Finally, the behaviour of the algorithm and the results obtained motivated the study of whether increasing the hyperparameters λ and μ could increase the algorithm's overall performance. In the fifth version, we analyse this hypothesis and perform an experiment to verify if it works.

It was surprising to find that the *high density of traps* caused by this hypothesis does not reduce the algorithm's overall performance. Although the results shown in table 6.11 discourage its usage in problems with few bits, it works remarkably well in large problems. In this way, the algorithm's capacity grew significantly, becoming capable of solving 62-bit keys in less than a day. The complete evolution of the algorithm's ability to solve the discrete logarithm problem can be seen in figure 7.1.

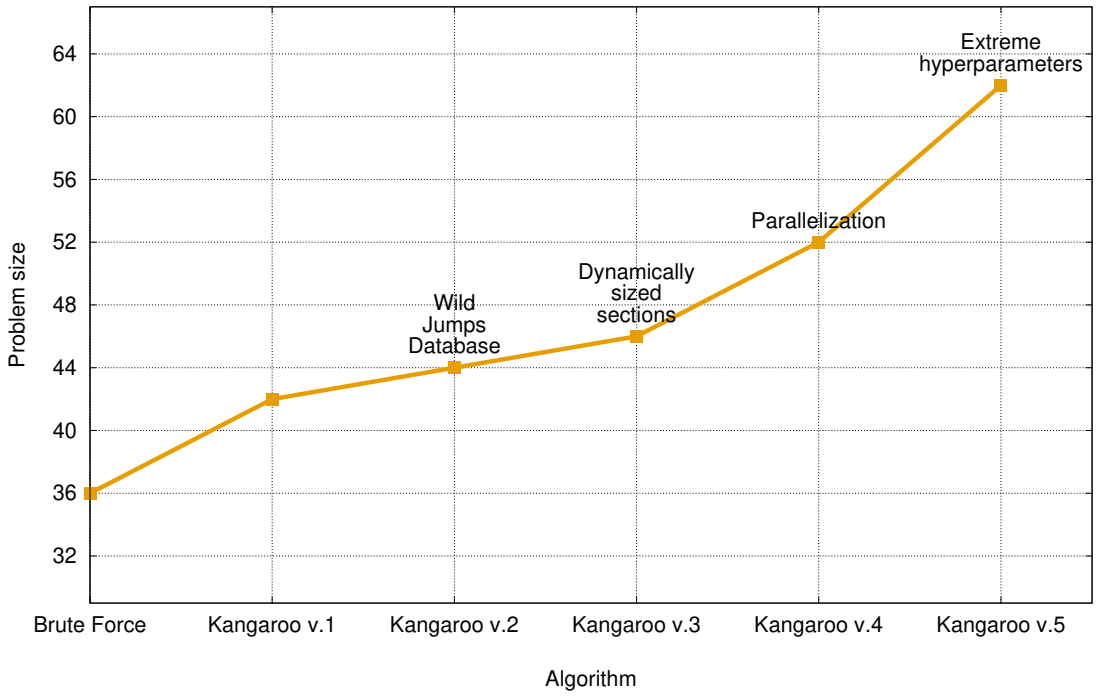


Figure 7.1: Largest problem solved in less than a day and a half with the different algorithms implemented and technique that has allowed such improvement.

CHAPTER 8
Conclusions

The discrete logarithm problem is one of the fundamental problems in cryptography today. Analyzing new methods to solve it implies analyzing new ways of carrying out cryptographic attacks on the protocols that implement it. In our case, we have studied *Pollard's Kangaroo Algorithm* for this purpose.

The study of already developed approaches is more logical than it may seem at first. As technology evolves, the computational power of the computers we have at our disposal improves, and the cost of hardware decreases. Thus, attacks that were previously economically unattainable can become efficient and viable after a technological breakthrough.

A great example of this in the world of cryptography is the DES encryption system. It was developed in the 1970s and was, for many years, the standardized symmetric encryption protocol. Its key-space is 56 bits, huge for its time but conspicuously insufficient today. In 1977, Whitfield Diffie and Martin Hellman estimated that an exhaustive key-finding machine could be built for about $20,000,000. In 1993, Michael Wiener proposed a new machine thanks to technological advances at the cost of approximately $1,000,000. In 1998, the *Deep Crack* machine to crack DES was built for less than $250,000 [20]. It is important to emphasize that it was only possible to build *Deep Crack* at a relatively low price because the hardware had become cheaper. In the 1980s, it would have been impossible to build it without spending millions of dollars [20]. In 2009, an article was published explaining how to build a DES cracking machine in less than nine days for €8,980 [21].

Throughout this project, we have presented different versions of *Pollard's Kangaroo Algorithm*, adapted to current technological improvements. Each variant presents specific enhancements and modifications to achieve progress in its temporal performance. Although we are dealing with an exponential algorithm, as can be seen, there is a wide margin for improvement. For each variant, we have analyzed how it behaved in a cryptographic attack with different key sizes and compared the results to prior versions.

With these improvements implemented, we have tried to make our own contribution to the different ways of configuring the algorithm that exists [3, 18, 22]. The *Wild Jumps Database* and the *Dynamically sized sections* have been shown to provide a satisfactory overall speedup to the algorithm.

In general, we have achieved attractive temporary reductions in each version designed of the algorithm, finally reaching notable differences in the time that the same cryptographic attack requires to complete successfully. The time gap between the first and fifth versions is remarkable, obtaining results more than a hundred thousand times faster in some cases. Figure 8.1 shows the time required by each version of the algorithm to perform a cryptographic attack with keys of 32, 44, and 60 bits, sizes that are pretty representative of the global results obtained.

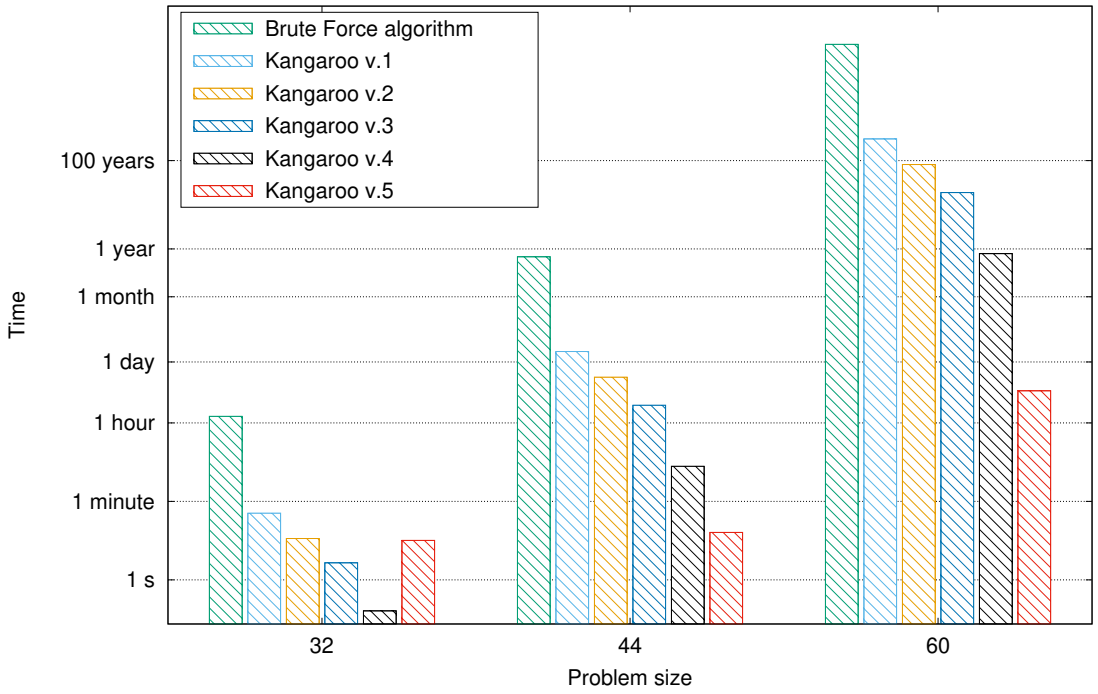


Figure 8.1: Time required to solve some discrete logarithm problems for each version developed.

It is worth noting the outstanding results of the latest version. Although we did not anticipate that using extreme values in the hyperparameters that model the algorithm's behaviour would lead to significant positive effects, the experimentation carried out has shown successful results, being able to solve 60-bit problems in a reasonable time.

When proposing this work of analysis and experimentation with keys of different sizes, we did not expect to attack keys of such a large size successfully. Based on the asymptotic estimates, we expected to solve problems of at most around 48 bits. Therefore, we can affirm that we have obtained satisfactory results with the successive versions presented.

To get an idea of the exorbitant size of the numbers we are working with, we can take as a reference the age of the universe. According to the Big Bang theory, the universe was created approximately 14 billion years ago. If we count it in seconds, the universe is about 4.41e+17 seconds old. This number is *only* 59 bits long. An example of a 60-bit problem can be seen in figure 8.2.

$$\alpha^k \mod p = \beta \qquad \begin{cases} \alpha = 1108981920138235052 \\ \beta = 167535827043038444 \\ p = 939623971882007407 \end{cases}$$

$$\Downarrow$$

$$1108981920138235052^k \mod 939623971882007407 = 167535827043038444$$

$$\Downarrow$$

$$k = 108235977948900510 \qquad \texttt{(Solved by Kangaroo v.5 after 3 hours)}$$

Figure 8.2: 60 bits discrete logarithm problem.

Regarding the objectives established in section 1.2, we present the following conclusions:

- The algorithms have been successfully implemented.
- The accuracy of the algorithm is generally 100%. However, if the algorithm cannot find the solution for a particular problem, increasing the value of σ can help to increase the accuracy.
- The maximum problem size that the algorithm can solve in a reasonable time is 62 bits, with an estimated time of 21 hours.
- As we have indicated before, our main contribution is the *Wild Jumps Database* and the *Dynamically sized sections*.
- The best combination of hyperparameters has been $\sigma = 10000000, \mu = 25$, and $\lambda = \lambda^*$ for problems of 40 bits or higher. For minor problems, $\sigma = 500, \mu = 5$, and $\lambda = \lambda^*$ has been shown to give the best results.

Finally, it is interesting to analyze these results from the current cryptographic security perspective. As expected, despite the elevated speedups we have achieved in the latest implemented versions, real-world problems are considerably larger than 60 bits and are far from being resolved in reasonable times. Note that this is a positive conclusion from a cryptographic security point of view. Therefore, we can conclude that the current protocols are still secure and are unlikely to be attacked with this algorithm, at least using a simple personal computer.

CHAPTER 9
Future work

Once the results have been commented on, we believe that it is interesting to analyze the possible points for improvement and the issues that have remained pending in this work, which could be of interest to deepen and further study in future versions.

First, a possible improvement that could improve the speed of solving problems is to use the *distinguished points* technique. We are basing our design on Pollard's original [1] design, storing the entire set of jumps that the tame kangaroo performs. Different later works [18, 19] use this technique to avoid storing all the traps.

The idea behind the *distinguished points* technique is to look for a collision between the kangaroos, but not between all their paths, but only between a small subset of positions that satisfy a particular property. The downside of this method is that it makes the implementation of the algorithm even more complex, and we did not believe that it was worthwhile for the analysis we wanted to perform. However, it is certainly a good point to continue this work.

On the other hand, the literature discusses a wide variety of techniques for dealing with kangaroos. We have only used the services of one wild and one tamed kangaroo, emphasizing the creation of our *Wild Jumps Database*. However, other authors [23, 22] propose using a different number of kangaroos and refer to techniques such as the *"Four kangaroo method"*. It would be interesting to see if these methods can be combined with our improvements to achieve even higher speedups.

Finally, another point where further analysis can be carried out is in the parallelization of the algorithm. Our parallelization technique has tried to be as simple and non-intrusive as possible, using the Python3 `multiprocessing` package. Our results reveal a speedup of approximately $\sqrt{p}$, where p is the number of processors used, rather than a speedup of p as achieved by other authors such as [19].

Although, as discussed in chapter 7, Oorschot and Wiener [18] seem to attribute this linear acceleration to the use of the *distinguished points* technique, it is still interesting to try to parallelize this algorithm with more professional tools such as OpenMP or MPI. This type of utility would allow the kangaroo method to be run on parallel computing clusters where more significant problems could be solved in reasonable times if such parallelization is satisfactory.

In conclusion, we have conducted exhaustive experimentation on *Pollard's Kangaroo Algorithm* throughout this work. However, as usual in research, there are always more techniques and methods that can be explored to try to achieve even better results.

Bibliography

[1] J. M. Pollard. Monte Carlo Methods for Index Computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.

[2] Ravi Montenegro. Kruskal Count and Kangaroo Method. https://faculty.uml.edu/rmontenegro/research/kruskal_count/index.html, 2009. Last visited on 4/5/2022.

[3] J. M. Pollard. Kangaroos, Monopoly and Discrete Logarithms. *Journal of Cryptology*, 13:437–447, 01 2000.

[4] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[5] Jennifer Santamaría Fernández and Daniel Sadornil Renedo. El logaritmo discreto y sus aplicaciones en Criptografía, 2012-2013.

[6] Douglas R Stinson. Cryptography: Theory and Practice, Third Edition (Discrete Mathematics and Its Applications). *CRC Press*, 2005.

[7] Shi Bai and Richard P. Brent. On the Efficiency of Pollard's Rho Method for Discrete Logarithms. In *Proceedings of the Fourteenth Symposium on Computing: The Australasian Theory - Volume 77*, CATS '08, pages 125–131, AUS, 2008. Australian Computer Society, Inc.

[8] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[9] WhatsApp & Open Whisper Systems. *WhatsApp Encryption Overview*. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. Last visited on 18/5/2022.

[10] Telegram. *MTProto Mobile Protocol*. https://core.telegram.org/mtproto/security_guidelines#diffie-hellman-key-exchange. Last visited on 22/5/2022.

[11] Signal. *Signal Technical information*. https://signal.org/docs/. Last visited on 22/5/2022.

[12] Josh Blum, Simon Booth, Brian Chen, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Jack O'Connor, Surya Rien, Miles Steele, Matthew Green, Lea Kissner, and Alex Stamos. *E2E Encryption for Zoom Meetings*, 2021.

[13] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January 2006.

[14] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

[15] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, August 2011.

[16] Sheila Frankel and Suresh Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071, February 2011.

[17] Taher El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, pages 10–18, Berlin, Heidelberg, 1985. Springer-Verlag.

[18] Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptol.*, 12(1):1–28, 1999.

[19] Edlyn Teske. Computing discrete logarithms with the parallelized kangaroo method. *Discrete Applied Mathematics*, 130(1):61–82, 2003. The 2000 Com2MaC Workshop on Cryptography.

[20] Christof Paar and Jan Pelzl. *Understanding Cryptography - A Textbook for Students and Practitioners.* Springer, 2010.

[21] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, Andy Rupp, and Manfred Schimmler. How to Break DES for € 8,980, August 2009.

[22] Alex Fowler and Steven Galbraith. Kangaroo Methods for Solving the Interval Discrete Logarithm Problem, 2015.

[23] Steven Galbraith, John Pollard, and Raminder Ruprai. Computing Discrete Logarithms in an Interval. *IACR Cryptology ePrint Archive*, 2010:617, 01 2010.

[24] Todd Rowland and Eric W. Weisstein. "Group." From *MathWorld* – A Wolfram Web Resource. https://mathworld.wolfram.com/Group.html. Last visited on 11/3/2022.

[25] Eric W. Weisstein. "Cyclic Group." From *MathWorld* – A Wolfram Web Resource. https://mathworld.wolfram.com/CyclicGroup.html. Last visited on 11/3/2022.

[26] Eric W. Weisstein. "Totient Function." From *MathWorld* – A Wolfram Web Resource. https://mathworld.wolfram.com/TotientFunction.html. Last visited on 11/3/2022.

[27] Eric W. Weisstein. "Primitive Root." From *MathWorld* – A Wolfram Web Resource. https://mathworld.wolfram.com/PrimitiveRoot.html. Last visited on 11/3/2022.

[28] Margherita. Barile. "Multiplicative Group." From *MathWorld* – A Wolfram Web Resource, created by Eric W. Weisstein. https://mathworld.wolfram.com/MultiplicativeGroup.html. Last visited on 11/3/2022.

[29] Eric W. Weisstein. "Modulo Multiplication Group." From *MathWorld* – A Wolfram Web Resource. https://mathworld.wolfram.com/ModuloMultiplicationGroup.html. Last visited on 11/3/2022.

[30] Eric W. Weisstein. "Discrete Logarithm." From *MathWorld* – A Wolfram Web Resource. https://mathworld.wolfram.com/DiscreteLogarithm.html. Last visited on 12/3/2022.

[31] Yutaka Nishiyama. The Kruskal principle. *International Journal of Pure and Applied Mathematics*, 85:983–992, 07 2013.

[32] Jeffrey C. Lagarias, Eric Rains, and Robert J. Vanderbei. The Kruskal Count, 2001.

[33] Ravi Montenegro and Prasad Tetali. How long does it take to catch a wild kangaroo?, 2010.

[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*, chapter 31. The MIT Press, 3rd edition, 2009.

[35] David Casacuberta. 70 años de los Derechos Humanos: Debemos garantizar el derecho a la privacidad. https://ethic.es/especiales/debemos-garantizar-el-derecho-a-la-privacidad/. Last visited on 25/5/2022.

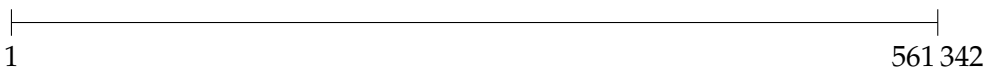
APPENDIX A
Tracing the parallel algorithm

In this appendix, the process of initialisation and search for the solution performed in the parallel versions of the algorithm is represented in a user-friendly and visual way.

In this case, we are going to solve a 20-bit problem although, for the sake of simplification, the hyperparameters are slightly smaller than they would actually be if we execute the algorithm:

$$\alpha^k \mod p = \beta \qquad \text{where} \quad \begin{cases} \alpha = 1\,015\,436 \\ p = 561\,343 \\ \beta = 410\,105 \end{cases} \qquad \text{and} \quad \begin{cases} \sigma = 100 \\ \mu = 4 \\ \lambda = 14 \end{cases}$$

Thus, the problem could be represented as follows: the solution k lies somewhere on the number line defined by the generator's order, i.e., between one and $p - 1$. The security of the discrete logarithm is based on the fact that we do not have any kind of information to orient our search. The modulo function brings chaotic behaviour to exponentiation. Therefore, the solution k can fall anywhere on the number line.

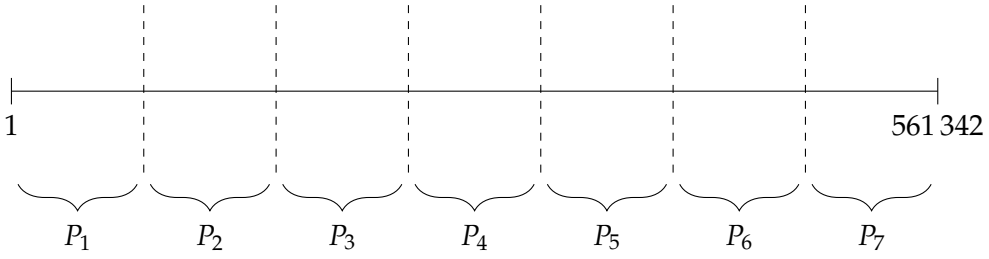


For this problem, we could directly apply the first version of *Pollard's Kangaroo Algorithm* developed, with $a = 1$ and $b = 561\,342$. However, we would be exploiting neither the acceleration of the algorithm design with a *Wild Jumps Database* nor the full computational power of a multi-core machine.

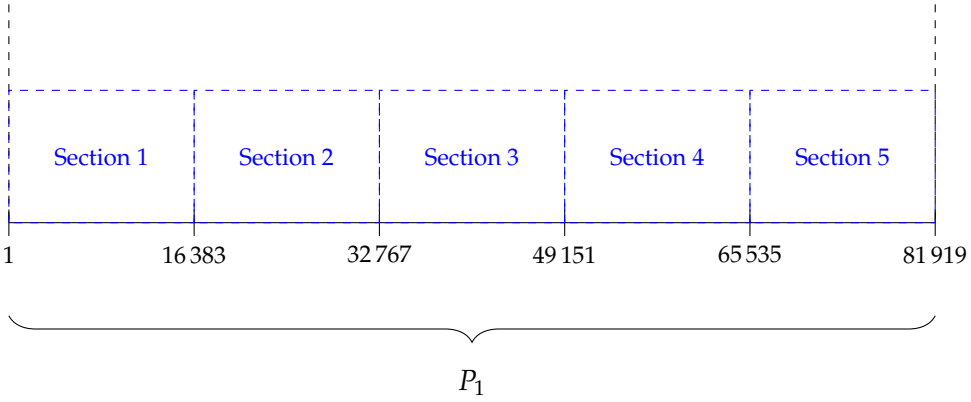
Therefore, let us look at the steps that would be performed by the fourth and fifth version of the algorithm, which employs a coordinator-worker design to parallelise the sequential algorithm with the *Wild Jumps Database*.

For this example, let us assume that the machine has four physical cores, providing eight execution threads ($th = 8$). One of them is designated as the coordinator, and the rest will be workers that will apply the kangaroo method.

Thus, the first thing the coordinator does is divide the number line into $th - 1$ equal intervals and communicate the splitting to each sub-process:

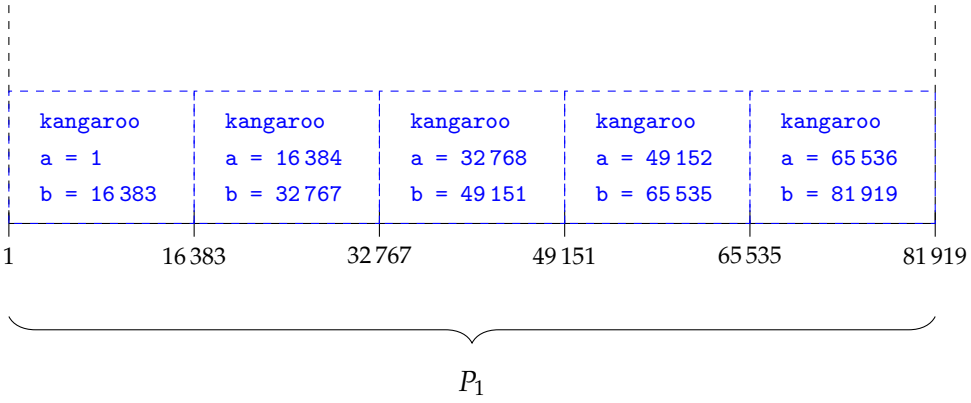


In turn, each thread, when it receives its interval, divides it into sections, as many as necessary, so that each section has a size of approximately 2^λ elements:



As can be seen, in this example, the interval `[1, 81 919]` is assigned to the first sub-process, which will divide it into 5 sections of $2^\lambda = 2^{14} = 16\,384$ elements.

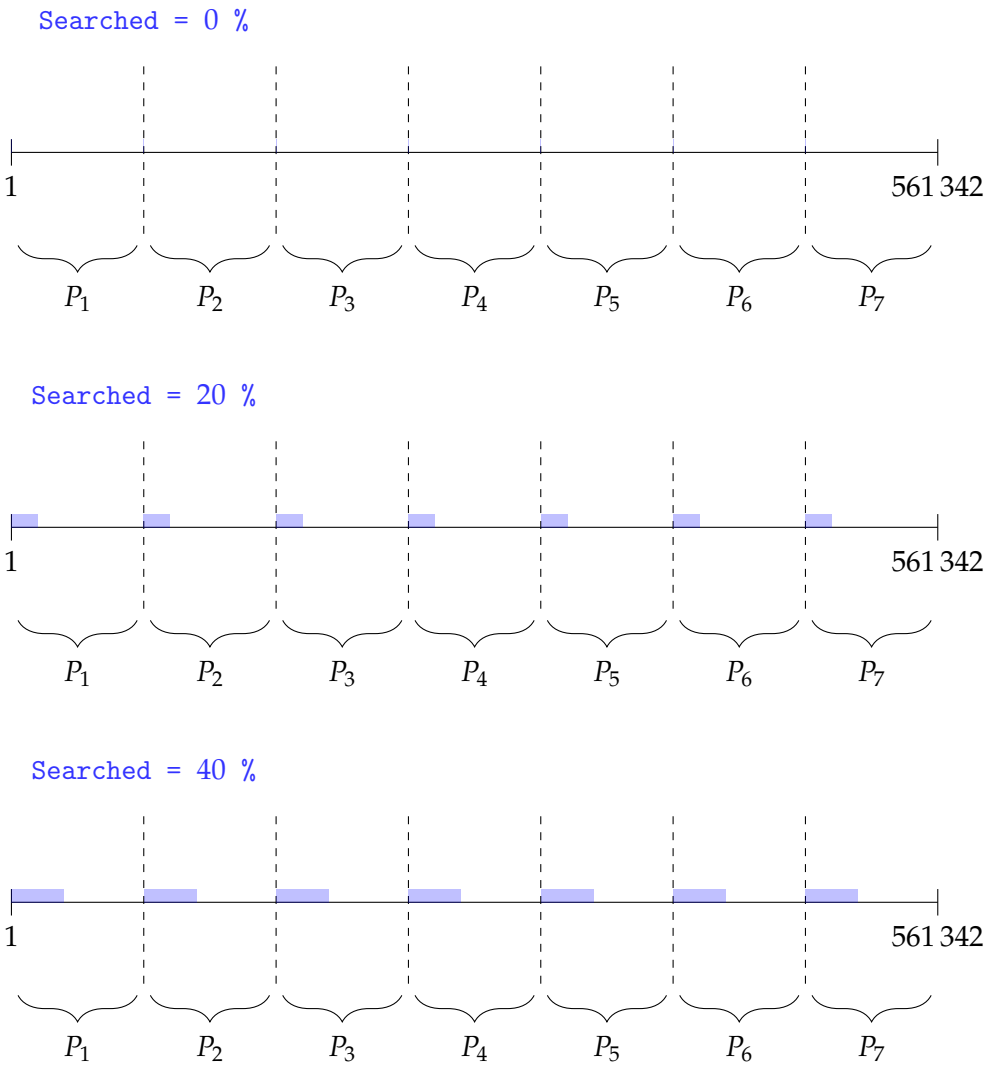
These sections will be the minimum search unit with which we are going to operate when solving the discrete logarithm. This means that the algorithm with the *Wild Jumps Database* will be applied directly to each section. For example, section 1 will be executed with $a = 1$ and $b = 16383$.



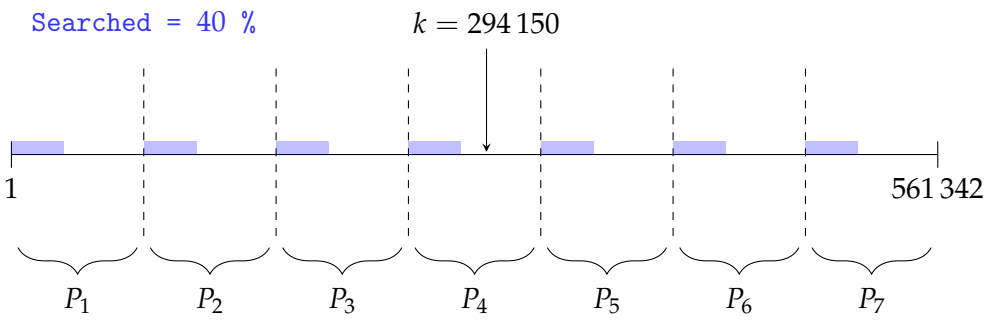
In fact, when implementing this strategy, instead of indicating each interval and then having the worker divide it into sections, the coordinator directly divides the entire real line into sections of 2^λ elements and simply tells each process which is its first section and how many sections correspond to it. It is equivalent but avoids rounding problems since in the worst case the last processes will simply have some sections less than the others.

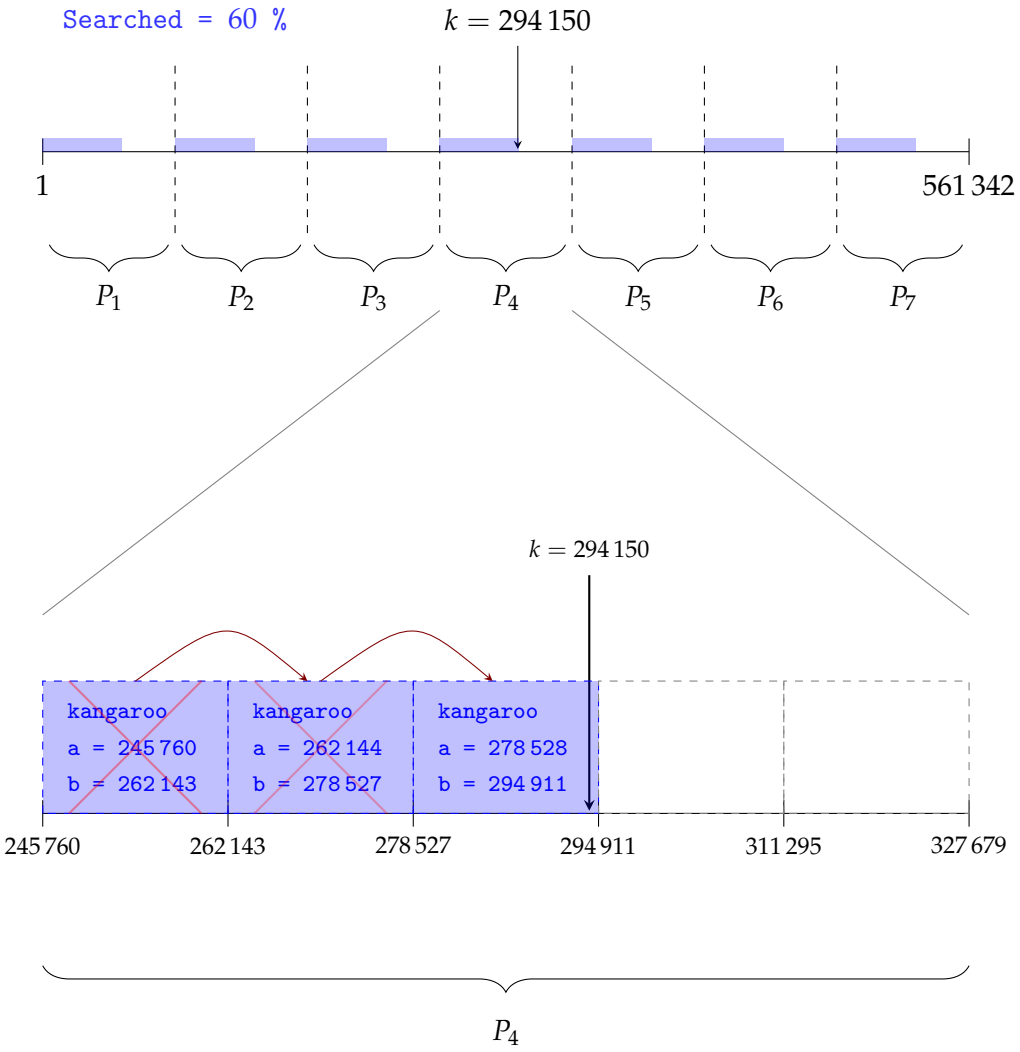
Once the distribution of intervals and sections is decided, the coordinator lets each worker do its tasks and waits to receive the results, as can be seen in algorithms 6.4 and 6.5.

This way, all workers will apply the algorithm in parallel, starting to search in its respective first sections. The following pictures illustrate this process:



Drawing the solution to this particular example, we see that it should be found by the fourth thread, specifically by searching in its third section.





This is indeed the case. Once k is found, it is sent to the coordinator to notify the rest of the processes that they must stop the search. Then, k is returned as the solution:

$$\alpha^k \mod p = \beta \qquad \text{where} \quad \begin{cases} \alpha = 1\,015\,436 \\ p = 561\,343 \\ \beta = 410\,105 \end{cases} \implies k = 294\,150$$

$$1\,015\,436^{294\,150} \mod 561\,343 = 410\,105$$

APPENDIX B
Sustainable Development Goals

B.1 Degree of relationship between the work and the United Nations Sustainable Development Goals

Sustainable Development Goals (SDGs)	High	Medium	Low	Not applicable
SDG 1. **No Poverty.**				✓
SDG 2. **Zero Hunger.**				✓
SDG 3. **Good Health and Well-Being.**				✓
SDG 4. **Quality Education.**	✓			
SDG 5. **Gender Equality.**				✓
SDG 6. **Clean Water and Sanitation.**				✓
SDG 7. **Affordable and Clean Energy.**				✓
SDG 8. **Decent Work and Economic Growth.**	✓			
SDG 9. **Industry, Innovation, and Infrastructure.**	✓			
SDG 10. **Reduced Inequalities.**				✓
SDG 11. **Sustainable Cities and Communities.**		✓		
SDG 12. **Responsible Consumption and Production.**				✓
SDG 13. **Climate Action.**				✓
SDG 14. **Life Below Water.**				✓
SDG 15. **Life on Land.**				✓
SDG 16. **Peace, Justice and Strong Institutions.**		✓		
SDG 17. **Partnerships for the Goals.**				✓

B.2 Discussion on the relationship between the work and the United Nations Sustainable Development Goals

This work fits perfectly into the field of cryptography. In this sense, it is clear that most of the United Nations Sustainable Development Goals related to poverty, inequalities, clean water, or land and underwater ecosystems, among others, are far from the scope of this work, and it is not appropriate to relate them to it. However, there are certain objectives that can be related to the work developed, although possibly on a smaller scale than the United Nations' desired goals.

To begin with, we can relate this paper to SDG number nine: *Industry, Innovation, and Infrastructure*. Information and communication technologies in general, and modern cryptography in particular, have always played a vital role in industry and innovation, since their inception in the 1960s and 1970s.

Throughout this work, we have presented a method and different improvements and optimisations to solve the discrete logarithm problem. As discussed in chapter 4 regarding its practical applications, this problem is vital in today's public-key cryptography and is used as a basis for designing secure cryptographic protocols.

In this sense, public-key cryptographic systems represent an essential qualitative improvement in the innovation and development of industry and commerce, bringing this science into fields that are essential today, such as digital signatures and identification systems. COVID-19 pandemic has highlighted the necessity of being able to identify ourselves or sign documents securely and unequivocally, without the requirement of being physically present.

From this perspective, we can also relate the work developed to Sustainable Development Goals 8 and 11: *Decent Work and Economic Growth* and *Sustainable Cities and Communities*. Digital signatures have clearly become one of the most widely used tools in the automation of processes and in the digitisation of companies, institutions, governments, hospitals, etc. In an increasingly urbanised world, where cities and metropolitan areas are the main economic growth centres, such identification forms have become essential. Some of the advantages we can relate to those SDGs are:

- They imply more speed in verifying information.
- They improve security in the exchange of sensitive information by allowing trust in the sender.
- They help to avoid unnecessary travel.
- They can facilitate further cost savings for companies.
- They allow for a better quality of services provided by companies.

In general, such systems provide not only reliability and security but also help with the economic growth of companies. They allow us to progress towards a more sustainable and environmentally friendly bureaucracy, reducing superfluous travel and excessive use of paper.

On the other hand, a SDG closely related to the primary motivation of this project is number four: *Quality Education*. One of the best ways to evaluate the security of a cryptographic system is to make the details of the system generally open to everyone. Making this information public facilitates someone discovering a weakness and notifying the cryptographic community. The more information is made public, the easier it is to find vulnerabilities, thereby allowing protocols to be improved.

Similarly, the more imaginative ways of attacking the mathematical problems on which cryptographic protocols rely are proposed by the scientific world, the better prepared society in general and students of cryptography in particular will be in the future. In this work, we have tried to offer our contribution to this end, analysing new efficient ways of implementing *Pollard's Kangaroo Algorithm*.

As mentioned in chapters 1 and 8, analysing algorithms designed years ago is still interesting from an educational and analytical point of view. The main reason for this is that, as time goes by, technology advances, and algorithms that were not applicable before, become applicable thanks to improvements in hardware and computational power.

Finally, we must also consider the so-called *"right to encrypt"* and its intimate relationship with SDG 16: *Peace, Justice and Strong Institutions*. Historically, the ability to encrypt a message to ensure its inviolability has been a privilege of states. Encryption and decryption systems were generally not accessible to the public. On the other hand, citizens had not felt any need to encrypt their communications either. With the development of ICTs, this started to change as a large part of our lives has become online. Thus, the need to use cryptographic methods to safeguard our privacy appeared.

Understanding encryption as a good concept, not as a form of secrecy, is essential. Cryptography is not a resource for criminals to prevent their conspiracies from being discovered. Cryptography is the science that allows us to ensure that our digital communications are secure and, therefore, that the integrity of our personal data and our privacy are being respected.

In this sense, the relationship between this work and SDG 16 can be appreciated. Ensuring that the mathematical foundations of cryptographic protocols remain solid despite the evolution of technology is crucial. Thus, we can be confident that no institution or government can access our encrypted data. Guaranteeing the right to privacy is and must remain a fundamental principle in our society.

This fact is so critical that, for some time now, the United Nations has been urged to modify the Universal Declaration of Human Rights. The proposal is to add to its 30 articles the following:

> "**Article 31**: Everyone has the right to protect his or her communications and data in digital format by using cryptographic methods. Such cryptographic methods must be freely accessible and secure, i.e. not containing backdoors or similar mechanisms that allow third parties to access the encrypted content.
>
> *David Casacuberta [35]" ”*

ETS Enginyeria Informàtica
Camí de Vera, s/n, 46022, València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es

In conclusion, we have seen the importance of this type of work and its relationship with different Sustainable Development Goals. Encouraging the study of current cryptographic protocols, reinventing them and checking that they are still secure is essential to maintain awareness of their security. By doing so, we increase their credibility and our confidence in them. This trust is key to the continued prosperity of technology in particular and our society in general.