Entropoid Based Cryptography

Danilo Gligoroski*

April 12, 2021

Abstract

The algebraic structures that are non-commutative and non-associative known as entropic groupoids that satisfy the *"Palintropic"* property i.e., $x^{\mathbf{AB}} = (x^{\mathbf{A}})^{\mathbf{B}} = (x^{\mathbf{B}})^{\mathbf{A}} = x^{\mathbf{BA}}$ were proposed by Etherington in '40s from the 20th century. Those relations are exactly the Diffie-Hellman key exchange protocol relations used with groups. The arithmetic for non-associative power indices known as Logarithmetic was also proposed by Etherington and later developed by others in the 50s-70s. However, as far as we know, no one has ever proposed a succinct notation for exponentially large non-associative power indices that will have the property of fast exponentiation similarly as the fast exponentiation is achieved with ordinary arithmetic via the consecutive rising to the powers of two.

In this paper, we define ringoid algebraic structures $(G, \boxplus, *)$ where $(G, \boxplus)$ is an Abelian group and $(G, *)$ is a non-commutative and non-associative groupoid with an entropic and palintropic subgroupoid which is a quasigroup, and we name those structures as Entropoids. We further define succinct notation for non-associative bracketing patterns and propose algorithms for fast exponentiation with those patterns.

Next, by an analogy with the developed cryptographic theory of discrete logarithm problems, we define several hard problems in Entropoid based cryptography, such as Discrete Entropoid Logarithm Problem (DELP), Computational Entropoid Diffie-Hellman problem (CEDHP), and Decisional Entropoid Diffie-Hellman Problem (DEDHP). We post a conjecture that DEDHP is hard in Sylow q-subquasigroups. Next, we instantiate an entropoid Diffie-Hellman key exchange protocol. Due to the non-commutativity and non-associativity, the entropoid based cryptographic primitives are supposed to be resistant to quantum algorithms. At the same time, due to the proposed succinct notation for the power indices, the communication overhead in the entropoid based Diffie-Hellman key exchange is very low: for 128 bits of security, 64 bytes in total are communicated in both directions, and for 256 bits of security, 128 bytes in total are communicated in both directions.

Our final contribution is in proposing two entropoid based digital signature schemes. The schemes are constructed with the Fiat-Shamir transformation of an identification scheme which security relies on a new hardness assumption: computing roots in finite entropoids is hard. If this assumption withstands the time's test, the first proposed signature scheme has excellent properties: for the classical security levels between 128 and 256 bits, the public and private key sizes are between 32 and 64, and the signature sizes are between 64 and 128 bytes. The second signature scheme reduces the finding of the roots in finite entropoids to computing discrete entropoid logarithms. In our opinion, this is a safer but more conservative design, and it pays the price in doubling the key sizes and the signature sizes.

We give a proof-of-concept implementation in SageMath 9.2 for all proposed algorithms and schemes in an appendix.

Keywords: Post-quantum cryptography, Discrete Logarithm Problem, Diffie-Hellman key exchange, entropic, Entropoid, Entropoid Based Cryptography

*Department of Information Security and Communication Technologies, Norwegian University of Science and Technology - NTNU

Contents

1 Introduction

The arithmetic of non-associative indices (shapes or patterns of bracketing with a binary operation $*$) has been defined as *"Logarithmetic"* by Etherington in the '40s of the 20th century. One of the most interesting properties that have been overlooked by modern cryptography is the Discrete Logarithm problem and the Diffie-Hellman key exchange protocol in the non-commutative and non-associative logarithmetic of power indices. In light of the latest developments in quantum computing, Shor's quantum algorithm that can solve DL problem in polynomial time if the underlying algebraic structures are commutative groups, and the post-quantum cryptography, it seems that there is an opening for a *rediscovery* of Logarithmetic and its applications in Cryptography.

Etherington himself, as well as other authors later ([1], [2]), noticed that the notation of shapes introduced in [3] quickly gets complicated (and from our point of view for using them for cryptographic purposes, incapable of handling exponentially large indices).

Inspired by the work of Etherington, in a series of works in '50s, '60s and '70s of the last century, many authors such as Robinson [4], Popova [5], Evans [6], Minc [7], Bollman [8], Harding [1], Dacey [2], Bunder [9], Trappmann [10], developed axiomatic number systems for non-associative algebras which in many aspects resemble the axiomatics of ordinary number theory. While the results from that development are quite impressive such as the fundamental theorem of non-associative arithmetic (prime factorization of indices [6]), or the analogue of the last Fermat Theorem [5, 6, 7], the construction of the shapes was essentially sequential. In cryptography, we need to operate with power indices of exponential sizes. Thus, we have to define shapes over non-associative and non-commutative groupoids that allow fast exponentiation, similarly as it is done with the consecutive rising to the powers of two in the standard modular arithmetic, while keeping the variety of possible outcomes of the calculations, as the flagship aspect of the non-commutative and non-associative structures.

1.1 Our Contribution

We first define a general class of groupoids $(G, *)$ (sets G with a binary operation $*$) over direct products of finite fields with prime characteristics $(\mathbb{F}_p)^L$ that are *"Entropic"* (for every four elements x, y, z and w, if $x * y = z * w$ then $x * z = y * w$). Then, for $L = 2$ we find instances of those operations $*$ that are nonlinear in $(\mathbb{F}_p)^2$, non-commutative and non-associative. In order to compute the powers x^a where $a \in \mathbb{Z}^+$, of elements $x \in G$, due to the non-associativity, we need to know some exact bracketing shape a_s, and we denote the power indices as pairs $\mathbf{A} = (a, a_s)$. Etherington defined the Logarithmetic of indices $\mathbf{A}$, $\mathbf{B}$, by defining their addition $\mathbf{A}+\mathbf{B}$ and multiplication $\mathbf{AB}$ as $x^{\mathbf{A}+\mathbf{B}} = x^{\mathbf{A}} * x^{\mathbf{B}}$ and $x^{\mathbf{AB}} = (x^{\mathbf{A}})^{\mathbf{B}}$. He further showed that the power indices of entropic groupoids, satisfy the *"Palintropic"* property i.e., $x^{\mathbf{AB}} = (x^{\mathbf{A}})^{\mathbf{B}} = (x^{\mathbf{B}})^{\mathbf{A}} = x^{\mathbf{BA}}$ which is the exact form of the Diffie-Hellman key exchange protocol.

We further analyze the chosen instances of entropic groupoids for how to find left unit elements, how to compute the multiplicative inverses, how to define addition in those groupoids, and how to find generators $g \in G$ that generate subgroupoids with a maximal size of $(p-1)^2$ elements. We show that these maximal multiplicative subgroupoids are quasigroups. We also define Sylow q-subquasigroups. Having all this, we define several hard problems such as Discrete Entropoid Logarithm Problem (DELP), Computational Entropoid Diffie-Hellman problem (CEDHP), and Decisional Entropoid Diffie-Hellman Problem (DEDHP). We post a conjecture that DEDHP is hard in Sylow q-subquasigroups. Next, we propose a new hard problem specific for the Entropoid algebraic structures: Computational Discrete Entropoid Root Problem (CDERP).

We propose instances of Diffie-Hellman key exchange protocol in those entropic, non-abelian and non-associative groupoids. Due to the hidden nature of the bracketing pattern of the power index (if chosen randomly from an exponentially large set of possible patterns), it seems that the current quantum algorithms for finding the discrete logarithms, but also all classical algorithms for solving DLP (such as Pollard's rho and kangaroo, Pohlig-Hellman, Baby-step giant-step, and others) are not suitable to address the CLDLP.

Based on CDERP, we define two post-quantum signature schemes.

Our notation for power indices that we introduce in this paper differs from the notation that

Etherington used in [3], and is adapted for our purposes to define operations of rising to the powers that have exponentially (suitable for cryptographic purposes) big values. However, for the reader's convenience we offer here a comparison of the corresponding notations: the shape s in [3] means a power index $\mathbf{A} = (a, a_s)$ here; degree $\delta(s)$ in [3] means a here; the notations of altitude $\alpha(s)$ and mutability $\mu(s)$ in [3] do not have a direct interpretation in the notation of $\mathbf{A} = (a, a_s)$ but are implicitly related to a_s.

2 Mathematical Foundations for Entropoid Based Cryptography

2.1 General definition of Logarithmetic

Definition 1. *A groupoid $(G, *)$ is an algebraic structure with a set G and a binary operation $*$ defined uniquely for all pairs of elements x and y i.e.,*

$$\forall x, y \in G, \quad \exists!\ (x * y) \in G. \tag{1}$$

The following definitions are taken and adapted for our purposes, from [3], [11], [12] and [13].

Definition 2. *We say the binary operation $*$ of the groupoid $(G, *)$ is entropic, if for every four elements $x, y, z, w \in G$, the following relation holds:*

$$\textit{If} \quad x * y = z * w \quad \textit{then} \quad x * z = y * w. \tag{2}$$

Definition 3. *Let $x \in G$ is an element in the groupoid $(G, *)$ and let $a \in \mathbb{N}$ is a natural number. A bracketing shape (pattern) for multiplying x by itself, a times is denoted by a_s i.e.*

$$a_s\colon \underbrace{(x * \ldots (x * x) \ldots)}_{a \text{ copies of } x}. \tag{3}$$

The pair $\mathbf{A} = (a, a_s)$ is called a power index.

Let us denote by S_a the set of all possible bracketing shapes a_s that use a instances of an element x i.e.

$$S_a(x) = \{a_s \mid a_s \text{ is a bracketing shape with } a \text{ instances of an element } x\}. \tag{4}$$

Proposition 1. *If $\mathbf{A} = (a, a_s)$ is a power index, then*

$$|S_a(x)| = C_{a-1} = \frac{1}{a}\binom{2a-2}{a-1}, \tag{5}$$

where C_{a-1} is the $(a-1)$-th Catalan number [14, Sequence A000108]. □

Two of those bracketing shapes are characteristic since they can be described with an iterative sequential process starting with $x * x$ absorbing the factors x one at a time either in the direction from left to right or from right to left (in [3] shapes that absorb the terms one at a time are called primary shapes).

Definition 4. *We say that the power index $\mathbf{A} = (a, a_s)$ has a primary left-to-right bracketing shape if*

$$a_s\colon \underbrace{((\ldots((x * x) * x) \ldots) * x)}_{a \text{ copies of } x}. \tag{6}$$

We shortly write it as: $\mathbf{A} \equiv x_{(a-1,x)\texttt{to-right}}$, denoting that it starts with x and applies $a-1$ right multiplications by x. For some generic and unspecified bracketing shape s the notation

$$s_{(k,x)\texttt{to-right}}$$

denotes a sequential extension of s by k right multiplications by x. This includes the formal notation $s \equiv s_{(0,x)\texttt{to-right}}$ which means the shape s itself extended with zero right multiplications by x.

We say that the power index $\mathbf{A} = (a, a_s)$ *has a primary right-to-left bracketing shape if*

$$a_s\colon \underbrace{(x * (\ldots (x * (x * x) \ldots))}_{a \text{ copies of } x}. \tag{7}$$

We shortly write it as: $\mathbf{A} \equiv x_{(x,a-1)\texttt{to-left}}$*, denoting that it starts with* x *and applies* $a-1$ *left multiplications by* x*. For some generic and unspecified bracketing shape* s *the notation*

$$s_{(x,k)\texttt{to-left}}$$

denotes a sequential extension of s *by* k *left multiplications by* x*. This includes the formal notation* $s \equiv s_{(x,0)\texttt{to-right}}$ *which means the shape* s *itself extended with zero left multiplications by* x*.*

Definition 5. *Every power index* $\mathbf{A} = (a, a_s)$ *can be considered as an endomorphism on* G *(a mapping of* G *to itself)*

$$\mathbf{A}\colon x \to x^{\mathbf{A}}. \tag{8}$$

If $\mathbf{A} = (a, a_s)$ *and* $\mathbf{B} = (b, b_s)$ *are two mappings, we define their sum* $\mathbf{A}+\mathbf{B}$ *as the power index of the product of two powers i.e.:*

$$x^{\mathbf{A}+\mathbf{B}} = x^{\mathbf{A}} * x^{\mathbf{B}}, \tag{9}$$

and we define their product $\mathbf{A}\times\mathbf{B}$ *(or shortly* $\mathbf{AB}$*) as the power index of an expression obtained by replacing every factor in the expression of the shape* b_s *by a complete shape* a_s *i.e.:*

$$x^{\mathbf{AB}} = (x^{\mathbf{A}})^{\mathbf{B}}. \tag{10}$$

Definition 6. *If the sum of any two endomorphisms of* G *is also and endomorphism of* G*, we say that* G *has additive endomorphisms.*

Definition 7. *Let* $(G, *)$ *be a given groupoid. Two power indices* $\mathbf{A} = (a, a_s)$ *and* $\mathbf{B} = (b, b_s)$ *are equal if and only if* $x^{\mathbf{A}} = x^{\mathbf{B}}$ *for all* $x \in G$*.*

Definition 8. *The Logarithmetic* $(L(G), +, \times)$ *is the algebra over the equated indices from Definition 7 with operations* $+$ *and* $\times$ *as defined in Definition 5.*

Definition 9. *If* $x^{\mathbf{AB}} = x^{\mathbf{BA}}$ *for all* $x \in G$ *and for all power indices* $\mathbf{A}$ *and* $\mathbf{B}$*, we say that the groupoid* $(G, *)$ *is palintropic.*

Theorem 1 (Etherington, [12])**.** *If the groupoid* $(G, *)$ *has additive endomorphisms, then (i) power indices are endomorphisms of* G*, (ii)* G *is palintropic.* □

Theorem 2 (Murdoch [15], Etherington, [12])**.** *If the groupoid* $(G, *)$ *is entropic, then for every* $x, y \in G$

$$(x * y)^{\mathbf{A}} = x^{\mathbf{A}} * y^{\mathbf{A}}, \tag{11}$$

and

$$x^{\mathbf{AB}} = (x^{\mathbf{A}})^{\mathbf{B}} = (x^{\mathbf{B}})^{\mathbf{A}} = x^{\mathbf{BA}}. \tag{12}$$

□

Definition 10. *We say that* $g \in G$ *is the generator of the groupoid* $(G, *)$ *if*

$$\forall x \in G, \quad \exists \mathbf{A}, \quad \textit{such that} \quad x = g^{\mathbf{A}}. \tag{13}$$

In that case we write

$$\langle g \rangle = G. \tag{14}$$

Note: In the following subsection and the rest of the paper, we will overload the operators $+$ and $\times$ for addition and multiplication of power indices defined in the previous definitions, with the same operators for the operations of addition and multiplication in the finite field $\mathbb{F}_p$. However, there will be no confusion since the operations act over different domains.

2.2 Entropic Groupoids Over $(\mathbb{F}_p)^L$

Let a set G is a direct product of L instances of the finite field $\mathbb{F}_p$ with p elements, where p is a prime number i.e.,

$$G = \underbrace{\mathbb{F}_p \times \ldots \mathbb{F}_p}_{L}. \tag{15}$$

Let $G = (\mathbb{F}_p)^L$ for $L \geq 2$, and let us represent elements $x, y, z, w \in G$ as symbolic tuples $x = (x_1, \ldots x_L)$, $y = (y_1, \ldots y_L)$, $z = (z_1, \ldots z_L)$ and $w = (w_1, \ldots w_L)$.

Definition 11 (Design criteria)**.** *The binary operation $*$ over G should meet the following:*

Design criterium 1: ***Entropic.*** *The operation $*$ should be entropic as defined in (2).*

Design criterium 2: ***Nonlinear.*** *The operation $*$ should be nonlinear with the respect of the addition and multiplication operations in the finite field $\mathbb{F}_p$.*

Design criterium 3: ***Non-commutative.*** *The operation $*$ should be non-commutative.*

Design criterium 4: ***Non-associative.*** *The operation $*$ should be non-associative.*

One simple way to find an operation that satisfies the design criteria from Definition 11 would be to define it for $L = 2$ for the most general quadratic $2L$-variate polynomials as follows:

$$\begin{aligned}
x * y = (x_1, x_2) * (y_1, y_2) =& (P_1(x_1, x_2, y_1, y_2), P_2(x_1, x_2, y_1, y_2)), && (16)\\
P_1(x_1, x_2, y_1, y_2) =& a_1 + a_2x_1 + a_3x_2 + a_4y_1 + a_5y_2 + a_6x_1y_1 + a_7x_1y_2 + a_8x_2y_1 + \\
&+ a_9x_2y_2 + a_{10}x_1^2 + a_{11}x_1x_2 + a_{12}x_2^2 + a_{13}y_1^2 + a_{14}y_1y_2 + a_{15}y_2^2, && (17)\\
P_2(x_1, x_2, y_1, y_2) =& b_1 + b_2x_1 + b_3x_2 + b_4y_1 + b_5y_2 + b_6x_1y_1 + b_7x_1y_2 + b_8x_2y_1 + \\
&+ b_9x_2y_2 + b_{10}x_1^2 + b_{11}x_1x_2 + b_{12}x_2^2 + b_{13}y_1^2 + b_{14}y_1y_2 + b_{15}y_2^2, && (18)
\end{aligned}$$

where 30 variables $a_1, \ldots, a_{15}$ and $b_1, \ldots, b_{15}$ are from $\mathbb{F}_p$, and their relations are to be determined such that (2) holds. It turns out that searching for those relations with 30 symbolic variables is not a trivial task for the modern computer algebra systems such as Sage [16] and Mathematica [17]. There are many ways how to simplify further the $2L$-variate polynomials (17) and (18) by removing some of their monomials. We present one such a simplification approach by defining the operation $*$ as follows:

$$\begin{aligned}
x * y = (x_1, x_2) * (y_1, y_2) =& (P_1(x_1, x_2, y_1, y_2), P_2(x_1, x_2, y_1, y_2)), && (19)\\
P_1(x_1, x_2, y_1, y_2) =& a_1 + a_2x_1 + a_3x_2 + a_4y_1 + a_6x_1y_1 + a_8x_2y_1, && (20)\\
P_2(x_1, x_2, y_1, y_2) =& b_1 + b_2x_1 + b_3x_2 + b_5y_2 + b_7x_1y_2 + b_9x_2y_2. && (21)
\end{aligned}$$

Open Problem 1. *Define a generic and systematic approach for finding solutions that will satisfy the design criteria of Definition 11 for higher dimensions ($L > 2$) and higher nonlinearity (the degree of the polynomials to be higher than 2).*

For the simplified system (19) - (21) one solution that satisfies all design criteria from Definition 11 is the following:

Definition 12. *Let $x = (x_1, x_2), y = (y_1, y_2)$ be two elements of the set $G = \mathbb{F}_p \times \mathbb{F}_p$. The operation $*$, i.e. $x * y$ is defined as:*

$$(x_1, x_2) * (y_1, y_2) = \left(\frac{a_3(a_8b_2 - b_7)}{a_8b_7} + a_3x_2 + \frac{a_8b_2y_1}{b_7} + a_8x_2y_1, -\frac{b_2(a_8 - a_3b_7)}{a_8b_7} + \frac{a_3b_7y_2}{a_8} + b_2x_1 + b_7x_1y_2 \right), \tag{22}$$

where $a_3, a_8, b_2, b_7 \in \mathbb{F}_p$, $a_8 \neq 0$ and $b_7 \neq 0$, and the operations $-$ and $/$ are the operations of subtraction and division in $\mathbb{F}_p$.

Open Problem 2. *Find efficient operations $*$ that satisfy the design criteria of Definition 11 and have as little as possible operations of addition, subtraction, multiplication and division in $\mathbb{F}_p$.*

The next Corollary can be easily proven by simple expression replacements.

Corollary 1. *Let $G = (\mathbb{F}_p)^2$, and let the operation $*$ is defined with Definition 12. Then:*

1. *The groupoid $(G, *)$ is entropic groupoid i.e., for every $x, y, z, w \in G$ if $x * y = z * w$ then $x * z = y * w$.*
2. *The element $\mathbf{0}_* = \left(-\frac{a_3}{a_8}, -\frac{b_2}{b_7}\right)$ is the multiplicative zero for the groupoid $(G, *)$, i.e.*

$$\forall x \in G, \quad x * \mathbf{0}_* = \mathbf{0}_* * x = \mathbf{0}_* \quad . \tag{23}$$

3. *The element $\mathbf{1}_* = \left(\frac{1}{b_7} - \frac{a_3}{a_8}, \frac{1}{a_8} - \frac{b_2}{b_7}\right)$ is the multiplicative left unit for the groupoid $(G, *)$, i.e.*

$$\forall x \in G, \quad \mathbf{1}_* * x = x \quad . \tag{24}$$

4. *For every $x = (x_1, x_2) \neq \mathbf{0}_*$ its inverse multiplicative element x_*^{-1} with the respect of the left unit element $\mathbf{1}_*$ is given by the following equation*

$$x_*^{-1} = \left(\frac{1 - a_3 b_2 - a_3 b_7 x_2}{a_8(b_2 + b_7 x_2)}, \frac{1 - a_3 b_2 - a_8 b_2 x_1}{b_7(a_3 + a_8 x_1)}\right) \quad , \tag{25}$$

for which it holds

$$x * x_*^{-1} = x_*^{-1} * x = \mathbf{1}_* \quad . \tag{26}$$

□

Proposition 2. *There are $p - 1$ distinct square roots of the left unity $\mathbf{1}_*$, i.e.*

$$\mathbb{S}(p) = \{x \mid x * x = \mathbf{1}_*\}, \quad \textit{and} \quad |\mathbb{S}(p)| = p - 1. \tag{27}$$

□

Definition 13. *Let $x = (x_1, x_2), y = (y_1, y_2)$ be two elements of the set $G = \mathbb{F}_p \times \mathbb{F}_p$. The additive operation $\boxplus$, i.e. $x \boxplus y$ is defined as:*

$$(x_1, x_2) \boxplus (y_1, y_2) = \left(x_1 + y_1 + \frac{a_3}{a_8}, x_2 + y_2 + \frac{b_2}{b_7}\right), \tag{28}$$

where $a_3, a_8, b_2, b_7 \in \mathbb{F}_p$, are defined in Definition 12.

Let us denote by $\boxminus$ the corresponding "inverse" of the additive operation $\boxplus$. Its definition is given by the following expression:

$$(x_1, x_2) \boxminus (y_1, y_2) = \left(x_1 - y_1 - \frac{a_3}{a_8}, x_2 - y_2 - \frac{b_2}{b_7}\right). \tag{29}$$

We can use the operation $\boxminus$ also as a unary operator. If we write $\boxminus x$ then we mean

$$\boxminus x \stackrel{def}{=} \mathbf{0}_* \boxminus x = \left(-\frac{a_3}{a_8} - x_1, -\frac{b_2}{b_7} - x2\right). \tag{30}$$

One can check that "minus one" i.e. $\boxminus \mathbf{1}_*$ is also a square root of the left unity i.e that it holds: $\boxminus \mathbf{1}_* * \boxminus \mathbf{1}_* = \mathbf{1}_*$.

A consequence of Corollary 1 and Definition 13 is the following

Corollary 2. *The algebraic structure $(G, \boxplus, *)$ is a ringoid where $(G, \boxplus)$ is an Abelian group with a neutral element $\mathbf{0}_*$, $(G, *)$ is a non-commutative and non-associative groupoid with a zero element $\mathbf{0}_*$, a left unit element $\mathbf{1}_*$ and $*$ is distributive over $\boxplus$ i.e.*

$$x * (y \boxplus z) = (x * y) \boxplus (x * z) \quad \textit{and} \quad (x \boxplus y) * z = (x * z) \boxplus (y * z). \tag{31}$$

Definition 14. *The ringoid $\mathbb{E}_{p^2} = (G, \boxplus, *)$ with operation $*$ defined with Definition 12 and operation $\boxplus$ defined with Definition 13 is called a* Finite Entropic Ring *or* Finite Entropoid *with p^2 elements. For given values of p, a_3, a_8, b_2 and b_7 it will be denoted as $\mathbb{E}_{p^2}(a_3, a_8, b_2, b_7)$.* [1]

In the rest of the text we will use interchangeably the notations $\mathbb{E}_{p^2}(a_3, a_8, b_2, b_7)$ and if in text context, the constants a_3, a_8, b_2, b_7 are not important, just $\mathbb{E}_{p^2}$. We will also assume that the choice for the parameters is $a_3 \neq 0$ and $b_2 \neq 0$. To shorten the mathematical expressions, when the meaning is clear from the context, we will also overload the symbol $\mathbb{E}_{p^2}$ with two interpretations: as a set $\mathbb{E}_{p^2} = G = \mathbb{F}_p \times \mathbb{F}_p$ and as an algebraic structure $\mathbb{E}_{p^2} = (G, \boxplus, *)$.

Definition 15. *Let us define the subset $\mathbb{E}^*_{(p-1)^2} \subset \mathbb{E}_p$ as*

$$\mathbb{E}^*_{(p-1)^2} = \Big(\Big(\mathbb{F}_p \setminus \Big\{ -\frac{a_3}{a_8} \Big\} \Big) \times \Big(\mathbb{F}_p \setminus \Big\{ -\frac{b_2}{b_7} \Big\} \Big) \Big). \tag{32}$$

We say that the groupoid $(\mathbb{E}^*_{(p-1)^2}, *)$ is the maximal multiplicative subgroupoid of $\mathbb{E}_{p^2}$.

It is clear from Definition 15 that the multiplicative subgroupoid $\mathbb{E}^*_{(p-1)^2}$ has $(p-1)^2$ elements $x = (x_1, x_2)$ such that x_1 is not the first component and x_2 is not the second component of the multiplicative zero $\mathbf{0}_*$. It has one additional property: it is a quasigroup, and that is stated in the following Lemma.

Lemma 1. *The maximal multiplicative groupoid $(\mathbb{E}^*_{(p-1)^2}, *)$ defined in Definition 15 with the operation $*$ defined in Definition 12 is a non-commutative and non-associative quasigroup with a left unit element $\mathbf{1}_*$.*

Proof. Non-commutativity, non-associativity and the proof about the left unit element come directly by the definition of the operation $*$, and Corollary 1. The only remaining part is to prove that for every $c = (c_1, c_2) \in \mathbb{E}^*_{(p-1)^2}$ and every $d = (d_1, d_2) \in \mathbb{E}^*_{(p-1)^2}$, the equations $c * x = d$ and $x * c = d$ have always solutions. It is left as an exercise for the reader to replace the values of c and d, to apply the definition of the operation $*$ annd with simple polynomial algebra to get get two equations for x_1 and x_2 with a unique solution $x = (x_1, x_2)$. □

We will use the notation $(\mathbb{E}^*_\nu, *)$ for the subgroupoids of $(\mathbb{E}^*_{(p-1)^2}, *)$, with ν elements. That means that if we are given $(\mathbb{E}^*_\nu, *)$, then $\mathbb{E}^*_\nu \subseteq \mathbb{E}^*_{(p-1)^2}$, $\forall x, y \in \mathbb{E}^*_\nu$, $x * y \in \mathbb{E}^*_\nu$ and $|\mathbb{E}^*_\nu| = \nu$. Using the Etherington terminology, we will say that the quasigroup $(\mathbb{E}^*_{(p-1)^2}, *)$ and all of its subquasigroups $(\mathbb{E}^*_\nu, *)$ are *entropic quasigroups.*

Proposition 3. *If $(\mathbb{E}^*_\nu, *)$ is a subgroupoid of $(\mathbb{E}^*_{(p-1)^2}, *)$, then ν divides $(p-1)^2$ i.e. $\nu | (p-1)^2$ and $(\mathbb{E}^*_\nu, *)$ is a subquasigroup of $(\mathbb{E}^*_{(p-1)^2}, *)$.* □

The following Proposition is a connection between the subgroups of the multiplicative group of a finite field, and the subgroupoids in the finite entropoid. It is a consequence of Proposition 3 and the Lagrange's theorem for groups:

Proposition 4. *Let $\mathbb{F}_p$ be the finite field over which an entropoid $\mathbb{E}_{p^2}$ is defined. Let $\Gamma \subseteq \mathbb{F}^*_p$ be a cyclic subgroup of order $|\Gamma|$ of the multiplicative group $\mathbb{F}^*_p$ and let $\gamma \neq -\frac{a_3}{a_8}$ and $\gamma \neq -\frac{b_2}{b_7}$ is its generator, then $g = (\gamma, \gamma)$ is a generator of a subgroupoid $(\mathbb{E}^*_\nu, *)$ and ν divides $|\Gamma|^2$.*

Next, for our subquasigroups $(\mathbb{E}^*_\nu, *)$ we will partially use the Smith terminology in his proposed Sylow theory for quasigroups [19].

[1] The ringoid $(G, \boxplus, *)$ is neither a neofield (since $(G, *)$ is not a group), nor a Lie ring (since the Jacobi identity is not satisfied), but is built with the operation $*$ given by the entropic identity (2). We are aware of the work of J.D.H. Smith and A.B. Romanowska in [18] that refer to the entropic Jonsson-Tarski algebraic varieties, but for our purposes, and to be more narrow with the definition of the algebraic structure that we will use, we give a formal name of this ringoid as "Entropoid".

Definition 16. *Let* $(\mathbb{E}^*_\nu, *)$ *be a subquasigroup of* $(\mathbb{E}^*_{(p-1)^2}, *)$ *and let* $(p-1)^2$ *be represented as product of the powers of its prime factors i.e.*$(p-1)^2 = 2^{e_1} q_1^{e_2} \dots q_k^{e_k}$. *We say that* $(\mathbb{E}^*_\nu, *)$ *is Sylow* q_i*-subquasigroup if* $\nu = q_i^{e_i}$ *for* $i \in \{2, \dots, k\}$.

Before we continue, let us give one example.

Example 1. Let the finite entropoid be defined as $\mathbb{E}_{7^2}(a_3 = 6, a_8 = 3, b_2 = 3, b_7 = 4)$. In that case the operation $*$ becomes:

$$x * y = (x_1, x_2) * (y_1, y_2) = (6 + 6x_2 + 4y_1 + 3x_2y_1, \quad y_2 + 3x_1 + 4x_1y_2).$$

All elements from $\mathbb{E}_{7^2}$ are presented in Table 1.

From Corollary 1 we get $\mathbf{0}_* = (5, 1)$ and $\mathbf{1}_* = (0, 6)$. The elements that are highlighted in Table 1 (elements in the row and column where the zero element $\mathbf{0}_*$ is positioned) are excluded from the multiplicative subgroupoid $\mathbb{E}^*_{6^2}$.

$\mathbb{E}_7$	0	1	2	3	4	5	6
0	$(0,0)$	$(0,1)$	$(0,2)$	$(0,3)$	$(0,4)$	$(0,5)$	$\mathbf{1}_*$ $(0,6)$
1	$(1,0)$	$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$	$(1,5)$	$(1,6)$
2	$(2,0)$	$(2,1)$	$(2,2)$	$(2,3)$	$(2,4)$	$(2,5)$	$(2,6)$
$a_8, b_2 \to 3$	$(3,0)$	$(3,1)$	$(3,2)$	$(3,3)$	$(3,4)$	$(3,5)$	$(3,6)$
$b_7 \to 4$	$(4,0)$	$(4,1)$	$(4,2)$	$(4,3)$	$(4,4)$	$(4,5)$	$(4,6)$
5	$(5,0)$	$\mathbf{0}_*$ $(5,1)$	$(5,2)$	$(5,3)$	$(5,4)$	$(5,5)$	$(5,6)$
$a_3 \to 6$	$(6,0)$	$(6,1)$	$(6,2)$	$(6,3)$	$(6,4)$	$(6,5)$	$(6,6)$

Table 1: Representation of all 49 elements of $\mathbb{E}_{7^2}$.

We can make several observations from the values in Table 2 and Table 3:

- As we would expect, $\mathbf{0}_*^a = \mathbf{0}_*$ and $\mathbf{1}_*^a = \mathbf{1}_*$ for all a and for all bracketing patterns a_s.
- Some values of x, raised to different patterns a_s for the same value a tend to produce more different values, while some generate less different values. For example, for $a = 4$, for $x = (0, 2)$ there are three different values $(3, 2)$, $(2, 4)$ and $(4, 3)$, while for $x = (0, 3)$ there are two different values $(3, 3)$ and $(0, 6)$.
- For any x, for a given fixed a, the number of different values generated with all patterns a_s do not exceed $a - 1$.
- When the number of different values is $a - 1$, the distribution of the patterns over those $a - 1$ values follows the distribution of Narayana numbers $N(n, k)$[14, Sequence A001263]:

  $a = 4$**:** 1, 3, 1

  $a = 5$**:** 1, 6, 6, 1

  $a = 6$**:** 1, 10, 20, 10, 1

One can raise to different powers with different patterns all elements of $\mathbb{E}_{7^2}$ and can count the total number of generated elements. In Table 4 we show the size of the generated sets with all elements of $\mathbb{E}_{7^2}$. The green highlighted elements with the value 36 generate the maximal multiplicative subgroupoid $(\mathbb{E}^*_{36}, *)$. For example, $(0, 2)$ is the generator of $(\mathbb{E}^*_{36}, *)$.

a	a_s i.e. bracketing of x (size of the associator class)	$(0,0)$	$(0,1)$	$(0,2)$	$(0,3)$	$(0,6)$	$(1,3)$	$(3,0)$	$(4,6)$	$(5,1)$	$(6,6)$
2	$(x*x)$	$(6,0)$	$(5,1)$	$(4,2)$	$(3,3)$	$(0,6)$	$(2,4)$	$(4,2)$	$(4,2)$	$(5,1)$	$(6,0)$
3	$(x*(x*x))$	$(2,0)$	$(5,1)$	$(2,2)$	$(0,3)$	$(0,6)$	$(1,2)$	$(1,0)$	$(4,4)$	$(5,1)$	$(6,4)$
	$((x*x)*x)$	$(6,4)$	$(5,1)$	$(4,4)$	$(3,6)$	$(0,6)$	$(4,5)$	$(6,5)$	$(2,2)$	$(5,1)$	$(2,0)$
4	$(x*(x*(x*x)))$ (1)	$(0,0)$	$(5,1)$	$(3,2)$	$(3,3)$	$(0,6)$	$(2,6)$	$(3,2)$	$(4,3)$	$(5,1)$	$(6,6)$
	$(x*((x*x)*x))$ (3)	$(2,4)$	$(5,1)$	$(2,4)$	$(0,6)$	$(0,6)$	$(6,0)$	$(2,4)$	$(2,4)$	$(5,1)$	$(2,4)$
	$((x*x)*(x*x))$	$(2,4)$	$(5,1)$	$(2,4)$	$(0,6)$	$(0,6)$	$(6,0)$	$(2,4)$	$(2,4)$	$(5,1)$	$(2,4)$
	$((x*(x*x))*x)$ (1)	$(6,6)$	$(5,1)$	$(4,3)$	$(3,3)$	$(0,6)$	$(0,4)$	$(4,3)$	$(3,2)$	$(5,1)$	$(0,0)$
	$(((x*x)*x)*x)$	$(2,4)$	$(5,1)$	$(2,4)$	$(0,6)$	$(0,6)$	$(6,0)$	$(2,4)$	$(2,4)$	$(5,1)$	$(2,4)$
5	$(x*(x*(x*(x*x))))$ (1)	$(6,0)$	$(5,1)$	$(6,2)$	$(0,3)$	$(0,6)$	$(1,5)$	$(4,0)$	$(4,0)$	$(5,1)$	$(6,0)$
	$(x*(x*((x*x)*x)))$ (6)	$(0,4)$	$(5,1)$	$(3,4)$	$(3,6)$	$(0,6)$	$(4,3)$	$(0,5)$	$(2,3)$	$(5,1)$	$(2,6)$
	$(x*((x*x)*(x*x)))$	$(0,4)$	$(5,1)$	$(3,4)$	$(3,6)$	$(0,6)$	$(4,3)$	$(0,5)$	$(2,3)$	$(5,1)$	$(2,6)$
	$(x*((x*(x*x))*x))$ (6)	$(2,6)$	$(5,1)$	$(2,3)$	$(0,3)$	$(0,6)$	$(3,2)$	$(1,6)$	$(3,4)$	$(5,1)$	$(0,4)$
	$(x*(((x*x)*x)*x))$	$(0,4)$	$(5,1)$	$(3,4)$	$(3,6)$	$(0,6)$	$(4,3)$	$(0,5)$	$(2,3)$	$(5,1)$	$(2,6)$
	$((x*x)*(x*(x*x)))$	$(0,4)$	$(5,1)$	$(3,4)$	$(3,6)$	$(0,6)$	$(4,3)$	$(0,5)$	$(2,3)$	$(5,1)$	$(2,6)$
	$((x*x)*((x*x)*x))$	$(2,6)$	$(5,1)$	$(2,3)$	$(0,3)$	$(0,6)$	$(3,2)$	$(1,6)$	$(3,4)$	$(5,1)$	$(0,4)$
	$((x*(x*x))*(x*x))$	$(2,6)$	$(5,1)$	$(2,3)$	$(0,3)$	$(0,6)$	$(3,2)$	$(1,6)$	$(3,4)$	$(5,1)$	$(0,4)$
	$(((x*x)*x)*(x*x))$	$(0,4)$	$(5,1)$	$(3,4)$	$(3,6)$	$(0,6)$	$(4,3)$	$(0,5)$	$(2,3)$	$(5,1)$	$(2,6)$
	$((x*(x*(x*x)))*x)$ (1)	$(6,0)$	$(5,1)$	$(4,0)$	$(3,6)$	$(0,6)$	$(1,5)$	$(6,2)$	$(6,2)$	$(5,1)$	$(6,0)$
	$((x*((x*x)*x))*x)$	$(2,6)$	$(5,1)$	$(2,3)$	$(0,3)$	$(0,6)$	$(3,2)$	$(1,6)$	$(3,4)$	$(5,1)$	$(0,4)$
	$(((x*x)*(x*x))*x)$	$(2,6)$	$(5,1)$	$(2,3)$	$(0,3)$	$(0,6)$	$(3,2)$	$(1,6)$	$(3,4)$	$(5,1)$	$(0,4)$
	$(((x*(x*x))*x)*x)$	$(0,4)$	$(5,1)$	$(3,4)$	$(3,6)$	$(0,6)$	$(4,3)$	$(0,5)$	$(2,3)$	$(5,1)$	$(2,6)$
	$((((x*x)*x)*x)*x)$	$(2,6)$	$(5,1)$	$(2,3)$	$(0,3)$	$(0,6)$	$(3,2)$	$(1,6)$	$(3,4)$	$(5,1)$	$(0,4)$

Table 2: Results of rising different values x to the powers from 2 to 5 with different bracketing patterns a_s. The green highlighted cells are for the representatives of a class of expressions that give the same result, and in the brackets is the size of that class.

a	a_s i.e. bracketing of x (size of the associator class)	$(0,2)$	$(3,0)$	$(6,6)$	a_s i.e. bracketing of x (size of the associator class)	$(0,2)$	$(3,0)$	$(6,6)$	a_s i.e. bracketing of x (size of the associator class)	$(0,2)$	$(3,0)$	$(6,6)$
6	$(x*(x*(x*(x*(x*x)))))$ (1)	$(1,2)$	$(1,2)$	$(6,4)$	$((x*x)*(x*(x*(x*x))))$	$(6,4)$	$(6,4)$	$(2,0)$	$((x*(x*(x*(x*x))))*x)$ (1)	$(4,5)$	$(4,5)$	$(2,0)$
	$(x*(x*(x*((x*x)*x))))$ (10)	$(6,4)$	$(6,4)$	$(2,0)$	$((x*x)*(x*((x*x)*x)))$	$(3,3)$	$(3,3)$	$(0,6)$	$((x*(x*((x*x)*x)))*x)$	$(2,0)$	$(2,0)$	$(6,4)$
	$(x*(x*((x*x)*(x*x))))$	$(6,4)$	$(6,4)$	$(2,0)$	$((x*x)*((x*x)*(x*x)))$	$(3,3)$	$(3,3)$	$(0,6)$	$((x*((x*x)*(x*x)))*x)$	$(2,0)$	$(2,0)$	$(6,4)$
	$(x*(x*((x*(x*x))*x)))$ (20)	$(3,3)$	$(3,3)$	$(0,6)$	$((x*x)*((x*(x*x))*x))$	$(2,0)$	$(2,0)$	$(6,4)$	$((x*((x*(x*x))*x))*x)$	$(3,3)$	$(3,3)$	$(0,6)$
	$(x*(x*(((x*x)*x)*x)))$	$(6,4)$	$(6,4)$	$(2,0)$	$((x*x)*(((x*x)*x)*x))$	$(3,3)$	$(3,3)$	$(0,6)$	$(((x*(x*x))*(x*(x*x)))*x)$	$(2,0)$	$(2,0)$	$(6,4)$
	$(x*((x*x)*(x*(x*x))))$	$(6,4)$	$(6,4)$	$(2,0)$	$((x*(x*x))*(x*(x*x)))$	$(3,3)$	$(3,3)$	$(0,6)$	$(((x*x)*(x*(x*x)))*x)$	$(2,0)$	$(2,0)$	$(6,4)$
	$(x*((x*x)*((x*x)*x)))$	$(3,3)$	$(3,3)$	$(0,6)$	$((x*(x*x))*((x*x)*x))$	$(2,0)$	$(2,0)$	$(6,4)$	$(((x*x)*((x*x)*x))*x)$	$(3,3)$	$(3,3)$	$(0,6)$
	$(x*((x*(x*x))*(x*x)))$	$(3,3)$	$(3,3)$	$(0,6)$	$(((x*x)*x)*(x*(x*x)))$	$(6,4)$	$(6,4)$	$(2,0)$	$(((x*(x*x))*(x*x))*x)$	$(3,3)$	$(3,3)$	$(0,6)$
	$(x*(((x*x)*x)*(x*x)))$	$(6,4)$	$(6,4)$	$(2,0)$	$(((x*x)*x)*((x*x)*x))$	$(3,3)$	$(3,3)$	$(0,6)$	$((((x*x)*x)*(x*x))*x)$	$(2,0)$	$(2,0)$	$(6,4)$
	$(x*((x*(x*x))*((x*x)*x)))$ (10)	$(2,0)$	$(2,0)$	$(6,4)$	$(((x*(x*(x*x)))*(x*x))*x)$	$(2,0)$	$(2,0)$	$(6,4)$	$((((x*(x*(x*x)))*x)*x)*x)$	$(6,4)$	$(6,4)$	$(2,0)$
	$(x*((x*((x*x)*x))*(x*x)))$	$(3,3)$	$(3,3)$	$(0,6)$	$(((x*((x*x)*x))*(x*x))*x)$	$(3,3)$	$(3,3)$	$(0,6)$	$((((x*((x*x)*x))*x)*x)*x)$	$(3,3)$	$(3,3)$	$(0,6)$
	$(x*(((x*x)*(x*x))*(x*x)))$	$(3,3)$	$(3,3)$	$(0,6)$	$(((x*x)*(x*x))*(x*(x*x)))$	$(3,3)$	$(3,3)$	$(0,6)$	$((((x*x)*(x*x))*x)*x)$	$(3,3)$	$(3,3)$	$(0,6)$
	$(x*(((x*(x*x))*x)*x))$	$(6,4)$	$(6,4)$	$(2,0)$	$(((x*(x*x))*x)*(x*x))$	$(6,4)$	$(6,4)$	$(2,0)$	$((((x*(x*x))*x)*x)*x)$	$(2,0)$	$(2,0)$	$(6,4)$
	$(x*((((x*x)*x)*x)*x))$	$(3,3)$	$(3,3)$	$(0,6)$	$((((x*x)*x)*x)*(x*x))$	$(3,3)$	$(3,3)$	$(0,6)$	$(((((x*x)*x)*x)*x)*x)$	$(3,3)$	$(3,3)$	$(0,6)$

Table 3: Results of rising different values x to the power of 6 with different bracketing patterns a_s.

The yellow highlighted cells in Table 4 are for the elements that do not belong to the multiplicative subgroupoid $((\mathbb{E}_{36}^*, *)$.

Notice that the prime number $p = 7$ can be represented as $p = 2q+1 = 2*3+1$, and that all elements of $\mathbb{E}_{7^2}$ can belong to different classes with cardinalities that are divisors of $(p-1)^2 = 2^2q^2 = 2^2 3^2$, i.e. the cardinalities are $\{1, 2, 3, 4, 6, 9, 18, 36\}$. □

	0	1	2	3	4	5	6
0	9	2	36	4	9	36	1
1	12	2	6	18	12	6	36
2	3	2	12	36	3	12	9
3	36	2	18	2	36	18	4
4	12	2	6	18	12	6	36
5	2	1	2	2	2	2	2
6	3	2	12	36	3	12	9

Table 4: The size of the sets $\langle (x_1, x_2) \rangle$ for $x = (x_1, x_2) \in \mathbb{E}_{7^2}$ where x_1 represents the row number and x_2 represents the column number.

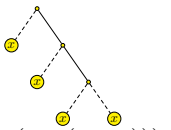
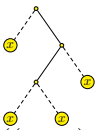
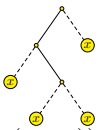
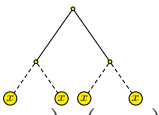
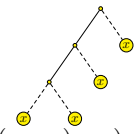
a	$i=1,\ N(3,1)=1$	$i=2,\ N(3,2)=3$	$i=3,\ N(3,3)=1$
4	 $(x*(x*(x*x)))$	 $(x*((x*x)*x))$	 $((x*(x*x))*x)$
		 $((x*x)*(x*x))$	
		 $(((x*x)*x)*x)$	

Table 5: Representation of all possible non-associative powers of $a=4$ as planar binary trees, and the corresponding clastering in subsets with Narayana numbers of elements.

Definition 17. *Let $2 \leq a$ be an integer, and let x be an element in G. An ordered list $R_a(x)$ of $a-1$ bracketing shapes is called the list of associative class representatives and is defined as:*

$$\begin{aligned} R_a(x) = [R_a(x)[0], R_a(x)[1], \ldots, R_a(x)[a-3], R_a(x)[a-2]], \textit{ where} \\ R_a(x)[0] = x_{(x,a-1)\,\texttt{to-left}}, \\ R_a(x)[1] = (x_{(x,1)\,\texttt{to-left}} * x)_{(x,a-3)\,\texttt{to-left}}, \\ R_a(x)[2] = (x_{(x,2)\,\texttt{to-left}} * x)_{(x,a-4)\,\texttt{to-left}}, \\ \ldots, \\ R_a(x)[a-3] = (x_{(x,a-3)\,\texttt{to-left}} * x)_{(x,0)\,\texttt{to-left}}, \\ R_a(x)[a-2] = (x_{(x,a-2)\,\texttt{to-left}} * x), \end{aligned} \tag{33}$$

Note: We use the zero-based indexing style for the list members.

Lemma 2. *The bracketing shapes for $R_a(x)$ and $R_{a+1}(x)$ are related with the following recurrent relations:*

$$\begin{aligned} R_{a+1}(x)[0] = (x * R_a(x)[0]), \\ R_{a+1}(x)[1] = (x * R_a(x)[1]), \\ R_{a+1}(x)[2] = (x * R_a(x)[2]), \\ \ldots, \\ R_{a+1}(x)[a-2] = (x * R_a(x)[a-2]), \\ R_{a+1}(x)[a-1] = (R_a(x)[0] * x), \end{aligned} \tag{34}$$

Proof. We will prove the lemma with induction by a. Let us first note that $R_2(x) = [R_2(x)[0] \equiv (x*x)]$. For $a=3$ we have $R_3(x) = [R_3(x)[0], R_3(x)[1]]$, where $R_3(x)[0] = x_{(x,3-1)\texttt{to-left}} = (x*(x*x)) = (x*R_2(x))$, and $R_3(x)[1] = (x_{(x,3-2)\texttt{to-left}} * x) = ((x*x)*x) = (R_2(x)[0]*x)$.

If we suppose that the recurrent relations (34) are true for a, then for $a+1$ we have

- $R_{a+1}(x)[0] \stackrel{def}{=} x_{(x,a)\texttt{to-left}} = (x * x_{(x,a-1)\texttt{to-left}}) = (x * R_a(x)[0])$,
- $R_{a+1}(x)[1] \stackrel{def}{=} (x_{(x,1)\texttt{to-left}} * x)_{(x,a+1-3)\texttt{to-left}} = (x * x_{(x,a-2)\texttt{to-left}}) = (x * R_a(x)[1])$,
- $\ldots$,
- $R_{a+1}(x)[a-2] \stackrel{def}{=} (x_{(x,a-2)\texttt{to-left}} * x)_{(x,0)\texttt{to-left}} = (x * R_a(x)[a-2])$,

- $R_{a+1}(x)[a-1] \stackrel{def}{=} (x_{(x,a-1)\texttt{to-left}} * x) = (R_a(x)[0] * x)$.

□

Example 2. Let us present the bracketing shapes highlighted in green in Table 2 and in Table 3, in Example 1 with the notation introduced in Definition 17.

$a = 4$ **:** Out of 5 bracketing shapes, the following 3 are highlighted:

$i = 1$**:** $(x * (x * (x * x))) = x_{(x,3)\texttt{to-left}}$
$i = 2$**:** $(x * ((x * x) * x)) = (x_{(x,1)\texttt{to-left}} * x)_{(x,0)\texttt{to-left}}$
$i = 3$**:** $((x * (x * x)) * x) = (x_{(x,2)\texttt{to-left}} * x)$

$a = 5$ **:** Out of 14 bracketing shapes, the following 4 are highlighted:

$i = 1$**:** $(x * (x * (x * (x * x)))) = x_{(x,4)\texttt{to-left}}$
$i = 2$**:** $(x * (x * ((x * x) * x))) = (x_{(x,1)\texttt{to-left}} * x)_{(x,2)\texttt{to-left}}$
$i = 3$**:** $(x * ((x * (x * x)) * x)) = (x_{(x,2)\texttt{to-left}} * x)_{(x,1)\texttt{to-left}}$
$i = 4$**:** $((x * (x * (x * x))) * x) = (x_{(x,3)\texttt{to-left}} * x)$

$a = 6$ **:** Out of 42 bracketing shapes, the following 5 are highlighted:

$i = 1$**:** $(x * (x * (x * (x * (x * x))))) = x_{(x,5)\texttt{to-left}}$
$i = 2$**:** $(x * (x * (x * ((x * x) * x)))) = (x_{(x,1)\texttt{to-left}} * x)_{(x,3)\texttt{to-left}}$
$i = 3$**:** $(x * (x * ((x * (x * x)) * x))) = (x_{(x,2)\texttt{to-left}} * x)_{(x,2)\texttt{to-left}}$
$i = 4$**:** $(x * ((x * (x * (x * x))) * x)) = (x_{(x,3)\texttt{to-left}} * x)_{(x,1)\texttt{to-left}}$
$i = 5$**:** $((x * (x * (x * (x * x)))) * x) = (x_{(x,4)\texttt{to-left}} * x)$

Let us also present in Table 5 and Table 6 the bracketing shapes for $a = 4$ and $a = 5$ as planar binary trees (for $a = 6$ it would be an impractically tall table with 20 trees in one column). We see that the first rows in the tables are exactly the highlighted green shapes from Table 2.

□

Since for any fixed integer a, the set $S_a(x)$ of all bracketing shapes $\mathbf{A} = (a, a_s)$ have $C_{a-1} = \frac{1}{a}\binom{2a-2}{a-1}$ elements (equation (5)), for a generic non-commutative and non-associative groupoid $(G, *)$, with a multiplicative operation $*$ we would expect that for some elements $x \in G$ there would be up to C_{a-1} different power values in the set $\{x^{\mathbf{A}}\}$. However, for entropic groupoids ($\mathbb{E}^*_{p^2}$ defined by Definition 11 and by the equation (22) that is not the case, and the size of the set $\{x^{\mathbf{A}}\}$ is limited to $a - 1$ as we see it in Example 1 and Example 2.

Definition 18. *Let $\mathbb{E}_{p^2}$ be a given finite entropoid. We define an integer b_{max} by the following expression:*

$$b_{max} = \min\left\{ b \mid \frac{1}{b}\binom{2b-2}{b-1} > (p-1)^2 \right\}. \tag{35}$$

The value b_{max} is the smallest integer for which the Catalan number $C_{b-1} = \frac{1}{b}\binom{2b-2}{b-1}$ surpasses the number of elements in the maximal multiplicative subgroupoid (G^*, g). We will need it in the proof of the next theorem.

Theorem 3 (Equivalent classes and their representatives)**.** *Let $\mathbb{E}_{p^2}$ be given, and let g is a generator of its maximal multiplicative subgroupoid $(\mathbb{E}^*_{(p-1)^2}, *)$, where $*$ is defined by Definition 11 and by the equation (22). For every $3 \leq a < b_{max}$, evaluating the bracketing shapes of the set $S_a(g)$ gives a partitioning in $a - 1$ equivalent classes $S_{1,a(g)}$, $S_{2,a(g)}$, ..., $S_{a-2,a(g)}$, $S_{a-1,a(g)}$, whose corresponding*

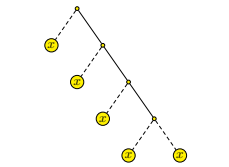
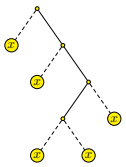
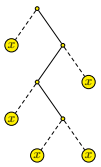
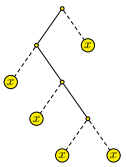
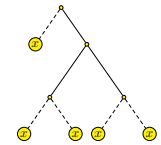
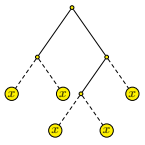
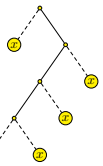
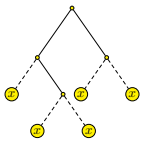
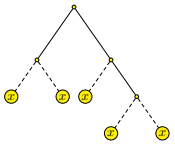
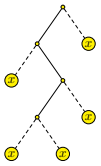
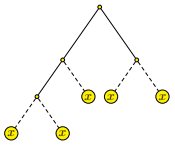
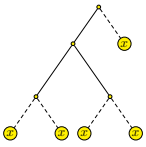
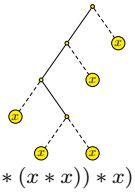
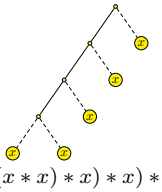
a	$i=1,\ N(4,1)=1$	$i=2,\ N(4,2)=6$	$i=3,\ N(4,3)=6$	$i=4,\ N(4,4)=1$
5	 $(x*(x*(x*(x*x))))$	 $(x*(x*((x*x)*x)))$	 $(x*((x*(x*x))*x))$	 $((x*(x*(x*x)))*x)$
	 $(x*((x*x)*(x*x)))$	 $((x*x)*((x*x)*x))$		
	 $(x*(((x*x)*x)*x))$	 $((x*(x*x))*(x*x))$		
	 $((x*x)*(x*(x*x)))$	 $((x*((x*x)*x))*x)$		
	 $(((x*x)*x)*(x*x))$	 $(((x*x)*(x*x))*x)$		
	 $(((x*(x*x))*x)*x)$	 $((((x*x)*x)*x)*x)$		

Table 6: Representation of all possible non-associative powers of $a=5$ as planar binary trees, and the corresponding clastering in subsets with Narayana number of elements.

representatives are given by the corresponding elements of $R_a(g)$ defined by the relation (33) and the cardinality of the sets $S_{i,a(g)}$ for $i=1,\ldots,a-1$ is given by the following expression

$$|S_{i,a(g)}| = N(a-1,i) = \frac{1}{a-1}\binom{a-1}{i}\binom{a-1}{i-1}, \tag{36}$$

where $N(a-1,i)$ are the Narayana numbers.

Proof. We will prove the theorem by induction on a for the range $3 \leq a < b_{max}$. For $a \geq b_{max}$ there are not enough elements to be classified in classes which total number surpass $(p-1)^2$ (as it is the number of elements of $(\mathbb{E}^*_{(p-1)^2}, *)$).

For $a=3$ the theorem is trivially true since there are just $a-1=2$ associative shapes. The first non-trivial case is for $a=4$. Taking $g=(g_1,g_2)$, and using the definition for the operation $*$ given in equation (22), by basic symbolic and algebraic expression replacements we get that for all three

shapes the end result is:

$$R_4(g)[1] \equiv (g*((g*g)*g)) = ((g*g)*(g*g)) = (((g*g)*g)*g) =$$
$$= \left(\begin{array}{l} \dfrac{a_8^3 g_1^2 \left(b_7 g_2 + b_2\right)^2 + 2a_3 a_8^2 g_1 \left(b_7 g_2 + b_2\right)^2 + a_3 \left(2a_3 a_8 b_7 b_2 g_2 + a_3 a_8 b_7^2 g_2^2 + a_3 a_8 b_2^2 - b_7\right)}{a_8 b_7}, \\ \dfrac{a_3^2 b_7^3 g_2^2 + 2a_3^2 b_2 b_7^2 g_2 + a_8^2 b_7 g_1^2 \left(b_7 g_2 + b_2\right)^2 + 2a_3 a_8 b_7 g_1 \left(b_7 g_2 + b_2\right)^2 + b_2 \left(a_3^2 b_2 b_7 - a_8\right)}{a_8 b_7} \end{array} \right)$$

We can also check that $R_4(g)[0] \equiv (g*(g*(g*g))) \neq (g*((g*g)*g)) \equiv R_4(g)[1]$, $R_4(g)[2] \equiv ((g*(g*g))*g) \neq (g*((g*g)*g)) \equiv R_4(g)[1]$, and $R_4(g)[0] \equiv (g*(g*(g*g))) \neq ((g*(g*g))*g) \equiv R_4(g)[2]$. Thus for the case $a = 4$ we have $S_{1,4(g)} = \{R_4(g)[0]\}$, $S_{2,4(g)} = \{R_4(g)[1] \equiv (g*((g*g)*g)), ((g*g)*(g*g)), (((g*g)*g)*g)\}$ and $S_{3,4(g)} = \{R_4(g)[2]\}$.

Let us now suppose that the claims of the theorem are true for a, i.e. that the expressions (37) and (36) hold. Then, for $a+1$ we have the set $S_{a+1}(g)$ that has $C_a = \frac{1}{a+1}\binom{2a}{a}$ elements, where C_a is the (a)-th Catalan number. We can now partition the set $S_{a+1}(g)$ as follows:

$$S_{a+1}(g) = S_{1,a(g)} \bigcup S_{2,a(g)} \bigcup \ldots \bigcup S_{a-1,a(g)} \bigcup S_{a,a(g)}, \tag{37}$$

where $x_{(x,a-1)\texttt{to-left}} \in S_{1,a(g)}$, $(x_{(x,1)\texttt{to-left}} * x)_{(x,a-3)\texttt{to-left}} \in S_{2,a(g)}$, …, $(x_{(x,a-3)\texttt{to-left}} * x)_{(x,1)\texttt{to-left}} \in S_{a-2,a(g)}$ and $((x_{(x,a-2)\texttt{to-left}} * x) \in S_{a-1,a(g)}$.

The conditions that g is a generator of the $(\mathbb{E}^*_{(p-1)^2}, *)$, and that $*$ is non-commutative and non-associative operation are essential for obtaining that the element $R_{a+1}(x)[a-1] = (R_a(x)[0] * x)$ of the subset $S_{a,a(g)}$ is different from the element $R_{a+1}(x)[0] = (x * R_a(x)[0])$ of the subset $S_{1,a(g)}$.

Then,if we represent the Catalan number C_a as a sum of Narayana numbers $N(a,i)$:

$$C_a = N(a,1) + N(a,2) + \ldots + N(a,a-1) + N(a,a),$$

and if we use the recurrent relations for the representatives of the equivalent classes given in Lemma 2 to get expressions for the representatives of the partitions of $S_{a+1}(g)$, we obtain that the relations (37) and (36) hold also for $a+1$. □

Open Problem 3. *In the proof of the Theorem 3, as a jump-start for the induction we used the concrete instance for the entropic operation $*$ defined by the equation (22). We do not know does the same partitioning in equivalent classes hold for general entropic operations $*$, and it would be an interesting mathematical problem to further investigate.*

2.3 Succinct Notation for Exponentially Large Bracketing Shapes

Notation: In the rest of the text we assume that we work in finite entropoid $\mathbb{E}_{p^2}$ and $g \in \mathbb{E}_{p^2}$ is a generator of some multiplicative subgroupoid $(\mathbb{E}^*_\nu, *)$, where $*$ is defined by Definition 11 and by the equation (22). In Definition 17, Lemma 2 and Theorem 3 we use the symbol a to represent any number (possibly exponentially large) of multiplicative factors in the process of rising to the power of a. In our pursuit for succinct representation of exponentially large indices, we will keep the symbol a, but for smaller bracketing patterns of size $\mathfrak{b}$ the role of the symbol a in Definition 17, Lemma 2 and Theorem 3 will be replaced by $\mathfrak{b} \geq 2$. Further, we will denote the indices either as pairs $(\mathbf{A}, \mathfrak{b})$ or as triplets $(a, a_s, \mathfrak{b})$ meaning $(\mathbf{A}, \mathfrak{b}) = (a, a_s, \mathfrak{b})$. The set of all power indices will be denoted by $\mathbb{L}$ in resemblance to the logarithmetic of power indices $(L(G), +, \times)$). We will write $(\mathbf{A}, \mathfrak{b}) \xleftarrow{\mathbf{r}} \mathbb{L}$ to denote that $(\mathbf{A}, \mathfrak{b})$ was chosen uniformly at random from the set $\mathbb{L}$ if $\mathfrak{b} \xleftarrow{\mathbf{r}} \mathbb{Z}_{>1}$, $a \xleftarrow{\mathbf{r}} \mathbb{Z}^*_p$, and $a_s \xleftarrow{\mathbf{r}} \mathbb{L}[a, \mathfrak{b}]$ where $\mathbb{L}[a, \mathfrak{b}]$ is the set of all bracketing shapes defined in Definition 19.

Definition 19 (Succinct power indices)**.** *Let integer $\mathfrak{b} \geq 2$ be a base. The power index $(\mathbf{A}, \mathfrak{b}) = (a, a_s, \mathfrak{b})$ is defined as a triplet consisting of two lists and an integer $\mathfrak{b}$. Let $a \in \mathbb{Z}^+$ be represented in base $\mathfrak{b}$ as $a = A_0 + A_1\mathfrak{b} + \ldots + A_k\mathfrak{b}^k$, or in little-endian notation with the list of digits $0 \leq A_i \leq \mathfrak{b} - 1$, as $a = [A_0, A_1, \ldots, A_k]_\mathfrak{b}$. Let the bracketing pattern a_s be represented with a list of digits $a_s =$*

$[P_0, P_1, \ldots, P_k]_{\mathfrak{b}-1}$ *where* $0 \leq P_i \leq \mathfrak{b}-2$ *for* $i = 0, \ldots, k$. *For base* $\mathfrak{b}$ *and* $a = [A_0, A_1, \ldots, A_k]_{\mathfrak{b}} \in \mathbb{Z}^+$ *we denote the set of all possible patterns* a_s *by*

$$\mathbb{L}[a, \mathfrak{b}] = \{a_s \ | a_s = [P_0, \ldots, P_k]_{\mathfrak{b}-1}, \text{ for all } P_j \in \mathbb{Z}_{\mathfrak{b}-1}, j = 0, \ldots, k\}. \tag{38}$$

Definition 20 (Non-associative and non-commutative exponentiation)**.** *For any* $x \in \mathbb{E}_p$ *we define* $x^{(\mathbf{A},\mathfrak{b})}$ *as follows:*

$$\begin{aligned} w_0 &= x, \\ w_i &= R_b(w_{i-1})[P_i], \text{ for } i = 1, \ldots, k, \\ j &= \text{index of the first nonzero digit } A_j \end{aligned} \tag{39}$$

$$x_j = \begin{cases} w_j & , \text{ if } A_j = 1, \\ R_{A_j}(w_j)[P_j \mod (A_j - 1)] & , \text{ if } A_j > 1. \end{cases} \tag{40}$$

for $i = j+1, \ldots, k$

$$x_i = \begin{cases} x_{i-1} & , \text{ if } A_i = 0, \\ w_i * x_{i-1} & , \text{ if } A_i = 1 \text{ and } P_{i-1} \text{ is even}, \\ x_{i-1} * w_i & , \text{ if } A_i = 1 \text{ and } P_{i-1} \text{ is odd}, \\ R_{A_i}(w_i)[P_i \mod (A_i - 1)] * x_{i-1} & , \text{ if } A_i > 1 \text{ and } P_{i-1} \text{ is even}, \\ x_{i-1} * R_{A_i}(w_i)[P_i \mod (A_i - 1)] & , \text{ if } A_i > 1 \text{ and } P_{i-1} \text{ is odd}, \end{cases} \tag{41}$$

$$x^{(\mathbf{A},\mathfrak{b})} = x_k. \tag{42}$$

Proposition 5. *For every base* $\mathfrak{b} \geq 2$, *every* $x \in \mathbb{E}_{p^2}$, *every* $a \in \mathbb{Z}^+$ *and every bracketing pattern* $a_s = [P_0, P_1, \ldots, P_k]_{\mathfrak{b}-1}$, *the value* $x^{(\mathbf{A},\mathfrak{b})}$ *is a product of* a *multiplications of* x

$$x^{(\mathbf{A},\mathfrak{b})} = x^{(a,a_s,\mathfrak{b})} = \underbrace{(x * \ldots (x * x) \ldots)}_{a \text{ copies of } x}.$$

If $O^*_{\mathfrak{b}}$ *denotes the number of operations* $*$ *used to compute the result* $x^{(\mathbf{A},\mathfrak{b})}$, *then its value is given by the following expression*

$$O^*_{\mathfrak{b}} = k(\mathfrak{b}-1) - 1 + \sum_{A_i \neq 0} A_i \quad . \tag{43}$$

Proof. For the first part, let us define a counter $M(i)$ for $i = 0, 1, \ldots, k$, that counts the number of times x was multiplied in the procedure described in equations (39) - (42) for a number $m_i = [A_0, A_1, \ldots, A_i]_{\mathfrak{b}}$ consisting of the first i digits of a.

Let us first notice that every value w_i, for $i = 0, 1, \ldots, k$ in (39) is a result of $\mathfrak{b}^k$ multiplications of x. Now, our counting should start at (40) with the value x_j where j is the index of the first nonzero digit A_j. This means that if $j = 0$, $M(0) = A_0 = m_0$, and if $j > 0$, $M(j) = A_j\mathfrak{b}^j = m_j$. Then, for every next digit A_i for $i = j+1, \ldots, k$ we have that $M(j) = M(j-1) + A_j\mathfrak{b}^j$, where the factor $\mathfrak{b}^j$ comes from the fact that w_j is used either as $1\mathfrak{b}^j$ or as $A_j\mathfrak{b}^j$ if $A_j > 1$ in the part $[P_i \mod (A_i - 1)]$ of (41). The final result is that $M(k) = \sum_{i=0}^{k} A_j\mathfrak{b}^j = a$.

The second part can be proved by counting the number of multiplications performed in parts (39) and in parts (40)-(41). In the expression (39) we have a fixed value $\mathfrak{b}$ - so total number of multiplications is exactly $k(\mathfrak{b}-1)$. Next, in (40) we have either 0 multiplications (if $A_j = 0$), or $A_j - 1$ multiplications (if $A_j > 1$). Finally, in (41), for $i = j+1, \ldots, k$ we apply 1 multiplication with x_{i-1} and $A_i - 1$ multiplications in the expression $R_{A_i}(w_i)[P_i \mod (A_i - 1)]$, which proves the equation (43). □

Proposition 5 ensures that Definition 20 defines a succinct notation for exponentially large power indices with their bracketing shapes since:

1. It is consistent procedure of rising to a power where the number of times that x is multiplied is exactly a, and
2. An efficient procedure where the number of performed operations $*$ is $O(\mathfrak{b} \log_{\mathfrak{b}} a)$.

The contribution of the non-associativity of the operation $*$ to the final result comes from the expressions $w_i = R_b(w_{i-1})[P_i]$ of (39), and from $R_{A_j}(w_j)[P_j \mod (A_j - 1)]$ of (40) and $R_{A_i}(w_i)[P_i \mod (A_i - 1)]$ of (41). On the other hand, the contribution of the non-commutativity of the operation $*$ comes from the multiplication by x_{i-1} from left or from right in (41).

The next Theorem is a direct consequence from Theorem 2 and the Proposition 5.

Theorem 4 (Basic theorem for exponentiation)**.** *Let $x, y \in \mathbb{E}^*_{(p-1)^2}$. For every $(\mathbf{A}, \mathfrak{b}_1), (\mathbf{B}, \mathfrak{b}_2) \overset{\mathbf{r}}{\leftarrow} \mathbb{L}$*

$$(x * y)^{(\mathbf{A},\mathfrak{b}_1)} = x^{(\mathbf{A},\mathfrak{b}_1)} * y^{(\mathbf{A},\mathfrak{b}_1)}, \tag{44}$$

and

$$x^{(\mathbf{A},\mathfrak{b}_1)(\mathbf{B},\mathfrak{b}_2)} = \left(x^{(\mathbf{A},\mathfrak{b}_1)}\right)^{(\mathbf{B},\mathfrak{b}_2)} = \left(x^{(\mathbf{B},\mathfrak{b}_2)}\right)^{(\mathbf{A},\mathfrak{b}_1)} = x^{(\mathbf{B},\mathfrak{b}_2)(\mathbf{A},\mathfrak{b}_1)}. \tag{45}$$

□

Note that different values of bases $\mathfrak{b}_1$ and $\mathfrak{b}_2$ in Theorem 4 do not affect the application of Theorem 2, since Theorem 2 holds true for any bracketing shapes $\mathbf{A}$ and $\mathbf{B}$.

The arithmetic (Logarithmetic) for indices $(\mathbf{A}, \mathfrak{b}_1)$ and $(\mathbf{B}, \mathfrak{b}_2)$ is thus implicitly defined with Theorem 2 and Definition 5. Namely, $(\mathbf{C}, \mathfrak{b}_3) = (\mathbf{A}, \mathfrak{b}_1) + (\mathbf{B}, \mathfrak{b}_2)$ iff $x^{(\mathbf{C},\mathfrak{b}_3)} = x^{(\mathbf{A},\mathfrak{b}_1)} * x^{(\mathbf{B},\mathfrak{b}_2)}$, and $(\mathbf{D}, \mathfrak{b}_4) = (\mathbf{A}, \mathfrak{b}_1)(\mathbf{B}, \mathfrak{b}_2)$ iff $x^{(\mathbf{D},\mathfrak{b}_4)} = (x^{(\mathbf{A},\mathfrak{b}_1)})^{(\mathbf{B},\mathfrak{b}_2)}$ for all $x \in \mathbb{E}_{p^2}$.

Proposition 6. *Let $(\mathbf{A}, \mathfrak{b}_1) = (a, a_s, \mathfrak{b}_1)$ and $(\mathbf{B}, \mathfrak{b}_2) = (b, b_s, \mathfrak{b}_2)$ be two power indices where $a, b \in \mathbb{Z}^+$, and let $(c, c_s, \mathfrak{b}_3) = (\mathbf{C}, \mathfrak{b}_3) = (\mathbf{A}, \mathfrak{b}_1) + (\mathbf{B}, \mathfrak{b}_2)$ and $(d, d_s, \mathfrak{b}_4) = (\mathbf{D}, \mathfrak{b}_3) = (\mathbf{A}, \mathfrak{b}_1)(\mathbf{B}, \mathfrak{b}_2)$. Then, $c = a + b$ and $d = ab$.*

Proof. The proof is by direct application of Theorem 2 for powers in entropic groupoids. □

While we know the values of c and d in $(\mathbf{C}, \mathfrak{b}_3)$ and $(\mathbf{D}, \mathfrak{b}_4)$, we do not know explicitly the shapes c_s and d_s.

Open Problem 4. *For a given entropoid $\mathbb{E}_{p^2}$ and two power indices $(\mathbf{A}, \mathfrak{b}_1)$ and $(\mathbf{B}, \mathfrak{b}_2)$ as in Definition 19 find the explicit forms for $(\mathbf{C}, \mathfrak{b}_1)$ and $(\mathbf{D}, \mathfrak{b}_2)$ such that $(\mathbf{C}, \mathfrak{b}_3) = (\mathbf{A}, \mathfrak{b}_1) + (\mathbf{B}, \mathfrak{b}_2)$ and $(\mathbf{D}, \mathfrak{b}_3) = (\mathbf{A}, \mathfrak{b}_1)(\mathbf{B}, \mathfrak{b}_2)$.*

Despite the efficiency of the procedure for non-associative and non-commutative exponentiation given in Definition 20, we have to notice that for a fixed base $\mathfrak{b}$ there are many bracketing shapes that are omitted and can not be produced with the expressions (39) - (42). Apparently, for $\mathfrak{b} = 2$ the pattern $a_s = [P_0, P_1, \ldots, P_k]_{\mathfrak{b}-1}$ where $0 \leq P_i \leq \mathfrak{b} - 2$ for $i = 0, \ldots, k$ becomes a constant pattern $a_s = [0, 0, \ldots, 0]$, (that we denote shortly by $[\mathbf{0}]$) and only the conditions for even P_{i-1} apply in (41).

Still, we can use the limitations of Definition 20 for good: the fixed pattern related to the base $\mathfrak{b} = 2$ can help us propose heuristics for efficient finding of generators for $(\mathbb{E}^*_{(p-1)^2}, *)$. For that purpose, let us first give several definitions and propositions about the subgroupoids generated by elements of $\mathbb{E}_{p^2}$ with $(\mathbf{A}, 2) = (a, a_s, 2) = (a, [\mathbf{0}], 2)$ indices and with general indices $\mathbf{A}$.

Definition 21. *For every $x = (x_1, x_2) \in \mathbb{E}_{p^2}$, we define the set $\langle x \rangle_2$ as*

$$\langle x \rangle_2 = \{x^{(\mathbf{A},2)} \mid (\mathbf{A}, 2) = (a, [\mathbf{0}], 2), a \in \mathbb{Z}^+\}, \tag{46}$$

and the set $\langle x \rangle$ as

$$\langle x \rangle = \{x^{\mathbf{A}} \mid \mathbf{A} = (a, a_s), a \in \mathbb{Z}^+, a_s \text{ a bracketing shape}\}. \tag{47}$$

We say the subgroupoid $(\langle x \rangle_2, *) = (\mathbb{E}^*_{|\langle x \rangle_2|}, *)$ is a multiplicative cyclic subgroupoid of $\mathbb{E}_{p^2}$ of order $|\langle x \rangle_2|$, and $(\langle x \rangle, *) = (\mathbb{E}^*_{|\langle x \rangle|}, *)$ is a multiplicative subgroupoid of $\mathbb{E}_{p^2}$ of order $|\langle x \rangle|$.

Without a proof which is left as an exercise, we give the following two propositions

Proposition 7. *If $s(x)_2 = |\langle x\rangle_2|$ is the size of the set $\langle x\rangle_2$, then*

$$s(x)_2 \mid 2(p-1), \tag{48}$$

and

$$s_{max}(x)_2 = \max_{x\in\mathbb{E}_{p^2}}(s(x)_2) = 2(p-1). \tag{49}$$

□

Proposition 8. *If $s(x) = |\langle x\rangle|$ is the size of the set $\langle x\rangle$, then*

$$s(x) \mid (p-1)^2 \tag{50}$$

and

$$s_{max}(x) = \max_{x\in\mathbb{E}_p} s(x) = (p-1)^2. \tag{51}$$

□

The problem of finding an explicit analytical expression for the group generators is still an open problem. One version of that problem is the famous Artin's conjecture on primitive roots [20]. However, there are efficient heuristic algorithms that find generators of the multiplicative group of a finite field (for example, see [21][Alg. 4.84, Note 4.82, Alg. 4.86]). We especially point out the efficient algorithm Alg 4.86 in [21] for finding a generator of $\mathbb{F}_p^*$ where p is a safe prime. Inspired by that algorithm, we propose here an efficient heuristic algorithm for finding a generator g of the maximal multiplicative quasigroup $(\mathbb{E}^*_{(p-1)^2}, *)$.

Definition 22. *A safe prime p is a prime of the form $p = 2q+1$ where q is also a prime.*

Conjecture 1. *Let $\mathbb{E}_{p^2}$ be an entropoid defined with a λ bit safe prime p, and let $g \in \mathbb{E}^*_{(p-1)^2}$. If the following conditions are true*

$$g \neq g^{(p,[\mathbf{0}],2)} \tag{52}$$
$$g * g \neq g^{(p-1,[\mathbf{0}],2)} \tag{53}$$
$$(g*(g*g)) \neq g^{(p-2,[\mathbf{0}],2)} \tag{54}$$
$$(g*(g*g)) \neq ((g*g)*g) \tag{55}$$
$$(g*(g*(g*g))) \neq ((g*(g*g))*g), \tag{56}$$

then the probability that g is the generator of $\mathbb{E}^*_{(p-1)^2}$ is

$$Pr[g \text{ is generator of } G^*] > 1-\epsilon, \text{ where } \epsilon < \frac{1}{2^\lambda}. \tag{57}$$

The corresponding algorithm coming from Conjecture 1 is Algorithm 1.

For a detailed demonstration of the Proposition 7, Proposition 8 and Conjecture 1 see the Appendix A.

Open Problem 5. *For a given entropoid $\mathbb{E}_{p^2}$ prove the Conjecture 1 or construct another exact or probabilistic efficient algorithm for finding generator of $\mathbb{E}^*_{(p-1)^2}$.*

A direct consequence from Proposition 7 and Proposition 8 is the following corollary.

Corollary 3. *Let g obtained with the Algorithm 1 be a generator of $\mathbb{E}^*_{(p-1)^2}$, where $p = 2q+1$ is a safe prime. Let $y \xleftarrow{\mathbf{r}} \mathbb{E}^*_{(p-1)^2}$ be a randomly chosen element from $\mathbb{E}^*_{(p-1)^2}$. Then the probability that there exists a power index $(\mathbf{A}, 2)$ such that $y = g^{(\mathbf{A},2)}$ is*

$$Pr(\{\exists(\mathbf{A},2) \text{ and } y = g^{(\mathbf{A},2)}\}) = \frac{1}{q}. \tag{58}$$

Algorithm 1 `Gen`: Find a generator g for the multiplicative quasigroup $(\mathbb{E}^*_{(p-1)^2}, *)$.
Input: λ bit safe prime p and $a_3, a_8, b_2, b_7 \in \mathbb{F}_p$ that define $\mathbb{E}_{p^2}$;
Output: Generator g for $(\mathbb{E}^*_{(p-1)^2}, *)$.

repeat
  Set $Success \leftarrow$ `True`;
  Choose random element $g \in G^*$;
  Set $Success \leftarrow \big(Success$ `and` $(g \neq g^{(p,[\mathbf{0}],2)})\big)$
  Set $Success \leftarrow \big(Success$ `and` $(g * g \neq g^{(p-1,[\mathbf{0}],2)})\big)$
  Set $Success \leftarrow \big(Success$ `and` $(\ (g * (g * g)) \neq g^{(p-2,[\mathbf{0}],2)})\big)$
  Set $Success \leftarrow \big(Success$ `and` $(\ (g * (g * g)) \neq ((g * g) * g))\big)$
  Set $Success \leftarrow \big(Success$ `and` $(\ (g * (g * (g * g))) \neq ((g * (g * g)) * g))\big)$
until $Success$
Return g.

Proof. The proof of this Corollary is just a direct ratio between the size of the cyclic subgroupoid and the size of the maximal entropic quasigroup:

$$\frac{s_{max}(x)_2}{s_{max}(x)} = \frac{2(p-1)}{(p-1)^2} = \frac{1}{q}.$$

□

Having a heuristics for finding generators of the maximal quasigroup $\mathbb{E}^*_{(p-1)^2}$, where $p = 2q+1$ is a safe prime, it would be also very beneficial if we can find generators for the Sylow q-subquasigroup $\mathbb{E}^*_{q^2}$.

Conjecture 2. *Let g be a generator of $\mathbb{E}^*_{(p-1)^2}$, where $p \geq 11$. Then*

$$g_q = (g * (g * (g * ((g * g) * g)))), \tag{59}$$

is the generator of the Sylow q-subquasigroup $\mathbb{E}^*_{q^2}$.

The corresponding algorithm coming from Conjecture 2 is Algorithm 2.

Algorithm 2 `GenQ`: Find a generator g_q for the Sylow q-subquasigroup $\mathbb{E}^*_{q^2}$.
Input: λ bit safe prime p and $a_3, a_8, b_2, b_7 \in \mathbb{F}_p$ that define $\mathbb{E}_{p^2}$;
Output: Generator g_q for $\mathbb{E}^*_{q^2}$.

Set $g \leftarrow$ `Gen`$(\lambda, p, a_3, a_8, b_2, b_7)$;
Set $g_q \leftarrow (g * (g * (g * ((g * g) * g))))$;
Return g_q.

Proposition 9. *Let g be a generator of $\mathbb{E}^*_{(p-1)^2}$. Then for every bracketing shape $(\mathbf{A}, \mathfrak{b}) = (a, a_s, \mathfrak{b}) \in \mathbb{L}$ the following relation hold:*

$$\left(g^{(\mathbf{A},\mathfrak{b})}\right)^{(p-1,[\mathbf{0}],2)} = \begin{cases} \mathbf{1}_*, & \text{if } a \text{ is even,} \\ \boxminus\mathbf{1}_*, & \text{if } a \text{ is odd.} \end{cases} \tag{60}$$

Proof. We use the Theorem 4 and Proposition 6 to deduce that when working with base $b = 2$ we work with a cyclic groupoid. Then from Proposition 7 we know that the order of $x = g^{(\mathbf{A},\mathfrak{b})}$ is $2(p-1)$. That means that the order of $y = x^{(p-1,[\mathbf{0}],2)}$ is 2 i.e. $y^2 = y * y = \mathbf{1}_*$. Thus y belongs to the set of square roots of the left unit $\mathbb{S}(p)$. Now, if y takes any value other than $\mathbf{1}_*$ or $\boxminus\mathbf{1}_*$ it will lead to a result that the order of x is greater than $2(p-1)$ which is a contradiction. This leaves only two possibilities: $y = \mathbf{1}_*$ or $y = \boxminus\mathbf{1}_*$. Again from Theorem 4 and Proposition 6 we get that there must exist an index $(\mathbf{B}, 2) = (b, [\mathbf{0}], 2)$ such that $(\mathbf{B}, 2) = (\mathbf{A}, \mathfrak{b})(p-1, [\mathbf{0}], 2)$. Thus $b = a(p-1)$, and the result y will depend on the parity of a. □

Proposition 10. *Let g_q be a generator of the Sylow q quasigroup $\mathbb{E}^*_{q^2}$. Then for every bracketing shape $(\mathbf{A}, \mathfrak{b}) = (a, a_s, \mathfrak{b}) \in \mathbb{L}$ the following relations hold:*

$$\left(g_q^{(\mathbf{A},\mathfrak{b})}\right)^{(p-1,[\mathbf{0}],2)} = \mathbf{1}_* \tag{61}$$

$$\left(g_q^{(\mathbf{A},\mathfrak{b})}\right)^{(q,[\mathbf{0}],2)} = y, \tag{62}$$

where y belongs in a subset of the set of square roots of the left unit i.e. $y \in \mathbb{S}_q(p) \subseteq \mathbb{S}(p)$, such that $|\mathbb{S}_q(p)| = q$.

Proof. Similar arguments hold for this situation with a distinction that now, the set of all values $x = g_q^{(\mathbf{A},\mathfrak{b})}$ have order q instead of $2q$ that was in the previous case. That, means $x^{(p-1,[\mathbf{0}],2)} = x^{(2q,[\mathbf{0}],2)} = \mathbf{1}_*$ and again putting $y = x^{(q,[\mathbf{0}],2)}$ we have that $y^2 = y * y = \mathbf{1}_*$. So, y must belong to $\mathbb{S}(p)$. If $\mathbb{S}_q(p) \subseteq \mathbb{S}(p)$ is the subset from where y receives its values, its cardinality must be q, otherwise it will generate the whole multiplicative quasigroup $\mathbb{E}^*_{(p-1)^2}$. □

2.4 Dichotomy between odd and even bases

There are a few more issues with the bracketing shapes defined in Definition 20 that need to be discussed. As we see in Proposition 7 and Proposition 8, the choice of the base b influence the size of the generated sets when raising to powers. It also influence the probability an element $x \in \mathbb{E}^*_{(p-1)^2}$ to be an image of some generator raised to some power index i.e. $x = g^{(\mathbf{A},\mathfrak{b})} = g^{(a,a_s,\mathfrak{b})}$. This means, since g is a generator, there is certainly some power index $(\mathbf{A}, \mathfrak{b}) = (a, a_s, \mathfrak{b})$ for which $x = g^{(\mathbf{A},\mathfrak{b})}$, but when distributed over all patterns $a_s = [P_0, P_1, \ldots, P_k]_{\mathfrak{b}-1}$, for some elements x there will be many patterns that give $x = g^{(a,a_s,\mathfrak{b})}$, while for other elements x there will be a few. To formalize this discussion we adapt the approach that Smith proposed in [22] about the Shannon entropy of completely partitioned sets, and the relations between Shannon entropy and several instances of Rényi entropy (such as collision entropy and min-entropy) studied by Cachin in [23] (see also the work of Skórski [24]).

For a given base $2 \leq \mathfrak{b} \leq b_{max}$ and given generator $g \in \mathbb{E}^*_{(p-1)^2}$, let us investigate how the sequence of sets of shapes

$$\mathbb{L}(i) = \{a_s \mid a_s = [P_0, \ldots, P_i]_{\mathfrak{b}-1}, \text{ for all } P_j \in \mathbb{Z}_{\mathfrak{b}-1}, j = 0, \ldots, i\}, \tag{63}$$

each with $(\mathfrak{b}-1)^i$ elements, are partitioned into r_i sets of mutually exclusive subsets

$$\xi_i = \{C_{i,1}, \ldots, C_{i,r_i}\}. \tag{64}$$

The partitioning is done due to the following conditions:

$$\forall a_{s_1}, a_{s_2} \in C_{i,j}, \quad g^{(\mathfrak{b}^{i-1},a_{s_1},\mathfrak{b})} = g^{(\mathfrak{b}^{i-1},a_{s_2},\mathfrak{b})} = g_{i,j}, \tag{65}$$

$$\forall a_{s_1} \in C_{i,j_1}, \text{ and } \forall a_{s_2} \in C_{i,j_2}, \quad g^{(\mathfrak{b}^{i-1},a_{s_1},\mathfrak{b})} \neq g^{(\mathfrak{b}^{i-1},a_{s_2},\mathfrak{b})}, \text{ when } j_1 \neq j_2. \tag{66}$$

As a short notation we write $g^{(\mathfrak{b}^{i-1},\mathbb{L}(i),\mathfrak{b})}$ the set of all powers of g to $\mathfrak{b}^{i-1}$ with all possible shapes $\mathbb{L}(i)$.

If a shape a_s is sampled uniformly at random from $\mathbb{L}(i)$, i.e. if $a_s \xleftarrow{\mathbf{r}} \mathbb{L}(i)$, then $g^{(\mathfrak{b}^{i-1},a_s,\mathfrak{b})}$ determines the set $C_{i,j}$ where it belongs. The probability that a_s belongs to $C_{i,j}$ depends on the number of elements $n_{ij} = |C_{i,j}|$ and is calculated as

$$p_{ij} = p(C_{i,j}) = \frac{n_{ij}}{(b-1)^i}. \tag{67}$$

The Shannon entropy H_1 of the partitioned set ξ_i is defined with

$$H_1(\xi_i) = -\sum_{j=1}^{r_i} p(C_{i,j}) \log p(C_{i,j}). \tag{68}$$

Similarly, the Rényi entropy of order α for ξ_i is defined as

$$H_\alpha(\xi_i) = \frac{1}{1-\alpha} \log \sum_{j=1}^{r_i} p(C_{i,j})^\alpha, \tag{69}$$

with the special instance for $\alpha = 2$ which is called the Collision entropy

$$H_2(\xi_i) = -\log \sum_{j=1}^{r_i} p(C_{i,j})^2. \tag{70}$$

Min entropy is defined as

$$H_\infty(\xi_i) = -\log \max_{C_{i,j} \in \xi_i} p(C_{i,j}) = -\log \frac{\max n_{ij}}{(\mathfrak{b}-1)^i}. \tag{71}$$

The ordering relation (see [23, Lemma 3.2.]) among different entropies is

$$H_\infty(\xi_i) \leq H_2(\xi_i) \leq H_1(\xi_i) \tag{72}$$

Let us point to the fact that the grouping of the shapes is governed by the Narayana numbers given in equation (36) in Theorem 3. Thus, for even bases $\mathfrak{b}$ there are $\mathfrak{b}-1$ classes that partition the set of all possible shapes with cardinality expressed by the Narayana numbers $N(\mathfrak{b}-1, i) = \frac{1}{\mathfrak{b}-1}\binom{\mathfrak{b}-1}{i}\binom{\mathfrak{b}-1}{i-1}$. Since $\mathfrak{b}-1$ in that case is odd, there is one central dominant Narayana number, and there will be one class of shapes that will have a dominant number of members. For example, for $\mathfrak{b} = 6$, the five classes have cardinality $\{1, 10, 20, 10, 1\}$, so the central class is a dominant one with 20 elements. For $\mathfrak{b} = 8$ the seven classes have cardinality $\{1, 21, 105, 175, 105, 21, 1\}$, so the central class is a dominant one with 175 elements. On the other hand, for odd bases $\mathfrak{b}$, we have grouping in even number of $\mathfrak{b}-1$ classes, the sequences of Narayana numbers are completely symmetrical and there is not one but two dominant classes. For example, for $\mathfrak{b} = 7$, the six classes have cardinality $\{1, 15, 50, 50, 15, 1\}$, so two central classes are dominant with 50 elements.

For bigger bases $\mathfrak{b}$ we have observed the same pattern: for even bases $\mathfrak{b} = 2\mathfrak{b}_1$ the values of $\max n_{ij}$ that determine the min entropy H_∞ are increasing with the same exponential speed as the values of $(\mathfrak{b}-1)^i$ increase. Looking at the equation (71) it makes $\frac{\max n_{ij}}{(\mathfrak{b}-1)^i}$ to trend to 1 i.e. H_∞ trends to 0.

For odd bases $\mathfrak{b} = 2\mathfrak{b}_1 + 1$, while there is increase of $\max n_{ij}$ as i increases, the ratio $\frac{\max n_{ij}}{(\mathfrak{b}-1)^i}$ is actually decreasing, which makes H_∞ to increase.

See Appendix B for details about this observed dichotomy between even and odd bases.

We summarize this discussion with the following Conjecture

Conjecture 3. *Let $g \in \mathbb{E}^*_{(p-1)^2}$ be a generator of the maximal multiplicative subgroupoid of a given entropoid $\mathbb{E}_{p^2}$. For every even base $\mathfrak{b} = 2b_1 < b_{max}$ the values $n(\mathfrak{b}, i) = \max n_{ij}$ are given by the following relation:*

$$n(\mathfrak{b}, i) = (\mathfrak{b}-1)\Big((\mathfrak{b}-1)^{i-1} - (\mathfrak{b}-2)^{i-1}\Big), \tag{73}$$

which makes the following relation about the min entropy H_∞

$$H_\infty(\xi_i) = 1 - \left(\frac{\mathfrak{b}-2}{\mathfrak{b}-1}\right)^{i-1}. \tag{74}$$

The discussion so far was about the entropy of the pattern sets $\mathbb{L}(i)$ partitioned to sets ξ_i induced by rising a generator g to a special forms of powers $g^{(a,a_{s_1},\mathfrak{b})}$ where $a = \mathfrak{b}^{i-1}$. This basically means that in the procedure for rising to a power we use only the equations (39) and (40), since the numbers a in that case have in the little-endian notation the following forms $a = [0, 0, \ldots, 1]_\mathfrak{b}$. One might hope that the entropy of ξ for even bases will improve significantly if we work with generic numbers a with a lot of non-zero digits $a = [A_0, A_1, \ldots, A_k]_\mathfrak{b}$. However, that is not the case as it is showed in Figure 2 in Appendix B.

The experiments were conducted by generating a random entropoid $\mathbb{E}_{p^2}$ with safe prime number p with λ bits. After finding a generator g for the multiplicative quasigroup $\mathbb{E}^*_{(p-1)^2}$, we generated one random number $a \in \mathbb{Z}_{(p-1)^2}$, and then we run the procedure of rising to a power $g^{(a,a_s,\mathfrak{b})}$ for random shapes $a_s = [P_0, \ldots, P_i]_{\mathfrak{b}-1}$ until the first collision.

As the size of the finite entropoid $\mathbb{E}_{p^2}$ increases, the collision entropy for different even bases remains constant. On the other hand, with the odd bases the situation is completely different. We see in Figure 3 that as the size of the entropoid increases, the collision entropy increases as well. A loose observation is that for p being λ bits long, the collision entropy is $H_2(\xi) \approx \frac{\lambda}{2}$.

Open Problem 6. *For odd bases $\mathfrak{b}$ find proofs and find tighter bounds for the collision entropy $H_2(\xi)$.*

3 Hard Problems in Entropoid Based Cryptography

We now have enough mathematical understanding and heuristic evidence to precisely formulate several hard problems in entropoid based cryptography, in a similar fashion as the discrete logarithm problem, and computational and decisional Diffie-Hellman problems are defined within the group theory. We will use the notion of negligible function $\texttt{negl} : \mathbb{N} \mapsto \mathbb{R}$ for the function that for every $c \in \mathbb{N}$ there is an integer n_c such that $\texttt{negl}(n) \leq n^{-c}$ for all $n \geq n_c$.

Definition 23 (Discrete Entropoid Logarithm Problem (DELP))**.** *An entropoid $\mathbb{E}_{p^2}$ and a generator g_{q_i} of one of its Silow subquasigroups $\mathbb{E}^*_\nu$ are publicly known. Given an element $y \in \mathbb{E}^*_\nu$ find a power index $(\mathbf{A}, \mathfrak{b})$ such that $y = g_{q_i}^{(\mathbf{A},\mathfrak{b})}$.*

Definition 24 (Computational Entropoid Diffie–Hellman Problem (CEDHP))**.** *An entropoid $\mathbb{E}_{p^2}$ and a generator g_{q_i} of one of its Silow subquasigroups $\mathbb{E}^*_\nu$ are publicly known. Given $g_{q_i}^{(\mathbf{A},\mathfrak{b}_1)}$ and $g_{q_i}^{(\mathbf{B},\mathfrak{b}_2)}$, where $(\mathbf{A}, \mathfrak{b}_1), (\mathbf{B}, \mathfrak{b}_2) \xleftarrow{\mathbf{r}} \mathbb{L}$, compute $g_{q_i}^{(\mathbf{A},\mathfrak{b}_1)(\mathbf{B},\mathfrak{b}_2)}$.*

The similar reduction as with DLP and CDH is true for DELP and CEDHP: CEDHP $\leq$ DELP i.e. CEDHP is no harder than DELP. Namely, if an adversary can solve DELP, it can find $(\mathbf{A}, \mathfrak{b}_1)$ and $(\mathbf{B}, \mathfrak{b}_2)$ and compute $g_{q_i}^{(\mathbf{A},\mathfrak{b}_1)(\mathbf{B},\mathfrak{b}_2)}$.

Definition 25 (Decisional Entropoid Diffie–Hellman Problem (DEDHP))**.** *An entropoid $\mathbb{E}_{p^2}$ and a generator g_{q_i} of one of its Silow subquasigroups $\mathbb{E}^*_\nu$ are publicly known. Given $g_{q_i}^{(\mathbf{A},\mathfrak{b}_1)}$, $g_{q_i}^{(\mathbf{B},\mathfrak{b}_2)}$ and $g_{q_i}^{(\mathbf{C},\mathfrak{b}_3)}$, where $(\mathbf{A}, \mathfrak{b}_1), (\mathbf{B}, \mathfrak{b}_2), (\mathbf{C}, \mathfrak{b}_3) \xleftarrow{\mathbf{r}} \mathbb{L}$, decide if $(\mathbf{C}, \mathfrak{b}_3) = (\mathbf{A}, \mathfrak{b}_1)(\mathbf{B}, \mathfrak{b}_2)$ or $(\mathbf{C}, \mathfrak{b}_3) \xleftarrow{\mathbf{r}} \mathbb{L}$.*

Again, the similar reduction as with classical CDH and DDH, holds here: DEDHP $\leq$ CEDHP i.e. DEDHP is no harder than CEDHP, since if an adversary can solve CEDHP, it will compute $g_{q_i}^{(\mathbf{A},\mathfrak{b}_1)(\mathbf{B},\mathfrak{b}_2)}$ and will compare it with $g_{q_i}^{(\mathbf{C},\mathfrak{b}_3)}$.

As with the classical DDH for the multiplicative group $\mathbb{F}^*_p$ where DDH is easy problem, but for its quadratic residues subgroup $QR(p)$, DDH is hard, we have a similar situation for DEDHP which is stated in the following Lemma.

Lemma 3. *Let $\mathbb{E}_{p^2}$ be an entropoid and g be a generator of its maximal quasigroup $\mathbb{E}^*_{(p-1)^2}$. Then there is an efficient algorithm that solves DEDHP in $\mathbb{E}^*_{(p-1)^2}$.*

Proof. The distinguishing algorithm can be built based on the distinguishing property described in Proposition 9. □

On the other hand, based on Proposition 10 we can give the following plausible conjecture.

Conjecture 4. *Let $\mathbb{E}_{p^2}$ be an entropoid, where $p = 2q+1$ is a safe prime and g_q is the generator of its Sylow q-subquasigroup $\mathbb{E}^*_{q^2}$. Then there is no algorithm $\mathcal{A}$ that solves DEDHP in $\mathbb{E}^*_{q^2}$ with significantly higher advantage over the strategy of uniformly random guesses for making the decisions.*

Theorem 5. *If the DEDHP conjecture is true, then a Diffie-Hellman key exchange protocol over finite entropoids is secure in the Canetti-Krawczyk model of passive adversaries [25].*

We will make here a slight digression, and will relate the classical DLP with another problem over the classical group theory: finding roots. Then we will just translate it for the case of finite entropoids.

Definition 26 (Computational Discrete Root Problem (CDRP)). *A group G of order N and its generator g are publicly known. Given $y = x^b$ and b, where $b \xleftarrow{\mathbf{r}} \mathbb{Z}_N$, compute $x = \sqrt[b]{y}$.*

In general, CDRP is an easy problem. However, there are instances where this problem is still hard, and we will discuss those instances now.

One of the best generic algorithms for solving CDRP is by Johnston [26]. As mentioned there, CDRP can be reduced to solving the DLP in G, i.e. CDRP $\leq$ DLP. First of all b is supposed to divide N, otherwise due to the cyclic nature of the group G, it is a straightforward technique that finds the b-th root: $x = \sqrt[b]{y} = y^{\frac{1}{b} \bmod N}$. Let us denote $g_1 = g^b$, where g is the generator of G. If we have a DLP solver for G, and if there exists a solution for the equation $y = g_1^a$, then DLP will find a efficiently. Then we can compute x as $x = g^a$. Johnston noticed that if $\frac{N}{b}$ is small, then DLP solver will be efficient with a complexity $O(\sqrt{\frac{N}{b}})$. On the other hand if $\frac{N}{b}$ is not that small, Johnston made a reduction to another DLP solver, by heavily using the reach algebraic structure of the finite cyclic groups generated by a gennerator g. The other DLP solver computes a discrete log of $x^{\frac{N}{b^k}}$ using the generator $g^{\frac{N}{b^{k-1}}}$, where k is the largest power of b such that b^k still divides N. The generic complexity of this DLP solver is $O((k-1)\sqrt{b})$. Now let us work in finite field $\mathbb{F}_p$ with the following prime number: $p = 2q^3 + 1$ where q has λ bits. So, if fix the b-th root to be exactly $b = q$, then for the second DLP solver in the Johnson technique we have that $k = 3$, and it has an exponential complexity of $O((k-1)2^{\frac{\lambda}{2}})$.

Definition 27 (Computational Discrete Entropoid Root Problem (CDERP)). *An entropoid $\mathbb{E}_{p^2}$ and a generator g of its multiplicative quasigroups $\mathbb{E}^*_{(p-1)^2}$ are publicly known. Given $y = x^{(\mathbf{B},\mathfrak{b})}$ and $(\mathbf{B}, \mathfrak{b})$, where $x, y \in \mathbb{E}^*_{(p-1)^2}$ and $(\mathbf{B}, \mathfrak{b}) \xleftarrow{\mathbf{r}} \mathbb{L}$, compute $x = \sqrt[(\mathbf{B},\mathfrak{b})]{y}$.*

A similar discussion applies for CDERP that it is not harder than DELP, i.e., CDERP $\leq$ DELP. However, notice that it is not possible directly to extend the Johnson technique to finite entropoids due to the lack of the associative law and because $\mathbb{E}^*_{(p-1)^2}$ is not a cyclic structure. So, at this moment, we can make the following plausible conjecture.

Conjecture 5. *Let $\mathbb{E}_{p^2}$ be an entropoid, where $p = 2q + 1$ is a safe prime with λ bits and g is the generator of its multiplicative quasigroups $\mathbb{E}^*_{(p-1)^2}$. Let $\mathcal{A}$ be an algorithm that solves CDERP in $\mathbb{E}^*_{(p-1)^2}$. Then the probability, over uniformly chosen $(\mathbf{B}, \mathfrak{b}) \xleftarrow{\mathbf{r}} \mathbb{L}$ that $\mathcal{A}(x^{(\mathbf{B},\mathfrak{b})}, (\mathbf{B}, \mathfrak{b})) = x$ is $\mathtt{negl}(\lambda)$.*

We want to emphasize one essential comparison between CDRP in cyclic groups G of order N and CDERP in the multiplicative quasigroups $\mathbb{E}^*_{(p-1)^2}$. CDERP is easy problem for almost all root values b except when $b = q$ in groups that have orders N divisible by q^k where $k \geq 2$. CDERP is conjectured that is hard in $\mathbb{E}^*_{(p-1)^2}$ (that has order $4q^2$) for every randomly selected root $(\mathbf{B}, \mathfrak{b}) \xleftarrow{\mathbf{r}} \mathbb{L}$. The conjecture is based on the fact that currently, there is no developed Logarithmetic for the succinct power indices, but more importantly, that $\mathbb{E}^*_{(p-1)^2}$ is neither a group nor a cyclic structure.

Continuing with the comparisons, let us now compare DELP with the classical DLP in finite groups or in finite fields. Several differences are notable:

1. Operations for DELP are non-associative and non-commutative operations in an entropic quasigroup $(\mathbb{E}^*_\nu, *)$. At the same time, DLP is exclusively defined in groups G that are mostly commutative (there are also DLPs over non-commutative groups, such as the isogenies between elliptical curves defined over the finite fields).

2. All generic algorithms for solving DLP, exclusively without exceptions, use the fact that the group G is cyclic of order N, generated by some generator element g and that for every element $y \in G$ there is a unique index $i \in \{1, \ldots, N\}$ such that $y = g^i$ (in multiplicative notion). In DELP, there are generators for the multiplicative quasigroup $\mathbb{E}^*_{(p-1)^2}$ which has order $(p-1)^2$, but $\mathbb{E}^*_{(p-1)^2}$ is not a cyclic structure since for every $y \in \mathbb{E}^*_{(p-1)^2}$ there are many indices $(\mathbf{A}, \mathfrak{b})$

such that $y = g^{(\mathbf{A},\mathfrak{b})}$, and finding only one of them will solve the DELP. Thus, at first sight, it might seem DELP is an easier task than DLP. However, with Proposition 11 (given below), we show that DLP is no harder than DELP, i.e., DLP $\leq$ DELP.

3. We can take a conservative approach for modeling the complexity of solving DELP, and assume that eventually, an arithmetic (logarithmetic) for the power indices in finite entropoids will be developed (Open problem 4). In that case, an adaptation of the generic algorithms for solving DLP, such as Baby-step giant-step, Pollard rho, Pollard kangaroo, or Pohlig–Hellman for solving DELP, will address a problem with a search space size $N \approx p^2$. Since the complexity of a generic DLP algorithm is $O(\sqrt{N})$ we get that solving DELP with classical algorithms could possibly reach a complexity as low as $O(p)$. Extending this thinking for potential quantum algorithms that will solve DELP, we estimate that their complexity could potentially be as low as $O(p^{1/2})$.

Proposition 11. *Let $p = 2q + 1$ is a safe prime number with λ bits, and let $\mathbb{E}_{p^2}(a_3, a_8, b_2, b_7)$ is a given entropoid. If $\mathcal{A}$ is an efficient algorithm that solves DELP, then there exist an efficient algorithm $\mathcal{B}$ that solves DLP for every subgroup Γ of $\mathbb{F}_p^*$.*

Proof. Let us use the algorithm $\mathcal{A}$ for an entropoid $\mathbb{E}_p(a_3 = 0, a_8 = 1, b_2 = 0, b_7 = 1)$. In that entropoid the operation $*$ becomes

$$(x_1, x_2) * (y_1, y_2) = (x_2 y_1, x_1 y_2).$$

Let $\Gamma \subseteq \mathbb{F}_p^*$ is a nontrivial subgroup. Then, since $p = 2q + 1$ where q is a prime number, Γ is either the quadratic residue group $\Gamma = QR(p)$ of order q or $\Gamma = \mathbb{F}_p^*$ of order $(p - 1)$. Let γ be a generator of Γ. Apparently $\gamma \neq 0$ i.e. $\gamma \neq -\frac{a_3}{a_8}$ and $\gamma \neq -\frac{b_2}{b_7}$. Then from Proposition 4 it follows that $g = (\gamma, \gamma)$ is a generator of some subgroupoid $(\mathbb{E}_\nu^*, *)$, and the operation of exponentiation of g in the entropoid, reduces to exponentiation in a finite field i.e.

$$g^{(\mathbf{A},b)} = g^{(a,a_s,b)} = (\gamma^a, \gamma^a).$$

Thus, for every received γ^a, the algorithm $\mathcal{B}$ constructs the pair (γ^a, γ^a) and asks the algorithm $\mathcal{A}$ to solve it. $\mathcal{A}$ solves it efficiently and returns $(\mathbf{A}, \mathfrak{b})$, from which $\mathcal{B}$ extracts the discrete logarithm a. □

So, in its generality, and currently without the arithmetic for the succinct power indices defined with Definition 19, the best algorithms for solving DELP are practically the generic algorithms for random function inversion, i.e., the generic guessing algorithms. Two of them are given as Algorithm 3 and Algorithm 4.

Algorithm 3 Randomized search solver for (DELP)
Input: Entropoid $\mathbb{E}_{p^2}$, generator g of $\mathbb{E}_{(p-1)^2}^*$ and $y \in \mathbb{E}_{(p-1)^2}^*$.
Output:$(\mathbf{A}, \mathfrak{b})$ such that $y = g^{(\mathbf{A},\mathfrak{b})}$.

1: **repeat**
2: Set $(\mathbf{A}, \mathfrak{b}) \xleftarrow{\mathbf{r}} \mathbb{L}$, where $\mathfrak{b} \geq 3$;
3: **until** $y = g^{(\mathbf{A},\mathfrak{b})}$
4: Return $(\mathbf{A}, \mathfrak{b})$.

3.1 DELP is secure against Shor's quantum algorithm for DLP

Shor's quantum algorithm breaks algorithms that rely on the difficulty of DLP defined over finite commutative groups. One of Shor'salgorithm's crucial components is the part of its quantum circuit that calculates the modular arithmetic for raising g to any power, with the repeated squaring. That part of the Shor's quantum circuit for the repeated squaring works if the related group multiplication operation is associative and commutative. There are no variants of Shor's algorithm or any other quantum algorithm that will work if the underlying algebraic structure is non-commutative. Additionally, DELP is defined over entropoids that are both non-associative and non-commutative.

A designer of a quantum algorithm for solving DELP faces two challenges:

Algorithm 4 Brute force search solver for (DELP)
Input: Entropoid $\mathbb{E}_{p^2}$, generator g of $\mathbb{E}^*_{p^2}$ and $y \in \mathbb{E}^*_{p^2}$.
Output:$(\mathbf{A}, \mathfrak{b})$ such that $y = g^{(\mathbf{A},\mathfrak{b})}$.

```
Set 𝔟 = 2𝔟′ + 1, and 𝔟 ≤ b_max;
for a = 2 to (p − 1)^2 do
    for a_s ∈ 𝕃[a, 𝔟] do
        if y = g^(a,a_s,𝔟) then
            Return (A, 𝔟) = (a, a_s, 𝔟).
        end if
    end for
end for
```

1. Build quantum circuits that implement non-commutative operations of multiplication $*$.
2. Build quantum circuits that perform an unknown pattern of non-associative and non-commutative multiplications $*$, where the number of possible patterns is exponentially high.

We have to note that if the used base is $\mathfrak{b} = 2$, the bracketing pattern is known, and there is a possibility to "reuse" the Shor's circuit. However, as we see from Corollary 3 the probability that the answer from that circuit will be correct is $\frac{1}{q}$. Thus, for q being 128 or 256 bits, it would be a very inefficient quantum algorithm.

4 Concrete instances of Entropoid Based Key Exchange and Digital Signature Algorithms

4.1 Choosing Parameters For a Key Exchange Algorithm Based on DELP

Based on the discussion in Section 3 for achieving post-quantum security levels of 2^{64} and 2^{128} qubit operations we propose finite entropoids $\mathbb{E}_{p^2}$ to use safe prime numbers p with 128 and 256 bits. For estimating the number of $*$ operations for performing one power operation, we use the equation (43). We see that the number depends on the odd base $\mathfrak{b}$. Additionally, $*$ operation with a pre-computation of expressions that involve a_3, a_8, b_2, b_7 can be computed with six modular additions and six modular multiplications in $\mathbb{F}_p$. The expected number of modular operations and the total communication cost for two security levels are given in Table 7. We can see that the number of modular operations increases with $\mathfrak{b}$, while the communication costs in both directions in total are 64 and 128 bytes, respectively.

A formal description of an unauthenticated Diffie-Hellman protocol over finite entropoids is given as follows:

Agreed public parameters

1. Alice and Bob agree on parameters for $\mathbb{E}_{p^2}(a_3, a_8, b_2, b_7)$, where $p = 2q+1$ is a prime number with a size of $\lambda = 128$ or $\lambda = 256$ bits, q is also a prime number, the values $a_3, a_8, b_2, b_7 \in \mathbb{F}_p$ and operations for non-commutative and non-associative multiplication and exponentiation are defined as in Definition 12 and Definition 20
2. Alice and Bob agree on the generator $g_q = \mathtt{GenQ}(\lambda, p, a_3, a_8, b_2, b_7)$
3. Alice and Bob agree on odd base $\mathfrak{b}$

Ephemeral key exchange phase

1. Alice generates a random power index $(\mathbf{A}, \mathfrak{b}) = (a, a_s, \mathfrak{b})$ where $a \in \mathbb{Z}^*_p$
2. Alice computes $K_a = g_q^{(\mathbf{A},\mathfrak{b})}$ and sends it to Bob
3. Bob generates a random power index $(\mathbf{B}, \mathfrak{b}) = (b, b_s, \mathfrak{b})$ where $b \in \mathbb{Z}^*_p$
4. Bob computes $K_b = g_q^{(\mathbf{B},\mathfrak{b})}$ and sends it to Alice
5. Alice computes $K_{ab} = K_b^{(\mathbf{A},\mathfrak{b})}$

6. Bob computes $K_{ba} = K_a^{(\mathbf{B},\mathfrak{b})}$
7. $K_{ab} = K_{ba}$.

	$\lambda = 128$						$\lambda = 256$					
$\mathfrak{b}$:	9	17	33	65	129	257	9	17	33	65	129	257
Additions in $\mathbb{F}_p$	2952	4608	7488	12672	21888	36864	5832	9072	14688	24768	42624	73728
Multiplications in $\mathbb{F}_p$	2952	4608	7488	12672	21888	36864	5832	9072	14688	24768	42624	73728
Sub total[a]	5904	9216	14976	25344	43776	73728	11664	18144	29376	49536	85248	147456
Total[b]	11808	18432	29952	50688	87552	147456	23328	36288	58752	99072	170496	294912
Communication in both directions	$4\lambda = 512$ bits (64 bytes)						$4\lambda = 1024$ bits (128 bytes)					

[a] The number of modular operations for performing one entropoid exponentiation.
[b] Each party need two entropoid exponentiations.

Table 7: Some efficiency metrics for entropoid based Diffie-Hellman key exchange.

4.2 Digital Signature Scheme Based on CDERP

For defining a digital signature over finite entropoids, let us first fix the base $\mathfrak{b} = 257$. The reason for this is the fact that in that case for any power index $(\mathbf{B}, 257) = (b, b_s, 257)$, the bracketing pattern part $b_s = [P_0, \dots P_k]_{256}$ has a very convenient interpretation as a list of bytes. **Remark:** Choosing a base $\mathfrak{b} = 257$ might be too conservative and expensive - making the computations of exponentiation very slow. All proposed algorithms in this and the next sub-section can be carried out with $\mathfrak{b} = 17$, which is also convenient since the bracketing patterns, in that case, become 4-bit nibbles. However, since this is the first introduction of a new signature scheme based on a new hardness problem, we propose more conservative parameters.

In the rest of this sub-section for the power indices instead of writing $(\mathbf{B}, 257)$ we will omit the base part and will simply write the bold letter $\mathbf{B}$ meaning $\mathbf{B} = (b, b_s, 257)$. We also assume that the public parameters $\mathbb{E}_{p^2}(a_3, a_8, b_2, b_7)$, where $p = 2q + 1$ is a prime number with a size of $\lambda \in \{128, 192, 256\}$ bits, q is also a prime number and the values $a_3, a_8, b_2, b_7 \in \mathbb{F}_p$ are known and fixed.

We will use the NIST standardized cryptographic hash function `SHAXXX` where `XXX` $\in \{256, 384, 512\}$ and will use the following notation. Let M be any message, and let $\texttt{SHAXXX}(M) = h_1 || h_2$ where $|h_1| = |h_2| = \frac{\texttt{XXX}}{2}$ bits. We define $truncate_{\mathbb{L}}(\texttt{SHAXXX}(M)) = (h, h_s)$ where $h = h_1$ and the sequence of bits of h_1 are interpreted as a little-endian encoding for a number $h \in \mathbb{Z}_{2^{\frac{\texttt{XXX}}{2}}}$. We partition the bits of h_2 in a list of $k_{max} \in \{16, 24, 32\}$ bytes $h_2 = [P_0, \dots P_{k_{max}-1}]_{256}$. Then we compute $k = \log_{257} h - 1$. Finally we interpret h_s as a list of the first $k+1$ bytes $h_s = [P_0, \dots P_k]_{256}$. If $k < k_{max} - 1$ we truncate the remaining bytes. The set of all such power indices $\mathbb{L}[h, 257]$ is shortly denoted as $\mathbb{L}_{\texttt{257}}$. With this we defined a mapping $Hash_{\texttt{XXX}} : \{0,1\}^* \mapsto \mathbb{L}_{\texttt{257}}$ as follows:

$$Hash_{\texttt{XXX}}(M) = truncate_{\mathbb{L}}(\texttt{SHAXXX}(M)) = (h, h_s, 257). \tag{75}$$

The signature scheme is designed by a Fiat-Shamir transformation [27] of an identification scheme, and it looks similar to Schnorr identification scheme [28], but the security is based on the hardness of CDERP.

Note: In this first version of the paper, we are not giving clear formal proof of the scheme's security in the EUF-CMA security model (Existential Unforgeability under Chosen Message Attack). That proof will be given either as a separate work or in the updated versions of this paper. Instead, we are giving here an initial discussion about the security properties of the proposed signature scheme.

Let us call the key generation algorithm `GenKey`. It is given in Algorithm 5. We assume that a generator g for the quasigroup $\mathbb{E}^*_{(p-1)^2}$ is publicly known and standardized and that a power index $\mathbf{B} = Hash_{\texttt{XXX}}(\text{“}\texttt{abc}\text{”})$ is also predetermined and fixed. The message “`abc`” can be any string such as `"This is a seed message for fixing the value of the public root."`

Algorithm 5 `GenKey`: Generate a key pair (`PrivateKey`, `PublicKey`).
Input: ;
Output: (`PrivateKey`, `PublicKey`).

1: Set $x \xleftarrow{\mathbf{r}} \mathbb{E}^*_{(p-1)^2}$
2: Set `PrivateKey` $= x$
3: Set $y = x^{\mathbf{B}}$
4: Set `PublicKey` $= y$
5: Return (`PrivateKey`, `PublicKey`)

Let us now describe the following identification scheme:

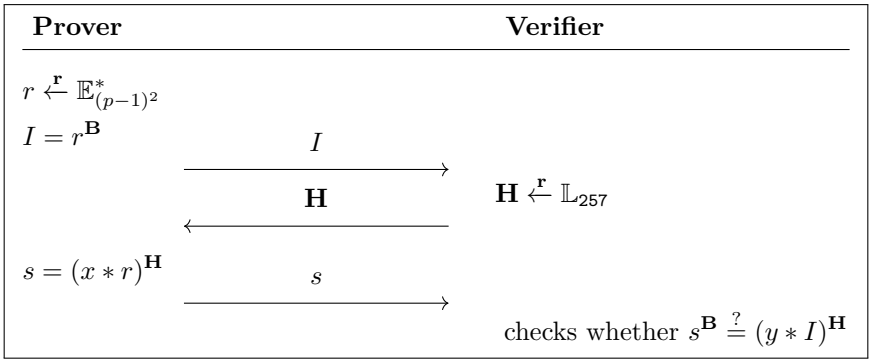


Figure 1: An ID scheme based on the hardness of CDERP

Since I is a uniformly random element from $\mathbb{E}^*_{(p-1)^2}$, and $\mathbf{H}$ is a uniformly random element from $\mathbb{L}_{257}$, the distribution of s is also uniformly random. Thus, an attacker can simulate the transcripts of honest executions by randomly producing triplets $(I, \mathbf{H}, s)$, without a knowledge of the private key. However, since $s = \sqrt[\mathbf{B}]{(y * I)^{\mathbf{H}}}$, if the produced transcripts are verified and true, it implies that the attacker can compute the $\mathbf{B}$-th root, i.e., can solve the CDERP. From this discussion, we give (without proof) the following Theorem:

Theorem 6. *If the computational discrete entropoid root problem is hard in $\mathbb{E}^*_{(p-1)^2}$, then the identification scheme given in Figure 1 is secure.*

The Fiat-Shamir transformation of the identification scheme presented in Figure 1 gives a digital signature scheme with three algorithms: `GenKey` for key generation (already presented in Algorithm 5), `Sign` for digital signing (given in Algorithm 6) and `Verify` for signature verification (given in Algorithm 7).

Algorithm 6 `Sign`: Sign a message.
Input: A message M, and `PrivateKey` $= x$;
Output: (M, σ) where $\sigma = (I, s)$ is the digital signature of the message M.

1: Set $r \xleftarrow{\mathbf{r}} \mathbb{E}^*_{(p-1)^2}$
2: Set $I = r^{\mathbf{B}}$
3: Set $\mathbf{H} = Hash_{\mathtt{XXX}}(I||M)$
4: Set $s = (x * r)^{\mathbf{H}}$
5: Set $\sigma = (I, s)$
6: Return (M, σ)

We check the correctness of the signature scheme as follows:

$$s^{\mathbf{B}} = \left((x * r)^{\mathbf{H}}\right)^{\mathbf{B}} = \left(x^{\mathbf{H}}\right)^{\mathbf{B}} * \left(r^{\mathbf{H}}\right)^{\mathbf{B}} = \left(x^{\mathbf{B}}\right)^{\mathbf{H}} * \left(r^{\mathbf{B}}\right)^{\mathbf{H}} = y^{\mathbf{H}} * I^{\mathbf{H}} = (y * I)^{\mathbf{H}}.$$

An attacker can forge signatures in one of the following ways

Algorithm 7 `Verify`: Verify a digital signature.
Input: A pair (M, σ), and $\texttt{PublicKey} = y$;
Output: `True` or `False`.

1: Set $\mathbf{H} = Hash_{\texttt{xxx}}(I||M)$
2: **if** $s^{\mathbf{B}} \stackrel{?}{=} (y * I)^{\mathbf{H}}$ **then**
3: Return `True`
4: **else**
5: Return `False`
6: **end if**

1. For an existing pair (M, σ), find a second preimage $I||M'$ such that $\mathbf{H} = Hash_{\texttt{xxx}}(I||M')$. In that case (M', σ) is a valid pair.

2. Compute a discrete entropoid $\mathbf{B}$-root of y. In that case the attacker will know the $\texttt{PrivateKey} = x = \sqrt[\mathbf{B}]{y}$.

3. Generate random I, random message M, and compute the corresponding $\mathbf{H} = Hash_{\texttt{xxx}}(I||M)$. Then compute a discrete entropoid $\mathbf{B}$-root $s = \sqrt[\mathbf{B}]{z}$ where $z = (y * I)^{\mathbf{H}}$.

We want to emphasize that finding a collision $\mathbf{H} = Hash_{\texttt{xxx}}(I_1||M_1) = Hash_{\texttt{xxx}}(I_2||M_2)$ is not enough to forge a signature since the attacker in order to produce s has to perform the operation of computing a discrete entropoid $\mathbf{B}$-root for $x = \sqrt[\mathbf{B}]{y}$ or for $s = \sqrt[\mathbf{B}]{z}$.

However, there is a collision finding strategy that will help the attacker to simulate the $\mathbf{B}$-root computation and that is the classical Diffie and Hellman Meet-in-the-middle attack [29]. For achieving the goal of having a probability $1/2$ for finding $\mathbf{B}$-root $s = \sqrt[\mathbf{B}]{z}$ where $z = (y * \sigma_1)^{\mathbf{H}}$ the attacker needs to build two tables T_1 and T_2 where T_1 will contain pairs of elements from $\mathbb{E}^*_{(p-1)^2}$ and T_2 will contain quadruples as described below:

$$T_1 = [(z_i, z_i^{\mathbf{B}}) \mid \text{for } z_i \stackrel{\mathbf{r}}{\leftarrow} \mathbb{E}^*_{(p-1)^2}, i \in \{1, \dots p-1\}] \tag{76}$$

and

$$T_2 = [(I_i, M_i, \mathbf{H}_i, (y * I_i)^{\mathbf{H}_i}) \mid \text{for } I_i \stackrel{\mathbf{r}}{\leftarrow} \mathbb{E}^*_{(p-1)^2}, M_i \stackrel{\mathbf{r}}{\leftarrow} \{0,1\}^*, \mathbf{H}_i = Hash_{\texttt{xxx}}(I_i||M_i), i \in \{1, \dots p-1\}] \tag{77}$$

During the build-up of the tables if the attacker is lucky, it can even find an entry in T_1 that has an item $z_i^{\mathbf{B}} = y$. In that case it found $\mathbf{B}$-root for $x = \sqrt[\mathbf{B}]{y} = z_i$. For the size of T_1 being $p-1$, the probability of this event is $\frac{p-1}{(p-1)^2} = \frac{1}{p-1}$. The attacker can also search for collisions $z_i^{\mathbf{B}} = (y * I_j)^{\mathbf{H}_j}$. If that happens, it would found $\mathbf{B}$-root for $s = \sqrt[\mathbf{B}]{z_i}$. For the size of T_1 and T_2 being $p-1$, the probability of this event is around 0.5. So the memory complexity for this attack is $O(2p) = O(2^{\lambda+1})$ and the time complexity is also $O(2^{\lambda+1})$.

λ, $\mathbb{F}_p$, $p = 2q+1$, $\lambda = \lceil \log p \rceil$	EUF-CMA classical security	EUF-CMA quantum security	PublicKey size (bytes)	PrivateKey size (bytes)	Signature size (bytes)
128	2^{128}	2^{85}	32	32	64
192	2^{192}	2^{128}	48	48	96
256	2^{256}	2^{171}	64	64	128

Table 8: A summary table for the characteristics of the entropoid digital signature scheme based on CDERP

For a similar quantum collision search, we first have to assume that the attacker has overcome the challenges discussed at the end of Section 3.1. While in this situation the associative pattern $\mathbf{B}$ used in table T_1 is known and fixed, for every entry in table T_2 the associative patterns $\mathbf{H}_i$ are entangled with the choices of I_i and M_i, and the output of the hash function $\mathbf{H}_i = Hash_{\texttt{xxx}}(I_i||M_i)$. So, the attacker faces again the two challenges: To build quantum circuits that implement non-commutative operations of multiplication $*$, and to build quantum circuits that perform an exponential number of non-associative and non-commutative multiplication patterns. If it overcomes those challenges, we can assume that the complexity for the quantum search [30] for entropoid collisions could be as low as $O(q^{\frac{2}{3}})$.

As a summary of all discussion in this section, we give a Table 8.

4.3 Digital Signature Scheme Based on reducing CDERP to DELP for Specific Roots B

The signature scheme proposed in the previous sub-section is based on the new assumption about the hardness of CDERP. In case that assumption turns out to be false, we propose here an alternative and more conservative signature scheme that relies its security on the hardness of solving the discrete entropoid logarithm problem.

The more conservative scheme is similar to the one given in the previous sub-section, with the following differences. The entropoid $\mathbb{E}_{p^2}(a_3, a_8, b_2, b_7)$, is defined with a prime number $p = 2q+1$ where the bit size of p is $\lambda \in \{256, 384, 512\}$ bits. The root $\mathbf{B}$ as a public parameter has the following format:[2] $\mathbf{B} = (q, b_s, 257)$, where q is the prime number in the construction of p, and $b_s = \texttt{SHAXXX}(\text{“}\texttt{abc}\text{”})$. The mapping $Hash_{\texttt{XXX}} : \{0,1\}^* \mapsto \mathbb{L}_{q,257}$ is now defined as:

$$Hash_{\texttt{XXX}}(M) = (q, \texttt{SHAXXX}(M), 257). \tag{78}$$

Now, with this different hashing, the algorithms `GenKey`, `Sign` and `Verify` are the same as in the previous case.

The same security analysis applies here, with one additional safety layer. Let us suppose that a Logarithmetic for our succinct power indices will be developed and that the Johnston root-finding algorithm will be adapted for the entropoids $\mathbb{E}_{p^2}$. Since the size of maximal multiplicative groupoid $\mathbb{E}_{(p-1)^2}$ is $4q^2$, and since we have a fixed value q in the public root value $\mathbf{B} = (q, b_s, 257)$, even the hypothetical version of the Johnston algorithm will reduce to finding discrete entropoid logarithm in $\mathbb{E}_{(p-1)^2}$.

The consequences of doubling the bit sizes of p are the doubling of the keys and signatures of the proposed signature scheme and are given in Table 9.

λ, $\mathbb{F}_p$, $p = 2q+1$, $\lambda = \lceil \log p \rceil$	EUF-CMA classical security	EUF-CMA quantum security	PublicKey size (bytes)	PrivateKey size (bytes)	Signature size (bytes)
256	2^{128}	2^{85}	64	64	128
384	2^{192}	2^{128}	96	96	192
512	2^{256}	2^{171}	128	128	256

Table 9: A summary table for the characteristics of the entropoid digital signature scheme based on reduction of CDERP to DELP

5 Conclusions

The algebraic structures that are non-commutative and non-associative known as entropic groupoids that satisfy the *"Palintropic"* property i.e., $x^{\mathbf{AB}} = (x^{\mathbf{A}})^{\mathbf{B}} = (x^{\mathbf{B}})^{\mathbf{A}} = x^{\mathbf{BA}}$ were proposed by Etherington in '40s from the 20th century. Those relations are exactly the Diffie-Hellman key exchange protocol relations used with groups. The arithmetic for non-associative power indices known as Logarithmetic was also proposed by Etherington and later developed by others in the period of '50s-'70s. However, there was never proposed a succinct notation for exponentially large non-associative power indices that will have the property of fast exponentiation similarly as the fast exponentiation is achieved with ordinary arithmetic via the consecutive rising to the powers of two.

In this paper, we defined ringoid algebraic structures $(G, \boxplus, *)$ where $(G, \boxplus)$ is an Abelian group and $(G, *)$ is a non-commutative and non-associative groupoid with an entropic and palintropic subgroupoid which is a quasigroup, and we named those structures as Entropoids. We further defined succinct notation for non-associative bracketing patterns and proposed algorithms for fast exponentiation with those patterns.

Next, by analogy with the developed cryptographic theory of discrete logarithm problems, we defined several hard problems in Entropoid based cryptography, and based on that, we proposed an entropoid Diffie-Hellman key exchange protocol and an entropoid signature schemes. Due to the non-commutativity and non-associativity, the entropoid based cryptographic primitives are supposed to be resistant to quantum algorithms. At the same time, due to the proposed succinct notation for the power indices, the communication overhead in the entropoid based Diffie-Hellman key exchange is very low: for 128 bits of security, 64 bytes in total are communicated in both directions, and for 256 bits of security, 128 bytes in total are communicated in both directions.

In this paper, we also proposed two entropoid based digital signature schemes. The schemes are constructed with the Fiat-Shamir transformation of an identification scheme which security relies on a new hardness assumption: computing roots in finite entropoids is hard. If this assumption withstands the time's test, the first proposed signature scheme has very attractive properties: for the classical security levels between 128 and 256

[2]The same remark about the possibility to use base $\mathfrak{b} = 17$ applies also for this signature scheme.

bits, the public and private key sizes are between 32 and 64, and the signature sizes are between 64 and 128 bytes. The second signature scheme reduces the finding of the roots in finite entropoids to computing discrete entropoid logarithms. In our opinion, this is a safer but more conservative design, and the price is in doubling the key sizes and the signature sizes.

We give a proof-of-concept implementation in SageMath 9.2 for all proposed algorithms and schemes in Appendix C.

We hope that this paper will initiate further research in Entropoid Based Cryptography.

References

[1] EF Harding. The probabilities of rooted tree-shapes generated by random bifurcation. *Advances in Applied Probability*, pages 44–77, 1971.

[2] Michael F Dacey. A non-associative arithmetic for shapes of channel networks. In *Proceedings of the June 4-8, 1973, national computer conference and exposition*, pages 503–508, 1973.

[3] IMH Etherington. On Non-Associative Combinations. *Proceedings of the royal society of Edinburgh*, 59:153–162, 1940.

[4] Abraham Robinson. On non-associative systems. *Proceedings of the Edinburgh Mathematical Society*, 8(3):111–118, 1949.

[5] Helen Popova. Logarithmetics of non associative algebras. *Annexe Thesis Digitisation Project 2019 Block 22*, 1951.

[6] Trevor Evans. Nonassociative number theory. *The American Mathematical Monthly*, 64(5):299–309, 1957.

[7] H Minc. Theorems on nonassociative number theory. *The American Mathematical Monthly*, 66(6):486–488, 1959.

[8] Dorothy Bollman et al. Formal nonassociative number theory. *Notre Dame Journal of Formal Logic*, 8(1-2):9–16, 1967.

[9] MW Bunder. Commutative non-associative number theory. *Proceedings of the Edinburgh Mathematical Society*, 20(2):133–136, 1976.

[10] Henryk Trappmann. *Arborescent numbers: higher arithmetic operations and division trees*. PhD thesis, Universität Potsdam, 2007.

[11] IMH Etherington. Transposed algebras. *Proceedings of the Edinburgh Mathematical Society*, 7(2):104–121, 1945.

[12] IMH Etherington. Groupoids with additive endomorphisms. *The American Mathematical Monthly*, 65(8P1):596–601, 1958.

[13] IMH Etherington. Quasigroups and cubic curves. *Proceedings of the Edinburgh Mathematical Society*, 14(4):273–291, 1965.

[14] Neil JA Sloane. The on-line encyclopedia of integer sequences. In *Towards mechanized mathematical assistants*, pages 130–130. Springer, 2007.

[15] DC Murdoch. Quasi-groups which satisfy certain generalized associative laws. *American Journal of Mathematics*, 61(2):509–522, 1939.

[16] William Stein and David Joyner. Sage: System for algebra and geometry experimentation. *Acm Sigsam Bulletin*, 39(2):61–64, 2005.

[17] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison Wesley Longman Publishing Co., Inc., 1991.

[18] Anna B Romanowska and Jonathan DH Smith. On Hopf algebras in entropic Jónsson-Tarski varieties. *Bulletin of the Korean Mathematical Society*, 52(5):1587–1606, 2015.

[19] Jonathan DH Smith. Sylow theory for quasigroups. *Journal of Combinatorial Designs*, 23(3):115–133, 2015.

[20] D. R. HEATH-BROWN. ARTIN'S CONJECTURE FOR PRIMITIVE ROOTS. *The Quarterly Journal of Mathematics*, 37(1):27–38, 03 1986.

[21] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[22] Jonathan DH Smith. Some observations on the concepts of information-theoretic entropy and randomness. *Entropy*, 3(1):1–11, 2001.

[23] Christian Cachin. *Entropy measures and unconditional security in cryptography*. PhD thesis, ETH Zurich, 1997.

[24] Maciej Skórski. Shannon entropy versus Renyi entropy from a cryptographic viewpoint. In *IMA International Conference on Cryptography and Coding*, pages 257–274. Springer, 2015.

[25] Ran Canetti and Hugo Krawczyk. Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2001.

[26] Anna M. Johnston. A Generalized q-th Root Algorithm. In Robert Endre Tarjan and Tandy J. Warnow, editors, *SODA*, pages 929–930. ACM/SIAM, 1999.

[27] Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 186–194, London, UK, 1987. Springer-Verlag.

[28] Claus P. Schnorr. Efficient Identification and Signatures for Smart Cards. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

[29] Whitfield Diffie and Martin E Hellman. Special feature exhaustive cryptanalysis of the nbs data encryption standard. *Computer*, 10(6):74–84, 1977.

[30] Seiichiro Tani. Claw finding algorithms using quantum walk. *Theoretical Computer Science*, 410(50):5285–5297, 2009.

A Examples for $\mathbb{E}_{11^2}$, $\mathbb{E}_{13^2}$, $\mathbb{E}_{19^2}$ and $\mathbb{E}_{23^2}$

Let us define the following finite entropoids:

1. $\mathbb{E}_{11^2}(a_3 = 9, a_8 = 1, b_2 = 8, b_7 = 9)$, which has $\mathbf{0}_* = (2, 4)$, $\mathbf{1}_* = (7, 5)$ and $x * y = (x_1, x_2) * (y_1, y_2) = (x_2y_1 + 9x_2 + 7y_1 + 10, \quad 9x_1y_2 + 8x_1 + 4y_2 + 10)$;
2. $\mathbb{E}_{13^2}(a_3 = 10, a_8 = 2, b_2 = 3, b_7 = 9)$, which has $\mathbf{0}_* = (8, 4)$, $\mathbf{1}_* = (11, 11)$ and $x * y = (x_1, x_2) * (y_1, y_2) = (2x_2y_1 + 10x_2 + 5y_1 + 7, \quad 9x_1y_2 + 3x_1 + 6y_2 + 6)$;
3. $\mathbb{E}_{19^2}(a_3 = 18, a_8 = 11, b_2 = 14, b_7 = 10)$, which has $\mathbf{0}_* = (7, 10)$, $\mathbf{1}_* = (9, 17)$ and $x*y = (x_1, x_2)*(y_1, y_2) = (11x_2y_1 + 18x_2 + 4y_1 + 17, \quad 10x_1y_2 + 14x_1 + 6y_2 + 7)$;
4. $\mathbb{E}_{23^2}(a_3 = 15, a_8 = 13, b_2 = 9, b_7 = 14)$, which has $\mathbf{0}_* = (13, 1)$, $\mathbf{1}_* = (18, 17)$ and $x*y = (x_1, x_2)*(y_1, y_2) = (13x_2y_1 + 15x_2 + 10y_1 + 21, \quad 14x_1y_2 + 9x_1 + 2y_2 + 22)$.

We present their elements as square $p \times p$ arrays as in Example 1 and in cells with coordinates (x_1, x_2) we put the values that are the size of the sets $\langle x \rangle_2$ and $\langle x \rangle$.

The colored cells has the following meaning:

1. The yellow highlighted elements do not belong to the multiplicative quasigroup $(\mathbb{E}^*_{p^2}, *)$;
2. The green highlighted elements $x = (x_1, x_2)$ are generators for both the maximal length cyclic subgroupoid $\langle x \rangle_2$ with $2(p-1)$ elements, and are generators of the multiplicative quasigroup $(\mathbb{E}^*_{p^2}, *)$;
3. The red highlighted elements $x = (x_1, x_2)$ are generators for a maximal length cyclic subgroupoid $\langle x \rangle_2$ with $2(p-1)$ elements, **but are not** generators of the multiplicative quasigroup $(\mathbb{E}^*_{p^2}, *)$.
4. Blue highlighted element for $\mathbb{E}_{11^2}$ and $\mathbb{E}_{23^2}$ denote the generators of the Sylow q-subgroupoids with 25 and 121 elements (11 and 23 are "safe primes" i.e. $11 = 2 \times 5 + 1$ and $23 = 2 \times 11 + 1$).

	0	1	2	3	4	5	6	7	8	9	10
0	20	4	10	20	2	10	20	2	5	10	20
1	10	10	20	10	2	20	10	20	20	4	2
2	2	2	2	2	1	2	2	2	2	2	2
3	20	20	5	20	2	10	20	10	10	2	4
4	10	2	20	10	2	20	10	4	20	20	10
5	20	20	2	20	2	10	4	10	10	5	20
6	4	20	10	20	2	10	20	5	2	10	20
7	20	20	10	4	2	1	20	10	10	10	20
8	10	10	20	2	2	4	10	20	20	20	10
9	2	10	20	10	2	20	10	20	4	20	10
10	10	10	4	10	2	20	2	20	20	20	10

Table 10: The size of the sets $\langle x \rangle_2$ for $x \in \mathbb{E}_{11^2}$

	0	1	2	3	4	5	6	7	8	9	10
0	20	20	25	100	2	25	100	5	5	25	100
1	50	50	20	50	2	100	10	100	100	20	10
2	2	2	2	2	1	2	2	2	2	2	2
3	100	100	5	100	2	25	20	25	25	5	20
4	10	10	100	50	2	100	50	20	20	100	50
5	100	100	5	100	2	25	20	25	25	5	20
6	20	20	25	100	2	25	100	5	5	25	100
7	100	100	25	4	2	1	100	25	25	25	100
8	50	50	100	2	2	4	50	100	100	100	50
9	10	10	100	50	2	100	50	20	20	100	50
10	50	50	20	50	2	100	10	100	100	20	10

Table 11: The size of the sets $\langle x \rangle$ for $x \in \mathbb{E}_{11^2}$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	12	6	24	2	2	4	24	12	6	8	24	24	8
1	12	2	8	6	2	12	8	4	6	24	24	24	24
2	6	4	8	12	2	6	8	2	12	24	24	24	24
3	6	12	24	4	2	2	24	6	12	8	24	24	8
4	24	8	2	24	2	24	4	8	24	12	12	6	3
5	8	24	12	24	2	24	6	24	8	6	2	4	12
6	2	12	24	12	2	6	24	6	4	24	8	8	24
7	24	24	12	8	2	8	6	24	24	2	6	12	4
8	2	2	2	2	1	2	2	2	2	2	2	2	2
9	24	24	3	8	2	8	12	24	24	4	12	6	2
10	4	6	24	6	2	12	24	12	2	24	8	8	24
11	8	24	6	24	2	24	12	24	8	12	4	1	6
12	24	8	4	24	2	24	2	8	24	6	6	12	12

Table 12: The size of the sets $\langle x \rangle_2$ for $x \in \mathbb{E}_{13^2}$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	36	12	48	12	2	12	48	12	36	48	144	144	48
1	36	12	48	12	2	12	48	12	36	48	144	144	48
2	36	12	48	12	2	12	48	12	36	48	144	144	48
3	36	12	48	12	2	12	48	12	36	48	144	144	48
4	144	48	3	48	2	48	12	48	144	12	36	9	3
5	16	144	36	144	2	144	18	144	16	18	2	4	36
6	4	36	144	36	2	36	144	36	4	144	16	16	144
7	144	48	12	48	2	48	6	48	144	6	18	36	12
8	2	2	2	2	1	2	2	2	2	2	2	2	2
9	144	48	3	48	2	48	12	48	144	12	36	9	3
10	4	36	144	36	2	36	144	36	4	144	16	16	144
11	16	144	9	144	2	144	36	144	16	36	4	1	9
12	144	48	12	48	2	48	6	48	144	6	18	36	12

Table 13: The size of the sets $\langle x \rangle$ for $x \in \mathbb{E}_{13^2}$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	18	36	18	36	36	12	36	6	2	36	2	18	4	12	9	6	18	18	36
1	6	12	18	36	12	36	4	18	9	36	2	18	36	36	2	18	6	18	36
2	18	36	3	12	36	36	36	18	18	4	2	2	36	36	18	18	18	6	12
3	6	12	18	36	4	36	12	18	18	36	2	18	36	36	6	9	2	18	36
4	36	18	4	6	18	18	18	36	36	6	2	12	18	18	36	36	36	12	2
5	36	18	12	2	18	18	18	36	36	6	2	12	18	18	36	36	36	4	6
6	18	36	18	36	36	4	36	6	6	36	2	18	12	12	18	2	9	18	36
7	2	2	2	2	2	2	2	2	2	2	1	2	2	2	2	2	2	2	2
8	36	18	36	18	18	2	18	12	12	18	2	36	6	6	36	4	36	36	18
9	18	36	6	4	36	36	36	18	18	12	2	6	36	36	18	18	18	1	12
10	18	36	2	12	36	36	36	18	18	12	2	3	36	36	18	18	18	6	4
11	12	6	36	18	2	18	6	36	36	18	2	36	18	18	12	36	4	36	18
12	36	18	12	6	18	18	18	36	36	2	2	4	18	18	36	36	36	12	6
13	12	6	36	18	6	18	2	36	36	18	2	36	18	18	4	36	12	36	18
14	36	18	36	18	18	6	18	12	4	18	2	36	2	6	36	12	36	36	18
15	9	36	18	36	36	12	36	2	6	36	2	18	12	4	18	6	18	18	36
16	4	2	36	18	6	18	6	36	36	18	2	36	18	18	12	36	12	36	18
17	2	4	18	36	12	36	12	9	18	36	2	18	36	36	6	18	6	18	36
18	36	18	36	18	18	6	18	4	12	18	2	36	6	2	36	12	36	36	18

Table 14: The size of the sets $\langle x \rangle_2$ for $x \in \mathbb{E}_{19^2}$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	27	108	81	324	108	108	36	27	9	324	2	81	36	108	9	27	27	81	324
1	27	108	81	324	108	108	36	27	9	324	2	81	36	108	9	27	27	81	324
2	81	324	3	36	324	324	324	81	81	12	2	3	324	324	81	81	81	9	12
3	27	108	81	324	36	36	108	27	27	324	2	81	108	108	27	9	9	81	324
4	324	162	12	18	162	162	162	324	324	6	2	12	162	162	324	324	324	36	6
5	324	162	36	2	162	162	162	324	324	18	2	36	162	162	324	324	324	4	18
6	27	108	81	324	36	36	108	27	27	324	2	81	108	108	27	9	9	81	324
7	2	2	2	2	2	2	2	2	2	2	1	2	2	2	2	2	2	2	2
8	108	54	324	162	18	18	54	108	108	162	2	324	54	54	108	36	36	324	162
9	81	324	9	4	324	324	324	81	81	36	2	9	324	324	81	81	81	1	36
10	81	324	3	36	324	324	324	81	81	12	2	3	324	324	81	81	81	9	12
11	108	54	324	162	18	18	54	108	108	162	2	324	54	54	108	36	36	324	162
12	324	162	12	18	162	162	162	324	324	6	2	12	162	162	324	324	324	36	6
13	108	54	324	162	54	54	18	108	36	162	2	324	18	54	36	108	108	324	162
14	108	54	324	162	54	54	18	108	36	162	2	324	18	54	36	108	108	324	162
15	9	36	81	324	108	108	108	9	27	324	2	81	108	36	27	27	27	81	324
16	36	18	324	162	54	54	54	36	108	162	2	324	54	18	108	108	108	324	162
17	9	36	81	324	108	108	108	9	27	324	2	81	108	36	27	27	27	81	324
18	36	18	324	162	54	54	54	36	108	162	2	324	54	18	108	108	108	324	162

Table 15: The size of the sets $\langle x \rangle$ for $x \in \mathbb{E}_{19^2}$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
0	44	2	22	22	22	22	44	22	44	2	11	44	44	22	22	44	4	22	44	22	44	44	44
1	4	2	2	22	11	22	44	22	44	22	22	44	44	22	22	44	44	22	44	22	44	44	44
2	2	2	4	44	44	44	22	44	22	44	44	22	22	44	44	22	22	44	22	44	22	22	22
3	22	2	44	44	44	44	22	44	22	4	44	22	22	44	44	22	2	44	22	44	22	22	22
4	44	2	22	22	22	22	44	22	44	11	2	44	44	22	22	4	44	22	44	22	44	44	44
5	44	2	22	11	22	22	44	22	44	22	22	4	44	22	2	44	44	22	44	22	44	44	44
6	22	2	44	44	44	44	2	44	22	44	44	22	22	44	44	22	22	44	22	4	22	22	22
7	44	2	22	2	22	22	44	22	44	22	22	44	44	22	11	44	44	22	44	22	44	44	4
8	22	2	44	44	44	44	22	44	2	44	44	22	22	44	44	22	22	4	22	44	22	22	22
9	44	2	11	22	2	22	44	22	44	22	22	44	44	22	22	44	44	22	44	22	44	4	44
10	44	2	22	22	22	2	44	22	44	22	22	44	44	22	22	44	44	22	44	11	4	44	44
11	44	2	22	22	22	22	44	2	44	22	22	44	44	11	22	44	44	22	4	22	44	44	44
12	44	2	22	22	22	22	44	11	44	22	22	44	4	2	22	44	44	22	44	22	44	44	44
13	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
14	22	2	44	44	44	44	22	44	22	44	44	22	2	4	44	22	22	44	22	44	22	22	22
15	22	2	44	44	44	44	22	4	22	44	44	22	22	44	44	22	22	44	2	44	22	22	22
16	22	2	44	44	44	4	22	44	22	44	44	22	22	44	44	22	22	44	22	44	2	22	22
17	22	2	44	44	4	44	22	44	22	44	44	22	22	44	44	22	22	44	22	44	22	2	22
18	44	2	22	22	22	22	44	22	4	22	22	44	44	22	22	44	44	1	44	22	44	44	44
19	22	2	44	4	44	44	22	44	22	44	44	22	22	44	44	22	22	44	22	44	22	22	2
20	44	2	22	22	22	11	4	22	44	22	22	44	44	22	22	44	44	22	44	2	44	44	44
21	22	2	44	44	44	44	22	44	22	44	44	2	22	44	4	22	22	44	22	44	22	22	22
22	22	2	44	44	44	44	22	44	22	44	4	22	22	44	44	2	22	44	22	44	22	22	22

Table 16: The size of the sets $\langle x \rangle_2$ for $x \in \mathbb{E}_{23^2}$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
0	484	2	121	121	121	121	484	121	484	11	11	484	484	121	121	44	44	121	484	121	484	484	484
1	44	2	11	121	11	121	484	121	484	121	121	484	484	121	121	484	484	121	484	121	484	44	484
2	22	2	44	484	44	484	242	484	242	484	484	242	242	484	484	242	242	484	242	484	242	22	242
3	242	2	484	484	484	484	242	484	242	44	44	242	242	484	484	22	22	484	242	484	242	242	242
4	484	2	121	121	121	121	484	121	484	11	11	484	484	121	121	44	44	121	484	121	484	484	484
5	484	2	121	11	121	121	484	121	484	121	121	44	484	121	11	484	484	121	484	121	484	484	44
6	242	2	484	484	484	44	22	484	242	484	484	242	242	484	484	242	242	484	242	44	22	242	242
7	484	2	121	11	121	121	484	121	484	121	121	44	484	121	11	484	484	121	484	121	484	484	44
8	242	2	484	484	484	484	242	484	2	484	484	242	242	484	484	242	242	4	242	484	242	242	242
9	44	2	11	121	11	121	484	121	484	121	121	484	484	121	121	484	484	121	484	121	484	44	484
10	484	2	121	121	121	11	44	121	484	121	121	484	484	121	121	484	484	121	484	11	44	484	484
11	484	2	121	121	121	121	484	11	484	121	121	484	44	11	121	484	484	121	44	121	484	484	484
12	484	2	121	121	121	121	484	11	484	121	121	484	44	11	121	484	484	121	44	121	484	484	484
13	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
14	242	2	484	484	484	484	242	44	242	484	484	242	22	44	484	242	242	484	22	484	242	242	242
15	242	2	484	484	484	484	242	44	242	484	484	242	22	44	484	242	242	484	22	484	242	242	242
16	242	2	484	484	484	44	22	484	242	484	484	242	242	484	484	242	242	484	242	44	22	242	242
17	22	2	44	484	44	484	242	484	242	484	484	242	242	484	484	242	242	484	242	484	242	22	242
18	484	2	121	121	121	121	484	121	4	121	121	484	484	121	121	484	484	1	484	121	484	484	484
19	242	2	484	44	484	484	242	484	242	484	484	22	242	484	44	242	242	484	242	484	242	242	22
20	484	2	121	121	121	11	44	121	484	121	121	484	484	121	121	484	484	121	484	11	44	484	484
21	242	2	484	44	484	484	242	484	242	484	484	22	242	484	44	242	242	484	242	484	242	242	22
22	242	2	484	484	484	484	242	484	242	44	44	242	242	484	484	22	22	484	242	484	242	242	242

Table 17: The size of the sets $\langle x \rangle$ for $x \in \mathbb{E}_{23^2}$

B Observation for the dichotomy between even and odd bases

Let us use the following Entropoid $\mathbb{E}_{49223^2}(a_3 = 33170, a_8 = 13052, b_2 = 12476, b_7 = 19648)$, which has $\mathbf{0}_* = (20898, 8427)$, $\mathbf{1}_* = (29739, 25115)$ and $x * y = (x_1, x_2) * (y_1, y_2) = (13052x_2y_1 + 33170x_2 + 24201y_1 + 34725, \quad 19648x_1y_2 + 12476x_1 + 14362y_2 + 19210)$. For a generator let us use $g = (21287, 34883)$.

For $\mathfrak{b} = 3$ and for $i = 2, 3, 4$ one can check that these are the following outcomes:

$i = 2$, $\mathbb{L}(i) = \{[0,0], [1,0], [0,1], [1,1]\}$.
$g^{(3,\mathbb{L}(i),\mathfrak{b})} = \{(22143, 3374), (22143, 3374), (9735, 2125), (9735, 2125)\}$. As we can see there are only two outcomes: $g_{2,1} = (22143, 3374)$ and $g_{2,2} = (9735, 2125)$, so we get $r_2 = 2$, $\xi_2 = \{C_{i,1}, C_{i,r_2}\}$, where $C_{i,1} = \{[0,0], [1,0]\}$ and $C_{i,2} = \{[0,1], [1,1]\}$. From this we get $H_\infty(\xi_2) = H_2(\xi_2) = H_1(\xi_2) = 1$.

$i = 3$, $\mathbb{L}(i) = \{[0,0,0], [1,0,0], [0,1,0], [1,1,0], [0,0,1], [1,0,1], [0,1,1], [1,1,1]\}$.
$g^{(9,\mathbb{L}(i),\mathfrak{b})} = \{(12320, 26593), (12320, 26593), (28416, 42082), (28416, 42082), (28416, 42082), (28416, 42082), (12320, 26593), (12320, 26593)\}$. As we can see there are again only two outcomes: $g_{3,1} = (12320, 26593)$ and $g_{3,2} = (28416, 42082)$. So, again we have $r_3 = 2$, and now $\xi_3 = \{C_{i,1}, C_{i,r_3}\}$, where $C_{i,1} = \{[[0,0,0], [1,0,0], [0,1,1], [1,1,1]]\}$ and $C_{i,2} = \{[0,1,0], [1,1,0], [0,0,1], [1,0,1]\}$. From this we get $H_\infty(\xi_3) = H_2(\xi_3) = H_1(\xi_3) = 1$.

$i = 4$, $\mathbb{L}(i) = \{[0,0,0,0], [1,0,0,0], \ldots, [1,1,1,1]\}$.
$g^{(27,\mathbb{L}(i),\mathfrak{b})} = \{(42159, 1249), (42159, 1249), (46373, 13249), \ldots\}$. One can see that again there are only two outcomes: $g_{4,1} = (42159, 1249)$ and $g_{4,2} = (46373, 13249)$. So, $r_4 = 2$, and now $\xi_4 = \{C_{i,1}, C_{i,r_4}\}$, where $|C_{i,1}| = 8$ and $|C_{i,2}| = 8$. From this we get $H_\infty(\xi_4) = H_2(\xi_4) = H_1(\xi_4) = 1$.

For $\mathfrak{b} = 4$ and for $i = 2, 3, \ldots 9$ a summary table of the obtained calculations is given in Table 18. As we can see, the sets ξ_i are partitioned always in 3 subsets, but the entropies tend to 0 as i is increasing.

$\mathfrak{b} = 4$											
$i = 2$						$i = 3$					
r_i	$g_{i,j}$	n_{ij}	H_∞	H_2	H_1	r_i	$g_{i,j}$	n_{ij}	H_∞	H_2	H_1
3	(2847, 43103)	3	1.585	1.585	1.585	3	(37676, 4224)	6	0.848	1.295	1.436
	(12306, 3250)	3					(14769, 4826)	6			
	(43283, 29857)	3					(27843, 29019)	15			
	Σ	9					Σ	27			
$i = 4$						$i = 5$					
r_i	$g_{i,j}$	n_{ij}	H_∞	H_2	H_1	r_i	$g_{i,j}$	n_{ij}	H_∞	H_2	H_1
3	(9873, 27342)	12	0.507	0.891	1.173	3	(10067, 22108)	24	0.317	0.592	0.914
	(44897, 4336)	12					(6487, 4975)	24			
	(31057, 15755)	57					(22832, 44737)	195			
	Σ	81					Σ	243			
$i = 6$						$i = 7$					
r_i	$g_{i,j}$	n_{ij}	H_∞	H_2	H_1	r_i	$g_{i,j}$	n_{ij}	H_∞	H_2	H_1
3	(19981, 22570)	48	0.204	0.391	0.694	3	(43901, 19938)	96	0.133	0.258	0.517
	(35514, 19869)	48					(2901, 22539)	96			
	(31074, 13020)	633					(1892, 14331)	1995			
	Σ	729					Σ	2187			
$i = 8$						$i = 9$					
r_i	$g_{i,j}$	n_{ij}	H_∞	H_2	H_1	r_i	$g_{i,j}$	n_{ij}	H_∞	H_2	H_1
3	(42829, 25216)	192	0.087	0.171	0.380	3	(9873, 27342)	384	0.057	0.114	0.277
	(18292, 35754)	192					(31057, 15755)	384			
	(44720, 23968)	6177					(44897, 4336)	18915			
	Σ	6561					Σ	19683			

Table 18: A summary table of calculations with base $\mathfrak{b} = 4$

For odd bases $\mathfrak{b} = 5$ and $\mathfrak{b} = 7$ things are getting more interesting, as the number r_i of partition parts for the sets ξ_i is increasing as i increases. The entropies H_∞, H_2 and H_1 are also increasing.

On the other hand, for even bases $\mathfrak{b} = 6$ and $\mathfrak{b} = 8$, we see that the number of partitions increases as well, but the entropies H_∞, H_2 and H_1, after an initial increase, start to decrease (and trend to zero).

The summary of the calculations is given in Table 19, where in order to keep a reasonable table space we omitted the representatives $g_{i,j}$ of the partitioned classes.

$\mathfrak{b}=5$							$\mathfrak{b}=6$						
i	r_i	$\min n_{ij}$	$\max n_{ij}$	H_∞	H_2	H_1	i	r_i	$\min n_{ij}$	$\max n_{ij}$	H_∞	H_2	H_1
2	4	4	5	2.000	2.000	2.000	2	5	5	5	2.322	2.322	2.322
3	6	8	16	2.000	2.415	2.500	3	7	10	45	1.474	2.276	2.543
4	8	16	48	2.415	2.678	2.811	4	9	20	305	1.035	1.841	2.439
5	10	32	192	2.415	2.871	3.031	5	11	40	1845	0.760	1.429	2.218
6	12	64	640	2.678	3.023	3.198	6	13	80	10505	0.573	1.104	1.961
7	14	128	2560	2.678	3.148	3.333	7	15	160	57645	0.439	0.857	1.704
8	16	256	8960	2.871	3.255	3.447	8	17	320	308705	0.340	0.669	1.464
9	18	512	35840	2.871	3.348	3.544	9	19	640	1625445	0.265	0.524	1.247
$\mathfrak{b}=7$							$\mathfrak{b}=8$						
i	r_i	$\min n_{ij}$	$\max n_{ij}$	H_∞	H_2	H_1	i	r_i	$\min n_{ij}$	$\max n_{ij}$	H_∞	H_2	H_1
2	6	6	6	2.585	2.585	2.585	2	7	7	7	2.807	2.807	2.807
3	12	12	24	3.170	3.433	3.503	3	13	14	91	1.914	3.054	3.408
4	20	24	144	3.170	3.971	4.124	4	21	28	889	1.433	2.622	3.548
5	30	48	576	3.755	4.360	4.580	5	31	56	7735	1.120	2.146	3.467
6	42	96	2880	4.018	4.666	4.933	6	43	112	63217	0.896	1.751	3.278
7	56	192	17280	4.018	4.918	5.220	7	57	224	496951	0.729	1.437	3.039
8	72	384	80640	4.380	5.132	5.459	8	73	448	3805249	0.599	1.188	2.780
9	90	768	430080	4.550	5.318	5.664	9	91	896	28596295	0.497	0.988	2.521

Table 19: A summary table of calculations with bases $\mathfrak{b}=5,6,7,8$

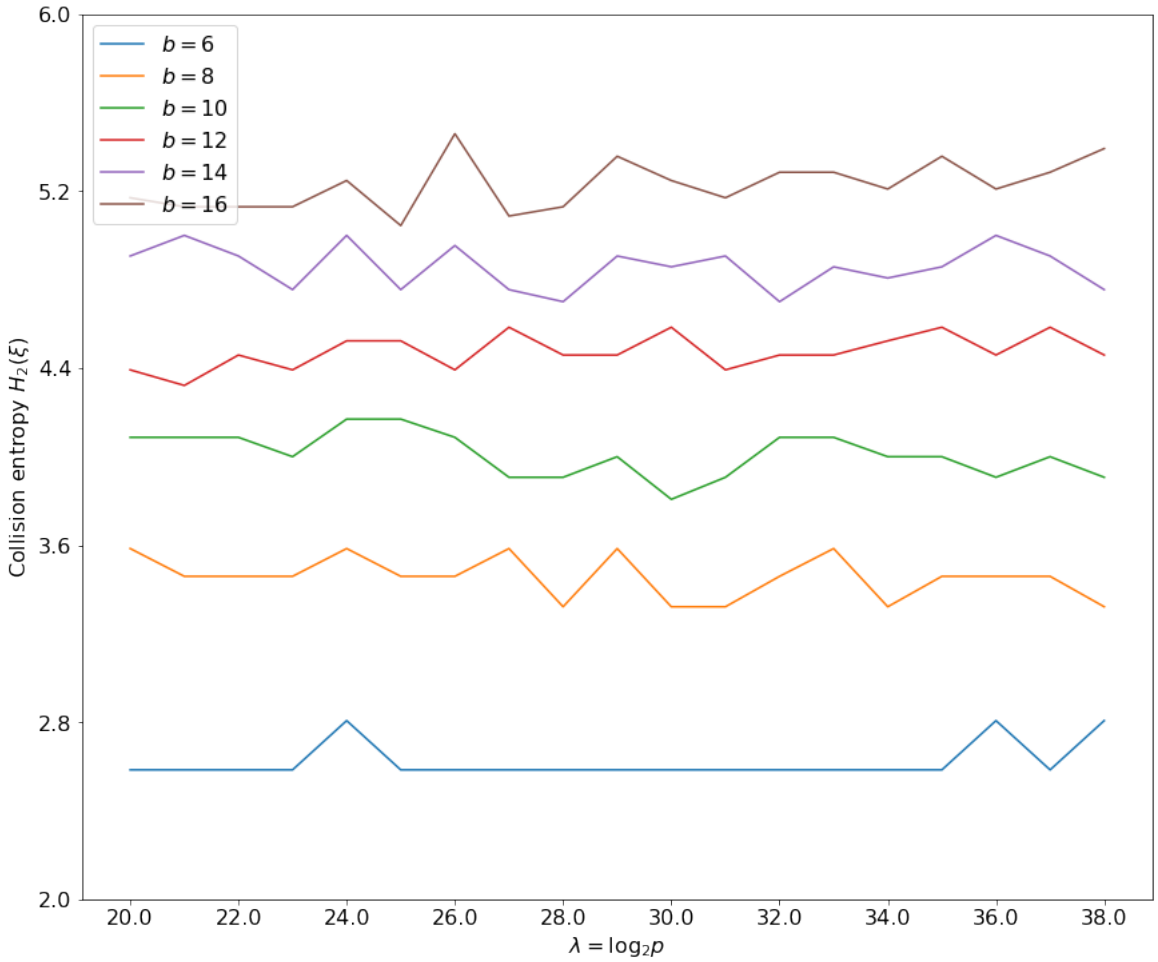


Figure 2: Collision entropy $H_2(\xi)$ experimentally calculated for bases $\mathfrak{b}=6,8,\ldots,16$, for different entropoids $\mathbb{E}_p$ where $\lambda=\log_2 p$ varies in the interval $[20,38]$. The collision entropy values were computed as an average of 100 experiments.

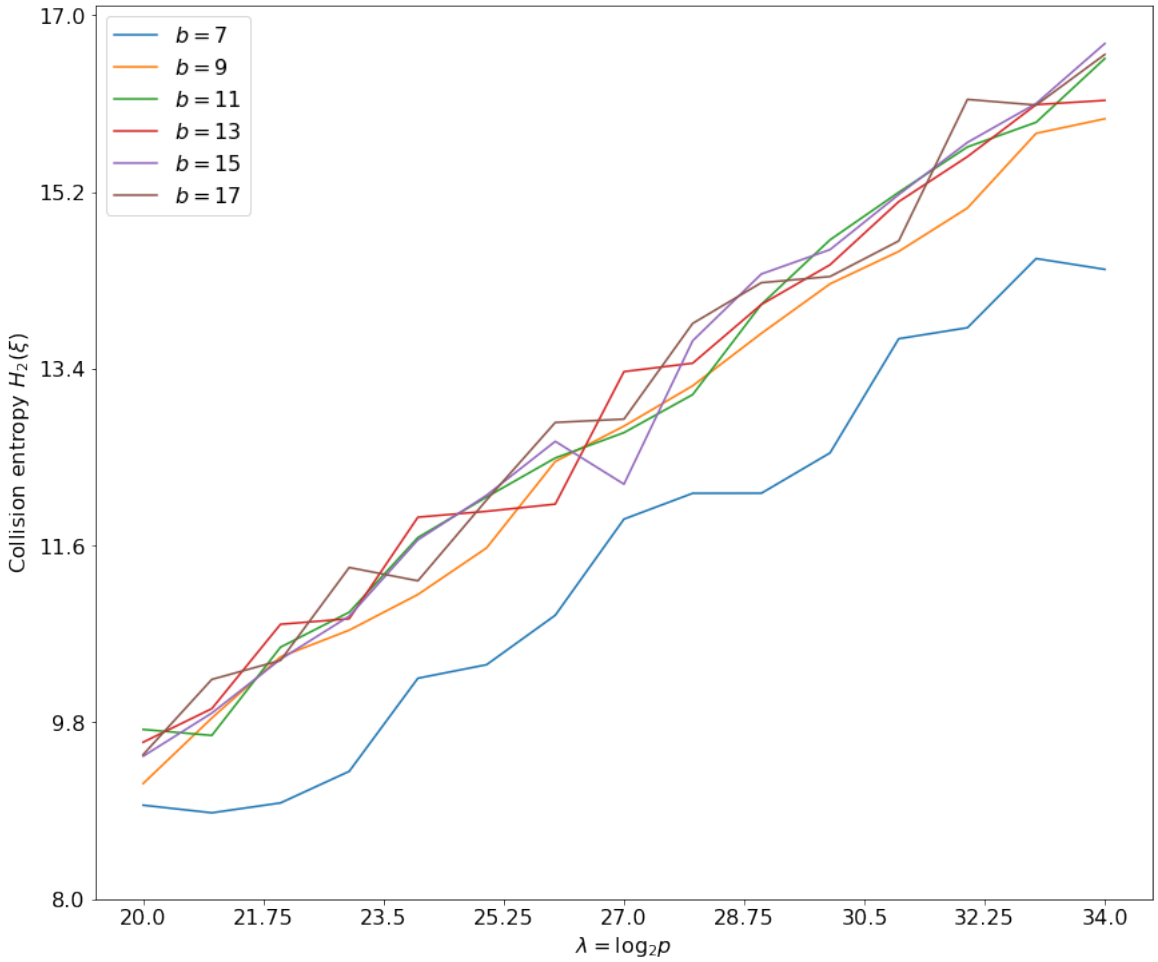


Figure 3: Collision entropy $H_2(\xi)$ experimentally calculated for bases $\mathfrak{b} = 7, 9, \ldots, 17$, for different entropoids $\mathbb{E}_p$ where $\lambda = \log_2 p$ varies in the interval $[20, 34]$. The collision entropy values were computed as an average of 10 experiments.

Appendix C
Proof-of-concept SageMath Jupyter implementation of the algorithms given in "Entropoid Based Cryptography"

In [1]:
```
#============================================================================================
#    Proof-of-concept SageMath 9.2 implementation of the algorithms
#    proposed in "Entropoid Based Cryptography"
#
#    Copyright 2020, 2021:
#    Danilo Gligoroski, <danilog@ntnu.no>
#    Department of Information Security and Communication Technology,
#    Faculty of Information Technology and Electrical Engineering,
#    Norwegian University of Science and Technology - NTNU,
#    O.S.Bragstads plass 2B,
#    N-7034 Trondheim, Norway
#
#    This program is free software: you can redistribute it and/or modify
#    it under the terms of the GNU General Public License as published by
#    the Free Software Foundation, either version 3 of the License, or
#    (at your option) any later version.
#
#    This program is distributed in the hope that it will be useful,
#    but WITHOUT ANY WARRANTY; without even the implied warranty of
#    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#    GNU General Public License for more details.
#
#    You should have received a copy of the GNU General Public License
#    along with this program.  If not, see <https://www.gnu.org/licenses/>.
#============================================================================================
```

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

from matplotlib.colors import LogNorm
from numpy import ma
from matplotlib import ticker, cm
from sage.plot.contour_plot import ContourPlot

import re
import scipy
import scipy.stats

import random

# Finds a prime p = 2 q + 1 such that q is the the smallest prime
# larger than 2^(lambda_bits - 2) + 2^(lambda_bits - 3)
def get_fixed_safe_prime(nbits = 128):
    # nbits should be >= 3
    global p, q
    q = next_prime(2^(nbits-2) + 2^(nbits - 3))
    search = True
    while search:
        p = 2*q + 1
        search = not(is_prime(p))
        q = next_prime(q)
    return p

def get_random_safe_prime(nbits = 128):
    # nbits should be >= 3
    global p, q
    q = random_prime(2^(nbits-2), false, 2^(nbits - 3))
    search = True
    while search:
        p = 2*q + 1
        search = not(is_prime(p))
        q = next_prime(q)
    return p

def get_random_entropoid_coefficients():
    global aa1, aa3, aa4, aa8, bb1, bb2, bb5, bb7, E_Zero, E_left_unit
    global F, p
    aa3 = F.random_element()
    aa8 = 0
    while aa8 == 0:
        aa8 = F.random_element();
    bb2 = F.random_element()
    bb7 = 0
    while bb7 == 0:
        bb7 = F.random_element();

    aa1 = (aa3*(aa8*bb2 - bb7))/(aa8*bb7)
    aa4 = (aa8*bb2)/bb7
    bb1 = -((bb2*(aa8 - aa3*bb7))/(aa8*bb7))
```

```
    bb5 = (aa3*bb7)/aa8

    E_Zero = vector((-(aa3/aa8), -(bb2/bb7)))
    E_left_unit = vector((-(aa3/aa8) + 1/bb7, 1/aa8 - bb2/bb7))
    pass

def mmult( X, Y ):
    global aa1, aa3, aa4, aa8, bb1, bb2, bb5, bb7
    f1 = aa8*X[1]*Y[0] + aa3*X[1] + aa4*Y[0] + aa1;
    f2 = bb7*X[0]*Y[1] + bb2*X[0] + bb5*Y[1] + bb1;
    return(vector((f1, f2)))

def pplus(x, y):
    global E_Zero
    return(vector(x)+vector(y)-vector(E_Zero))

def mminus(x, y):
    global E_Zero
    return(vector(x)-vector(y)+vector(E_Zero))

def iinverse(x):
    global aa3, bb2, bb7, aa8
    return(vector(((1 - aa3 * bb2 - aa3 * bb7 * x[1])/(aa8 * (bb2 + bb7 * x[1])), \
           (1 - aa3 * bb2 - aa8 * bb2 * x[0])/(bb7 * (aa3 + aa8 * x[0])))))

def Narayana_sequence(base):
    seq = [binomial(base - 1, i) * binomial(base - 1, i - 1)/(base - 1) for i in range(1, base)]
    return(seq)

def Catalan_number(b):
    return((binomial(2*b, b))/(b+1))

def get_base_max(p):
    maxvalue = (p-1)^2
    bmax = 1
    while Catalan_number(bmax) < maxvalue:
        bmax += 1
    return(bmax)

def get_all_bracketing(n, base):
    nndigits = n.digits(base-1)
    nrdigits = len(nndigits)
    maxvalue = ZZ([base - 2 for i in range(nrdigits-1)], base-1)
    PPall = [ZZ¶ Yp(i).digits(base-1, padto=nrdigits-1) for i in range(maxvalue+1)]
    return(PPall)

# A procedure that given an integer n, and a base b
# produces an array of integers with values between 0 and b - 2 (including b - 2)
# The length of the array is the same as the length of n interpreted in the base base
def get_random_bracketing(n, base):
    bracket_pattern = [ random.randint(0, base - 2) for i in range(len(n.digits(base))) ]
    return(bracket_pattern)

# A procedure that produces a random power index
# The range of a is between 1 and p - 1
# Once a is obtained, the bracketing is obtained by calling a procedure
# for random bracketing a in the used base.
def get_random_power_index(base):
    global p
    a = ZZ(random.randint(1, p))
    a_pattern = get_random_bracketing(a, base)
    return([a, a_pattern, base])

# If we want to generate the bracketing sapes we can use this procedure
def allassocrules(n):
    rules = ['mmult(gg,gg)']
    for i in range(n-2):
        qq = rules[0]
        tmp = 'mmult(' + qq + ',gg)'
        for j in range(len(rules)):
            rules[j] = 'mmult(gg,' + rules[j] + ')'
        rules.append(tmp)
    return (rules)

# Let us generate all basic bracketing shapes as defined in
# Definition 17 of the paper "Entropoid Based Cryptography"
limitbases = 258
AssocRulesForDifferentBases = [[] for i in range(limitbases)]
for i in range(2, limitbases):
    AssocRulesForDifferentBases[i] = allassocrules(i)

# This is a computational procedure that gives the same result as
# we would have evaluated the symbolic bracketing
AssocRulesForDifferentBases[base][digit]
def AssocRuleForBaseAndDigit(gg, base, digit):
    j = 0
    tmp = mmult(gg, gg)
```

```
    if digit == 0:
        while j < base - 2:
            tmp = mmult(gg, tmp)
            j += 1
        return(tmp)
    else:
        while j < digit - 1:
            tmp = mmult(gg, tmp)
            j += 1
        if j <= base - 2:
            tmp = mmult(tmp, gg)
            j += 1
        while j < base - 2:
            tmp = mmult(gg, tmp)
            j += 1
        return(tmp)


# This is the central and most important procedure for
# non-abelian and non-associative exponentiation of g
# to the power index nn = [n, n_pattern, base]
def nanapow(g, nn):
    n = nn[0]
    pattern = nn[1]
    base = nn[2]

    digitss = n.digits(base)

    # Compute equations (39) from the paper
    gg = g
    powlist = [gg]
    for i in range(1, len(pattern)):
        gg = AssocRuleForBaseAndDigit(gg, base, pattern[i])
        powlist.append(gg)

    # find the index of the first non-zero element
    ii = digitss.index(next(filter(lambda x: x!=0, digitss)))

    # Compute equations (40) from the paper
    if digitss[ii] == 1:
        result = powlist[ii]
    else:
        gg = powlist[ii]
        result = AssocRuleForBaseAndDigit(gg, digitss[ii], pattern[ii]%(digitss[ii] -1))

    # Compute equations (41) from the paper
    ii += 1
    while ii<len(digitss):
        if digitss[ii] != 0:
            if digitss[ii] == 1:
                tmp = powlist[ii]
            else:
                gg = powlist[ii]
                tmp = AssocRuleForBaseAndDigit(gg, digitss[ii], pattern[ii]%(digitss[ii]- 1))
            if pattern[ii-1]%2 == 0:
                result = mmult(tmp, result)
            else:
                result = mmult(result, tmp)
        ii += 1

    return(result)

# A procedure that finds a generator of the multiplicative quasigroup (\mathbb{E}_{(p-1)^2}^*, *)
def find_generator_p():
    global F, p
    base = 2
    search = True
    trials = 0
    while search:
        trials += 1
        Success = True
        g = vector((F.random_element(), F.random_element()))
        Success = Success and (g != nanapow(g, [ZZ(p), get_random_bracketing(ZZ(p),base), base]))
        tmp2 = mmult(g, g)
        Success = Success and (tmp2 != nanapow(g, [ZZ(p-1), get_random_bracketing(ZZ(p),base), base]))
        tmp3  = mmult(g, tmp2)
        Success = Success and (tmp3 != nanapow(g, [ZZ(p-2), get_random_bracketing(ZZ(p),base), base]))
        Success = Success and (tmp3 != mmult(tmp2, g))
        Success = Success and (mmult(g, tmp3) != mmult(tmp3, g))
        search = not(Success)
    return(g)



# A procedure that finds a generator of the Sylow q-subquasigroup (\mathbb{E}_{q^2}^*,*)
def find_generator_q():
    global F, p
```

```
    g = find_generator_p()
    return( mmult(g,mmult(g,mmult(g,mmult(mmult(g,g),g)))) )

def GenParameters_Diffie_Hellman(lambda_bits):
    global aa1, aa3, aa4, aa8, bb1, bb2, bb5, bb7, E_Zero, E_left_unit
    global F, p, g, g_q

    p = get_fixed_safe_prime(lambda_bits) # or get_random_safe_prime(lambda_bits)
    print("p = ", p)
    print("log_2(p) = ", ceil(log(p, 2)))
    print()

    F = GF(p)
    print("Information about the finite field: ", F)
    print()

    # Get some appropriate but random coefficients that define E_{p^2}
    get_random_entropoid_coefficients()
    print("aa1 = ", aa1)
    print("aa3 = ", aa3)
    print("aa4 = ", aa4)
    print("aa8 = ", aa8)
    print("bb1 = ", bb1)
    print("bb2 = ", bb2)
    print("bb5 = ", bb5)
    print("bb7 = ", bb7)
    print()

    # Some of the characteristic elements in the Entropoid
    print("E_Zero = ", E_Zero)
    print("E_left_unit = ", E_left_unit)
    E_minus_one = mminus(E_Zero, E_left_unit)
    print("E_minus_one = ", E_minus_one)
    print()

    # g is a generator for the multiplicative quasigroup of E_{p^2}
    g = find_generator_p()
    print("g = ", g)
    print()

    # g_q is a generator for the Silow q subquasigroup
    g_q = find_generator_q()
    print("g_q = ", g_q)
    print()

import binascii
import hashlib
def concatenate_entropoid_element_and_message(I, M, I_size_in_bytes):
    byte_array = b''
    for i in range(len(I)):
        byte_array = byte_array + int(I[i]).to_bytes(I_size_in_bytes, 'little')
    byte_array = byte_array + bytearray(M.encode())
    return byte_array

# Hash to power indices procedures for Signatures01
def Hash512_to_power_index(byte_array, base):
    if not(base in {17, 257}):
        print("ERROR: Base should be either 17 or 257")
        return(-2)
    ss = hashlib.sha512(byte_array).hexdigest()
    ss_first_half = ss[:64];
    bb1 = ZZ(int('0x'+ss_first_half,16))
    bb1_in_base = bb1.digits(base);
    ss_second_half = ss[64:];
    bb2 = ZZ(int('0x'+ss_second_half,16))
    bb2_in_base_min_1 = bb2.digits(base - 1)
    bb2_in_base_min_1 = bb2_in_base_min_1[:len(bb1_in_base)]
    return([bb1, bb2_in_base_min_1, base])

def Hash384_to_power_index(byte_array, base):
    if not(base in {17, 257}):
        print("ERROR: Base should be either 17 or 257")
        return(-2)
    ss = hashlib.sha384(byte_array).hexdigest()
    ss_first_half = ss[:48];
    bb1 = ZZ(int('0x'+ss_first_half,16))
    bb1_in_base = bb1.digits(base);
    ss_second_half = ss[48:];
    bb2 = ZZ(int('0x'+ss_second_half,16))
    bb2_in_base_min_1 = bb2.digits(base - 1)
    bb2_in_base_min_1 = bb2_in_base_min_1[:len(bb1_in_base)]
    return([bb1, bb2_in_base_min_1, base])

def Hash256_to_power_index(byte_array, base):
    if not(base in {17, 257}):
        print("ERROR: Base should be either 17 or 257")
        return(-2)
```

```
    ss = hashlib.sha256(byte_array).hexdigest()
    ss_first_half = ss[:32];
    bb1 = ZZ(int('0x'+ss_first_half,16))
    bb1_in_base = bb1.digits(base);
    ss_second_half = ss[32:];
    bb2 = ZZ(int('0x'+ss_second_half,16))
    bb2_in_base_min_1 = bb2.digits(base - 1)
    bb2_in_base_min_1 = bb2_in_base_min_1[:len(bb1_in_base)]
    return([bb1, bb2_in_base_min_1, base])

def GenKey_Signature01(lambda_bits, base):
    global aa1, aa3, aa4, aa8, bb1, bb2, bb5, bb7, E_Zero, E_left_unit
    global F, p, g, g_q, B, PrivateKey, PublicKey, x, y

    if not(lambda_bits in {128, 192, 256}):
        print("ERROR: lambda_bits should be either 128 or 192 or 256")
        return(-1)
    if not(base in {17, 257}):
        print("ERROR: Base should be either 17 or 257")
        return(-2)

    p = get_fixed_safe_prime(lambda_bits) # or get_random_safe_prime(lambda_bits)
    print("p = ", p)
    print("log_2(p) = ", ceil(log(p, 2)))
    print()

    F = GF(p)
    print("Information about the finite field: ", F)
    print()

    # Get some appropriate but random coefficients that define E_p
    get_random_entropoid_coefficients()
    print("aa1 = ", aa1)
    print("aa3 = ", aa3)
    print("aa4 = ", aa4)
    print("aa8 = ", aa8)
    print("bb1 = ", bb1)
    print("bb2 = ", bb2)
    print("bb5 = ", bb5)
    print("bb7 = ", bb7)
    print()

    # Some of the characteristic elements in the Entropoid
    print("E_Zero = ", E_Zero)
    print("E_left_unit = ", E_left_unit)
    E_minus_one = mminus(E_Zero, E_left_unit)
    print("E_minus_one = ", E_minus_one)
    print()

    # g is a generator for the multiplicative quasigroup of E_p
    g = find_generator_p()
    print("g = ", g)
    print()

    # g_q is a generator for the Silow q subquasigroup
    g_q = find_generator_q()
    print("g_q = ", g_q)
    print()

    # generate a random public parameter B
    # B is a power index
    M = "This is a seed message for fixing the value of the public root."
    if lambda_bits == 128:
        B = Hash256_to_power_index(bytearray(M.encode()), base)
    if lambda_bits == 192:
        B = Hash384_to_power_index(bytearray(M.encode()), base)
    if lambda_bits == 256:
        B = Hash512_to_power_index(bytearray(M.encode()), base)

    print("Public power index B = ", B)
    print()

    # generate the PrivateKey x
    # skip the coordinates of E_Zero
    x0 = E_Zero[0]
    while x0 == E_Zero[0]:
        x0 = F.random_element();
    x1 = E_Zero[1]
    while x1 == E_Zero[1]:
        x1 = F.random_element();
    x = vector((x0, x1))
    PrivateKey = x
    print("PrivateKey = ", PrivateKey)
    print()

    # generate the PublicKey y
    y = nanapow(x, B)
```

```
    PublicKey = y
    print("PublicKey = ", PublicKey)
    print()

def Sign_Signature01(PrivateKey, Message, base, lambda_bits):
    global B
    # generate a random r
    # skip the coordinates of E_Zero
    r0 = E_Zero[0]
    while r0 == E_Zero[0]:
        r0 = F.random_element();
    r1 = E_Zero[1]
    while r1 == E_Zero[1]:
        r1 = F.random_element();
    r = vector((r0, r1))

    I = nanapow(r, B)
    I_and_Message = concatenate_entropoid_element_and_message(I, Message, lambda_bits//8)

    if lambda_bits == 128:
        H = Hash256_to_power_index(I_and_Message, base)
    if lambda_bits == 192:
        H = Hash384_to_power_index(I_and_Message, base)
    if lambda_bits == 256:
        H = Hash512_to_power_index(I_and_Message, base)

    s = nanapow(mmult(PrivateKey, r), H)

    signature = (I, s)
    return signature

def Verify_Signature01(PublicKey, Message, signature, base, lambda_bits):
    global B

    I = signature[0]
    I_and_Message = concatenate_entropoid_element_and_message(I, Message, lambda_bits//8)

    if lambda_bits == 128:
        H = Hash256_to_power_index(I_and_Message, base)
    if lambda_bits == 192:
        H = Hash384_to_power_index(I_and_Message, base)
    if lambda_bits == 256:
        H = Hash512_to_power_index(I_and_Message, base)

    s_to_B = nanapow(signature[1], B)
    y_mult_I_to_H = nanapow(mmult(PublicKey, I), H)

    if s_to_B == y_mult_I_to_H:
        return True
    else:
        return False

# Hash to power indices for Signatures02
def Hash512_to_power_index_02(byte_array, base):
    if not(base in {17, 257}):
        print("ERROR: Base should be either 17 or 257")
        return(-2)
    ss = hashlib.sha512(byte_array).hexdigest()
    bb_s = ZZ(int('0x'+ss,16))
    bb_s_in_base_min_1 = bb_s.digits(base - 1)
    bb_s_in_base_min_1 = bb_s_in_base_min_1[:len(q.digits(base))]
    return([q, bb_s_in_base_min_1, base])

def Hash384_to_power_index_02(byte_array, base):
    if not(base in {17, 257}):
        print("ERROR: Base should be either 17 or 257")
        return(-2)
    ss = hashlib.sha384(byte_array).hexdigest()
    bb_s = ZZ(int('0x'+ss,16))
    bb_s_in_base_min_1 = bb_s.digits(base - 1)
    bb_s_in_base_min_1 = bb_s_in_base_min_1[:len(q.digits(base))]
    return([q, bb_s_in_base_min_1, base])

def Hash256_to_power_index_02(byte_array, base):
    if not(base in {17, 257}):
        print("ERROR: Base should be either 17 or 257")
        return(-2)
    ss = hashlib.sha256(byte_array).hexdigest()
    bb_s = ZZ(int('0x'+ss,16))
    bb_s_in_base_min_1 = bb_s.digits(base - 1)
    bb_s_in_base_min_1 = bb_s_in_base_min_1[:len(q.digits(base))]
    return([q, bb_s_in_base_min_1, base])

def GenKey_Signature02(lambda_bits, base):
    global aa1, aa3, aa4, aa8, bb1, bb2, bb5, bb7, E_Zero, E_left_unit
    global F, q, p, g, g_q, B, PrivateKey, PublicKey, x, y
```

```
    if not(lambda_bits in {256, 384, 512}):
        print("ERROR: lambda_bits should be either 128 or 192 or 256")
        return(-1)
    if not(base in {17, 257}):
        print("ERROR: Base should be either 17 or 257")
        return(-2)

    p = get_fixed_safe_prime(lambda_bits) # or get_random_safe_prime(lambda_bits)
    print("p = ", p)
    print("log_2(p) = ", ceil(log(p, 2)))
    print()
    print("q = ", q)
    print("log_2(q) = ", ceil(log(q, 2)))
    print()

    F = GF(p)
    print("Information about the finite field: ", F)
    print()

    # Get some appropriate but random coefficients that define E_{p^2}
    get_random_entropoid_coefficients()
    print("aa1 = ", aa1)
    print("aa3 = ", aa3)
    print("aa4 = ", aa4)
    print("aa8 = ", aa8)
    print("bb1 = ", bb1)
    print("bb2 = ", bb2)
    print("bb5 = ", bb5)
    print("bb7 = ", bb7)
    print()

    # Some of the characteristic elements in the Entropoid
    print("E_Zero = ", E_Zero)
    print("E_left_unit = ", E_left_unit)
    E_minus_one = mminus(E_Zero, E_left_unit)
    print("E_minus_one = ", E_minus_one)
    print()

    # g is a generator for the multiplicative quasigroup of E_{p^2}
    g = find_generator_p()
    print("g = ", g)
    print()

    # g_q is a generator for the Silow q subquasigroup
    g_q = find_generator_q()
    print("g_q = ", g_q)
    print()

    # generate a random public parameter B
    # B is a power index
    M = "This is a seed message for fixing the value of the public root."
    if lambda_bits == 256:
        B = Hash256_to_power_index_02(bytearray(M.encode()), base)
    if lambda_bits == 384:
        B = Hash384_to_power_index_02(bytearray(M.encode()), base)
    if lambda_bits == 512:
        B = Hash512_to_power_index_02(bytearray(M.encode()), base)

    print("Public power index B = ", B)
    print()

    # generate the PrivateKey x
    # skip the coordinates of E_Zero
    x0 = E_Zero[0]
    while x0 == E_Zero[0]:
        x0 = F.random_element();
    x1 = E_Zero[1]
    while x1 == E_Zero[1]:
        x1 = F.random_element();
    x = vector((x0, x1))
    PrivateKey = x
    print("PrivateKey = ", PrivateKey)
    print()

    # generate the PublicKey y
    y = nanapow(x, B)
    PublicKey = y
    print("PublicKey = ", PublicKey)
    print()



def Sign_Signature02(PrivateKey, Message, base, lambda_bits):
    global B
    # generate a random r
    # skip the coordinates of E_Zero
```

```
        r0 = E_Zero[0]
        while r0 == E_Zero[0]:
            r0 = F.random_element();
        r1 = E_Zero[1]
        while r1 == E_Zero[1]:
            r1 = F.random_element();
        r = vector((r0, r1))

        I = nanapow(r, B)
        I_and_Message = concatenate_entropoid_element_and_message(I, Message, lambda_bits//8)

        if lambda_bits == 256:
            H = Hash256_to_power_index_02(I_and_Message, base)
        if lambda_bits == 384:
            H = Hash384_to_power_index_02(I_and_Message, base)
        if lambda_bits == 512:
            H = Hash512_to_power_index_02(I_and_Message, base)

        s = nanapow(mmult(PrivateKey, r), H)

        signature = (I, s)
        return signature

    def Verify_Signature02(PublicKey, Message, signature, base, lambda_bits):
        global B

        I = signature[0]
        I_and_Message = concatenate_entropoid_element_and_message(I, Message, lambda_bits//8)

        if lambda_bits == 256:
            H = Hash256_to_power_index_02(I_and_Message, base)
        if lambda_bits == 384:
            H = Hash384_to_power_index_02(I_and_Message, base)
        if lambda_bits == 512:
            H = Hash512_to_power_index_02(I_and_Message, base)

        s_to_B = nanapow(signature[1], B)
        y_mult_I_to_H = nanapow(mmult(PublicKey, I), H)

        if s_to_B == y_mult_I_to_H:
            return True
        else:
            return False
```

I. EXPERIMENTS WITH ENTROPOID DIFFIE-HELLMAN KEY EXCHANGE

In [3]:
```
# The results of this code are produced in SageMath 9.2
# To ensure the reproducibility of the results, set a fixed initial seed
set_random_seed(0)
random.seed(0, version=2)
print("initial seed: ", initial_seed())

# Set lambda_bits for a safe prime p = 2 q + 1.
# Notice that the prime number q will have lambda_bits - 1 bits
lambda_bits = 128
print("lambda_bits = ", lambda_bits)
print()

GenParameters_Diffie_Hellman(lambda_bits)
```

```
initial seed:  0
lambda_bits =  128

p =  255211775190703847597530955573826162347
log_2(p) =  128

Information about the finite field:  Finite Field of size
255211775190703847597530955573826162347

aa1 =  761962570250668710028914716198486502 86
aa3 =  222398418873342498392506793217273317643
aa4 =  241322191999219874657044597267756778501
aa8 =  178720374391503507560897272085794393770
bb1 =  477177328774872485300433265975609537 14
bb2 =  617335074474983118920638760376302357 26
bb5 =  247856823686916680700123131434371243080
bb7 =  272288238100104871366876352907822676 14

E_Zero =  (259586272033860399782907165911176490 49,
211647247068710178442880791600586184585)
E_left_unit =  (174395086300108117424968928287131469871,
150946712368694434305233963210010197064)
E_minus_one =  (132733943297367810129143460468929990574,
171360065780220749829966644173360097 59)
```

```
g =  (134868144355889480566471806491775788889, 232946345282352956340474695993456704856)

g_q =  (128137692344645973157570713635711768143,
200296064834253079673787541112431333948)
```

```
In [4]: # The results of this code are produced in SageMath 9.2
        # To ensure the reproducibility of the results, set a fixed initial seed
        set_random_seed(0)
        random.seed(0, version=2)
        print("initial seed: ", initial_seed())
        print()

        # A simmulation of a Diffie-Hellman key exchange with g

        # First test: Alice and Bob know E_p, g, and work with a same base

        # From the Section 2.4 Dichotomy between odd and even bases
        # in the paper, the recommended values are odd bases
        base = 17

        base_alice = base
        base_bob = base
        print("Agreed common base = ", base)
        print()

        # Alice gets a random power index
        Alice = get_random_power_index(base_alice);
        print("Alice = ", Alice)
        # Alice produces her public part Ka by calling the procedure nanapow()
        # that uses two parameters g and the power index Alice in order to
        # calculate g^Alice
        Ka = nanapow(g, Alice)
        print("Ka = ", Ka)
        print()

        # Bob gets a random power index
        Bob = get_random_power_index(base_bob);
        print("Bob = ", Bob)
        # Bob produces his public part Kb by calling the procedure nanapow()
        # that uses two parameters g and the power index Bob in order to
        # calculate g^Bob
        Kb = nanapow(g, Bob)
        print("Kb = ", Kb)
        print()

        # After receiving Kb, Alice computes the shared key Kab
        Kab = nanapow(Kb, Alice)
        print("Kab = ", Kab)
        print()

        # After receiving Ka, Bob computes the shared key Kba
        Kba = nanapow(Ka, Bob)
        print("Kba = ", Kba)
        print()

        print("The shared keys are the same: ", Kab == Kba)
```

```
initial seed:  0

Agreed common base =  17

Alice =  [47392387440050222334387056025351624212, [9, 4, 3, 8, 4, 9, 3, 2, 10, 15, 3,
11, 13, 10, 6, 15, 14, 8, 1, 0, 2, 12, 0, 15, 10, 7, 10, 2, 6, 7, 7], 17]
Ka =  (120421544237529629217788738702892637460,
127399315981183519023472362443200305215)

Bob =  [27374292381938936809021261532918539202, [10, 15, 3, 9, 9, 3, 10, 6, 9, 14, 2,
12, 10, 7, 9, 5, 6, 5, 1, 8, 15, 2, 2, 4, 4, 1, 2, 12, 8, 7, 6], 17]
Kb =  (750262808249586931709390390556894694 63,
160666520418747394653273412070345743677)

Kab =  (125453479868602117491962117002259721097,
802316838230043240741807272943602871 90)

Kba =  (125453479868602117491962117002259721097,
802316838230043240741807272943602871 90)

The shared keys are the same:  True
```

```
In [5]: # The results of this code are produced in SageMath 9.2
        # To ensure the reproducibility of the results, set a fixed initial seed
        set_random_seed(0)
        random.seed(0, version=2)
```

```
        print("initial seed: ", initial_seed())
        print()

        # A simmulation of a Diffie-Hellman key exchange with g

        # Second test: Alice and Bob know E_p, g, and work with bases on their own choice

        base_alice = 9
        print("base_alice = ", base_alice)
        # Alice gets a random power index
        Alice = get_random_power_index(base_alice);
        print("Alice = ", Alice)
        # Alice produces her public part Ka by calling the procedure nanapow()
        # that uses two parameters g and the power index Alice in order to
        # calculate g  to  the power of Alice
        Ka = nanapow(g, Alice)
        print("Ka = ", Ka)
        print()

        base_bob = 33
        print("base_bob = ", base_bob)
        # Bob gets a random power index
        Bob = get_random_power_index(base_bob);
        print("Bob = ", Bob)
        # Bob produces his public part Kb by calling the procedure nanapow()
        # that uses two parameters g and the power index Bob in order to
        # calculate g  to the power of Bob
        Kb = nanapow(g, Bob)
        print("Kb = ", Kb)
        print()

        # After receiving Kb, Alice computes the shared key Kab
        Kab = nanapow(Kb, Alice)
        print("Kab = ", Kab)
        print()

        # After receiving Ka, Bob computes the shared key Kba
        Kba = nanapow(Ka, Bob)
        print("Kba = ", Kba)
        print()

        print("The shared keys are the same: ", Kab == Kba)
```

```
initial seed:  0

base_alice =  9
Alice =  [47392387440050222334387056025351624212, [4, 2, 1, 4, 2, 4, 1, 1, 5, 7, 1, 5,
6, 5, 3, 7, 7, 4, 0, 0, 1, 6, 0, 7, 5, 3, 5, 1, 3, 3, 3, 2, 7, 1, 1, 5, 7, 1, 4, 4],
9]
Ka =  (187978293522818904347073968325418320167, 40318840796186523217530828093636825 74)

base_bob =  33
Bob =  [113229132117011277211194849153838560638, [13, 18, 28, 5, 24, 20, 15, 18, 11,
12, 11, 2, 16, 30, 4, 5, 8, 9, 2, 5, 25, 17, 15, 13, 26, 17], 33]
Kb =  (215810056254095917586510141651352944519,
58026588231455569467905033989562444838)

Kab =  (180300066404931116932038460669304112827,
140824513012217402735374548102038516194)

Kba =  (180300066404931116932038460669304112827,
140824513012217402735374548102038516194)

The shared keys are the same:  True
```

```
In [6]: # The results of this code are produced in SageMath 9.2
        # To ensure the reproducibility of the results, set a fixed initial seed
        set_random_seed(0)
        random.seed(0, version=2)
        print("initial seed: ", initial_seed())

        # Test with 192-bit prime numbers
        lambda_bits = 192
        print("lambda_bits = ", lambda_bits)
        print()

        GenParameters_Diffie_Hellman(lambda_bits)
        print()

        base_alice = 19
        print("base_alice = ", base_alice)
        # Alice gets a random power index
        Alice = get_random_power_index(base_alice);
        print("Alice = ", Alice)
        # Alice produces her public part Ka by calling the procedure nanapow()
        # that uses two parameters g and the power index Alice in order to
```

```
        # calculate g  to  the power of Alice
        Ka = nanapow(g, Alice)
        print("Ka = ", Ka)
        print()

        base_bob = 65
        print("base_bob = ", base_bob)
        # Bob gets a random power index
        Bob = get_random_power_index(base_bob);
        print("Bob = ", Bob)
        # Bob produces his public part Kb by calling the procedure nanapow()
        # that uses two parameters g and the power index Bob in order to
        # calculate g  to the power of Bob
        Kb = nanapow(g, Bob)
        print("Kb = ", Kb)
        print()

        # After receiving Kb, Alice computes the shared key Kab
        Kab = nanapow(Kb, Alice)
        print("Kab = ", Kab)
        print()

        # After receiving Ka, Bob computes the shared key Kba
        Kba = nanapow(Ka, Bob)
        print("Kba = ", Kba)
        print()

        print("The shared keys are the same: ", Kab == Kba)
```

```
initial seed:  0
lambda_bits =  192

p =  4707826301540010572876842067405749812076766583348025913783
log_2(p) =  192

Information about the finite field:  Finite Field of size
4707826301540010572876842067405749812076766583348025913783

aa1 =  257405322579552167145111504184189350941678348657606363000
aa3 =  119532791761269064716056035369728457738265053964404214027
aa4 =  739437136086943117194679493097867279546663219437537419926
aa8 =  329680900715761964011930877086999895860619508510387851126 9
bb1 =  127123825939707486963464396172505465970612452499399491586 4
bb2 =  280667772476280910759459937997540111851273487919409188564 6
bb5 =  459916388844202609415506244236040625798377825566564316646 3
bb7 =  949464039987904334513186654173522437107012951521210331273

E_Zero =  (267483832740753264872659450016843624242161662993967512484,
166053016173828599236689319982104781016237267598145924346 8)
E_left_unit =  (366547362505620205879025426766826981452971526528507670914 5,
272210090230589196512439373949759014447748940243816802813 8)
E_minus_one =  (168420302975886323866293473266860267031351799459485831 5823,
598959421170680019609392660144505475847255949524750458798)

g =  (878294255096851485610734722356052461437011464090979274925,
203730394788874113391655749229123012503133113833327558159 02)

g_q =  (269607523802418676881796240188295135748024524350363395887 4,
237049577975276765003791688064943945293891794224237172038 3)


base_alice =  19
Alice =  [449711953145234411268392775391383145652814819654039913657, [6, 14, 7, 14,
16, 5, 10, 4, 15, 17, 1, 17, 2, 16, 10, 0, 2, 3, 13, 11, 14, 10, 12, 16, 11, 3, 4, 10,
0, 5, 4, 0, 10, 6, 1, 13, 1, 9, 12, 1, 5, 11, 2, 13, 1], 19]
Ka =  (281764272077163471895777390452868590278235997030115907531,
191024527646352477314678649810975994381214591478548920 5426)

base_bob =  65
Bob =  [224475026280920032200709411808330373083543673929505487552 7, [12, 15, 3, 44, 2,
22, 51, 1, 41, 58, 63, 60, 10, 6, 51, 33, 3, 12, 10, 42, 45, 12, 60, 4, 19, 36, 4, 0,
48, 43, 20, 18], 65]
Kb =  (213466990206113312918447949631194829122923184507059675506 5,
316464677611448463946710924740188562748878247977624760461)

Kab =  (194791965136784447558149621713329666493912369868705707252 4,
344130877253281509509408074822858569257658007353336745519 6)

Kba =  (194791965136784447558149621713329666493912369868705707252 4,
344130877253281509509408074822858569257658007353336745519 6)

The shared keys are the same:  True
```

In [7]:
```
        # The results of this code are produced in SageMath 9.2
        # To ensure the reproducibility of the results, set a fixed initial seed
        set_random_seed(0)
```

```
        random.seed(0, version=2)
        print("initial seed: ", initial_seed())

        # Test with 256-bit prime numbers
        lambda_bits = 256
        print("lambda_bits = ", lambda_bits)
        print()

        GenParameters_Diffie_Hellman(lambda_bits)
        print()

        base_alice = 19
        print("base_alice = ", base_alice)
        # Alice gets a random power index
        Alice = get_random_power_index(base_alice);
        print("Alice = ", Alice)
        # Alice produces her public part Ka by calling the procedure nanapow()
        # that uses two parameters g and the power index Alice in order to
        # calculate g  to  the power of Alice
        Ka = nanapow(g, Alice)
        print("Ka = ", Ka)
        print()

        base_bob = 65
        print("base_bob = ", base_bob)
        # Bob gets a random power index
        Bob = get_random_power_index(base_bob);
        print("Bob = ", Bob)
        # Bob produces his public part Kb by calling the procedure nanapow()
        # that uses two parameters g and the power index Bob in order to
        # calculate g  to the power of Bob
        Kb = nanapow(g, Bob)
        print("Kb = ", Kb)
        print()

        # After receiving Kb, Alice computes the shared key Kab
        Kab = nanapow(Kb, Alice)
        print("Kab = ", Kab)
        print()

        # After receiving Ka, Bob computes the shared key Kba
        Kba = nanapow(Ka, Bob)
        print("Kba = ", Kba)
        print()

        print("The shared keys are the same: ", Kab == Kba)
```

```
initial seed:  0
lambda_bits =  256

p =  86844066927987146567678238756515930889952488499230423029593188005934847271147
log_2(p) =  256

Information about the finite field:  Finite Field of size
86844066927987146567678238756515930889952488499230423029593188005934847271147

aa1 =  32020853807654887644334957222306269398498608370866698553183453753385508071789
aa3 =  21006824032566107844541320689058119045737974467270935413688009986168488070826
aa4 =  78901668574955584791111916221370791050852265697346349228460454742358577919791
aa8 =  17514520152847203018279441394225361894014762419869662416889631828284746070238
bb1 =  21504609816581240890771596944610981247772392172380469982872574919132312646024
bb2 =  25223174955288415702340049551314795160644138670364684787245633274934513695975
bb5 =  51809049454113147465743186155015416837084822644379520043428264433478357770134
bb7 =  54522689072505696568590869193235206153532724699456527123048120076369379523 92

E_Zero =
(51894466488054851875251155944967472764493767614264395765203671293569617713236,
85989701923192530215918400608340282133450885001582874472320302659310552000477)
E_left_unit =
(63900928990625806614587970778417819593629835846896396669812155416150920745560,
51307550079261792743025881937137191345137761333097166359818337956730920664544)
E_minus_one =
(39888003985483897135914341111517125935357699381632394860595187170988314680912,
33827786839136121121132680523027442031811520170838159555229079355955336065263)

g =  (63870838387673686408399046990538467647890408044698845654006807288140639667 38,
69213513481711846442415867053262310462001757214589976425812813821794688022734)

g_q =  (52479429458270479956162254184736425846351402051441173451606348112480762619828,
11050458649956768945033688668132742127771498323039623993159230275819424398685)


base_alice =  19
Alice =
[50494267262122769030118119523953001112367345010310560200607286122927005644528, [9,
10, 6, 10, 13, 4, 3, 7, 11, 15, 14, 5, 11, 13, 13, 13, 11, 3, 2, 17, 14, 4, 3, 0, 16,
12, 15, 7, 12, 3, 13, 1, 9, 5, 9, 8, 16, 3, 17, 10, 0, 4, 0, 2, 10, 10, 14, 6, 7, 11,
```

```
14, 8, 17, 5, 14, 12, 0, 3, 0, 3], 19]
Ka =  (583198879010136977621923600783408813233461247836965522448382197792508271387 47,
71728981921072099159143151661375446742549903685523854955320855926837388591226)

base_bob =  65
Bob =  [85742694800052454927885104198579961590739048763645092626887756186393799161434,
[7, 31, 14, 21, 56, 21, 21, 59, 33, 21, 56, 3, 48, 2, 15, 12, 39, 10, 48, 26, 32, 49,
29, 59, 47, 23, 54, 1, 53, 29, 36, 44, 40, 60, 59, 32, 44, 34, 20, 23, 28, 59, 41],
65]
Kb =  (623841798604556297498877241332914320984086388340108381678206567108888400695872,
34601202818684419307831067107179224327961058089663236540130514156571880145504)

Kab =  (194898727193510805355757010102201425335395291942887520812983981118680247414 81,
55783088627265246592571548430229513640432066609268166341998182809194562182853)

Kba =  (194898727193510805355757010102201425335395291942887520812983981118680247414 81,
55783088627265246592571548430229513640432066609268166341998182809194562182853)

The shared keys are the same:  True
```

II. EXPERIMENTS WITH DIGITAL SIGNATURES 01

```
In [8]: # The results of this code are produced in SageMath 9.2
        # To ensure the reproducibility of the results, set a fixed initial seed
        set_random_seed(0)
        random.seed(0, version=2)
        print("initial seed: ", initial_seed())

        lambda_bits = 128
        base = 257
        GenKey_Signature01(lambda_bits, base)

        print()
        Message = "Try to sign this message!"
        print("Signing the message: ", Message)
        signature = Sign_Signature01(PrivateKey, Message, base, lambda_bits)
        print("signature = ", signature)
        print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
        print()
        # Corrupt the Message
        Message = "Try to sign this message"
        print("An attampt to verify a signature of a corrupted message")
        print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
```

```
initial seed:  0
p =  25521177519070384759753095557382616 2347
log_2(p) =  128

Information about the finite field:  Finite Field of size
25521177519070384759753095557382616 2347

aa1 =  76196257025066871002891471619848650286
aa3 =  22239841887334249839250679321727 3317643
aa4 =  24132219199921987465704459726775 6778501
aa8 =  17872037439150350756089727208579 4393770
bb1 =  47717732877487248530043326597560953714
bb2 =  61733507447498311892063876037630235726
bb5 =  24785682368691668070012313143437 1243080
bb7 =  27228823810010487136687635290782267614

E_Zero =  (25958627203386039978290716591117649049,
21164724706871017844288079160058618 4585)
E_left_unit =  (17439508630010811742496892828713 1469871,
15094671236869443430523396321001019 7064)
E_minus_one =  (13273394329736781012914346046 8929990574,
17136006578022074982996664417336009759)

g =  (13486814435588948056647180649177578889, 23294634528235295634047469599345 6704856)

g_q =  (12813769234464597315757071363571 1768143,
20029606483425307967378754111243133 3948)

Public power index B =  [33074435428350873753220117677699672 7082, [122, 144, 252, 40,
232, 192, 61, 58, 106, 191, 17, 204, 84, 142, 219, 140], 257]

PrivateKey =  (18769952426748430136604997612423969 2348,
13234408806419945051978838754289632 6862)

PublicKey =  (12408698176943960432513119973443877 2689,
63633811805890784069039228612491201 419)


Signing the message:  Try to sign this message!
```

```
signature =  ((107903183123075764427828763979029799787,
229642977360973535707709533387469458206), (538197266480606935201992821076611343 26,
175890934206597718523061341884515052675))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False
```

```
In [9]: # The results of this code are produced in SageMath 9.2
        # To ensure the reproducibility of the results, set a fixed initial seed
        set_random_seed(0)
        random.seed(0, version=2)
        print("initial seed: ", initial_seed())

        lambda_bits = 128
        base = 17
        GenKey_Signature01(lambda_bits, base)

        print()
        Message = "Try to sign this message!"
        print("Signing the message: ", Message)
        signature = Sign_Signature01(PrivateKey, Message, base, lambda_bits)
        print("signature = ", signature)
        print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
        print()
        # Corrupt the Message
        Message = "Try to sign this message"
        print("An attampt to verify a signature of a corrupted message")
        print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
```

```
initial seed:  0
p =  255211775190703847597530955573826162347
log_2(p) =  128

Information about the finite field:  Finite Field of size
255211775190703847597530955573826162347

aa1 =  761962570250668710028914716198486502 86
aa3 =  222398418873342498392506793217273317643
aa4 =  241322191999219874657044597267756778501
aa8 =  178720374391503507560897272085794393770
bb1 =  477177328774872485300433265975609537 14
bb2 =  617335074474983118920638760376302357 26
bb5 =  247856823686916680700123131434371243080
bb7 =  272288238100104871366876352907822676 14

E_Zero =  (259586272033860399782907165911176490 49,
211647247068710178442880791600586184585)
E_left_unit =  (174395086300108117424968928287131469871,
150946712368694434305233963210010197064)
E_minus_one =  (132733943297367810129143460468929990574,
171360065780220749829966644173360097 59)

g =  (134868144355889480566471806491775788 89, 232946345282352956340474695993456704856)

g_q =  (128137692344645973157570713635711768143,
200296064834253079673787541112431333948)

Public power index B =  [330744354283508737532201176776996727082, [10, 7, 0, 9, 12,
15, 8, 2, 8, 14, 0, 12, 13, 3, 10, 3, 10, 6, 15, 11, 1, 1, 12, 12, 4, 5, 14, 8, 11,
13, 12, 8], 17]

PrivateKey =  (187699524267484301366049976124239692348,
132344088064199450519788387542896326862)

PublicKey =  (571772974277474236953196517943011 10642,
150211926636526683836991675796943445263)


Signing the message:  Try to sign this message!
signature =  ((229260805685363144288679791250418504870,
135760263038118728463907594768162086282), (197465655642382975061767623344022586735,
217905194456207085163822482362355764477))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False
```

```
In [10]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 192
```

```
        base = 257
        GenKey_Signature01(lambda_bits, base)

        print()
        Message = "Try to sign this message!"
        print("Signing the message: ", Message)
        signature = Sign_Signature01(PrivateKey, Message, base, lambda_bits)
        print("signature = ", signature)
        print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
        print()
        # Corrupt the Message
        Message = "Try to sign this message"
        print("An attampt to verify a signature of a corrupted message")
        print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
```

```
initial seed:  0
p =  47078263015400105728768420674057498120767665833480259137 83
log_2(p) =  192

Information about the finite field:  Finite Field of size
47078263015400105728768420674057498120767665833480259137 83

aa1 =  257405322579552167145111504184189350941678348657606363000
aa3 =  119532791761269064716056035369728457738265053964404214027
aa4 =  739437136086943117194679493097867279546663219437537419926
aa8 =  329680900715761964011930877086999895860619508510387851126 9
bb1 =  127123825939707486963464396172505465970612452499399491586 4
bb2 =  280667772476280910759459937997540111851273487919409188564 6
bb5 =  459916388844202609415506244236040625798377825566564316646 3
bb7 =  949464039987904334513186654173522437107012951521210331273

E_Zero =  (267483832740753264872659450016843624242161662993967512484,
166053016173828599236689319982104781016237267598145924346 8)
E_left_unit =  (366547362505620205879025426766826981452971526528507670914 5,
272210090230589196512439373949759014447748940243816802813 8)
E_minus_one =  (168420302975886323866293473266860267031351799459485831582 3,
598959421170680019609392660144505475847255949524750458798)

g =  (878294255096851485610734722356052461437011464090979274925,
203730394788874113391655749229123012503133113833275581590 2)

g_q =  (269607523802418676881796240188295135748024524350363395887 4,
237049577975276765003791688064943945293891794224237172038 3)

Public power index B =  [461137594522449029654502449403633289063987371220352922337 8,
[14, 139, 36, 239, 242, 210, 23, 59, 178, 110, 127, 242, 54, 195, 17, 243, 207, 160,
53, 120, 123, 168, 14, 52], 257]

PrivateKey =  (310054162383228864108817400690344053227988772040930478043 4,
182708553463714052150946867493212972311006807015788383224 3)

PublicKey =  (280092543415166204392317681471889874846673367550755543149 17,
447744457761024000143513454437415110219798418902357296506 8)


Signing the message:  Try to sign this message!
signature =  ((369144933696088526530657908540872732915882555691825533463 5,
346481150266980653037822741237174490175594870353806094513 0),
(315002716837672802656396919562402748570449794395699423394 8,
551145588699858476103609975900732141365342832596107957403))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False
```

```
In [11]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 192
         base = 17
         GenKey_Signature01(lambda_bits, base)

         print()
         Message = "Try to sign this message!"
         print("Signing the message: ", Message)
         signature = Sign_Signature01(PrivateKey, Message, base, lambda_bits)
         print("signature = ", signature)
         print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
         print()
         # Corrupt the Message
         Message = "Try to sign this message"
         print("An attampt to verify a signature of a corrupted message")
         print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
```

```
initial seed:  0
p =  4707826301540010572876842067405749812076766583348025913783
log_2(p) =  192

Information about the finite field:  Finite Field of size
4707826301540010572876842067405749812076766583348025913783

aa1 =  257405322579552167145111504184189350941678348657606363000
aa3 =  119532791761269064716056035369728457738265053964404214027
aa4 =  739437136086943117194679493097867279546663219437537419926
aa8 =  32968090071576196401193087708699989586061950851037851 1269
bb1 =  127123825939707486963464396172505465970612452499399491 5864
bb2 =  280667772476280910759459937997540111851273487919409188 5646
bb5 =  459916388844202609415506244236040625798377825566564316 6463
bb7 =  949464039987904334513186654173522437107012951521210331273

E_Zero =  (267483832740753264872659450016843624242161662993996751 2484,
166053016173828599236689319982104781016237267598145924346 8)
E_left_unit =  (366547362505620205879025426766826981452971526528507670 9145,
272210090230589196512439373949759014447748940243816802813 8)
E_minus_one =  (168420302975886323866293473266860267031351799459485 15823,
598959421170680019609392660144505475847255949524750458798)

g =  (878294255096851485610734722356052461437011464090979274925,
203730394788874113391655749229123012503133113833275581 5902)

g_q =  (269607523802418676881796240188295135748024524350363395887 4,
237049577975276765003791688064943945293891794224237172038 3)

Public power index B =  [461137594522449029654502449403633289063987371220352922337 8,
[14, 0, 11, 8, 4, 2, 15, 14, 2, 15, 2, 13, 7, 1, 11, 3, 2, 11, 14, 6, 15, 7, 2, 15, 6,
3, 3, 12, 1, 1, 3, 15, 15, 12, 0, 10, 5, 3, 8, 7, 11, 7, 8, 10, 14, 0, 4], 17]

PrivateKey =  (310054162383228864108817400690344053227988772040930478043 4,
182708553463714052150946867493212972311006807015788383224 3)

PublicKey =  (356232703563530572171274440928932589574613591291914909031 3,
161356213141249154050313053110353835999870178502374713362 1)


Signing the message:  Try to sign this message!
signature =  ((248568201938671624839060599826442114854666780187361385144 2,
464987881208094050953929773437566309115134138336968502462 4),
(296977471380044191050725025247358335931871617317281734987 3,
788465226406924846486964593856040528030591503330253706126))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False
```

```
In [12]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 256
         base = 257
         GenKey_Signature01(lambda_bits, base)

         print()
         Message = "Try to sign this message!"
         print("Signing the message: ", Message)
         signature = Sign_Signature01(PrivateKey, Message, base, lambda_bits)
         print("signature = ", signature)
         print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
         print()
         # Corrupt the Message
         Message = "Try to sign this message"
         print("An attampt to verify a signature of a corrupted message")
         print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
```

```
initial seed:  0
p =  868440669279871465676782387565159308899524884992304230295931880059348472711 47
log_2(p) =  256

Information about the finite field:  Finite Field of size
868440669279871465676782387565159308899524884992304230295931880059348472711 47

aa1 =  320208538076548876443349572223062693984986083708666985531834537533855080 71789
aa3 =  210068240325661078445413206890581190457379744672709354136880099861684880 70826
aa4 =  789016685749555847911119162213707910508522656973463492284604547423585779 19791
aa8 =  175145201528472030182794413942253618940147624198696624168896318282847460 70238
bb1 =  215046098165812408907715969446109812477723921723804699828725749191321326 46024
bb2 =  252231749552884157023400495513147951606441386703646847872456332749345136 95975
bb5 =  518090494541131474657431861550154168370848226443795200434282644334783577 70134
```

```
bb7 =  54522689072505696568590869193235206153532724699456527123048120076369379523 92

E_Zero =
(51894466488054851875251155944967472764493767614264395765203671293569617713236,
85989701923192530215918400608340282133450885001582874472320302659310552000477)
E_left_unit =
(63900928990625806614587970778417819593629835846896396669812155416150920745560,
51307550079261792743025881937137191345137761333097166359818337956730920664544)
E_minus_one =
(39888003985483897135914341111517125935357699381632394860595187170988314680912,
33827786839136121121132680523027442031811520170838159555229079355955336065263)

g =  (63870838387673686408399904699053846764789040804469884565400680728814063966738,
69213513481711846442415867053262310462001757214589976425812813821794688022734)

g_q =  (52479429458270479956162254184736425846351402051441173451606348112480762619828,
11050458649956768945033688668132742127771498323039623993159230275819424398685)

Public power index B =
[10834628919310687640339009583926248589311165052804325515504719748837622045 45, [203,
163, 166, 12, 244, 239, 238, 229, 142, 4, 175, 71, 61, 200, 166, 197, 187, 230, 83,
141, 218, 111, 175, 77, 187, 111, 28, 26, 1, 106, 148, 206], 257]

PrivateKey =
(79434508831788634043760869072494127777062337554518938032538523069944444174397,
77230632318947480536928206345632336245652795533833568674974941863932896893241)

PublicKey =
(28978861783619100818021748198806204873613038697361726607864281258512799244400,
84928212033650259396160334508849581925728314604467764543880179357985745454112)


Signing the message:  Try to sign this message!
signature =
((15530121100324659082859276230222367477129958353136917933796888465928054605453,
48223616658863571626769036631391933752232640424038986121722066066495996230200),
(57455838207194233708911344162984875757160709103528003486982291520878455 14981,
74881944811037829547846373882737642226503182387553640918274359945292069447866))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False


In [13]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 256
         base = 17
         GenKey_Signature01(lambda_bits, base)

         print()
         Message = "Try to sign this message!"
         print("Signing the message: ", Message)
         signature = Sign_Signature01(PrivateKey, Message, base, lambda_bits)
         print("signature = ", signature)
         print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))
         print()
         # Corrupt the Message
         Message = "Try to sign this message"
         print("An attampt to verify a signature of a corrupted message")
         print("Verify: ", Verify_Signature01(PublicKey, Message, signature, base, lambda_bits))

initial seed:  0
p =  86844066927987146567678238756515930889952488499230423029593188005934847271147
log_2(p) =  256

Information about the finite field:  Finite Field of size
86844066927987146567678238756515930889952488499230423029593188005934847271147

aa1 =  32020853807654887644334957222306269398498608370866698553183453753385508071789
aa3 =  21006824032566107844541320689058119045737974467270935413688009986168488070826
aa4 =  78901668574955584791111916221370791050852265697346349228460454742358577919791
aa8 =  17514520152847203018279441394225361894014762419869662416889631828284746070238
bb1 =  21504609816581240890771596944610981247772392172380469982872574919132312646024
bb2 =  25223174955288415702340049551314795160644138670364684787245633274934513695975
bb5 =  51809049454113147465743186155015416837084822644379520043428264433478357770134
bb7 =  54522689072505696568590869193235206153532724699456527123048120076369379523 92

E_Zero =
(51894466488054851875251155944967472764493767614264395765203671293569617713236,
85989701923192530215918400608340282133450885001582874472320302659310552000477)
E_left_unit =
(63900928990625806614587970778417819593629835846896396669812155416150920745560,
```

```
513075500792617927430258819371371913451377613330971663598183379567309206645 44)
E_minus_one =
(39888003985483897135914341111517125935357699381632394860595187170988314680912,
33827786839136121121132680523027442031811520170838159555229079355955336065263)

g =  (63870838387673686408399904699053846764789040804469884565400680728814063966738,
69213513481711846442415867053262310462001757214589976425812813821794688022734)

g_q =  (52479429458270479956162254184736425846351402051441173451606348112480762619828,
11050458649956768945033688668132742127771498323039623993159230275819424398685)

Public power index B =
[10834628919310687640339009583926248589311165052804325515504719748837622045 45, [11,
12, 3, 10, 6, 10, 12, 0, 4, 15, 15, 14, 14, 14, 5, 14, 14, 8, 4, 0, 15, 10, 7, 4, 13,
3, 8, 12, 6, 10, 5, 12, 11, 11, 6, 14, 3, 5, 13, 8, 10, 13, 15, 6, 15, 10, 13, 4, 11,
11, 15, 6, 12, 1, 10, 1, 1, 0, 10, 6, 4], 17]

PrivateKey =
(79434508831788634043760869072494127777062337554518938032538523069944444174397,
77230632318947480536928206345632336245652795533833568674974941863932896893241)

PublicKey =
(50814889348371133837889307524154547437277902185700767674212171611740531411876,
46949590751533512461830114803249529664218807618103476066759642559152825906 57)


Signing the message:  Try to sign this message!
signature =
((68979615736317796500575480543083060773800271114761757967532985762366052518227,
57427382978879981616724749934032808176436234884933255065190111654279111617353),
(35101598652680936208523824038416223507416657771158181571444506759487709707097,
61652918848301993265393449903131687923650163283792172101505363900041686497119))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False
```

III. EXPERIMENTS WITH DIGITAL SIGNATURES 02

```
In [14]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 256
         base = 257
         GenKey_Signature02(lambda_bits, base)

         print()
         Message = "Try to sign this message!"
         print("Signing the message: ", Message)
         signature = Sign_Signature02(PrivateKey, Message, base, lambda_bits)
         print("signature = ", signature)
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
         print()
         # Corrupt the Message
         Message = "Try to sign this message"
         print("An attampt to verify a signature of a corrupted message")
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))

initial seed:  0
p =  868440669279871465676782387565159308899524884992304230295931880059348472711 47
log_2(p) =  256

q =  434220334639935732838391193782579654449762442496152115147965940029674236359 59
log_2(q) =  255

Information about the finite field:  Finite Field of size
868440669279871465676782387565159308899524884992304230295931880059348472711 47

aa1 =  320208538076548876443349572223062693984986083708666985531834537533855080 71789
aa3 =  210068240325661078445413206890581190457379744672709354136880099861684880 70826
aa4 =  789016685749555847911119162213707910508522656973463492284604547423585779 19791
aa8 =  175145201528472030182794413942253618940147624198696624168896318282847460 70238
bb1 =  215046098165812408907715969446109812477723921723804699828725749191323126460 24
bb2 =  252231749552884157023400495513147951606441386703646847872456332749345136959 75
bb5 =  518090494541131474657431861550154168370848226443795200434282644334783577701 34
bb7 =  545226890725056965685908691932352061535327246994565271230481200763693795239 2

E_Zero =
(518944664880548518752511559449674727644937676142643957652036712935696177132 36,
859897019231925302159184006083402821334508850015828744723203026593105520004 77)
```

```
E_left_unit =
(6390092899062580661458797077841781959362983584689639666981215541615092074556 0,
5130755007926179274302588193713719134513776133309716635981833795673092066454 4)
E_minus_one =
(3988800398548389713591434111151712593535769938163239486059518717098831468091 2,
3382778683913612112113268052302744203181152017083815955522907935595533606526 3)

g =  (6387083838767368640839990469905384676478904080446988456540068072881406396 6738,
6921351348171184644241586705326231046200175721458997642581281382179468802273 4)

g_q =  (5247942945827047995616225418473642584635140205144117345160634811248076261 9828,
1105045864995676894503368866813274212777149832303962399315923027581942439868 5)

Public power index B =
[434220334639935732838391193782579654449762442496152115147965940029674236359 59, [122,
144, 252, 40, 232, 192, 61, 58, 106, 191, 17, 204, 84, 142, 219, 140, 42, 169, 98,
180, 176, 181, 92, 12, 57, 92, 32, 120, 171, 11, 211, 248], 257]

PrivateKey =
(7943450883178863404376086907249412777706233755451893803253852306994444417439 7,
7723063231894748053692820634563233624565279553383356867497494186393289689324 1)

PublicKey =
(6640348923620773127283319133891144233822095879059417152120652779245208240032 0,
8020479543440713423540066999757307306003194330079073413758779505563883039109 2)


Signing the message:  Try to sign this message!
signature =
((9909336984919732581132989217304955290022411626014224927110011499153768839787,
2362295755409966455009662168624046707397750393194852586848129770819498396171 1),
(5393855229347941561442121979634065929035131582963574892010083983760321769203,
2062410206935397371108292588195067838893668915230260328882639293637041619391 1))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False
```

```
In [15]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 256
         base = 17
         GenKey_Signature02(lambda_bits, base)

         print()
         Message = "Try to sign this message!"
         print("Signing the message: ", Message)
         signature = Sign_Signature02(PrivateKey, Message, base, lambda_bits)
         print("signature = ", signature)
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
         print()
         # Corrupt the Message
         Message = "Try to sign this message"
         print("An attampt to verify a signature of a corrupted message")
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
```

```
initial seed:  0
p =  8684406692798714656767823875651593088995248849923042302959318800593484727114 7
log_2(p) =  256

q =  4342203346399357328383911937825796544497624424961521151479659400296742363595 9
log_2(q) =  255

Information about the finite field:  Finite Field of size
8684406692798714656767823875651593088995248849923042302959318800593484727114 7

aa1 =  3202085380765488764433495722230626939849860837086669855318345375338550807178 9
aa3 =  2100682403256610784454132068905811904573797446727093541368800998616848807082 6
aa4 =  7890166857495558479111191622137079105085226569734634922846045474235857791979 1
aa8 =  1751452015284720301827944139422536189401476241986966241688963182828474607023 8
bb1 =  2150460981658124089077159694461098124777239217238046998287257491913231264602 4
bb2 =  2522317495528841570234004955131479516064413867036468478724563327493451369597 5
bb5 =  5180904945411314765743186155015416837084822644379520043428264433478357770134
bb7 =  5452268907250569656859086919323520615353272469945652712304812007636937952392

E_Zero =
(5189446648805485187525115594496747276449376761426439576520367129356961771323 6,
8598970192319253021591840060834028213345088500158287447232030265931055200047 7)
E_left_unit =
(6390092899062580661458797077841781959362983584689639666981215541615092074556 0,
5130755007926179274302588193713719134513776133309716635981833795673092066454 4)
E_minus_one =
```

```
(39888003985483897135914341111517125935357699381632394860595187170988314680912,
338277868391361211211326805230274420318115201708381595552290793559553360652 63)

g =  (63870838387673686408399904699053846764789040804469884565400680728814063966738,
692135134817118464424158670532623104620017572145899764258128138217946880227 34)

g_q =  (52479429458270479956162254184736425846351402051441173451606348112480762619828,
110504586499567689450336886681327421277714983230396239931592302758194243986 85)

Public power index B =
[434220334639935732838391193782579654449762442496152115147965940029674236359 59, [10,
7, 0, 9, 12, 15, 8, 2, 8, 14, 0, 12, 13, 3, 10, 3, 10, 6, 15, 11, 1, 1, 12, 12, 4, 5,
14, 8, 11, 13, 12, 8, 10, 2, 9, 10, 2, 6, 4, 11, 0, 11, 5, 11, 12, 5, 12, 0, 9, 3, 12,
5, 0, 2, 8, 7, 11, 10, 11, 0, 3, 13, 8], 17]

PrivateKey =
(794345088317886340437608690724941277770623375545189380325385230699444441743 97,
772306323189474805369282063456323362456527955338335686749749418639328968932 41)

PublicKey =
(703610341200315110826254238856701160377048612308770438462341413267594560256 66,
686275191536943465934772462692297161036812635110473144293814791638791776415)


Signing the message:  Try to sign this message!
signature =
((160222221506821933008419858114826105016490325712517888361169235255414689645 75,
817258707863604247059388068702136673461019938439056433618196706740577678329 83),
(699587611694869254317826526498650128269733719263505326176174157542241638868 47,
713054718366669192381827264971977631715088701105142496149267998304863788570 98))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False
```

```
In [16]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 384
         base = 257
         GenKey_Signature02(lambda_bits, base)

         print()
         Message = "Try to sign this message!"
         print("Signing the message: ", Message)
         signature = Sign_Signature02(PrivateKey, Message, base, lambda_bits)
         print("signature = ", signature)
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
         print()
         # Corrupt the Message
         Message = "Try to sign this message"
         print("An attampt to verify a signature of a corrupted message")
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
```

```
initial seed:  0
p =  29551504647295859409209280075107710353809804452849085000961220053184291328622 9079
585606996911636867306049709927669 87
log_2(p) =  384

q =  14775752323647929704604640037553855176904902226424542500480610026592145664311 4539
792803498455818433653024854963835 37
log_2(q) =  383

Information about the finite field:  Finite Field of size 295515046472958594092092800
510771035380980445284908500096122005318429132862290795856069969116368673060497099276 69
87

aa1 =  758239485613261762035030169128756268166517831911599621671148879899429632126000 8
236339887422691230485947234341760057
aa3 =  206944055400675342865714076148520139492269160193367901908138681909785489665302 2
516194029684490886329402703646603905 1
aa4 =  284045077502967194804011112531013916505894527964265870722465458951327032990022 5
899717872407820365939690463622295892 8
aa8 =  595988237309532315740388168846994021514742101893720676337673154601540630600307 4
498066795537946055833375823442482254
bb1 =  970805433433573488628847016502327039713088170854348337712897400456343872104072 8
394296437688023842510725858098674699
bb2 =  127883641468025528054087781323655020024526952064988930808917956090909292326892 9
881357650976248838602054548660014711 7
bb5 =  689685254091244378568211266090565980805996574997855548809216355160361529637785 1
849089897959840682434076336518477491
bb7 =  185531096884866254751629036567897218924133782396759621962264410399489519812012 9
456985441139251130011571250928006434
```

```
E_Zero =  (19450661778312382300674102788293458197448865527683863015992542397266643992060820952613385009292599918830460797934729, 27629038716076565781719371295404141061192727438711839308322395460937724888892561528643018833679393789082568786506064)
E_left_unit =  (26133549342078802675975766132942905658437114655693895885601935155856030465690313995339771892523496768789223333456637, 64083220359050313659033774647982018764748199613983121464734826082880834338567866409935145889521918440304295196981630758431327909886998126061703068871698262412821, 19298250748952240788326085050902369892100830463176281469210088260403075015305428457731823387242909003529737060546978)

g =  (19462415207596296181403497597901846757011564296245864283403624939672531843924062973237647528699203754610023101922697, 19746431298753066688567257865373298328723479079943385228189334756829597831813369098424767629075882663988075443550131)

g_q =  (81251876739280741158671275351705805696333095976882020983595322817178839407768225999098512429967997893983350972678 54, 84012499203395259159168490544512904808137864346801912233294864732037120441283848955774803053258731097976271068832 18)

Public power index B =  [14775752323647929704604640037553855176904902226424542504808610026592145664311450397928034984558184336530248549638353 7, [14, 139, 36, 239, 242, 210, 23, 59, 178, 110, 127, 242, 54, 195, 17, 243, 207, 160, 53, 120, 123, 168, 14, 52, 210, 124, 36, 110, 186, 181, 130, 235, 30, 20, 87, 159, 2, 116, 112, 207, 72, 92, 111, 48, 228, 2, 17, 188], 257]

PrivateKey =  (90573347823195991861509439224096503311223517596894182768778231755963305965697821481870262346851516127942193656602 75, 51418182909936739544265082532226416368576135190736007450680686362147067999951281961051181023324154635758758833656 32)

PublicKey =  (10472944270833701475233302041248761042908320634557351476682253871136051192793912846563383951515720427714559248508517, 62051868762016112859167936332127997775988726876964677728531878779604261837351662944041952320520310699538052763776)


Signing the message:  Try to sign this message!
signature =  ((20816498159712888228293579688931303691134563263701329328575641173089562990287721825455485588091606682787259994597229, 27699602934393329656882967015174186464807950288674842081601989941056922257958493904534641153034265781724671387692 81), (98847148082974993867540310013176563766179352395288804373272587416750615955850724381561286361995288533678627770703584, 18591392290541456229781236019227229392274330481011480612873098401345027878605903097252976054994199332184696017770708))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False
```

```
In [17]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 384
         base = 17
         GenKey_Signature02(lambda_bits, base)

         print()
         Message = "Try to sign this message!"
         print("Signing the message: ", Message)
         signature = Sign_Signature02(PrivateKey, Message, base, lambda_bits)
         print("signature = ", signature)
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
         print()
         # Corrupt the Message
         Message = "Try to sign this message"
         print("An attampt to verify a signature of a corrupted message")
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
```

```
initial seed:  0
p =  29551504647295859409209280075107710353809804452849085000961220053184291328622907 95856069969116368673060497099276698 7
log_2(p) =  384

q =  14775752323647929704604640037553855176904902226424542500480610026592145664311453979280349845581843365302485496383537
log_2(q) =  383

Information about the finite field:  Finite Field of size 29551504647295859409209280075107710353809804452849085000961220053184291328622907958560699691163686730604970992766987

aa1 =  75823948561326176203503016912875626816651783191159962167114887989942963212600082363398874226912304859472343417600 57
aa3 =  20694405540067534286571407614852013949226916019336790190813868190978548966530225161940296844908863294027036466039051
aa4 =  28404507750296719480401111253101391650589452796426587072246545895132703299002 25
```

```
899717872407820365939690463622295892 8
aa8 =  595988237309532315740388168846994021514742101893720676337673154601540630600307 4
498066795537946055833375823442482254
bb1 =  970805433433573488628847016502327039713088170854348337712897400456343872104072 8
394296437688023842510725858098674699
bb2 =  127883641468025528054087781323655020024526952064988930808917956090909292326892 9
881357650976248838602054548660014711 7
bb5 =  689685254091244378568211266090565980805996574997855548809216355160361529637785 1
849089897959840682434076336518477491
bb7 =  185531096884866254751629036567897218924133782396759621962264410399489519812012 9
456985441139251130011571250928006434

E_Zero =  (194506617783123823006741027882934581974488655276838630159925423972666439920
608209526133850092925999188304607979734729, 276290387160765657817193712954041410611927 2
743871183930832239546093772488889256152864301883367939378908256878650606 4)
E_left_unit =  (261335493420788026759757661329429056584371146556938958856019351558560 3
046569031399533977189252349676878922333345663 7, 640832203590503136590337746479820187 6 4
748199613983121464734826082880834338567866409935145889521918440304295196981 6 3)
E_minus_one =  (127677742145459619253724394436440107364606163996738301463814963867725
751843132790988699812606170306887169826241282 1, 192982507489522407883260850509023698 9 2
100830463176281469210088260403075015305428457731823387242909003529737060546978)

g =  (194624152075962961814034975979018467570115642962458642834036249396725318439240 6 2
973237647528699203754610023101922697, 197464312987530666885672578653732983287234790799
433852281893347568295978318133690984247676290758826639880754435501 3 1)

g_q =  (812518767392807411586712753517058056963330959768820209835953228171788394077682
259990985124299679978939833509726785 4, 840124992033952591591684905445129048081378643 4 6
801912233294864732037120441283848955774803053258731097976271068832 1 8)

Public power index B =  [14775752323647929704604640037553855176904902226424542500480 6 1
002659214566431145397928034984558184336530248549638353 7, [14, 0, 11, 8, 4, 2, 15, 14,
2, 15, 2, 13, 7, 1, 11, 3, 2, 11, 14, 6, 15, 7, 2, 15, 6, 3, 3, 12, 1, 1, 3, 15, 15,
12, 0, 10, 5, 3, 8, 7, 11, 7, 8, 10, 14, 0, 4, 3, 2, 13, 12, 7, 4, 2, 14, 6, 10, 11,
5, 11, 2, 8, 11, 14, 14, 1, 4, 1, 7, 5, 15, 9, 2, 0, 4, 7, 0, 7, 15, 12, 8, 4, 12, 5,
15, 6, 0, 3, 4, 14, 2, 0, 1, 1], 17]

PrivateKey =  (905733478231959918615094392240965033112235175968941827687782317559633 0 5
965697821481870262346851516127942193656602 7 5, 514181829099367395442650825322264163685 7
613519073600745068068636214706799995128196105118102332415463575875883365632)

PublicKey =  (181766747132077943612498109745569748587050449756526211633534026578992 2 8
755602559946444926742283780432979398934106 2 8, 240875620898241592085323877600821262936 7
097345689327037042428759424164196419873499817912194757523416584964439308250 4)


Signing the message:  Try to sign this message!
signature =  ((241912587271365567771010710465942707051787202410524253071803093844987 6 1
085178404041807764247666868060403657784264822, 175523898200678278493309092066786042769
910808841597910379896739782017863975572809023763454713275861302192107905947 7 4), (89421
671448471651566214721718176561800763453862245243789900407928406281957409688596289981 6 2
932676747984959035564679, 303567368239930897046317578984482387900626235120202554882456
644425979673173340942733115902441474147456887274199834))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False


In [18]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 512
         base = 257
         GenKey_Signature02(lambda_bits, base)

         print()
         Message = "Try to sign this message!"
         print("Signing the message: ", Message)
         signature = Sign_Signature02(PrivateKey, Message, base, lambda_bits)
         print("signature = ", signature)
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
         print()
         # Corrupt the Message
         Message = "Try to sign this message"
         print("An attampt to verify a signature of a corrupted message")
         print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))

initial seed:  0
p =  100558559474569478246805187486543845956095243654442950332926710827913230225551602
326014057236251775707675238936398645381403154121089599274598252367545971 9 9
log_2(p) =  512

q =  502792797372847391234025937432719229780476218272214751664633554139566151127758011
```

```
6300702861812588785383761946819932269070157706054479963729912618377298809
log_2(q) =  511

Information about the finite field:  Finite Field of size 100558559474569478246805187 4
865438459560952436544429503329267108279132302255516023260140572362517757076752389363986
645381403154121089599274598252367545971 99

aa1 =  508352123849918161610710343554011746355586386849972571734528661837837057976130 3
457915289965842449989320867780251139340372938505409275428692727806563961170
aa3 =  243242404297149321525975271640936052173212336989759498353809515851082532603787 6
699129932315236342603157409792578609891994907734721051329531278165610547467
aa4 =  262243230541992905997048818059552157669000185257906221273885731759144989666290 4
619799952403996535930094107878467894339952834965446029306692688592915881137
aa8 =  631329607854202420502662867145665318978808213097829273181140158739538567864927 7
368876913583499084548362205489272342293542385933707548386627002145836222887
bb1 =  428477434346242182794385424818140919573918965330002982197075064675476436189218 8
335813755072306168585850275872439684355166336583770831050474250855359294942
bb2 =  739573781824771236423602020176670183093316182616825455348286904685479054455683 70
593333790859521700466724364752969780641673509981915176519917524470852720801
bb5 =  510382526006415325585980917171428015706436160338358088350573088930937817588854 0
769658573790799402085716793034357461718549622609044105958925015508559177331
bb7 =  511653057772527860215326595441963923981275988615618656624904922719101745443980 4
347024626691866080512704659468525933164826837067743034167529124014613706958

E_Zero =  (306322666255279299117145364422886150981159281448773363151934657812776334839 5
725047883297655915853405585851830102629292243349356223433243941560676199106 41, 671815
608789572042977841920137125508991874502201434796672427560748358232273669087287960363 3
353236619172226135239971159345827826256533812461234115924758 69)
E_left_unit =  (211605022637636961392925065467786236052498462785043456647813996978 2578
540816032418386753933680624610527263054410939832341060210028599625586587703091997 49, 6
144340745982368747943866281826965920705004789655290698313096862959229220684970787754 73
462635368571980724065777272357631817501293353270854916099072151378 0)
E_minus_one =  (401040309872921636841365663377986065909820100112503269656055318647294 8
155975417677379841378151082200644440605794318752145638502418266862295633649306215 33, 7
291971429809072111612972120915544259132485254373405235135454352007935424788410958004 47
264031696160402720456927527065555099055231777405394308583246343795 8)

g =  (234192202806132359274091888964324067706651365341994425333681071552947755864032 78
874089554962024758081959478945405449298478706408839937167758426604301293 22, 3082051317
723056156737907829456221478610697838887434848349554565178422378930893709998548934624 38
507189115702289276909387505162085230237246930418039434578 3)

g_q =  (753977035802701235943706185323042237812339564575739679635113936471902506962003 9
472616899596556463324601697849260815733995012576239359781143982818056107202, 36076595
228283331533789150572294988888586875114445805545482930252164024568100364460950355138 95
226320529271359437132338074364498193032144979755885960984394)

Public power index B =  [502792797372847391234025937432719229780476218272214751664633 5
541395661511277580116300702861812588785383761946819932269070157706054479963729912618 37
7298809, [203, 163, 166, 12, 244, 239, 238, 229, 142, 4, 175, 71, 61, 200, 166, 197,
187, 230, 83, 141, 218, 111, 175, 77, 187, 111, 28, 26, 1, 106, 148, 206, 129, 111,
197, 116, 241, 28, 157, 234, 156, 236, 186, 247, 87, 183, 21, 187, 241, 151, 105, 50,
216, 38, 205, 119, 154, 41, 89, 58, 224, 55, 101, 2], 257]

PrivateKey =  (751131897839397881142150556919725456145945157093760606937701312013978 91
249717968662625802397350895134338610096302846482528426150822694203652541852857236 87, 3
624786729732340756087298060344710483472215618250275593867958658646450172550034329976 74
708360456729554209211516604765624831433463427869036798344382652598 2)

PublicKey =  (282397933402739624851353339743454110489448364984710341351728711830953997 0
891321949531419387450511796827232693835259819272114042369841015781044603687575417, 59
048758014626434623886909609234856682420449823767325716542569390001546072795360954481 16
684465955641911604101880825115486830870540875238160009001375141885)


Signing the message:  Try to sign this message!
signature =  ((775850414173953667326401163243167461614777264719959593725186413443843 87
220653596072924783018511509288025891134729809215658603033061657606684244309532604 43, 4
065833214001259882876533384956770551746127111082536770807222133490856428068243000120 41
208251110004906324827177588763335216744930182653503965003237506197 0), (723965930260528
331576370725477810588666871366076214384892530596641607541878353962841488063208405479 68
889848928285275028451733920359894086358023229854562 36, 942064461608271496975342528954 4
288980632515619835182328997021037171011559867671688402309106564452218056894253708439 19
128799424719526811569736450277244583 7))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False


In [76]: # The results of this code are produced in SageMath 9.2
         # To ensure the reproducibility of the results, set a fixed initial seed
         set_random_seed(0)
         random.seed(0, version=2)
         print("initial seed: ", initial_seed())

         lambda_bits = 512
```

```
        base = 17
        GenKey_Signature02(lambda_bits, base)

        print()
        Message = "Try to sign this message!"
        print("Signing the message: ", Message)
        signature = Sign_Signature02(PrivateKey, Message, base, lambda_bits)
        print("signature = ", signature)
        print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
        print()
        # Corrupt the Message
        Message = "Try to sign this message"
        print("An attampt to verify a signature of a corrupted message")
        print("Verify: ", Verify_Signature02(PublicKey, Message, signature, base, lambda_bits))
```

```
p =  10055855947456947824680518748654384595609524365444295033292671082791323022555160232601405723625177570767523893639832604057236251775707675238936398645381403154121089599274598252367545971990
log_2(p) =  512

q =  50279279737284739123402593743271922978047621827221475166463355413956615112775801163007028618125887853837619468199322690701577060544799637299126183772988090
log_2(q) =  511

Information about the finite field:  Finite Field of size 10055855947456947824680518748654384595609524365444295033292671082791323022555160232601405723625177570767523893639864538140315412108959927459825236754597199

aa1 =  83935918861901974858474028487116401335883804353076103369044876900480495562486931813176486958330752330825582453682271667505092519710631641187845815430204490
aa3 =  17315533069134108558273843555971238892857381371061189083674566127556202224786850849448736292603841223456589543568035951240117617020776166943316895576306700
aa4 =  44146175638470886709187271664613089661753652275953466872688023416061462129692931830693602077182479682997002335121311863507326131792499551758098187118637030
aa8 =  12346935758284025454231765804837216308495771586250190682855324600197298965808849664507449024836183706276752096238972039732103027024166103126763293962050210
bb1 =  67196689544311280037300004323700907418530876214104806223219748394985754776578539491358530203173981575440920917638977428699724254371549595092160273175168600
bb2 =  11411609848460511975991390430460534796134642902247416501149967876135693229836427233554839387356176873542780906034808700148621711128447046870634932946891440
bb5 =  72194136384935669263777342913031843922139792243611940147561811492499984888624966807018022386055669893507180054999211661270908055548303615439184147944575700
bb7 =  11489227562021563290837180286937392022381090610618623562654939751302531856468723105708103240917066711421574226342695750938748042059611576942405848212919232552
E_Zero =  (71477627116264972676229290623827397652285038568229493189070554680371886362530765479581954038027110217245379625910505346003604236780802095699205158805617380765352475247329831370557216327167044553201461764714996580059159775856568207480374571845794642562788130001576223205214524244597761892214083554608380082022030)
E_left_unit =  (84515832718324880302480234058253105457965881493748212524508286990145766876544162280143548252756653537410380235351307205594554852830350204263087995466504820606929496624927642414408801116030848097267228219949798277562849821501847157626713213495390873598563151802928515221945314993810029759341593605939025578954)
E_minus_one =  (58439421514205065049978347189401689846604195642710773853632822370590085848517368679020359823297566897080379016469703486412653620731253987135322322144729940,319036540538409080701651571566299346693350075927875448444159802255848877178328796855425781401378149208098842248595642139361033023972354765749050023622825 3)

g =  (71068140849682752539647752707098382922684741070234575901556786798357877735250429001600572194451875181366070789722519287129807631709737622762982045641545620, 9719630868722778956523617763374675869954047334242973194568651459723915405300258958386090093179178125395727878350290706472036478462601680987913167659442 00)

g_q =  (46256750126570640908585290942959680885992243893333995616750942281988638174309976961889170190608374079498826451251224294324919959943512726995201683032240 7, 82639469055992730350373538165454867414427589645661579642255188610807294410506570556594577657589238468187558784997665457637246410768321755782643972422287 3)

Public power index B =  [50279279737284739123402593743271922978047621827221475166463355413956615112775801163007028618125887853837619468199322690701577060544799637299126183772988090, [11, 12, 3, 10, 6, 10, 12, 0, 4, 15, 15, 14, 14, 14, 5, 14, 14, 8, 4, 0, 15, 10, 7, 4, 13, 3, 8, 12, 6, 10, 5, 12, 11, 11, 6, 14, 3, 5, 13, 8, 10, 13, 15, 6, 15, 10, 13, 4, 11, 11, 15, 6, 12, 1, 10, 1, 1, 0, 10, 6, 4, 9, 14, 12, 1, 8, 15, 6, 5, 12, 4, 7, 1, 15, 12, 1, 13, 9, 10, 14, 12, 9, 12, 14, 10, 11, 7, 15, 7, 5, 7, 11, 5, 1, 11, 11, 1, 15, 7, 9, 9, 6, 2, 3, 8, 13, 6, 2, 13, 12, 7, 7, 10, 9, 9, 2, 9, 5, 10, 3, 0, 14, 7, 3, 5], 17]

PrivateKey =  (96019602048467288116915418429241365458419425255988323197832047188143102777386902025613274284142366860647073969842854313467256966807124667527667962372679 14, 693890311738547768609917766168063308976265902299139159278232256610397662549588851092775691943472210960360578513917612986161023016797885543987728878384059)

PublicKey =  (52249586273890011208243576684997976917079146375997270292971295532892959319002595505268018752596705933370273514204182990992396017317325478554596215413217 4, 15802902353925410164839577522610516475571445729711654310875020796287638803360219730445147904281094475789057953289153052295011358460992623598930778649882 1)
```

```
Signing the message:  Try to sign this message!
signature =  ((828636380858467621807495607606573790216468931964603854804867569342281515
580110510660606656204896455040313291195700104541800793426469832730242319582350685 40, 3
282157584662531229347005487854262586687925818000069018922806719677877646315712844341 02
414280910398096838918579463493102361949307104489617324325638498414 9), (290940477350525
938881176192878970102692208385388849636663246315068030363020836932872556731043002698 17
298345723918677870649857525001246676065310495373136 21, 736730977654411229416938095102 8
395914703177912370356620417356652682007016442905245468286070778375333162590792057547 95
122520643784198454972355216242655763 1))
Verify:  True

An attampt to verify a signature of a corrupted message
Verify:  False
```